Forms

Gustav Meglicki (code and weave)

February 5, 2010

Contents

1 Copyright and Authors **6**

2 Introduction **7**
2.1 Building the Code . . . 8
2.1.1 Parallel Build and Runs . . . 12
2.2 The Input File . . . 15
2.2.1 The `grid` group . . . 15
2.2.2 The `pml` group . . . 16
2.2.3 The `signal` group . . . 16
2.2.4 The `media` group . . . 19
2.2.5 The `iterate` group . . . 25
2.2.6 The `output` group . . . 25
2.2.7 The `post` group . . . 27
2.2.8 The `watch` group . . . 33
2.3 Postprocessing and Visualization . . . 35
2.3.1 Chombovis . . . 35
2.3.2 VisIt . . . 36
2.4 Defining Scheme Functions in C . . . 40
2.4.1 Two Dielectrics . . . 41
2.4.2 A Drude Metal . . . 43
2.5 Input for Parallel Execution . . . 59
2.6 Tests . . . 60
2.6.1 Plane Wave Propagation . . . 60
2.6.2 Refraction . . . 62
2.6.3 Mie Scattering . . . 63
2.6.4 Mie Scattering on Silver . . . 67
2.6.5 Fluxes . . . 69
2.6.6 Mie Flux . . . 73
2.6.7 Virtual Measurements . . . 75
2.6.8 Fourier Analysis . . . 80
2.7 The Library . . . 87
2.7.1 Constants . . . 87
2.7.2 Vectors . . . 90
2.7.3 Signals . . . 95
2.7.4 Media . . . 101
2.7.5 Measurements . . . 106
2.7.6 GSL—GNU Scientific Library . . . 122
2.7.7 Mie . . . 125
2.8 Annotated Examples . . . 128
2.8.1 Snell's Law . . . 128
2.8.2 Fourier Analysis . . . 130
2.8.3 Mie Scattering . . . 132

2.8.4 Mie Scattering on Silver . . . 135
2.8.5 Mie Scattering on Silver in Parallel . . . 137
2.8.6 Power Flux Map . . . 139

3 Numerics **144**
3.1 Maxwell Equations . . . 145
3.2 Discretization . . . 147
3.3 Signal Injection . . . 149
3.3.1 Relationship to Formulæ in Taflove and Hagness . . . 159
3.4 UPML Boundary . . . 160
3.5 Fourier Transforms and Fluxes . . . 166
3.5.1 Energy Flux . . . 167
3.5.2 Evaluation of Fluxes . . . 169
3.5.3 Parallel Utilities for Scheme . . . 181
3.5.4 Grid Field Interpolation on an Arbitrary Point within the Cell . . . 183

4 Guile Wrap **192**

5 Code Outline **193**
5.1 Process an Input File . . . 195
5.2 Build Initial Condition . . . 197
5.2.1 Build a Multigrid . . . 197
5.2.2 Fill the Multigrid . . . 197
5.3 Iterate . . . 198

6 Initialization **201**
6.1 Build Levels . . . 202
6.1.1 Variable Definitions . . . 202
6.1.2 Read Grid Specifications . . . 204
6.1.3 Build Level 0 Grid . . . 206
6.1.4 Build Higher Levels . . . 208
6.2 Fill Levels . . . 213

7 Iteration **219**
7.1 Advance the $\boldsymbol{D}$ field . . . 220
7.1.1 Advance Level Zero (with UPML) . . . 220
7.1.2 Advance Higher Levels . . . 222
7.2 Inject the $\boldsymbol{D}$ field . . . 224
7.2.1 Incident Field Functions . . . 226
7.3 Convert $\boldsymbol{D}$ to $\boldsymbol{E}$ . . . 229
7.3.1 Function `d_to_e` . . . 229
7.3.2 Function `d_to_e_0` . . . 229
7.3.3 Function `d_to_e_0_0` . . . 230
7.3.4 Function `d_to_e_0_n` . . . 231
7.3.5 Function `d_to_e_n` . . . 233
7.3.6 Function `d_to_e_n_0` . . . 234
7.3.7 Function `d_to_e_n_n` . . . 235
7.3.8 Scheme Driven $\boldsymbol{D}$-to-$\boldsymbol{E}$ Conversion . . . 236
7.4 Advance the $\boldsymbol{B}$ field . . . 238
7.4.1 Advance Level Zero (with UPML) . . . 238
7.4.2 Advance Higher Levels . . . 240
7.5 Inject the $\boldsymbol{B}$ field . . . 242
7.6 Convert $\boldsymbol{B}$ to $\boldsymbol{H}$ . . . 245
7.6.1 Function `b_to_h` . . . 245
7.6.2 Function `b_to_h_0` . . . 245

7.6.3 Function b_to_h_n . . . 246

8 Scheme Functions **248**
8.1 Draw Box . . . 249
8.1.1 Drawing Box on Media . . . 250
8.1.2 Drawing Box on Tags . . . 252
8.2 Draw Ball . . . 253
8.2.1 Drawing Ball on Media . . . 254
8.2.2 Drawing Ball on Tags . . . 256
8.3 Draw Ellipsoid . . . 257
8.3.1 Drawing Ellipsoid on Media . . . 258
8.3.2 Drawing Ellipsoid on Tags . . . 261
8.4 Draw Cylinder . . . 262
8.4.1 Drawing Cylinder on Media . . . 263
8.4.2 Drawing Cylinder on Tags . . . 266
8.5 Draw Cone . . . 267
8.5.1 Drawing Cone on Media . . . 268
8.5.2 Drawing Cone on Tags . . . 271
8.6 Draw Disk . . . 272
8.6.1 Drawing Disk on Media . . . 273
8.6.2 Drawing Disk on Tags . . . 275
8.7 Draw Arbitrary Ellipsoid . . . 276
8.7.1 Drawing Arbitrary Ellipsoid on Media . . . 277
8.7.2 Drawing Arbitrary Ellipsoid on Tags . . . 279
8.8 Draw Parallelepiped . . . 280
8.8.1 Drawing Parallelepiped on Media . . . 281
8.8.2 Drawing Parallelepiped on Tags . . . 283
8.9 Draw Torus . . . 284
8.9.1 Drawing Torus on Media . . . 285
8.9.2 Drawing Torus on Tags . . . 287
8.10 Draw Lens . . . 288
8.10.1 Drawing Lens on Media . . . 289
8.10.2 Drawing Lens on Tags . . . 291
8.11 Auxiliary Scheme Procedures . . . 292
8.11.1 Is a Symbol Defined . . . 292
8.12 Functions for MPI Connectivity . . . 293
8.13 Interpolation . . . 295
8.14 Initialize Scheme . . . 299

9 IO Functions **304**
9.1 Dump Data . . . 306
9.2 Precompute Fields for Output . . . 309
9.2.1 Space Interpolate Electric Fields . . . 309
9.2.2 Space and Time Interpolate Magnetic Fields . . . 310
9.2.3 Evaluate Energy . . . 311
9.2.4 Evaluate Poynting Vector . . . 311
9.2.5 Space Interpolate Electric Media Distribution . . . 312
9.2.6 Space Interpolate Magnetic Media Distribution . . . 312
9.2.7 Evaluate User Defined Fields . . . 313
9.3 Transfer Fields to the Output Object . . . 314

10 Auxiliary Functions **315**
10.1 Is It a Power of Two . . . 316
10.2 Is File Readable . . . 317
10.3 Effective Grid Bounds . . . 318
10.4 Make Tag Set . . . 320
10.5 The UPML sigma . . . 322
10.6 Marking Regions . . . 323
10.7 Return Third Direction . . . 325
10.8 Mark Total Field Region Faces . . . 326
10.9 Fill PML Arrays . . . 329
10.10 Swap New-Old Indexes . . . 331

11 Header File **332**
11.1 Chombo Includes . . . 333
11.2 Program Constants . . . 334
11.3 C++ Functions . . . 336
11.4 Scheme Functions . . . 337
11.4.1 Drawing Figures . . . 337
11.5 Fortran Functions . . . 341
11.5.1 Update Functions . . . 341
11.5.2 $\boldsymbol{D}$-to-$\boldsymbol{E}$ Conversion Functions . . . 342
11.5.3 $\boldsymbol{B}$-to-$\boldsymbol{H}$ Conversion Functions . . . 343
11.5.4 Signal Injection Functions . . . 344
11.5.5 Output Functions . . . 345
11.6 Structure **level** . . . 347

12 Chombo Extensions **349**
12.1 Scheme Parser . . . 350
12.1.1 **SCMParmParse** Class Headers . . . 350
12.1.2 **SCMParmParse** Class Implementation . . . 353
12.2 **EdgeFab**⟨**T**⟩ Class Template . . . 398
12.2.1 **EdgeFab**⟨**T**⟩ Headers . . . 400
12.2.2 **EdgeFab**⟨**T**⟩ Implementation . . . 403
12.3 **FaceFab**⟨**T**⟩ Class Template . . . 414
12.3.1 **FaceFab**⟨**T**⟩ Headers . . . 414
12.3.2 **FaceFab**⟨**T**⟩ Implementation . . . 415
12.4 **CurlBox** Class . . . 421
12.4.1 **CurlBox** Class Headers . . . 421
12.4.2 **CurlBox** Class Implementation . . . 422

13 Fortran Subroutines **428**
13.1 Subroutines `update_d` and `update_d_upml` . . . 429
13.2 Subroutines `update_b` and `update_b_upml` . . . 433
13.3 $\boldsymbol{D}$-to-$\boldsymbol{E}$ Conversion Subroutines . . . 437
13.4 $\boldsymbol{B}$-to-$\boldsymbol{H}$ Conversion Subroutines . . . 448
13.5 Fortran Subroutines for Drawing on Chombo Data Structures . . . 453
13.6 Subroutine `inject_d` . . . 486
13.7 Subroutine `inject_b` . . . 491
13.8 Output Subroutines . . . 496

14 Makefile and srcMakefile **506**

15 Closing Comments **515**

Bibliography **516**

1 Copyright and Authors

This software is copyrighted © by the Argonne National Laboratory and Indiana University. Permission is granted to reproduce this software for non-commercial purposes provided that this notice is left intact.

This software is provided as a professional and academic contribution for joint exchange. Thus it is experimental, is provided "as is", with no warranties of any kind whatsoever, no support, no promise of updates, or printed documentation. By using this software, you acknowledge that the Argonne National Laboratory and Indiana University shall have no liability with respect to the infringement of other copyrights by any part of this software.

Portions of this code, namely, the **EdgeFab** and **FaceFab** templates, the **CurlBox** class, and related utilities derive—with some minor modifications—from the Chombo **FluxBox** code and Dan Martin's **EdgeDataBox** code. Both carry the following copyright note:

> This software is copyrighted © by the Lawrence Berkeley National Laboratory. Permission is granted to reproduce this software for non-commercial purposes provided that this notice is left intact.
>
> It is acknowledged that the U.S. Government has rights to this software under Contract DE-AC03-765F00098 between the U.S. Department of Energy and the University of California.
>
> This software is provided as a professional and academic contribution for joint exchange. Thus it is experimental, is provided "as is", with no warranties of any kind whatsoever, no support, no promise of updates, or printed documentation. By using this software, you acknowledge that the Lawrence Berkeley National Laboratory and Regents of the University of California shall have no liability with respect to the infringement of other copyrights by any part of this software.

This document, is a *weave* of an original WEB source, which was written by Zdzisław Meglicki of Indiana University for the Argonne National Laboratory. CWEB-3.64 by Silvio Levy and Donald E. Knuth, as well as LaTeX classes RCS-2.10 by Joachim Schrod and Jeffrey Goldberg and CWEB-3.6 by Joachim Schrod had been used in its preparation.

2 Introduction

The Forms code solves Maxwell equations in vacuum and in the Lorentz media in three dimensions using the Finite Difference Time Domain (FDTD) method. It is hoped that the code will add multigrid integration eventually, and parts of the code have been already developed for this purpose, but for the time being they are in limbo.

The program uses Fortran stubs for basic FDTD computations [20, 21], Chombo C++ wrapper for general logic and multigrid operations [4], and Guile, which is a dialect of Scheme [11], for media layout specifications, for media properties specifications, for injected signal specifications, for output specifications, and for virtual measurements. This layer of software is further enriched by provision of an external Scheme library, which is discussed in Section 2.7.

The standard Chombo library is additionally enhanced by new templates and classes that provide basic utilities for face and edge mounted data in parallel computing context. These borrow from functions of the **staggeredChombo** package mercifully contributed to the project by Dan Martin of the Berkeley National Lab. Bless him! The **staggeredChombo** classes implement divergence-free averaging from fine to coarse grids, curl-refluxing corrections on the coarse side of the fine/coarse grid boundaries, and piecewise linear interpolation filling of cells on the fine side of the fine/coarse grid boundaries for face-mounted data. See [1, 15, 16] for more details.

The program also builds on experiences gained and ideas developed in an earlier two dimensional AMR FDTD program called Shapes, which was described in [18].

Throughout this document, issues and code fragments that are slippery are marked with the Knuth dangerous bend sign placed in the margin. Issues and code fragments that call for special attention, but are not exactly in the slippery category, are marked with the black triangle pointing left—as shown in the margin of this paragraph.

2.1 Building the Code

The code is written as a collection of CWEB files complemented by some raw F77code. Consequently, the build comprises two steps. First, we need to convert the CWEB input to plain C++ and C code. Then we need to compile C, C++, and F77 and link it all with Chombo. The two steps are automated by invoking two Makefiles, which are listed in Section 14, page 506.

The first Makefile, called `srcMakefile`, extracts the program source from this CWEB document, then prettifies it by stripping CWEB comments and blank lines with `sed` and by indenting it with `astyle`. Finally, it moves most of the source so generated to the `src` subdirectory leaving only `Forms.cpp` and `Forms.H` in the main code directory, which is as Chombo wants it nowadays. The original CWEB files, and any auxiliary files generated on the way, are removed. The procedure leaves a clean C++/C/Fortran source behind. Another thing that the target `all` of `srcMakefile` does is to format this document and leave it on `doc/Forms.pdf`. Another similar target `big-all` does the same, but adds a split document to the `doc` directory, too. The split document separates the code from the Introduction, Numerics, and Bibliography (written on `doc/Forms-1.pdf`) onto `doc/Forms-2.pdf`.

> A user or an installer should not be bothered about this step, because the program is distributed with `make -f srcMakefile` already run, and the source ready for compilation. Few production systems have all the software required by `srcMakefile` available. ◀

Other specific targets that `srcMakefile` knows about are `src-only`, `src`, `doc`, `clean`, and `clobber`. The latter removes everything and restores the directory to its virgin condition, with all files in the RCS directory. Target `clean` removes temporary and auxiliary files, but leaves the source and the PDF document behind.

GNU `make` should know how to handle RCS directories and any files within. Hence

```
$ make −f srcMakefile
```

should accomplish all of the above, without the installer having to pull anything from the RCS directory manually. The command should extract `srcMakefile` first and then follow steps specified within.

The generated source files should be as follows:

`Forms.cpp` The C++ code mainlines. Discussed in Sections 4, page 192, and 5, page 193.

`Forms.H` The C++ code headers.

`src/Auxiliary.cpp` This file contains code for various auxiliary functions, as defined in Section 10, page 315.

`src/Conversion.c` This file contains plain C wrappers for calling Scheme functions that convert the $\boldsymbol{D}$ fields to $\boldsymbol{E}$ fields. The wrappers are invoked from Fortran, which is why they have to be done in plain C. In future $\boldsymbol{B} \to \boldsymbol{H}$ conversions may be added as well. Discussed in Section 7.3.8, page 236.

`src/Conversion.h` Headers used by `src/Conversion.c`

`src/CurlBox.H` This file contains headers used by the **CurlBox** class.

`src/CurlBox.cpp` The C++ implementation of the **CurlBox** class. Discussed in Section 12.4, page 421.

`src/EdgeFab.H` Headers used by the **EdgeFab** template.

`src/EdgeFabImplem.H` Implementation of the **EdgeFab** template. Discussed in Section 12.2, page 398.

`src/FaceFab.H` Headers used by the **FaceFab** template.

`src/FaceFabImplem.H` Implementation of the **FaceFab** template. Discussed in Section 12.3, page 414.

`src/IO.cpp` C++ code for dumping three-dimensional HDF5 field images. In future, generation and reading of restart files will be added too. Discussed in Section 9, page 304.

`src/Initialize.cpp` This file contains code that generates and fills Chombo levels. At present, the code uses one level only, but functions defined here know how to build more. Discussed in Section 6, page 201.

`src/Injection.c` This file contains C wrappers for Scheme functions, *ex_lambda*, *ey_lambda*, and *ez_lambda*, to be defined by the user. The C wrappers are called by Fortran functions defined on `inject_b.f` and `inject_d.f`, which are, in turn, called from `Iteration.cpp`. The wrappers are discussed in Section 7.2.1, page 226.

`src/Injection.h` This file contains C language headers required by `src/Injection.c`. It is discussed in Section 7.2.1, page 228.

`src/Iteration.cpp` This file contains Chombo wrappers that invoke the computational guts of the program: Fortran routines that advance the $\boldsymbol{D}$ and $\boldsymbol{B}$ fields, routines that inject and extract the incident field into and from the total field region, and routines that convert $\boldsymbol{D}$ to $\boldsymbol{E}$ and $\boldsymbol{B}$ to $\boldsymbol{H}$. Its content is discussed in Section 7, page 219.

`src/SCMParmParse.H` The headers for the **SCMParmParse** class.

`src/SCMParmParse.cpp` This file contains the implementation of the **SCMParmParse** C++/Chombo class. The class is reminiscent of the Chombo **ParmParse** class, but it works against the Scheme space and provides additional Scheme related methods. It can be compiled in a stand-alone mode that does not depend on Chombo. It is discussed in Section 12.1, page 350.

`src/Scheme.cpp` This file contains C++ definitions of native Scheme functions, such as *draw_box*, *draw_ball*, *draw_ellipsoid*, and other that may yet be defined. These are really wrappers that may call other, more specific C++ functions, for example, to operate on Chombo objects, and these are also defined here. A function that initializes Scheme at the beginning, *initialize_scheme*( ) is defined on this file, as well. These are all discussed in Section 8, page 248.

`src/b_to_h.f` This Fortran file contains definitions of functions that convert $\boldsymbol{B}$ to $\boldsymbol{H}$. The conversion within the UPML region is included. Listed in Section 13.4, page 448. It is here that user provided Scheme code is invoked through plain C wrappers.

`src/d_to_e.f` This Fortran file contains definitions of functions that convert $\boldsymbol{D}$ to $\boldsymbol{E}$. The conversion within the UPML region is included. Listed in Section 13.3, page 437. It is here that user provided Scheme code is invoked through plain C wrappers.

`src/draw.f` Fortran subroutines in this file draw media on Chombo fields. They are listed in Section 13.5, page 453.

`src/inject_b.f` Fortran subroutines in this file carry out the $\boldsymbol{B}$ field injection/extraction on the total field boundary. They call user provided Scheme code through plain C wrappers defined on `Injection.c`. They are listed in Section 13.7, page 491.

`src/inject_d.f` Fortran subroutines in this file carry out the $\boldsymbol{D}$ field injection/extraction on the total field boundary. They call user provided Scheme code through plain C wrappers defined on `Injection.c`. They are listed in Section 13.6, page 486.

`src/output.f` These are auxiliary Fortran subroutines used in computing cell-centered fields for output on the HDF5 files. Additional computations that evaluate energy and Poynting vector are defined here, as well. They are listed in Section 13.8, page 496.

`src/update_b.f` These subroutines implement leap-frog time step for the $\boldsymbol{B}$ field with or without UPMLs. They are listed in Section 13.2, page 433.

`src/update_d.f` These subroutines implement leap-frog time step for the $\boldsymbol{D}$ field with or without UPMLs. They are listed in Section 13.1, page 429.

Once the code has been fully extracted from RCS and Cweb files, the binaries are built by invoking a standard Chombo Makefile (provided in RCS as well). Because Chombo puts a lot of its own hocus-pocus into its Makefile system, it proved impractical to merge source building and binary building functions into it—hence the use of a separate `srcMakefile`. The Chombo Makefile is called `Makefile`, thus typing

$ make Forms

at the prompt builds the binary, which, depending on which specific Chombo library the `Makefile` links with, will have a long name such as

`Forms3d.CYGWIN.g++.g77.OPTHIGH.ex`

where `3d` means that this is a code for 3-dimensional simulations, and where `CYGWIN`, the operating system flag, may be replaced with `Linux` or `Darwin`, `g++` and `g77` with other C++ and Fortran-77 compiler names, for example `mpicxx` and `gfortran`, and the `OPTHIGH` label may be replaced with `DEBUG`, `PROF`, `MPI` or a combination thereof. This way the binary name contains information about what its configuration is and what system it has been compiled for.

On the Argonne Jazz cluster, which is Linux-2.4.29-rc2 currently, it is best to compile Chombo (the current version is 2.0-Oct2007) with `icpc` and `ifort`. These are standard Intel C++ and Fortran compilers. The latter does an excellent job, automatically vectorizing Fortran loops for the small vector units that are attached to Jazz node CPUs. The installer should also tell `make` to use the Intel `icc` C compiler in preference to `/usr/bin/cc`. The `make` command will therefore be ◀

$ CC=icc make Forms

The code will not run if linked against a 2-dimensional Chombo system. There are places in the code that check for this and stop its execution.

What will you need to compile the code successfully? Chombo-2.0 or later—currently Chombo-3.0—is required with HDF5 utilities compiled in. HDF5 version should be 1.6.6 or later...

... but not 1.8.x. The newly released HDF5-1.8.x is incompatible with the earlier verions, and this results in compilation errors. This can be fixed if HDF5-1.8.x is built with the 1.6 compatibility flag.

Additionally, the code links with GNU Guile. There are two flags to this effect, defined on the `Makefile`:

`XTRALIBFLAGS` which evaluates to whatever is returned by `guile-config link`, and

`XTRACXXFLAGS` which evaluates to whatever is returned by `guile-config compile`.

Guile version must be 1.8.x or later. Earlier versions of Guile do not have some of the functions that Forms uses. If Guile has to be built from scratch, GNU multiple precision library, which is a separate product, must be installed first.

The Chombo file `mk/Make.rules` must be modified to add rules for depending and building plain C programs, because the wrappers that provide a bridge between Fortran and Scheme are written in plain C. A version of `Make.rules` that works with Chombo-2.0 and *another one* that works with Chombo-3.0 are provided in the `Patches` subdirectory of the Forms directory tree. The `Patches` subdirectory is currently organized into `Chombo-2.0` and `Chombo-3.0` trees. The `Chombo-3.0` tree contains further subdirectories for (1) `All` systems, then specific ones for (2) `Cygwin`, and three more for `x86_64 GNU/Linux` systems I have worked with, (3) IU Quarry, (4) Purdue Steele, and (5) NCSA Abe. ◀

In the `All/src/BaseTools` directory, there are three patched files, which compilers complained about on several occasions. I have merely added missing includes to them.

Directories `Cygwin`, `Quarry`, `Steele` and `Abe` contain `mk` files for various code configurations, `OPT`, `DEBUG`, with `MPI` and without. The patched `Make.rules` for Chombo-3.0 and `Make.defs.locals` can be found there. Additionally, the `Cygwin` directory contains a patched `check` script, which should go into `mk`, too.

On the 64-bit systems, the `USE_64` flag in `Make.defs.local` must be set to `TRUE`, otherwise Chombo defaults to 32-bits, and the generated libraries cannot be linked together with the rest of the system.

The configuration on the `Abe` cluster was compiled with Intel's `icpc/icc` and `ifort`. The `OPT` version is highly optimized, as the compiler has *vectorized* all suitable loops automatically, both in the C++ shell of Chombo, and in Fortran subroutines.

Table 1: Utilities required to extract the source from the CWEB files and typeset the document—above the dividing line, and compile the Chombo program—below.

utility	version	notes
		Required for building the source
GNU RCS	5.7	
GNU `make`	3.81	
`ctangle`	3.5.4	
`cweave`	3.5.4	
`sed`	4.1.5	
`astyle`	1.21	
LaTeX	3.141592-1.21a-2.2	
`cweb.cls`	3.6	*LaTeX class, patch it for LaTeX2e*
`cweb.cls.patch`		*patch for LaTeX2e*
`rcs.sty`	2.10	
`dvipdfm`	0.13.2c	
		Required for building the binary
`g++`	3.4.4 or 4.4.0	`icpc 8.1` *is fine*
`g77`	3.4.4	`ifort 8.1` *is fine*
`gfortran`	4.2.2 or 4.4.0	
HDF5	1.6.6	*but not 1.8.x, unless built for 1.6 compatibility*
Chombo	2.0-Oct2007 or 3.0-Apr2009	
`Make.rules`		*lives in* `Forms/Patches`
`check`		*if building on Cygwin, lives in* `Forms/Patches`
`Forms/Patches`		*other files therein, also examples of* `Make.defs.local`
Guile	1.8.2	*must be 1.8.x at least*

Table 1 lists utilities required to extract the source and build the code. The CWEB file includes the following LaTeX packages: `cweb`, `epic`, `eepic`, `here`, `amssymb`, `amsbsy`, `graphicx`, `rcs`, `fp`, and `listings`. Every well configured research system should have most of these, perhaps with the exception of Chombo, since they're all part of Linux standard and the present day research canon.

> The LaTeX `cweb` package may be an exception. It can be downloaded from CTAN archives. When installing it to work with LaTeX2e a patch, which is provided on `cweb.cls.patch` in the top level directory of the distribution, must be applied. ◀

Once the code has been compiled, the application can be run by typing

```
$ make run
```

This will work on sequential systems only, assuming that the program's input file is in the directory, where `make run` has been invoked. Parallel build and runs are discussed in the next section.

2.1.1 Parallel Build and Runs

To build the code for parallel execution, HDF5 and Chombo must be compiled for parallel execution first. This is accomplished by

1. Enabling "Parallel HDF5" on HDF5 `configure`.
2. Pointing HDF5 to MPI C compiler, `CC=mpicc`, both on `configure` and `make`.
3. HDF5 C++ interface has to be disabled for parallel compilation.
4. Defining `MPI = TRUE` on the Chombo's `mk/Make.defs.local` file.
5. Defining `MPICXX = mpicxx` on the Chombo's `mk/Make.defs.local` file.
6. Defining `HDFMPIINCFLAGS` and `HDFMPILIBFLAGS` on the Chombo's `mk/Make.defs.local` file and pointing them towards the parallel HDF5 location, for example,

   HDFMPIINCFLAGS = −I/soft/apps/packages/hdf5−1.6.5−parallel/include
   HDFMPILIBFLAGS = −L/soft/apps/packages/hdf5−1.6.5−parallel/lib −lhdf5 −lz −lm

7. Pointing the FORMS `Makefile` `CHOMBO_HOME` variable towards the parallel Chombo library, for example,

   CHOMBO_HOME =
   /home/meglicki/Chombo−2.0−Oct2007/lib3d.Linux.mpicxx.gfortran.OPT.MPI

Normally, parallel HDF5 should be installed on an MPI system by administrators already, so it's a matter of finding where it is. As has been remarked above, HDF5-1.8.x will not work with Chombo-2.0/3.0, unless built for 1.6 compatibility. Same holds for the parallel version.

Given all the above, `make Forms` should complete the job and deliver an MPI binary named something like

```
Forms3d.Linux.64.mpicxx.gfortran.OPT.MPI.ex
```

which can be moved to the user's `bin` directory. The command `make run` will not run the program in this case. It must be submitted through a batch system and `mpirun` instead.

Running Parallel FORMS on Jazz

The batch execution and configuration depends on the batch system in use. PBS is one of the most commonly used, and installed on the ANL Jazz cluster. It is also important that data is written by FORMS on a parallel file system and not on the NFS. The latter is extremely inefficient and may flood the normally limited NFS mounted volume with production data. It must *not* be done.

The following lists a Jazz PBS script that I have used to run FORMS recently:

```
#!/bin/bash
#PBS -m abe
#PBS -M gustav@indiana.edu
#PBS -l walltime=03:00:00
#PBS -l nodes=8
#PBS -A nanophotonics
#PBS -N Forms
#PBS -q shared
#
cd /pvfs/scratch/meglicki
[ -d Dust ] || mkdir Dust
cd Dust
cp /home/meglicki/src/Forms-Apr09-1723/Examples/Dust.scm .
NN=`wc -l $PBS_NODEFILE | awk ' { print $1 } '`
echo Got $NN nodes on $PBS_NODEFILE:
cat $PBS_NODEFILE
export LD_LIBRARY_PATH=/home/meglicki/lib:/soft/apps/packages/hdf5-1.6.5-parallel/lib:$LD_LIBRARY_PATH
mpirun -np $NN -machinefile $PBS_NODEFILE \
   /home/meglicki/bin/Forms3d.Linux.mpiCC.ifort.OPT.MPI.ex \
   Dust.scm
```

The script begins with the shell specification, which is followed by a number of PBS keywords. These are

-m abe Send mail on abort, begin and end.

-M gustav@indiana.edu Mail is to be sent to *gustav@indiana.edu*.

-l walltime=03:00:00 Reserve up to three wall-time hours for the job.

-l nodes=8 Allocate eight nodes to the job.

-A nanophotonics The job is to be charged against the "nanophotonics" account (this is a PBS account, normally given to research groups, not a UNIX user account).

-N Forms The job is to be called "Forms". This is how it will appear in listings produced by `qstat`.

-q shared Submit the job to the `shared` queue. This queue is for test runs. For production work `pri0` through `pri9` queues should be used. If no queue is specified PBS will assign one of the production queues, depending on the user's privileges, automatically.

The first shell command is to change the directory to the one that lives on the Parallel File System (PVFS). Every Jazz user can have a directory there. Then we check if a PVFS subdirectory for this job exists and if it does not it is created. Having done so, we change to that subdirectory, and copy the FORMS input file, called `Dust.scm`, to it.

The command

```
NN=`wc -l $PBS_NODEFILE | awk ' { print $1 } '`
```

returns the actual number of nodes that have been allocated to the job. The next two commands list the number and the allocated node names on standard output, which is going to be saved on the `Forms.o??????` file (where `??????` expands to the job number) in the user's working directory, that is, the directory from which the PBS script has been submitted.

Before running the job itself, we *must* tell the dynamic loading system about the location of dynamic libraries that the FORMS binary needs. In this case Guile libraries live in

```
/home/meglicki/lib
```

and the HDF5 libraries live in

```
/soft/apps/packages/hdf5-1.6.5-parallel/lib
```

At the same time we must not drop the system defined `LD_LIBRARY_PATH` from our enlarged `LD_LIBRARY_PATH`, because that is where other dynamic libraries live that FORMS needs, for example, Intel Fortran, MPI and C libraries.

Finally, the MPI run command itself is issued. Here we specify the number of nodes allocated to the job with the `-np` switch and the names of the nodes with the `-machinefile` switch. This must be followed by the full path of the FORMS binary and the input file, `Dust.scm`, which here the program expects to find in the current PVFS subdirectory.

The script is submitted with the PBS command

$ **qsub** jazz_submit.sh

Its status can be viewed with

$ **qstat** −u meglicki

On successful execution, two files should be created in the user's original working directory, called `Forms.e??????` and `Forms.o??????`, that will contain data written by the script and FORMS on standard error and standard output respectively, the data that FORMS itself does not write by the means of the Chombo *pout*( ) function. Fortran and Scheme diagnostics are in this category. These two files are created by the PBS, not by FORMS.

All messages that are written by the means of the *pout*( ) function, which are all C++ messages, go to files called `pout.??` (where `??` expands to the MPI process number) in the program's PVFS working directory.

Image data generated by FORMS are written in parallel on HDF5 files in the PVFS directory.

The above invocations are *not enough* to ensure that the job really runs in parallel. For this to happen the computational domain must be additionally partitioned on the FORMS input file. This is discussed in more detail in Section 2.5, page 59.

2.2 The Input File

FORMS expects to read a single file that contains Scheme code on entry. This is discussed in more detail in Section 5.1, page 195. Specifically, if no file name is provided on the command line, the program aborts.

If a file name has been provided, it is handled by the **SCMParmParse** class constructor, which is discussed in Section 12.1.2, page 353, module ⟨ **SCMParmParse** implementation: constructors and destructors 274 ⟩. The constructor inspects the file name. If it is OK, that is, not null, and the file pointed to by the name exists and is readable, it is loaded into the Scheme by a call to *scm_c_primitive_load* (cf. "The Guile Reference Manual", Section 5.13.4).

And this is all.

Whatever is on the file is evaluated. This may include various preliminary computations, definitions of constants and definitions of lambda expressions. Having done this Scheme does not go away. It's there always in the background, ready to be interrogated or asked to perform auxiliary computations. When the job runs in parallel, there is a separate Scheme instance associated with every MPI process. Procedures are provided for the instances to communicate with each other, see Section 2.6.5, page 69, and Section 8.12, page 293, for more information about these.

Methods provided with the class **SCMParmParse** are used throughout the code to access the Scheme symbol table and values bound to the symbols either for reading or for writing. In some cases, the expected values may be lambda expressions, to be evaluated within the Scheme process. This is not going to be efficient, but it adds to the program's flexibility. Much effort has been invested in making the symbol table access operations as fast as possible.

Scheme symbol names that FORMS knows about conform to Chombo **ParmParse** class specification. They are divided into groups defined by a common prefix terminated by a dot. In place of the = character of **ParmParse** here we use Scheme's `define`, but this is the only difference. Semicolons inserted at the beginning of the line signify comments, as is traditional in Scheme and Lisp.

2.2.1 The `grid` group

A typical FORMS input file may begin with grid specification. For example,

(**define** forms.grid.level0.cells '(128 128 128))
(**define** forms.grid.level0.origin '(.0 .0 .0))
(**define** forms.grid.level0.delta 1.0)

Where **ParmParse** would expect an array of numbers, here we can use either a list, a vector, or a uniform vector. So, the first line of above could also be coded by

(**define** forms.grid.level0.cells #(128 128 128))

or, for a vector of signed 64-bit integers,

(**define** forms.grid.level0.cells #s64(128 128 128))

Prefix `forms.grid.level0` in this case says that the variables refer to the level 0 grid of program FORMS. The prefix is somewhat long, but this helps us avoid possible overwrites of FORMS variables. All program specific variables have prefix `forms.`, which is additionally extended by the specific role of the variable. Program users may define any other variables they want, and use them in auxiliary evaluations, but these should not begin with the same prefix, which should be considered *reserved.*

Binding `'(128 128 128)` with `forms.grid.level0.cells` tells FORMS that the level 0 grid should have 128 cells in each principal direction, x, y and z. We also tell FORMS that the origin of the grid, meaning, the physical space point that corresponds to the grid's `'(0 0 0)` coordinate is, in this case, $(0.0, 0.0, 0.0)$. Finally, we tell FORMS that the grid constant is 1.0.

The grid constant in FORMS is the same in all three directions. This ensures best computational accuracy and simplifies coding.

2.2.2 The `pml` group

The next group of parameters describes the UPML region at the computational domain's boundary. The UPML formalism used by the program is discussed in Section 3.4, page 160, and its implementation is discussed in Section 7.1.1, page 220, Section 7.4.1, page 238, Section 7.3.2, page 229, and Section 7.6.2, page 245. Additionally, Fortran listings of relevant UPML operations are provided in Section 13.1, page 429, Section 13.2, page 433, Section 13.3, page 437, and Section 13.4, page 448.

The prefix for this group is `forms.pml` and the required specifications are as in the following example:

```
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
```

Here `lo` and `hi` stand for low and high corners of the *non*-UPML region, that is, of the interior of the computational domain. Everything outside this box is UPML. The `sigma_max` parameter is $\sigma_{\max}$ from equation (272), page 160. We find in Section 2.6.6, page 73, that it works best when set low, e.g., to 0.5 or 0.6. Finally, the `print_arrays` parameter tells FORMS to list arrays C_{0x} through C_{4z} as defined by equations (330–344), page 164, when it is set to 1 and not print them when set to 0.

2.2.3 The `signal` group

The incident signal is defined by the `forms.signal` group. This time the specification is more involved, because an arbitrary plane wave signal propagating in an arbitrary direction can be defined functionally.

The mathematics of the procedure is discussed in Section 3.3, page 149. Its implementation is discussed in Section 7.2, page 224, and Section 7.5, page 242. The related Fortran functions are listed in Section 13.6, page 486, and Section 13.7, page 491.

The general idea here is that FORMS provides a Chombo/C++ wrapper that calls Fortran functions on the Chombo data structures. The Fortran functions, in turn, refer to plain-C wrappers, which, in turn, invoke user provided Scheme lambda expressions that characterize the signal. The reason why we cannot do this directly from the Chombo/C++ wrapper is because the Chombo data structures are stored on Fortran arrays, so accessing them from C++ or C is inefficient. But Fortran cannot call Scheme expressions directly, hence the need for the plain-C wrapper.

It's complicated and in the ideal world we wouldn't do it like this. But this is not the ideal world. Chombo was designed for Fortran specifically, not for Scheme, in the anticipation that users would write their modules pluggable into the Chombo shell in Fortran. But this would require the code to be recompiled and relinked prior to execution, and the Chombo/Fortran interface, the programmer would have to know and understand, is pretty clumsy. Our choice here is to hide this interface and ask the user/programmer to specify just the signal itself in Scheme. This process can be facilitated further by calling one of many predefined signals from the Forms library, see Section 2.7.3, page 95, and an example at the end of this section.

Here is an example of a `signal` input group:

```
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
(define forms.signal.direction '( 1.0 1.0 1.0))
;;
(define wavelength 30.0)
(define alpha 0.2)
(define delay 20.0)
(define pi (* 2.0 (asin 1.0)))
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (* 0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
```

```
  (lambda (zeta)
    (* -0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.normalized 1)
```

The `lo` and `hi` parameters specify positions of the low and high corner of the total field region. The `direction` vector (which can be provided as a list) specifies the direction of signal propagation. Then we have a list of user defined and named parameters that pertain to the injected signal. The injected signal is

$$E_x(\zeta) = \frac{1}{2}\left(1 - \tanh(\alpha(\zeta+\delta))\right)\sin\left(\frac{2\pi(\zeta+\delta)}{\lambda}\right), \tag{1}$$

$$E_y(\zeta) = -E_x(\zeta), \tag{2}$$

$$E_z(\zeta) = 0, \tag{3}$$

where $\zeta = \boldsymbol{n}\cdot\boldsymbol{r} - t$, $\boldsymbol{n}$ is a normalized signal direction vector, and $\boldsymbol{r} = (x, y, z)$. See Section 3.3, page 149, for the discussion. In the above we have also specified that

$$\lambda = 30.0, \tag{4}$$

$$\alpha = 0.2, \tag{5}$$

$$\delta = 20.0, \tag{6}$$

$$\pi = 2\arcsin 1. \tag{7}$$

The last equation sets π to the machine accuracy.

Field $\boldsymbol{E}$ is clearly perpendicular to $\boldsymbol{n}$,

$$\boldsymbol{E}(\zeta)\cdot\boldsymbol{n} = E_x(\zeta)\cdot 1 + (-E_x(\zeta))\cdot 1 + 0\cdot 1 = 0, \tag{8}$$

for all values of ζ. The program would enforce this anyway, but this is costly. So we advise Forms that this is already the case by setting the `normalized` parameter to 1.

The program assumes that $\boldsymbol{H}(\zeta) = \boldsymbol{n}\times\boldsymbol{E}(\zeta)$

The formula provided will be invoked at every point of the total/scattered field boundary, as summarized by equations (241–264), page 157, at every time step. The user may request that garbage collection be triggered after every time step by setting `garbage_collect` to 1. This is not strictly necessary, because Scheme invokes garbage collection when needed anyway.

Because variable lookups are expensive in Scheme, it is more efficient to replace calls to `wavelength`, `alpha`, and `pi`, in the above formula with the actual numbers. It is also possible to define the functions in C and precompile them, and then load the binaries into Scheme. We discuss this in Section 2.4, page 40. So what's presented here is at once the most inefficient, but also the most readable way of doing it.

Because the operation is only applied at the two-dimensional boundary of the total field region, it is less costly, generally, than operations that apply to the whole volume of the computational domain, and this is visible in performance figures, that are, in this case, tolerable.

The waveform specified here can be viewed with `gnuplot` by issuing the following command:

```
# Gnuplot file for drawing the waveform.
# Draw the form on the display.
#
set samples 192
plot [z = -128:64] \
   f(z) = 0.5 * (1.0 - tanh(a * (z + d))) * sin(2 * pi * (z + d) / w), \
   w = 30, a = 0.2, d = 20, pi = 3.14, f(z)
#
```

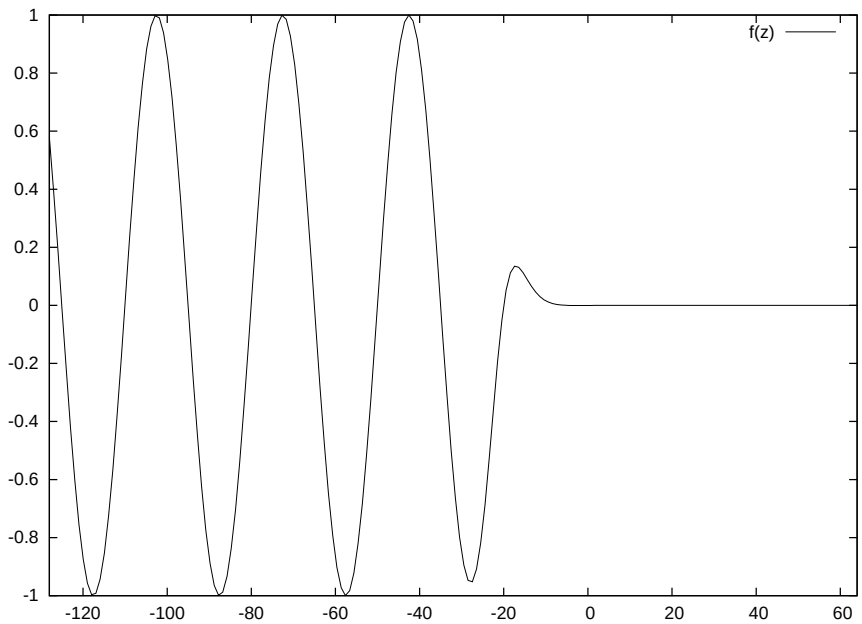


Figure 1: Gnuplot graph of the incident signal defined by equation (1).

```
# Now redraw on an Encapsulated Postscript file.
#
set term postscript eps
set output "signal.eps"
replot
```

The last three commands create an encapsulated PostScript file that contains a plot of the waveform, shown here in Figure 1. We have included it in this very document with

```
\begin{center}
  \includegraphics*[width=0.7\textwidth]{signal.eps}
  \caption{Gnuplot graph of the incident signal defined by
    equation~(\ref{e:signal}).}
  \label{f:waveform}
\end{center}
```

When specifying the signal, the user must ensure that it is zero or nearly zero within the whole computational domain at $t = 0$. Otherwise Maxwell equations are going to be broken on the total/scattered field boundary with unpredictable consequences. The signal presented in Figure 1 satisfies the requirement to within 2.5×10^{-7}. The user can check this easily by evaluating

```
(forms.signal.ex_lambda 18.0)
```

in an interactive Scheme session. A small inconsistency like this will be propagated away and absorbed by UPMLs eventually.

If we were to use the library to specify the signal, the corresponding Scheme input might look as follows:

```
;; Definition of an injected signal using the Forms library.
(set! %load-path (cons "/home/gustav/src/Forms/lib" %load-path))
;;
(load-from-path "constants")
(load-from-path "chirps")
(load-from-path "windows")
;;
(define forms.signal.lo '( 18.0 18.0 18.0))
```

```
(define forms.signal.hi '(109.0 109.0 109.0))
;;
(define forms.signal.direction '( 1.0 0.0 0.0))
;;
(define period_0 40.0)
(define alpha 0.002)
(define width 400.0)
(define omega_0 (/ (* 2 pi) period_0))
;;
(define forms.signal.ex_lambda (lambda (zeta) 0.0))
;;
(define forms.signal.ey_lambda (lambda (zeta) 0.0))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    (* (exponential-chirp-2 zeta omega_0 alpha)
       (hann zeta width))))
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.normalized 1)
```

This time we inject a signal like the one shown in example 5, Section 2.7.3, page 101. It is a shrinking exponential chirp in a Hann window, which propagates in the $\boldsymbol{e}_x$ direction and is polarized in the $\boldsymbol{e}_z$ direction. Care must be taken to ensure that even its highest frequency vibrations are adequately resolved.

2.2.4 The `media` group

The next group of parameters defines the media. Both the types of the media and their distribution must be defined here, and both are defined by lambda expressions. If a given medium is characterized by an auxiliary differential equation, the user must provide the equation solver itself, so that the program obtains a complete $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion rule from the expression.

This requirement is simplified by media expressions predefined in the FORMS library, which is discussed in Section 2.7, page 87. These cover dielectrics, Drude metals, Lorentz media with one and two poles, and mixed Drude-Lorentz models with one and two Lorentz poles.

FORMS defines two reserved symbols, `forms.media.model.e_lambda` and `forms.media.distribution.lambda`, that must be specified further by the user. Additionally, FORMS provides several low level functions, namely `draw-box`, `draw-ball`, `draw-ellipsoid`, `draw-cylinder`, `draw-cone`, `draw-parallelepiped`, `draw-lens` and `draw-disk`, that "draw" the actual media distribution on the Chombo data structures. Furthermore, FORMS defines the following symbols that can be used in user provided media specifications and that are bound to appropriate values during program execution:

`forms.direction` The field component that is currently worked on. The value is set by the program on calling the lambda. It evaluates to 0, 1 or 2.

`forms.time_e` The current time of the electric field time slice.

`forms.dt` The length of the time step. This and `forms.time_e` may have to be used by an auxiliary differential equation solver.

`forms.medium` The number of the medium within which the computation is taking place. These numbers are defined by the user in `forms.media.distribution.lambda`.

`forms.E` The value of the $\boldsymbol{E}$ field in the cell in the direction given by `forms.direction` at time $t =$ `forms.time_e`. The value is set by the program on calling the lambda.

`forms.E_old` The value of the $\boldsymbol{E}_{\text{old}}$ field in the cell in the direction given by `forms.direction` at time $t =$ `forms.time_e` $-$ `forms.dt`. The value is set by the program on calling the lambda.

`forms.D` The value of the $\boldsymbol{D}$ field in the cell in the direction given by `forms.direction` at time $t =$ `forms.time_e`. The value is set by the program on calling the lambda.

`forms.D_old` The value of the $\boldsymbol{D}_{\text{old}}$ field in the cell in the direction given by `forms.direction` at time $t =$ `forms.time_e` $-$ `forms.dt`. The value is set by the program on calling the lambda.

`forms.S` An auxiliary field array. These can be used for storing values of $\boldsymbol{E}$ or $\boldsymbol{D}$ even older than $\boldsymbol{E}_{\text{old}}$ or $\boldsymbol{D}_{\text{old}}$, or for storing values of those pesky auxiliary fields that show up in ADEs and that mask induced currents. The user must tell FORMS up-front how many of these will be needed by defining `forms.media.number_of_auxiliary_fields`. Also, the user must ensure that `forms.S` is correctly updated by the scripts provided. FORMS does not do this on its own, because it wouldn't know how.

The job of `forms.media.model.e_lambda` is to return E_i, for $i =$ `forms.direction` and other parameters listed above. This may involve an auxiliary differential equation step.

The predefined symbols are bound at the beginning of the program and their values modified or read by referencing variables bound to them, The variable addresses are acquired by FORMS on initialization. This way we avoid costly, both in terms of memory and performance, redefines of Scheme symbols and variable lookups through the symbol table. Whereas painful to the programmer the methodology should be transparent to FORMS users.

A simple example of `forms.media.model.e_lambda` may look as follows:

```
;; Two dielectrics.
;; Define dielectric constants.
;;
(define epsilon_1 4.0)
(define epsilon_2 2.0)
;;
;; Precompute inverses.
;;
(define one_by_epsilon_1 (/ 1.0 epsilon_1))
(define one_by_epsilon_2 (/ 1.0 epsilon_2))
;;
;; The D -> E script for media numbers 1 and 2.
;;
(define forms.media.model.e_lambda
  (lambda ()
    (case forms.medium
      ((1) (* one_by_epsilon_1 forms.D))
      ((2) (* one_by_epsilon_2 forms.D)))))
```

The above states that where the medium type is

1: $E_i = D_i/\epsilon_1$ in any direction $i \in \{0, 1, 2\}$, for any time $t =$ `time_e` and for any $\mathrm{d}t$;

2: $E_i = D_i/\epsilon_2$ in any direction $i \in \{0, 1, 2\}$, for any time $t =$ `time_e` and for any $\mathrm{d}t$;

where $\epsilon_1 = 4$ and $\epsilon_2 = 2$.

Symbols `forms.medium`, `forms.direction`, `forms.time_e`, `forms.dt`, `forms.E`, `forms.E_old`, `forms.D`, and `forms.D_old` are global. The values they are bound to are picked up from Chombo Fortran arrays and FORMS internal parameters for every call on a given grid cell.[1]

We do not have here explicit dependence on (x, y, z). This is because the dependence is encoded in the medium number, which is here called `medium`, and which FORMS refers to internally as `color`. It is the role of `forms.media.distribution.lambda` to tell FORMS where the medium of type `medium` (or of some specific `color`) is.

The $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion is discussed in Section 7.3, page 229.

Basically various C++/Chombo wrappers here assess the context (level 0 versus a higher level, with or without auxiliary fields) and call an appropriate Fortran routine, all of which are listed in Section 13.3, page 437.

[1] The current formalism does not really allow for a full tensoral dependence of $\boldsymbol{D}$ on $\boldsymbol{E}$, although it does allow for different actions on the principal directions x, y and z.

Because for level 0 there is also a special $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion within the UPML region, the region is assigned medium number (or `color`) of -2 by default and the Fortran routine invokes the UPML conversion there automatically. The user cannot overwrite this action. Vacuum is assigned medium number 0, and within this medium the routines assign $\boldsymbol{E} = \boldsymbol{D}$. This action is automatic too and cannot be modified. However, users can "draw" media with media number set to zero, on top of other media, which is equivalent to punching holes in the other media.

A user's script is invoked for media numbered from +1 onwards. This is done by calling a plain C wrapper from within Fortran, which is listed in Section 7.3.8, page 236.

The $\boldsymbol{B}$ field is similarly converted to the $\boldsymbol{H}$ field, with the exception that, at this stage, there is no provision for the users to submit their own conversion scripts (we plan to alter this in the future). So, the only conversion that happens is the one within the UPML region and in vacuum, where $\boldsymbol{B} = \boldsymbol{H}$. The $\boldsymbol{B}$-to-$\boldsymbol{H}$ conversion is discussed in Section 7.6, page 245, and the corresponding Fortran routines are listed in Section 13.4, page 448.

As we have remarked above, defining media is facilitated by the FORMS library provided. There are library procedures that encapsulate auxiliary differential equations for Drude, one and two-pole Lorentz and mixed Drude-Lorentz models. Each procedure has documentation attached, which can be read by evaluating `procedure-documentation` on the procedure itself. The media library procedures are discussed in Section 2.7.4, page 101. Here we include a simple example:

```
;; Definition of media types using the Forms library
(set! %load−path (cons "/home/gustav/src/Forms/lib" %load−path))
;;
(load−from−path "constants")
(load−from−path "media")
;;
(define forms.media.number_of_auxiliary_fields 2)
;;
(define one_by_epsilon_drude (/ 1.0 epsilon_LH_Ag_300_400))
(define omega_drude_square (* (/ omega_LH_Ag_300_400 unit_of_time)
                              (/ omega_LH_Ag_300_400 unit_of_time)))
(define gamma_drude (/ gamma_LH_Ag_300_400 unit_of_time))
(define one_by_epsilon_dielectric (/ 1.0 epsilon_Si3N4))
;;
(define forms.media.model.e_lambda
  (lambda ()
    (case forms.medium
      ((1) (dielectric one_by_epsilon_dielectric))
      ((2) (drude one_by_epsilon_drude omega_drude_square gamma_drude)))))
;;
```

We define two media, silicon nitride, which is a dielectric, and which is assigned number 1, and silver, which is assigned number 2, and described by the Drude model with ϵ_∞. Material constants used are provided by the `"constants"` library. The Drude model requires the use of two auxiliary fields. The epsilons are unitless, but the omegas are defined in radians per second in the `"constants"` library. These will not normally be our units of time, which is why we must scale both ω_D^2 and γ_D before use. Function `drude` takes care of managing and updating `forms.S`.

Section 2.4.2, page 43, discusses in detail (1) the Drude model, (2) its C-language implementaton, which can be loaded into Scheme as a primitive function (this is most efficient, but also most difficult for users themselves to provide), and (3) its explicit implementation in Scheme.

Here is an example of a media distribution script:

```
(define forms.media.distribution.lambda
  (lambda ()
    (draw−box (f64vector 60 60 60) (f64vector 68 68 68) 1)
    (draw−ball (f64vector 40 40 40) 5.0 1)
    (draw−ellipsoid (f64vector 70 70 70) (f64vector 80 80 80) 19.0 2)))
```

The expression draws a box on the Chombo data structures with its low corner at $(60, 60, 60)$ and its high corner at $(68, 68, 68)$ and assigns medium #1 to all cells within the box. Then it draws a ball of radius 5.0, centered on $(40, 40, 40)$ and assigns medium #1 to all cells within it, too. Finally, it draws an ellipsoid with its

foci at $(70, 70, 70)$ and $(80, 80, 80)$, the major axis of which is 19.0 units long, and assigns medium #2 to all cells within it.

Users can replicate the basic shapes provided by FORMS by invoking any valid Scheme constructs, such as `let` or `do` loops, `case`, `cond`, `if`, and the like. Predefined structure "drawings" can be loaded and further enhanced. Holes can be punctured by drawing with medium #0, that is, with vacuum. Media can be layered on top of each other.

The following more complex example distributes 512 elliptical glazed metal grains with the ellipsoid long axis aligned diagonally within the total field region of a $256 \times 256 \times 256$ domain:

```
(define dd 3)
(define dq 0)
(define sum1 12)
(define sum2 8)
(define delta 25)
(define i0 38)
(define j0 38)
(define k0 38)
(define imax 213)
(define jmax 213)
(define kmax 213)
;;
(use-modules (ice-9 format))
;;
(define forms.media.distribution.lambda
  (lambda ()
    (let ((count 0))
      (do ((i i0 (+ i delta)))
          ((> i imax))
        (do ((j j0 (+ j delta)))
            ((> j jmax))
          (do ((k k0 (+ k delta)))
              ((> k kmax))
            (let* ((f1x i)
                   (f1y j)
                   (f1z k)
                   (f2x (+ i dd))
                   (f2y (+ j dd))
                   (f2z (+ k dd))
                   (f3x (+ f1x dq))
                   (f3y (+ f1y dq))
                   (f3z (+ f1z dq))
                   (f4x (- f2x dq))
                   (f4y (- f2y dq))
                   (f4z (- f2z dq)))
              (draw-ellipsoid (f64vector f1x f1y f1z) (f64vector f2x f2y f2z) sum1 1)
              (draw-ellipsoid (f64vector f3x f3y f3z) (f64vector f4x f4y f4z) sum2 2)
              (set! count (+ count 1))))))
      (format #t "~a grains positioned" count)
      (newline))))
```

It is here that the power and flexibility of adding an extention language (and even more so as good a language as Scheme) to the code becomes apparent. The application can be used for simulating meta-materials and other large and complex systems. An interactive Scheme session can be used beforehand to evaluate, test and debug any complex expressions that are to be used in the FORMS input.

Further down, in Section 2.4, page 40, we discuss coding Scheme functions directly in C for improved performance. But there is no need to do this for the distribution lambda, because this code is executed once only. On the other hand, the `e_lambda` code, which converts $\boldsymbol{D}$ to $\boldsymbol{E}$ is executed at every non-vacuum point of the grid at every iteration. This code must be optimal for the best performance and it is here that it pays to express the function in C.

The drawing procedures that are provided for the use in the distribution lambda are rather complex and time consuming, because the drawing has to be done for every one of the six basic face and edge mounted fields separately. This is discussed in Section 6.2, page 213, where the user provided lambda is invoked for every level of the multigrid. The actual media drawing, Scheme callable C++ functions are discussed in Section 8, page 248, and the specific Fortran routines invoked listed in Section 13.5, page 453.

Scheme garbage collection after every iteration can be forced by setting

(**define** forms.media.garbage_collect 1)

Garbage collection is not invoked manually (Scheme does it anyway) when this parameter is left at 0.

The following media drawing routines are provided at present.

draw-box Draws a rectangular box.

synopsis (**draw−box** lo−corner hi−corner medium−number)

documentation The box is drawn between the low and the high corners, which have to be specified as `f64vector`s. The routine checks if the low corner is indeed lower than the high one and aborts if it isn't. `medium-number` must be an integer; it is assigned to all field attachment locations, both for $\boldsymbol{E}$ and $\boldsymbol{B}$, within the box.

draw-ball Draws a ball.

synopsis (**draw−ball** center radius medium−number)

documentation A ball of radius `radius`, which is a real number, and center `center`, which is an `f64vector`, is drawn. All field attachment locations within it are assigned the `medium-number` provided.

draw-ellipsoid Draws a prolate spheroid, i.e., something like a rugby ball.

synopsis (**draw−ellipsoid** focus−1 focus−2 sum medium−number)

documentation A prolate spheroid, that is an ellipsoid that is produced by rotating an ellipse by 360° about its *long* axis, with foci at `focus-1` and `focus-2`, which must be `f64vector`s, is drawn. `sum` is the sum of distances from a point on the ellipsoid's surface to the two foci. This number is the same for every point on the ellipsoid's surface, and thus characterizes it. It must be larger than the distance between the two foci. The function checks for this and aborts if it isn't. Every field attachment location within the ellipsoid is assigned the `medium-number` provided.

draw-disk Draws an oblate spheroid, i.e., a disk.

synopsis (**draw−disk** center normal radius thickness medium−number)

documentation An oblate spheroid, that is an ellipsoid that is produced by rotating an ellipse by 360° about its *short* axis, is drawn. Since there are no foci in this case (there is a focal ring instead) the ellipsoid is specified by providing its `center` and its `normal`, both `f64vector`s. The `normal` is a direction along the short axis of the ellipsoid. We also provide two double precision numbers, the radius and the thickness of the disk, which are a half of the long axis and the *whole* short axis of the rotated ellipse. Every field attachment location within the disk is assigned the `medium-number` provided. If the `thickness` of the ellipsoid is more than two times its `radius`, the ellipsoid becomes a prolate spheroid, as above, in which case the function provides an alternative method of parametrizing it.

draw-arbitrary-ellipsoid Draws an arbitrary and arbitrarily oriented ellipsoid.

synopsis (**draw−ary−ellipsoid** r_0 e_1 c_1 e_2 c_2 e_3 c_3 medium−number)

documentation Draws an arbitrary ("ary") and arbitrarily oriented ellipsoid, with its center at $\boldsymbol{r}_0$, and characterized by its three radii, c_1, c_2, and c_3, along its three principal directions $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, and $\boldsymbol{e}_3$. Of the three principal directions we expect only that they should be linearly independent, that is, that the volume $\boldsymbol{e}_1 \wedge \boldsymbol{e}_2 \wedge \boldsymbol{e}_3 \neq 0$. The resulting equation that defines the ellipsoid is

$$\frac{\left((\boldsymbol{r}-\boldsymbol{r}_0)\cdot\boldsymbol{e}_1\right)^2}{c_1^2} + \frac{\left((\boldsymbol{r}-\boldsymbol{r}_0)\cdot\boldsymbol{e}_2\right)^2}{c_2^2} + \frac{\left((\boldsymbol{r}-\boldsymbol{r}_0)\cdot\boldsymbol{e}_3\right)^2}{c_3^2} = 1 \tag{9}$$

Every field attachment location within the ellipsoid is assigned the `medium-number` provided

draw-cylinder Draws a cylinder.

synopsis (**draw−cylinder** end−1 end−2 radius medium−number)

documentation `end-1` and `end-2` are the centers of the two disks at both ends of the cylinder. They must be provided as `f64vector`s. The cylinder's radius is `radius`, which is a real number. Every field attachment location within the cylinder is assigned the `medium-number` provided.

draw-cone Draws a cone.

synopsis (**draw−cone** end−1 end−2 radius medium−number)

documentation `end-1` is the center of the disk at the base of the cone. `end-2` is the apex of the cone. Both points must be provided as `f64vector`s. `radius` is the radius of the disk at the base of the cone. All field attachment locations within the cone are assigned the `medium-number` provided.

draw-parallelepiped Draws a parallelepiped.

synopsis (**draw−parallelepiped** corner v−1 v−2 v−3 medium−number)

documentation The parallelepiped is spanned by the three vectors, `v-1`, `v-2`, and `v-3`, attached at the point referred to as the `corner`. All four must be specified as `f64vector`s. All field attachment locations within the cone are assigned the `medium-number` provided. The parallelepiped is a generalization of a box. Unlike a box though, it can be oriented arbitrarily. So, if one needs a normal box, but oriented arbitrarily, the parallelepiped construction can be used in its place.

draw-torus Draws a torus.

synopsis (**draw−torus** center normal radius−1 radius−2 medium−number)

documentation A torus with its center at `center`, which must be provided as an `f64vector` is drawn. The axis of the torus is given by `normal`, also an `f64vector`. `radius-1` is the inner radius of the torus and `radius-2` is the outer one. Both must be provided as reals. All field attachment locations within the torus are assigned the `medium-number` provided.

draw-lens Draws a lens.

synopsis (**draw−lens** center−1 radius−1 center−2 radius−2 medium−number)

documentation A lens is produced when two balls intersect. So, the arguments of the function are the centers and the radii of the balls. The centers must be provided as `f64vector`s and the radii as double precision reals. All field attachment locations within the lens are assigned the `medium-number` provided. For a lens to form, the distance between the ball centers must be less than the sum of their radii. The function checks for this and aborts if this is not the case.

Ellipsoids, disks, cylinders, cones, parallelepipeds, tori and lenses can be oriented arbitrarily. Boxes are always oriented along the principal directions, x, y and z. If one needs to draw a differently oriented box, one can simply draw a parallelepiped.

One can combine the predefined forms in order to produce a greater variety of shapes. For example, to construct a spherical shell, one would first draw a ball and fill it with a medium-number that corresponds to a metal. One would then draw a smaller ball within it, centered at the same point, and fill it with

medium-number zero, which is vacuum. To punch a hole in the shell, we would simply draw a cylinder crossing the shell perpendicularly, and fill it with vacuum, too. A barrel can be constructed by drawing an ellipsoid first, then cutting off its top and its bottom with vacuum filled cylinders. Similarly, we can cut off the top of a cone. We can produce a conical shell by drawing a vacuum filled cone within a metal filled one. Forms of the same `medium-number` can be made to overlap to produce even more exotic shapes, such as a droplet, which could be produced by overlapping a cone and a ball.

2.2.5 The `iterate` group

The next group describes iterations. Here is an example:

(**define** forms.iterate.number_of_steps 512)
(**define** forms.iterate.stride 2)
(**define** forms.iterate.image_frequency 16)
(**define** forms.iterate.t0 0.0)

We request a total of 512 time steps, such that 2 steps are needed to advance the incident signal across a single grid cell. This is what the `stride` parameter describes. We are going to dump images every 16 time steps, and we set the initial time to $t_0 = 0.0$.

In future releases of FORMS we will add system dump on signal and program restart from the dump.

2.2.6 The `output` group

The program's output at present is in the form of logs and HDF5 images. FORMS' internal fields are all face and edge centered, but CHOMBOVIS, a tool that is used to view the images, can only handle cell centered fields. The internal fields of FORMS are therefore interpolated onto cell centers, before they are written on the HDF5 files.

This is done on request. The user must first tell FORMS which fields must be precomputed. This implies cell centering, but also additional computations, when needed. Table 2 lists fields which FORMS knows about by default.

The output procedure, which is discussed in Section 9, page 304, works as follows. The user specifies fields to be precomputed, in such order as is required for the operation, by defining an ordered list of strings, for example

(**define** forms.output.precompute '("Ey" "Ez" "Hy" "Hz" "Px"))

In this case we precompute E_y, E_z, H_y and H_z, before we can precompute $P_x = E_y H_z - E_z H_y$, so if we were to specify

(**define** forms.output.precompute '("Ey" "Ez" "Px" "Hy" "Hz"))

the output procedure would flag an error, write a fairly informative message about what's wrong, and abort. The same applies to the energy density field $\mathcal{E}$, which requires precomputation of all components of all basic fields, $\boldsymbol{E}$, $\boldsymbol{D}$, $\boldsymbol{B}$ and $\boldsymbol{H}$.

In the future, users will be able to define their own string symbols within the `precompute` list, and supplement them with lambda scripts that will tell FORMS how to evaluate the requested fields by using the previously listed (and evaluated) ones. Fortran listings of predefined field operations are provided in Section 13.8, page 496.

Some of the precomputed fields may be auxiliary and of no more interest to us, once they've been used to evaluate other fields. For example, we may be only interested in P_x, but not in $\boldsymbol{E}$ or $\boldsymbol{H}$. For this reason, we define another list that specifies which of the precomputed fields are to be written on the HDF5 file. Here's an example:

(**define** forms.output.**write** '("Px"))

If the list contains more than one string, fields will be output on the HDF5 file in such order as is specified by the list.

Three-dimensional computations can generate enormous amount of data. For this reason it pays to be judicious in deciding what to output.

The output files are named `<root>_ddd.hdf5`, where `<root>` evalutes to a string provided on the `filename.root` variable, for example,

Table 2: Predefined cell-centered fields.

string	meaning
`"Ex"`	E_x
`"Ey"`	E_y
`"Ez"`	E_z
`"Dx"`	D_x
`"Dy"`	D_y
`"Dz"`	D_z
`"Bx"`	B_x
`"By"`	B_y
`"Bz"`	B_z
`"Hx"`	H_x
`"Hy"`	H_y
`"Hz"`	H_z
`"En"`	$\mathcal{E} = \frac{\boldsymbol{E}\cdot\boldsymbol{D}}{2} + \frac{\boldsymbol{B}\cdot\boldsymbol{H}}{2}$
`"Px"`	$P_x = (\boldsymbol{E} \times \boldsymbol{H})_x$
`"Py"`	$P_y = (\boldsymbol{E} \times \boldsymbol{H})_y$
`"Pz"`	$P_z = (\boldsymbol{E} \times \boldsymbol{H})_z$
`"eMdx"`	Media distribution for E_x
`"eMdy"`	Media distribution for E_y
`"eMdz"`	Media distribution for E_z
`"hMdx"`	Media distribution for H_x
`"hMdy"`	Media distribution for H_y
`"hMdz"`	Media distribution for H_z

(**define** forms.output.filename.root "Poynting")

and `ddd` is replaced by a null padded image number. The total number of digits used is specified by defining the `filename.number_of_digits` symbol, for example

(**define** forms.output.filename.number_of_digits 3)

If specified as above, the file names will be

```
Poynting_001.hdf5
Poynting_002.hdf5
...
```

For every field that needs to be precomputed FORMS can find outliers and their grid location, and report them. This is activated by defining

(**define** forms.output.report_outliers 1)

The media distribution can be dumped, too, by precomputing and then writing out any of `"eMdx"`, `"eMdy"`, `"eMdz"`, `"hMdx"`, `"hMdy"`, and `"hMdz"`. These symbols produce three-dimensional maps of media numbers (colors) located at the points where E_x, E_y, E_z, H_x, H_y, and H_z are attached, respectively. The media numbers are *interpolated* from their locations on cell edges or faces onto cell centers.

2.2.7 The `post` group

This is a group of lambdas to be invoked after the completion of *each* iteration, and after the completion of *all* requested iterations, before the program exits. The lambdas are *thunks,* which means that they do not take any arguments. They can return anything, but FORMS checks for a `#f` on return, and if such is detected, FORMS writes an error message and aborts.

forms.post.iteration.lambda A thunk to be executed after each iteration. It can be used to accumulate Fourier transforms, calculate fluxes, inspect the accuracy of the computational process, inject a hard source signal, etc.

forms.post.all.lambda A thunk to be executed after all iterations have been completed, just before the program exits. It can be used to cut a slice out of data and dump it on a file, or to evaluate fluxes from Fourier accumulated quantities, etc.

The main low-level utility that users may invoke here is a FORMS-Scheme function `interpolate`, which is discussed in Section 2.7.5, pages 106 and 116, Section 3.5.4, page 183, and then in Section 8.13, page 295, module ⟨Interpolation 198⟩. It provides interpolated access to all major Chombo fields. Additionally, elementary parallel programming utilities are provided as well, in case some quantities should be accumulated across the process pool or Scheme output written on process specific files. These are discussed in Section 3.5.3, page 181, and then in Section 8.12, page 293, module ⟨MPI Functions 197⟩.

There are two library modules designed for this group, `"3D-vectors"`, discussed in Section 2.7.2, page 90, and `"measurements"`, discussed in Section 2.7.5, page 106. The former defines complex-valued three-dimensional vectors as a class and provides a plethora of utilities, including dot and cross products, norm, arithmetic operations, complex number operations, and vector constants. The latter defines arbitrary two-dimensional surfaces as an operational Scheme class, and provides utilities for computation of fluxes, accumulation of Fourier transforms on surface plaquettes, as well as for generation of commonly used surfaces such as planes, disks, caps, spheres, and hemispheres.

The following lists a simple example that illustrates how to use `interpolate` to inspect the accuracy of signal propagation within the total field region.

```
;; Forms input file
;;
;; $Id: PlaneWave.scm,v 1.4 2008/05/30 20:23:00 gustav Exp $
;;
;; Inject a plane wave signal into the total field region. Test the
;; solution by probing and comparing against the analytical formula at
```

```
;; selected points. All formulations are in plain Scheme.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Auxiliary modules
;;
(use-modules (ice-9 format))
;;
;; The Grid group
;;
;; Level 0 parameters
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Signal group
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 0.0 0.0 1.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define wavelength 30.0)
(define delay -14.0)
(define pi (* 2.0 (asin 1.0)))
;;
(define heaviside
  ;; This is an inverted Heaviside, which is 0 for positive
  ;; numbers and zero for negative ones.
  (lambda (x)
    (if (< x 0.0)
        1.0
        0.0)))
;;
(define forms.signal.ex_lambda
  ;; We want the signal to be zero for zeta in [ -delay, +infty ]
  ;; and switched on to a harmonic wave for zeta < -delay.
```

```
  ;; For every point in the computational domain zeta is positive
  ;; initially and becomes increasingly negative, as time goes
  ;; by. Quite like myself, come to think of it.
  (lambda (zeta)
    (* (heaviside (+ zeta delay))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Media group
;;
;; We do not distribute any media for this run, so
;; forms.media.distribution.lambda is undefined. This will
;; trigger a warning, but the program will run.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
(define forms.iterate.number_of_steps 1028)
(define forms.iterate.stride 8)
(define forms.iterate.image_frequency 64)
(define forms.iterate.t0 0.0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Postprocessing group
;;
(define sample-points
  #(
    #(64 64 15) #(15 64 15)
    #(64 64 20) #(15 64 20)
    #(64 64 30) #(15 64 30)
    #(64 64 40) #(15 64 40)
    #(64 64 50) #(15 64 50)
    #(64 64 60) #(15 64 60)))
;;
(define nx (list-ref forms.signal.direction 0))
(define ny (list-ref forms.signal.direction 1))
(define nz (list-ref forms.signal.direction 2))
(define outside
  (lambda (point)
    (let ((xlo (list-ref forms.signal.lo 0))
          (ylo (list-ref forms.signal.lo 1))
          (zlo (list-ref forms.signal.lo 2))
          (xhi (list-ref forms.signal.hi 0))
          (yhi (list-ref forms.signal.hi 1))
          (zhi (list-ref forms.signal.hi 2))
          (x (vector-ref point 0))
          (y (vector-ref point 1))
          (z (vector-ref point 2)))
```

```
      (or (or (< x xlo) (> x xhi))
          (or (< y ylo) (> y yhi))
          (or (< z zlo) (> z zhi))))))
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format #t "post.iteration: t = ~a~%" forms.time_e)
    (do ((number-of-points (vector-length sample-points))
         (count 0 (1+ count)))
        ((>= count number-of-points) #t)
      (let* ((point (vector-ref sample-points count))
             (x (vector-ref point 0))
             (y (vector-ref point 1))
             (z (vector-ref point 2))
             (Ex (interpolate "Ex" x y z))
             (Ey (interpolate "Ey" x y z))
             (Ez (interpolate "Ez" x y z))
             (zeta (- (+ (* nx x) (* ny y) (* nz z)) forms.time_e))
             (E-expected (if (outside point)
                             #(0 0 0)
                             (vector (forms.signal.ex_lambda zeta)
                                     (forms.signal.ey_lambda zeta)
                                     (forms.signal.ez_lambda zeta))))
             (Ex-expected (vector-ref E-expected 0))
             (Ey-expected (vector-ref E-expected 1))
             (Ez-expected (vector-ref E-expected 2)))
        (format #t "post.iteration: at ~a:~%" point)
        (format #t " Ex: is ~a, expected ~a~%" Ex Ex-expected)
        (format #t " Ey: is ~a, expected ~a~%" Ey Ey-expected)
        (format #t " Ez: is ~a, expected ~a~%" Ez Ez-expected)))))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Output group
;;
;; There is nothing here: we don't want any images dumped.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Watch group. This is just for debugging.
;;
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
```

```
(define forms.watch.fill_PML_arrays 1)
(define forms.watch.scheme 0)
;;
;; end
;;
```

In this computation we specify the injected signal explicitly in Scheme. The injected wave is a step-function switched sine wave. The `forms.post.iteration.lambda` makes use of some externally defined globals that include function `outside`. The function returns `#t` if a point is outside the total field region, and `#f` otherwise.

The lambda iterates over the sampling points in the list and extracts $\boldsymbol{E}$ for each point from Chombo fields. Then it uses the lambdas provided in the `signal` group to find the expected $\boldsymbol{E}$ for this point, and prints both values on standard output.

There is no media distributed in this model and no images are dumped on HDF5 files. Relative errors observed are within about ±0.8%, and do not appear to grow. The following listing shows part of the output generated by the lambda.

```
post.iteration: t = 128.0
post.iteration: at #(64 64 15):
   Ex: is −1.93409799851751e−5, expected 0
   Ey: is 5.09128500952072e−7, expected 0
   Ez: is 2.66775666574887e−5, expected 0
post.iteration: at #(15 64 15):
   Ex: is 5.16944686224109e−5, expected 0
   Ey: is −8.80573088890072e−8, expected 0
   Ez: is 1.75827337129027e−4, expected 0
post.iteration: at #(64 64 20):
   Ex: is −0.403658730273346, expected −0.406736643075798
   Ey: is 6.29730745629832e−7, expected 0.0
   Ez: is −3.66319327376315e−5, expected 0.0
post.iteration: at #(15 64 20):
   Ex: is 3.19872185509662e−4, expected 0
   Ey: is −3.55634434937554e−7, expected 0
   Ez: is 4.28928271708507e−4, expected 0
post.iteration: at #(64 64 30):
   Ex: is 0.988409435595339, expected 0.994521895368273
   Ey: is −1.0802588447781e−6, expected 0.0
   Ez: is −8.20706324455865e−5, expected 0.0
post.iteration: at #(15 64 30):
   Ex: is −2.67406011953885e−4, expected 0
   Ey: is 3.29926116390102e−7, expected 0
   Ez: is −0.00153652376513669, expected 0
post.iteration: at #(64 64 40):
   Ex: is −0.591484583875268, expected −0.587785252292474
   Ey: is 5.86970903953327e−7, expected 0.0
   Ez: is 1.25357155609607e−4, expected 0.0
post.iteration: at #(15 64 40):
   Ex: is −0.00235729100613782, expected 0
   Ey: is 3.42300847388829e−6, expected 0
   Ez: is 2.9930778627703e−4, expected 0
post.iteration: at #(64 64 50):
   Ex: is −0.394697291173291, expected −0.406736643075798
   Ey: is 6.60377126333706e−6, expected 0.0
   Ez: is 5.81628253284223e−4, expected 0.0
post.iteration: at #(15 64 50):
   Ex: is 0.00419307730165257, expected 0
   Ey: is 2.04382808918296e−6, expected 0
   Ez: is 0.00260341693855163, expected 0
post.iteration: at #(64 64 60):
   Ex: is 0.986963208923196, expected 0.994521895368273
   Ey: is −8.43209979815623e−6, expected 0.0
```

```
    Ez: is 0.0011708265679426, expected 0.0
post.iteration: at #(15 64 60):
    Ex: is −0.00115764928088425, expected 0
    Ey: is −8.05525728300801e−6, expected 0
    Ez: is −0.00293205694363568, expected 0
```

Points where either x, y, or z is 15 are outside the total field region, and any signal observed there is due to the discrepancy between the analytically injected signal and its numerical propagation along the total/scattered field boundary. Similarly, E_y and E_z within the total field region should be zero, but aren't on account of the discrepancy and numerical dispersion. The size of the E_y and E_z signal within the total field region provides us therefore with an additional measure of the code's accuracy. As we are going to show below, reduction in the grid spacing, reduces this signal, as well.

If the job runs on a parallel system, then we should make Scheme log on separate files for each process. The following illustrates the required changes.

```
(define my−rank (proc−id))
(define pool−size (num−proc))
(define my−log (open−file (format #f "scm_out.~a" my−rank) "a"))
(format my−log "Guile log: process number: ~a~%" my−rank)
(format my−log " pool size: ~a~%" pool−size)
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format my−log "post.iteration: t = ~a~%" forms.time_e)
    (do ((number−of−points (vector−length sample−points))
         (count 0 (1+ count)))
        ((>= count number−of−points) #t)
      (let* ((point (vector−ref sample−points count))
             (x (vector−ref point 0))
             (y (vector−ref point 1))
             (z (vector−ref point 2))
             (Ex (interpolate "Ex" x y z))
             (Ey (interpolate "Ey" x y z))
             (Ez (interpolate "Ez" x y z))
             (zeta (− (+ (* nx x) (* ny y) (* nz z)) forms.time_e))
             (E−expected (if (outside point)
                             #(0 0 0)
                             (vector (forms.signal.ex_lambda zeta)
                                     (forms.signal.ey_lambda zeta)
                                     (forms.signal.ez_lambda zeta))))
             (Ex−expected (vector−ref E−expected 0))
             (Ey−expected (vector−ref E−expected 1))
             (Ez−expected (vector−ref E−expected 2)))
        (format my−log "post.iteration: at ~a:~%" point)
        (format my−log " Ex: is ~a, expected ~a~%" Ex Ex−expected)
        (format my−log " Ey: is ~a, expected ~a~%" Ey Ey−expected)
        (format my−log " Ez: is ~a, expected ~a~%" Ez Ez−expected)))))
;;
(define forms.post.all.lambda
  (lambda ()
    (format my−log "post.all: finished~%")
    (close−port my−log)))
```

Here we have also provided a simple example of how to use `forms.post.all.lambda`. In this case the thunk logs a farewell message and closes the file (or the `port` in Scheme parlance).

Although it is possible to run a $128 \times 128 \times 128$ resolution job on a single node, a $256 \times 256 \times 256$ resolution job no longer fits. This should not be surprising, because the latter asks for eight times more memory than the former.

The utility presented here can be used to investigate the code's accuracy in function of lattice spacing. And so, we find, for example, that when the job runs on a $128 \times 128 \times 128$ cell grid, with stride set to 4, we get

```
post.iteration: t = 128.0
post.iteration: at #(64 64 15):
    Ex: is −7.8968175476453e−5, expected 0
    Ey: is −1.83430933495469e−6, expected 0
    Ez: is 1.24310100767323e−5, expected 0
...
post.iteration: at #(64 64 60):
    Ex: is 0.986933627467093, expected 0.994521895368273
    Ey: is 9.12358218421598e−6, expected 0.0
    Ez: is −0.00104007380852946, expected 0.0
...
```

which yields relative error in $E_x(64, 64, 60)$ at $t_e = 128$ of ≈ 0.0076.

On the other hand, if the job is run with exactly the same physical parameters, but with half the lattice spacing (and the stride still 4), we find the following:

```
post.iteration: t = 128.0
post.iteration: at #(64 64 15):
    Ex: is 6.18749154726525e−6, expected 0
    Ey: is −2.79278037932539e−7, expected 0
    Ez: is −1.20615055878475e−5, expected 0
...
post.iteration: at #(64 64 60):
    Ex: is 0.992894816374051, expected 0.994521895368273
    Ey: is −9.06036290963048e−6, expected 0.0
    Ez: is 3.27713712045805e−5, expected 0.0
...
```

which yields a relative error in $E_x(64, 64, 60)$ at $t_e = 128$ of ≈ 0.0016. Thus, doubling the resolution, reduces the error 4.7 times. The signal level in $E_z(64, 64, 60)$ at $t_e = 128$ (which we have attributed to the discrepancy between the analytical and the numerical propagation of the injected signal) is reduced nearly 32 times, from ≈ 0.001040 to ≈ 0.00003277.

More elaborate examples that show how to compute fluxes, Fourier transforms, and scattering coefficients with Forms are discussed in Section 2.6, page 60, and in Section 2.8, page 128.

2.2.8 The `watch` group

The last group of input parameters can be used to make Forms log its actions within every one of its functions with varying degrees of detail. For parallel execution each MPI process logs its own actions on its own separate log file. For sequential execution, the log is written on standard output.

Here is an example:

```
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
```

(**define** forms.watch.fill_PML_arrays 1)

Binding a symbol to any integer value greater than 0 activates the log for a given function. The larger the number, the more copious the log. Binding the symbol to 0, makes the function run quietly.

Logging should be used with caution and for debugging purposes only. Setting any log number to more than 1 may generate huge output.

2.3 Postprocessing and Visualization

2.3.1 Chombovis

Once upon a time we used to view and analyze HDF5 organized data dumped by FORMS with CHOMBOVIS. CHOMBOVIS could slice 3-dimensional data sets in all principal directions, it could produce isosurfaces and contour plots on the slices. It could even do volume rendering, if slowly.

Animations

Once a particular view of the data had been obtained, the state of the visualization engine could be saved on a state file, which could then be reloaded while viewing another HDF5 file. This was useful when producing images for animation from the data dumped by FORMS.

It would work as follows. Assuming that FORMS generated HDF5 files all lived in the working directory, the following Python script, when loaded into CHOMBOVIS, would extract an image from every HDF5 file, viewed according to settings saved on a state file called, in this case, `crests.state`, and would write it in the PPM format on a file, whose name preserved the original stem, but replaced the `.hdf5` suffix with `.ppm`:

```
import os
import chombovis
c = chombovis.latest()
for filename in os.listdir('.'):
        if filename [−5:] == '.hdf5':
                c.reader.loadHDF5( filename )
                c.misc.restoreState( 'crests.state' )
                c.misc.hardCopy( outfile_name = filename[:−5] + '.ppm', mag_factor = 1 )
```

If the original HDF5 files were as listed in Section 2.2.6, page 25, then the PPM files would be

```
Poynting_001.ppm
Poynting_002.ppm
...
```

To combine the PPM images produced into a GIF animation, we had to convert them to GIF files first and then merge into a single animated GIF file. The following shell script shows how this used to be done:

```
#!/bin/bash
. /home/ANL/.bashrc
PREFIX=$1
if [ −f script.py −a −f crests.state ]
then
    chombovis user_script=script.py
fi
for i in $PREFIX_∗.ppm
do
  ppmquant 256 $i | ppmtogif −sort −nolzw > ‘basename $i .ppm‘.gif
done
gifsicle −−colors 256 −−delay 15 −−loopcount ${PREFIX}_∗.gif > ${PREFIX}_movie.gif
```

The script invoked CHOMBOVIS on the Python script listed above, then converted each PPM image to a GIF image, and finally assembled GIF file sinto a single moving GIF file. Program `gifsicle` was not quite kosher (it used some copyrighted GIF code) and therefore not widely available.

CHOMBOVIS was a dog to compile and install, in particular under Cygwin, and is no longer supported, so all the above comments are of historic interest mostly, unless the users have access to CHOMBOVIS and still want to use it.

2.3.2 VisIt

More recently the Chombo team has been recommending the use of VisIt. VisIt is more like a real application, whereas Chombovis was like a Python script, hacked by a summer scholar.[2] No changes are needed to Forms or Chombo to use VisIt.

VisIt can be downloaded from `https://wci.llnl.gov/codes/visit/home.html`. Most importantly, it can be downloaded as proper Windows or Mac executables, without the need for a build, which is often difficult for substantial graphics applications. Product manuals and tutorials can be downloaded from the same site.

The currently distributed version of VisIt, 1.11.2, knows about Chombo, and the way Chombo writes data on HDF5 files. It will recognize such files on load.

To illustrate this, let us consider HDF5 files generated by the following input:

```
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.signal.direction '( 1.0 1.0 0.0))
(define forms.signal.normalized 1)
(define forms.signal.ex_lambda
  (lambda (zeta)
    0.0))
(define forms.signal.ey_lambda
  (lambda (zeta)
    0.0))
(define delay -15.0)
(define wavelength 30.0)
(define omega (/ (* 2.0 pi) wavelength))
(define forms.signal.ez_lambda
  (lambda (zeta)
    (let ((delayed-zeta (+ zeta delay)))
      (* (heaviside delayed-zeta) (sin (* omega delayed-zeta))))))
(define t_begin 0)
(define t_end 256)
(define forms.iterate.stride 4)
(define forms.iterate.number_of_steps
  (* (/ (- t_end t_begin) forms.grid.level0.delta) forms.iterate.stride))
(define forms.iterate.image_frequency (* 2 forms.iterate.stride))
(define forms.iterate.t0 t_begin)
(define forms.output.precompute '("Ez"))
(define forms.output.write '("Ez"))
(define forms.output.filename.root "Ez")
(define forms.output.filename.number_of_digits 3)
```

There is nothing here in the `media` group: a plane, harmonic wave propagates diagonally in the $\boldsymbol{e}_x + \boldsymbol{e}_y$ direction through the total field region. The program generates 3D data dumps on `Ez_001.hdf5` through `Ez_128.hdf5`.

To open, say, `Ez_128.hdf5` in VisIt, we just click on its name in the `Selected files` window of VisIt, and then click on `Open`. Selecting `File information` from the main `File` menu opens a window, which confirms that VisIt knows about Chombo. It recognizes the file format as `Chombo_1.0`. It recognizes the size of the region, also the name of the field.

To volume render the field, we click on the `Plot` button. This unfolds a menu, from which we select `Volume->Ez`. An entry is created in the `Active plots` window. While the entry is selected, pressing on `Draw` renders the data in the graphics window. In this case, however, the data does not show well, because the view of the field is obstructed by the scattered field region, within which there is little of interest to us.

We can use VisIt's operators to strip the scattered field region from the data displayed. To do so we click on `Operators` and select `Clip`. We must then click on `OpAtts`, that is on operator attributes, select `Clip` again, and enter `Origin 116 0 0` and `Normal 1 0 0` for `Plane 1`, which we should activate, and nothing for the remaining two planes. This will chop off all data for which $x > 116$. We now need to dismiss the window, and generate the next `Clip` operator, as before, for which the operator attributes are `Origin 11 0 0` and `Normal`

[2] Which is probably what it was.

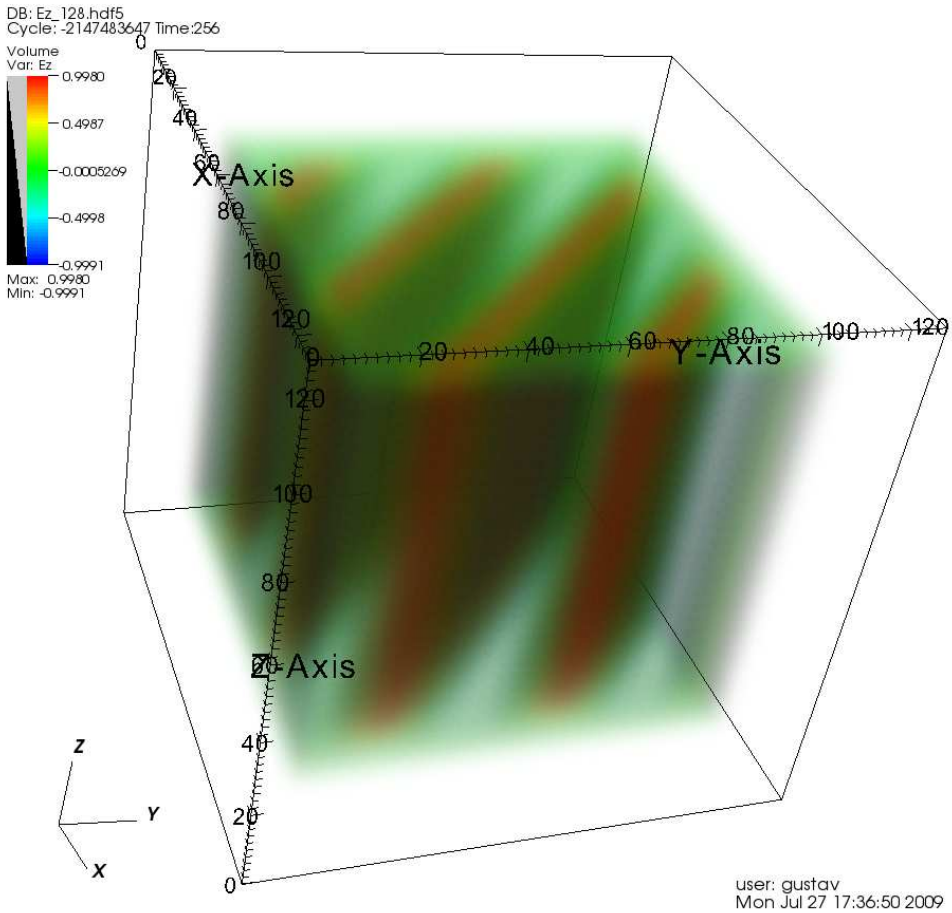


Figure 2: Plane wave propagating diagonally through the total field region: volume rendered by VisIt from `Ez_128.hdf5`.

`-1 0 0` for `Plane 1`, which should be activated, and nothing for the remaining two planes. This chops off all data for $x < 11$.

We need to repeat this operation for the y and z directions, thus accumulating six `Clip` operators acting on `Ez`. The `Active plots` window should diplay

```
128:Volume - Clip(Clip(Clip(Clip(Clip(Clip(Ez))))))
```

Clicking on the `PlotAtts`, i.e., plot attributes, button and selecting `Volume` lets us adjust various parameters, including `Attenuation`, dropping the latter to, say, 10%, so that we can see through better. Eventually, this should produce an image like the one shown in Figure 2.

Figure 3 shows data on a slice $z = 64$ also generated by VisIt from `Ez_128.hdf5`. To do this, we proceed as before, to invoke volume rendering. But the operator applied afterwards is `Slice`. The operator attributes are `Z Axis`, `Origin Intercept 64`, and `Project to 2D`. In addition, we have requested `Smooth Data` in the Volume plot attributes window.

Because so much information can be extracted from HDF5 files generated by Forms with VisIt, it is not necessary for Forms itself to do anything other than compute evolution of the fields and dump their 3D images. All else can be then carried out by separate post-processing applications. Nevertheless, we provide some more specialized virtual measurements and diagnostic tools with the Forms Scheme library, which can be invoked during program execution. These are discussed in Sections 2.6.7 and 2.7.5.

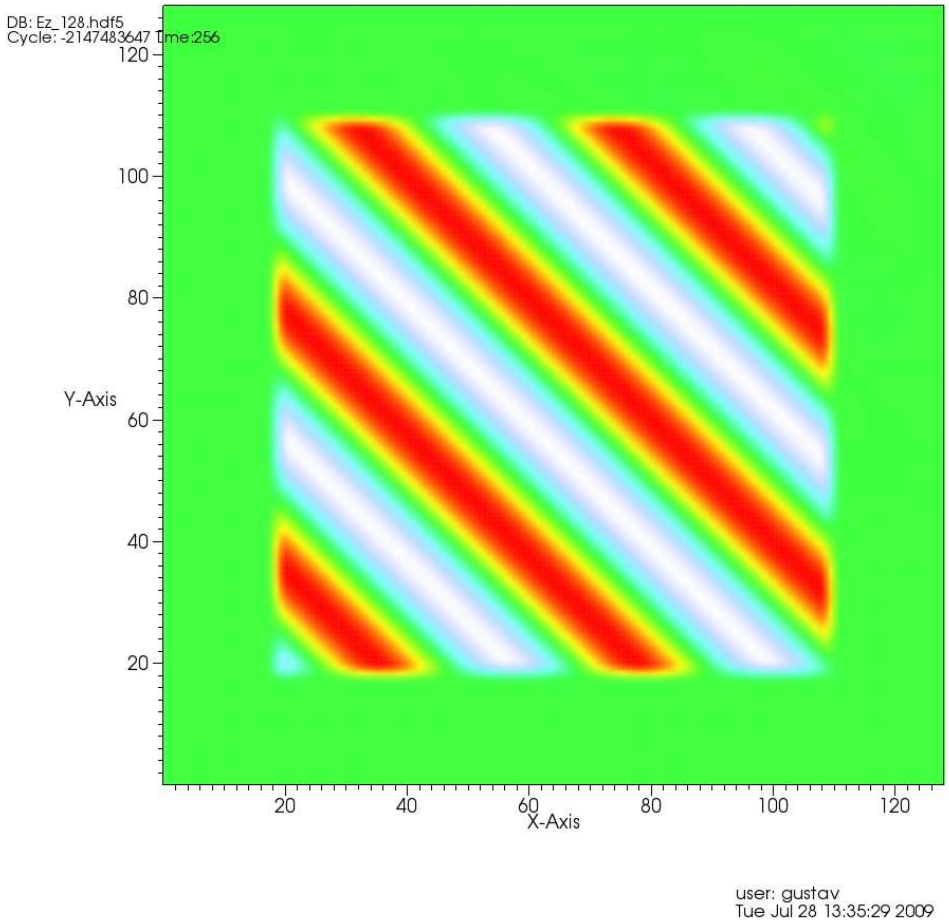


Figure 3: Plane wave propagating diagonally through the total field region: data on a slice $z = 64$ generated by VisIt from `Ez_128.hdf5`.

Animations

VisIt allows for automation by the means of Python or Java programs, as does Chombovis. Luckily, for simple activities, there is a sufficient push-button interface that can be used to produce animations. The first step in animating HDF5 files produced by Forms is to group them into a single data base. This is done in the `File selection` window, invoked by pressing `Select file` in the `File` menu. Within that window one should select all HDF5 files that correspond to the animation—the files are shown in the `Files` column—then press `Group`. This will generate a file with extension `.visit`, e.g., `Ez_.visit`, which should show in the `Files` window, once VisIt has done grouping.

Then, instead of selecting the HDF5 files, we select just the group file, e.g., `Ez_.visit`. Once selected, it should be opened. In the `Selected files` window of VisIt, the individual HDF5 files will show as `cycle 0000`. To prepare for an animation, one should double click on the one that is certain to have some meaningful data within, in our case the last one. Then we procede as above to produce a plot. The one that will result in a simple animation is `Contour`, the attributes of which can be set to `Contour colors: Single`, `Select by: N levels 10`, `Limits: Max 1.1, Min -1.1`, and `Scale: Linear`. One should then press `Make default` and `Apply`. In a contour plot of a 3D field, outermost shells obscure the inner ones. But if we slice the image through $z = 64$, we obtain a planar contour plot that corresponds to field strength within the plane. To do so we apply the `Slice` operator and set its attributes to `Normal: Othogonal Z Axis`, `Origin: Intercept 64`, `Up Axis: Project to 2D`, and `Interactive`. One should press `Make default` again, and then `Apply`. Now we are ready to press `Draw` in the VisIt's main command window, which should result in a contour plot for `Ez_128.hdf5` in `Window 1`.

Because we have loaded the data as a single data base, which contains multiple time-slices, the VCR buttons in the `VisIt` command window are active. Once the defaults are set, pressing on ▶ plays the animation in `Window 1`.

Selecting `Save movie` in the `File` menu, invokes a `Save movie wizard`. Answering its questions step-by-step, eventually constructs and runs a Python script, which generates a movie, e.g., in the MPEG format, for viewing outside VisIt.

This does not appear to work with certain types of plots though, in particular with volume rendering—hence the choice of `Contour` in the above example. A volume rendering window will not print correctly either, unless `Screen capture` is activated in the `Set save options` window, selected from the `File` menu. These settings are read by the `Save window` printer, but not by the `Save movie` wizard. The wizard leaves a Python script behind, and the setting here for each frame is `SaveWindowAtts.screenCapture = 0`. To work around this difficulty one would need to edit and re-run the script.

2.4 Defining Scheme Functions in C

FORMS extends Scheme by defining its own Scheme callable C++ functions that access Chombo fields. These are discussed in Section 8, page 248.

Once defined, they are loaded by function *initialize_scheme*( ), which is discussed in Section 8.14, page 299, whereupon they are available to users.

A C or C++ function registration with Scheme is accomplished by calling *scm_c_define_gsubr*, as in

```
scm_c_define_gsubr("draw−box", 2, 1, 0, (SCM (∗) ()) draw_box);
```

Here we tell Scheme that when users call `draw-box`, with two required parameters, one optional, and zero rest, it should reach for a preloaded C++ function *draw_box*, which is declared here as a function whose return value is of type **SCM**. The rather inscrutable cast operator, (**SCM**(∗)( )), tells the C++ compiler just this, though not in so many words.

The *draw_box* function itself has the following interface,

```
SCM draw_box(SCM lower_corner, SCM upper_corner, SCM color)
```

All its parameters must be of just one type, **SCM**, and its returned type is **SCM**, too. If the third parameter is not specified by the user, Scheme will set it to *unbound,* which the function can test for by calling the `SCM_UNBNDP` predicate.

What's inside *draw_box* is discussed in Section 8.1, page 249. It is too complicated for most users, because the function has to *draw* a box on Chombo fields, either media or tags, and this is a convoluted process.

But users may wish to write their own Scheme functions that, for example, manage ADE computation for absorbing media. This may be a computationally intense procedure, which should be much faster when C-compiled than when interpreted by Scheme, at the same time, not as difficult to write as *draw_box*.

This section explains how to do this and discusses examples.

The following is an example[3] listing of a simple wrapper that invokes the `libm.a` defined Bessel function J_0.

```
#include <math.h>
#include <libguile.h>

SCM j0_wrapper(SCM x)
{
  return scm_from_double (j0(scm_to_double(x)));
}

void init_bessel()
{
  scm_c_define_gsubr("j0", 1, 0, 0, (SCM(∗)())j0_wrapper);
}
```

Before we get to discussing it, first we want to compile it into a module that's dynamically loadable into Scheme and show what to do next. The following simple `Makefile` contains compilation instructions for Linux and Cygwin.

```
linux: libguile−bessel.so

cygwin: libguile−bessel.dll

libguile−bessel.so: bessel.c
        gcc ‘guile−config compile‘ −shared −o libguile−bessel.so −fPIC bessel.c ‘guile−config link‘

libguile−bessel.dll: bessel.c
        gcc ‘guile−config compile‘ −shared −o libguile−bessel.dll bessel.c ‘guile−config link‘

clean:
        rm −f libguile−bessel.dll libguile−bessel.so ∗.o ∗~
```

Typing

[3] This example is taken from the "The Guile Reference Manual" by the Free Software Foundation.

```
$ make cygwin
```

generates a DLL file under Cygwin. Under Linux this has to be a `.so` file. In both cases, the file is produced by a normal `gcc` compilation invoked with the `-shared` switch—under Linux we have to add the `-fPIC` option, too—and linked with Guile libraries, which are listed by the `guile-config link` command. Under MacOS the compilation and link steps are

```
$ gcc −fPIC −DPIC −dynamic −c ‘guile−config compile‘ bessel.c
$ gcc −dynamiclib −o libguile−bessel.so bessel.o ‘guile−config link‘
```

The DLL (or `.so`) file can be loaded into Scheme, assuming the file resides in the working directory, as follows:

```
guile> (load−extension ”libguile−bessel” ”init_bessel”)
guile> (j0 2)
0.223890779141236
guile>
```

The `load-extension` form takes two arguments, both are strings. The first one is the name of the shared library. It can be passed on without extension and a fully qualified path under Cygwin, but Guile 1.8.4 on Linux 2.4.29 (this is the Jazz cluster) wants full name, as in

```
guile> (load−extensions ”./libguile−bessel.so” ”init_bessel”)
```

It understands the dot, but does not know about the tilde expansion.

The second argument is the name of the function in the library that must be executed on loading in order to activate the other functions defined therein.

In this case, the name of the library is `"libguile-bessel"` and the name of the initialization function is `"init_bessel"`. The function, as can be seen from the code listed above, invokes *scm_c_define_gsubr*, which registers string `"j0"` with Scheme, as a symbol that is bound to the C function *j0_wrapper*, which, in turn, just calls *j0* from `libm.a`. The `guile-config link` command generates a string (it may be long) of linking options that are needed to link a Guile program on a system on which it is installed.

2.4.1 Two Dielectrics

Our first example is a C implementation of the double dielectric model described by the Scheme code presented on page 20. The code is only a little more complicated, but it should run much faster.

```
#include <libguile.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

#define EPS_1 4.0
#define EPS_2 2.0

SCM dielectrics()
{

  static int first = TRUE;
  static double one_by_eps_1, one_by_eps_2;
  static SCM medium_var, D_var;

  int medium;
  double E, D;
```

```
  if (first)
    {
      one_by_eps_1 = 1.0 / EPS_1;
      one_by_eps_2 = 1.0 / EPS_2;
      medium_var = scm_c_lookup("forms.medium");
      D_var = scm_c_lookup("forms.D");

      first = FALSE;
    }

  medium = scm_to_int(scm_variable_ref(medium_var));
  D = scm_to_double(scm_variable_ref(D_var));

  switch (medium)
    {
    case 1:
      E = one_by_eps_1 * D;
      break;
    case 2:
      E = one_by_eps_2 * D;
      break;
    }

  return scm_from_double(E);

}

void init_e_lambda()
{
  scm_c_define_gsubr("forms.media.model.e_lambda", 0, 0, 0, (SCM(*)()) dielectrics);
}
```

The code uses some **static** variables, which are evaluated on its first invocation only, and then held in memory. Apart from *one_by_eps_1* and *one_by_eps_2*, which stand for $1/\epsilon_1$ and $1/\epsilon_2$, we have two very important variables here, namely *medium_var* and *D_var*. These two point to variables that are bound to Scheme symbols `"forms.medium"` and `"forms.D"`. The *scm_c_lookup* operation is costly, so by finding the requisite variables on the first invocation only, and then reusing them on consecutive calls, we save much time.

These two variables must be bound to their symbols before the first invocation of the function, which is called *dielectrics* in C and, which is bound to the `"foms.media.model.e_lambda"` symbol in Scheme. The values of the variables may be then changed *only* by using `set!` of Scheme, or *scm_variable_set_x* of C.

The values of the variables are extracted by the calls to *scm_variable_ref*, and then by appropriate conversions from **SCM** to **int** or **double**. And this is how we get *medium* and D. Depending on the value of *medium*, we activate either case 1 or case 2, and return an appropriate value of E, which must be converted to **SCM** before the function exits.

We compile the file, here called `"dielectric.c"`, under Cygwin by

```
$ gcc `guile-config compile` -O2 -shared -o libguile-dielectric.dll
dielectric.c \
`guile-config link`
```

Under Linux, we'd have to add the `-fPIC` switch. Then we test it interactively in Scheme as follows:

```
guile> (load-extension "libguile-dielectric.dll" "init_e_lambda")
guile> (define forms.medium 1)
guile> (define forms.D 0.5)
guile> (forms.media.model.e_lambda)
0.125
guile> (* 0.5 (/ 1.0 4.0))
0.125
guile> (set! forms.medium 2)
```

```
guile> (forms.media.model.e_lambda)
0.25
guile> (* 0.5 (/ 1.0 2.0))
0.25
guile>
```

We can make function *dielectrics* more flexible by making it obtain ϵ_1 and ϵ_2 from the Scheme environment, like this:

```
one_by_eps_1 = 1.0 /
scm_to_double(scm_variable_ref(scm_c_lookup("epsilon_1")));
one_by_eps_2 = 1.0 /
scm_to_double(scm_variable_ref(scm_c_lookup("epsilon_2")));
```

Function *does_scm_symbol_exist*, which is discussed in Section 8.11, page 292, can be used to check if `epsilon_1` and `epsilon_2` have been defined. If not some evasive action can be taken, for example, substituting `EPS_1` and `EPS_2` as defaults.

Having compiled and tested the function, we would `load-extension` in the Scheme input file instead of defining the lambda.

By doing things this way we not only make the code run faster, but also avoid possible memory leak in the Scheme space.[4]

2.4.2 A Drude Metal

A Drude metal model is a specific case of a Debye medium described by the following equation:

$$\epsilon(\omega) = \epsilon_\infty + \frac{\sigma}{\mathrm{i}\omega\epsilon_0} + \frac{\chi}{1+\mathrm{i}\omega\tau}, \tag{10}$$

where τ is the collision time, σ and χ are phenomenological conductivities, and ϵ_∞ is the dielectric constant for very high frequencies, $\omega \to \infty$.

Substituting

$$\chi = -\omega_0^2\tau^2 \quad \text{and} \tag{11}$$

$$\sigma = \omega_0^2\tau\epsilon_0 \tag{12}$$

yields the Drude expression,

$$\begin{aligned}\epsilon(\omega) &= \epsilon_\infty + \frac{\omega_0^2\tau}{\mathrm{i}\omega} - \frac{\omega_0^2\tau^2}{1+\mathrm{i}\omega\tau} = \epsilon_\infty + \frac{\omega_0^2}{-\omega^2+\mathrm{i}\omega/\tau} \\ &= \epsilon_\infty + \frac{\omega_0^2}{\omega(\mathrm{i}/\tau-\omega)}.\end{aligned} \tag{13}$$

This is also called the unmagnetized plasma model. Strictly speaking we should have $\epsilon_\infty = 1$, but there is no harm in giving it more flexibility. Phenomenological models of metals may have several terms, including Lorentz terms, Drude terms, and dielectric terms, as well. The collision time τ may be replaced by its inverse, collision frequency $\nu = 1/\tau$.

The $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion at a given frequency would then be

$$\boldsymbol{D}(\omega) = \epsilon(\omega)\boldsymbol{E}(\omega) = \left(\epsilon_\infty + \frac{\omega_0^2}{\omega(\mathrm{i}/\tau-\omega)}\right)\boldsymbol{E}(\omega) = \epsilon_\infty\boldsymbol{E}(\omega) + \boldsymbol{S}(\omega), \tag{14}$$

where

$$\boldsymbol{S}(\omega) = \frac{\omega_0^2}{\omega(\mathrm{i}/\tau-\omega)}\boldsymbol{E}(\omega), \tag{15}$$

which implies that

$$\mathrm{i}\omega\nu\boldsymbol{S} - \omega^2\boldsymbol{S} = \omega_0^2\boldsymbol{E}. \tag{16}$$

[4] This can be controlled by calling garbage collect (gc). Guile should do this automatically, when needed.

Assuming that $\boldsymbol{S}(\omega) = \boldsymbol{S}_0 e^{\mathrm{i}\omega t}$ we find that

$$\frac{\partial}{\partial t}\boldsymbol{S} = \mathrm{i}\omega\boldsymbol{S} \quad \text{and} \tag{17}$$

$$\frac{\partial^2}{\partial t^2}\boldsymbol{S} = -\omega^2\boldsymbol{S}, \tag{18}$$

which lets us rewrite equation (16) as

$$\nu\frac{\partial}{\partial t}\boldsymbol{S} + \frac{\partial^2}{\partial t^2}\boldsymbol{S} = \omega_0^2\boldsymbol{E}. \tag{19}$$

We can approximate the derivatives with the following centered finite difference terms:

$$\frac{\partial^2}{\partial t^2}\boldsymbol{S} \approx \frac{\boldsymbol{S}^{(n+1)} - 2\boldsymbol{S}^{(n)} + \boldsymbol{S}^{(n-1)}}{\Delta t^2} \quad \text{and} \tag{20}$$

$$\frac{\partial}{\partial t}\boldsymbol{S} \approx \frac{\boldsymbol{S}^{(n+1)} - \boldsymbol{S}^{(n-1)}}{2\Delta t}. \tag{21}$$

Substituting these in equation (19) yields

$$\nu\frac{\boldsymbol{S}^{(n+1)} - \boldsymbol{S}^{(n-1)}}{2\Delta t} + \frac{\boldsymbol{S}^{(n+1)} - 2\boldsymbol{S}^{(n)} + \boldsymbol{S}^{(n-1)}}{\Delta t^2} = \omega_0^2\boldsymbol{E}^{(n)}. \tag{22}$$

Now let us multiply both sides by $2\Delta t^2$ and let us group the terms for given time slices together:

$$\boldsymbol{S}^{(n+1)}\left(2 + \nu\Delta t\right) - 4\boldsymbol{S}^{(n)} + \boldsymbol{S}^{(n-1)}\left(2 - \nu\Delta t\right) = 2\omega_0^2\Delta t^2\boldsymbol{E}^{(n)}. \tag{23}$$

Now we shift this equation by one step into the past, $n \to n-1$, which yields the following solution for $\boldsymbol{S}^{(n)}$ in function of $\boldsymbol{S}^{(n-1)}$, $\boldsymbol{S}^{(n-2)}$ and $\boldsymbol{E}^{(n-1)}$:

$$\boldsymbol{S}^{(n)} = \frac{4}{2+\nu\Delta t}\boldsymbol{S}^{(n-1)} - \frac{2-\nu\Delta t}{2+\nu\Delta t}\boldsymbol{S}^{(n-2)} + \frac{2\omega_0^2\Delta t^2}{2+\nu\Delta t}\boldsymbol{E}^{(n-1)}. \tag{24}$$

Returning to equation (14) and discretizing it in the time domain we find that

$$\boldsymbol{D}^{(n)} = \epsilon_\infty\boldsymbol{E}^{(n)} + \boldsymbol{S}^{(n)}, \tag{25}$$

hence

$$\boldsymbol{E}^{(n)} = \frac{1}{\epsilon_\infty}\left(\boldsymbol{D}^{(n)} - \boldsymbol{S}^{(n)}\right), \tag{26}$$

which together with equation (24) provides us with detailed prescription for how to find $\boldsymbol{E}^{(n)}$ given

1. $\boldsymbol{D}^{(n)}$,
2. $\boldsymbol{E}^{(n-1)}$,
3. $\boldsymbol{S}^{(n-1)}$, and
4. $\boldsymbol{S}^{(n-2)}$.

Once $\boldsymbol{S}^{(n-2)}$ has been used to find $\boldsymbol{S}^{(n)}$ it is no longer needed. Instead $\boldsymbol{S}^{(n-1)}$ becomes the new $\boldsymbol{S}^{(n-2)}$ and $\boldsymbol{S}^{(n)}$ becomes the new $\boldsymbol{S}^{(n-1)}$. Similarly, once $\boldsymbol{E}^{(n-1)}$ has been used to find the new $\boldsymbol{E}^{(n)}$, it is no longer needed, and can be replaced by $\boldsymbol{E}^{(n)}$.

The procedure therefore calls for two additional fields, $\boldsymbol{S}^{(n-1)}$ and $\boldsymbol{S}^{(n-2)}$, which we can call $\boldsymbol{S}$ and $\boldsymbol{S}_{\text{old}}$. In three dimensions, this will translate into six 3-dimensional arrays, S_x, S_y, S_z, $S_{x\,\text{old}}$, $S_{y\,\text{old}}$, and $S_{z\,\text{old}}$ that must be laid out and sized the same way as their corresponding $\boldsymbol{E}$ fields.

The $\boldsymbol{S}$ field's implementation within the program is defined, alongside with $\boldsymbol{E}$ and $\boldsymbol{D}$, in Section 11.6, page 347, module ⟨ Structure level 266 ⟩. We can see there that S is a **LevelData**⟨**CurlBox**⟩. Each single S slot, therefore, defines one full $\boldsymbol{S} = (S_x, S_y, S_z)$. For $\boldsymbol{S}$ and $\boldsymbol{S}_{\text{old}}$ we need to reserve two slots.

This indented subsection, about three pages long, discusses in much detail the mechanisms through which user provided code is inserted into the program. It is highly technical and may be skipped, perhaps, on first reading. On the other hand, users who plan on writing their own modules in C should study it. The section end is similarly flagged. ◀◀

Space reservation for $\boldsymbol{S}$ is discussed in Section 6.2, page 213, module ⟨ Function *fill_levels* 94 ⟩. It is a long module, and the relevant entries to how $\boldsymbol{S}$ space is allocated are:

```
if (media_parameter_parser.query("number_of_auxiliary_fields", auxiliaries))
{
  if (watch_fill_levels)
    pout() << "\tread: number_of_auxiliary_fields = " << auxiliaries << endl;
  if ((auxiliaries < N_AUXILIARIES_MIN) || (auxiliaries > N_AUXILIARIES_MAX))
    {
      pout() << "\n\tERROR(fill_levels): number_of_auxiliary_fields out of range" << endl;
      return_status = ERR_FILL_LEVELS;
      goto exit;
    }
}
```

and then

```
if (auxiliaries)
{
  if (watch_fill_levels) pout() << " S" << flush;
  levels[n]->S.define(level_box_layout, auxiliaries, ghost_margin);
}
```

These tell us that in order to get space for $\boldsymbol{S}$ and $\boldsymbol{S}_{\text{old}}$ we need to declare

```
(define forms.media.number_of_auxiliary_fields 2)
```

on the input file in the first place.

To see what happens next we need to turn to Section 7.3.2, page 229, module ⟨ Function *d_to_e_0* 116 ⟩. There we find that the top level function that does the $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion at level 0, checks if auxiliary fields have been requested and if so, it invokes the *d_to_e_0_n* function to do the job. This function is discussed in Section 7.3.4, page 231, module ⟨ Function *d_to_e_0_n* 118 ⟩. All it does is to prepare things for calling the Fortran subroutine *d_to_e_0_n_* on each box of its current level. This, however, includes defining which are going to be $(\boldsymbol{E}, \boldsymbol{D})$ and which are going to be $(\boldsymbol{E}_{\text{old}}, \boldsymbol{D}_{\text{old}})$ in this current conversion call, since these swap around on each consecutive iteration.

However, the function does not do this to the multi-slot $\boldsymbol{S}$ field, leaving the responsibility for managing movement of data between $\boldsymbol{S}$ and $\boldsymbol{S}_{\text{old}}$ to the programmer.

Also, before calling the Fortran subroutine, function *d_to_e_0_n* extracts the single component **FArrayBox** of $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{S}$ for a given direction, as follows:

```
for (int dir = 0;dir < SpaceDim;dir++)
{
    FArrayBox&E_dir = E[dir];
    FArrayBox&D_dir = D[dir];
    FArrayBox&S_dir = S[dir];
    BaseFab<int> &medium_E_dir = medium_E[dir];
    d_to_e_0_n_(
        &SpaceDim, &dir, &ghost_margin, &number_of_auxiliary_fields,
        &(level_ptr->E_idx[0]), &(level_ptr->D_idx[0]),
        &time_e, &dt,
        &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0],
        &Cy_lo, &Cy_hi, &C0y[0], &C1y[0], &C3y[0], &C4y[0],
        &Cz_lo, &Cz_hi, &C0z[0], &C1z[0], &C3z[0], &C4z[0],
        &(D_dir.loVect()[0]), &(D_dir.loVect()[1]), &(D_dir.loVect()[2]),
        &(D_dir.hiVect()[0]), &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]),
        D_dir.dataPtr(), E_dir.dataPtr(), S_dir.dataPtr(),
        medium_E_dir.dataPtr(),
```

```
            &watch_d_to_e, &return_status
        );

        if (return_status != EXIT_SUCCESS)
        {
            pout() << "\t\t\tERROR(d_to_e_0_n): failure in d_to_e_0_n_: "
            << return_status << endl;
            goto exit;
        }
    }
```

The Fortran subroutine *d_to_e_0_n* is listed in Section 13.3, page 437. The important part of the listing is

```
                do n = 1, n_aux_fields
                   S_vector(n) = S(i,j,k,n)
                end do
                E(i,j,k,new_E) = scm_d_to_e_n(
& dir, n_aux_fields, medium_E(i,j,k),
& E(i,j,k,old_E),
& D(i,j,k,new_D), D(i,j,k,old_D),
& S_vector, time_e, dt)
                do n = 1, n_aux_fields
                   S(i,j,k,n) = S_vector(n)
                end do
```

In summary, for a given (i, j, k) the Scheme wrapper *scm_d_to_e_n* will receive

1. the direction of the field, *dir*;
2. the *medium* number at the (i, j, k) location for the direction *dir*—the number may differ for each direction, because the corresponding field components are mounted at different locations;
3. $E_{\text{dir old}}(i, j, k, t_e) = E_{\text{dir}}(i, j, k, t_e - \Delta t)$, $D_{\text{dir}}(i, j, k, t_e)$, $D_{\text{dir old}}(i, j, k, t_e) = D_{\text{dir}}(i, j, k, t_e - \Delta t)$, as individual numbers;
4. $S_{\text{dir}}(i, j, k, t_e)$, $S_{\text{dir old}}(i, j, k, t_e) = S_{\text{dir}}(i, j, k, t_e - \Delta t)$, $S_{\text{dir older}}(i, j, k, t_e) = S_{\text{dir}}(i, j, k, t_e - 2\Delta t)$, etc., as a vector;
5. t_e and Δt.

Before the call to *scm_d_to_e_n*, the $\boldsymbol{S}$-vector is initialized with numbers taken from the $\boldsymbol{S}$ field, and on return, the possibly updated values stored in the $\boldsymbol{S}$-vector are returned to the $\boldsymbol{S}$ field. The user provided Scheme function is not expected to do anything to $E_{\text{dir old}}(i, j, k, t_e)$, $D_{\text{dir}}(i, j, k, t_e)$, and $D_{\text{dir old}}(i, j, k, t_e)$, and is expected to return $E_{\text{dir}}(i, j, k, t_e)$ on exit.

The reason why $\boldsymbol{S}$ numbers are arranged into a vector, whereas $\boldsymbol{E}$ and $\boldsymbol{D}$ are not, is because we do not know a priori the number of auxiliary fields the user may wish to utilize.

Now let us have a look at *scm_d_to_e_n_*. This is a plain C-wrapper that is discussed in Section 7.3.8, page 236, module ⟨ `Conversion.c_std` 122 ⟩. This module creates a separate file. The function interface looks as follows:

```
Real scm_d_to_e_n_(const int*const direction,
                   const int*const n_aux_fields,
                   const int*const medium,
                   const Real*const E_old,
                   const Real*const D,
                   const Real*const D_old,
                   Real*const S,
                   const Real*const t,
                   const Real*const dt)
```

All parameters are defined as pointers, because Fortran passes variables by reference. All are constant, i.e., cannot be changed, with the exception of S, which must be updated.

SCM-type variables *E_old_var*, *D_var*, *D_old_var*, *medium_var*, *direction_var*, *t_e_var*, *dt_var* and *S_var_ref* that appear in the declaration

```
extern SCM E_old_var, D_var, D_old_var, S_var_ref, medium_var,
  direction_var, t_e_var, dt_var;
```

are program globals, which are defined in Section 8.14, page 299, module ⟨ Initialize Scheme 199 ⟩. In case of the ***S*** field we need the variable reference, because it is an array. This is generated inside *initialize_scheme*( ) by

```
if (number_of_auxiliary_fields)
{
   char command[BUFSIZ];

   sprintf(command, "(define forms.S (make−f64vector %d 0.0))\n",
            number_of_auxiliary_fields);
   scm_c_eval_string(command);
   S_var = forms_parser.variable("S");
   S_var_ref = scm_variable_ref(S_var);
}
```

Returning to *scm_d_to_e_n_*, the wrapper transfers values received from Fortran directly into the Scheme variables, so that no new bindings need to be created or existing ones looked up from the binding table:

```
scm_variable_set_x(D_var, scm_from_double(*D));
scm_variable_set_x(E_old_var, scm_from_double(*E_old));
scm_variable_set_x(D_old_var, scm_from_double(*D_old));
scm_variable_set_x(medium_var, scm_from_int(*medium));
scm_variable_set_x(direction_var, scm_from_int(*direction));
scm_variable_set_x(t_e_var, scm_from_int(*t));
scm_variable_set_x(dt_var, scm_from_int(*dt));
```

Because ***S*** is an array, we have to fuss more with it. Here the transfer is

```
S_ptr = S;

for (count = 0;count < *n_aux_fields;count++)
  scm_uniform_vector_set_x
    (S_var_ref, scm_from_int(count), scm_from_double((double)*S_ptr++));
```

Next, the wrapper calls the user defined Scheme function, which is referred to by *e_lambda*:

```
E_ret = scm_to_double(scm_call_0(e_lambda));
```

The function is expected to return $E_{\text{dir}}(i, j, k, t_e)$, which is passed back to Fortran, which will place it in the appropriate location of the appropriate **FArrayBox**. At the same time whatever is now in the Scheme S array is transferred to the Fortran array:

```
S_ptr = S;

for (count = 0;count < *n_aux_fields;count++)
  *S_ptr++ = (Real)
    scm_to_double(scm_uniform_vector_ref(S_var_ref, scm_from_int(count)));
```

and Fortran, as we have seen above, transfers these values back into an appropriate location of an appropriate ***S*** **FArrayBox**.

In spite of operating on Scheme variable addresses there is still much data movement involved, which is the unavoidable price for using Scheme as an interpreted extension language. Where we gain in flexibility and utility, we lose in performance. The real issue here is if the performance loss is still acceptable to the program user.

Here the technical section ends and we return to discuss the Drude model. ◀◀

So, now that we know how the user provided Scheme or C-function couples with the computational process, how would we C-code the Drude model?

We begin by rewriting equations (24) and (26) in terms of $\boldsymbol{S}$, $\boldsymbol{S}_{\text{old}}$, $\boldsymbol{E}_{\text{old}}$ and $\boldsymbol{D}$, and adding $\boldsymbol{S}$ update steps at the same time:

$$\boldsymbol{S}_{\text{tmp}} \leftarrow \frac{4}{2+\nu\Delta t}\boldsymbol{S} - \frac{2-\nu\Delta t}{2+\nu\Delta t}\boldsymbol{S}_{\text{old}} + \frac{2\omega_0^2\Delta t^2}{2+\nu\Delta t}\boldsymbol{E}_{\text{old}} \tag{27}$$

$$\boldsymbol{E} \leftarrow \frac{1}{\epsilon_\infty}\left(\boldsymbol{D} - \boldsymbol{S}_{\text{tmp}}\right) \tag{28}$$

$$\boldsymbol{S}_{\text{old}} \leftarrow \boldsymbol{S} \tag{29}$$

$$\boldsymbol{S} \leftarrow \boldsymbol{S}_{\text{tmp}} \tag{30}$$

And this leads to the following C-function:

```
#include <libguile.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

SCM drude()
{
  /*

  Provide definitions for two media:

  medium #1 is a dielectric,
  medium #2 is a Drude metal.

  Required user defines are "nu", "omega_0", and "epsilon_infty"
  for the Drude metal, and "epsilon_1" for the dielectric.

  */

  static int first = TRUE;
  static SCM medium_var, D_var, E_old_var, S_var, S_var_ref, dt_var;
  static double C1, C2, C3, nu, omega_0, dt, eps_infty, one_by_eps_infty,
    eps_1, one_by_eps_1;

  int medium;
  double E, E_old, D, S, S_old, S_tmp;

  if (first)
    {
      /* Forms provided variables */

      medium_var = scm_c_lookup("forms.medium");
      D_var = scm_c_lookup("forms.D");
      E_old_var = scm_c_lookup("forms.E_old");
      S_var = scm_c_lookup("forms.S");
      S_var_ref = scm_variable_ref(S_var);
      dt_var = scm_c_lookup("forms.dt");

      dt = scm_to_double(scm_variable_ref(dt_var));

      /* User provided variables for this module */

      nu = scm_to_double(scm_variable_ref(scm_c_lookup("nu")));
```

```
      omega_0 = scm_to_double(scm_variable_ref(scm_c_lookup("omega_0")));
      eps_infty = scm_to_double(scm_variable_ref(scm_c_lookup("epsilon_infty")));
      one_by_eps_infty = 1.0 / eps_infty;
      eps_1 = scm_to_double(scm_variable_ref(scm_c_lookup("epsilon_1")));
      one_by_eps_1 = 1.0 / eps_1;

      /* Evaluate coefficients C1, C2 and C3. */

      C1 = 4.0/(2.0 + nu * dt);
      C2 = (2.0 - nu * dt) / (2.0 + nu * dt);
      C3 = (2 * omega_0 * omega_0 * dt * dt) / (2.0 + nu * dt);

      first = FALSE;
    }

  medium = scm_to_int(scm_variable_ref(medium_var));
  D = scm_to_double(scm_variable_ref(D_var));

  switch (medium)
    {
    case 1:

      E = one_by_eps_1 * D;
      break;

    case 2:

      E_old = scm_to_double(scm_variable_ref(E_old_var));
      S = scm_to_double(scm_f64vector_ref(S_var_ref, scm_from_int(0)));
      S_old = scm_to_double(scm_f64vector_ref(S_var_ref, scm_from_int(1)));

      S_tmp = C1 * S - C2 * S_old + C3 * E_old;
      E = one_by_eps_infty * (D - S_tmp);
      S_old = S;
      S = S_tmp;

      scm_f64vector_set_x(S_var_ref, scm_from_int(0), scm_from_double(S));
      scm_f64vector_set_x(S_var_ref, scm_from_int(1), scm_from_double(S_old));

      break;
    }

  return scm_from_double(E);

}

void init_e_lambda()
{
  scm_c_define_gsubr("forms.media.model.e_lambda", 0, 0, 0, (SCM(*)()) drude);
}
```

We compile this function, as before, with

```
$ gcc `guile-config compile` -O2 -shared -o libguile-drude.dll drude.c \
`guile-config link`
```

and then load it into Scheme with

```
guile> (load-extension "libguile-drude.dll" "init_e_lambda")
```

How to test the function other than through its inclusion in the actual FORMS run, which may or may not produce enlightening results?

The easiest way is to load the extension into an interactive Scheme session and see if it does what it's supposed to. But how do we know that the numbers produced, whatever they may be, are correctly evaluated? To check this we can implement the computation in Scheme entirely and then run a comparison.

The Scheme version of `drude.c` may look as follows:

```
(define test.e_lambda
  ;;
  ;; required global variables:
  ;; (define epsilon_1 1.0)
  ;; (define epsilon_infty 2.0)
  ;; (define nu 3.0)
  ;; (define omega_0 4.0)
  ;; (define forms.dt 0.25)
  ;; (define forms.medium 2)
  ;; (define forms.D 1.0)
  ;; (define forms.E_old 0.0)
  ;; (define forms.S (make-f64vector 2 0.0))
  ;;
  (lambda ()
    (case forms.medium
      ((1) (let ((one_by_epsilon_1 (/ 1.0 epsilon_1)))
             (* one_by_epsilon_1 forms.D)))
      ((2) (let* ((one_by_epsilon_infty (/ 1.0 epsilon_infty))
                  (factor (* nu forms.dt))
                  (denom (+ 2.0 factor))
                  (C1 (/ 4.0 denom))
                  (C2 (/ (- 2.0 factor) denom))
                  (C3 (/ (* 2.0 omega_0 omega_0 forms.dt forms.dt) denom))
                  (S (f64vector-ref forms.S 0))
                  (S_old (f64vector-ref forms.S 1))
                  (S_tmp (+ (* C1 S) (* -1.0 C2 S_old) (* C3 forms.E_old)))
                  (E (* one_by_epsilon_infty (- forms.D S_tmp))))
             (f64vector-set! forms.S 1 S)
             (f64vector-set! forms.S 0 S_tmp)
             E)))))
```

We first load the Scheme version and subject it to a test. We load the file, then we define the constants as suggested in the comment.

```
guile> (load "drude.scm")
guile> (define epsilon_1 1.0)
guile> (define epsilon_infty 2.0)
guile> (define nu 3.0)
guile> (define omega_0 4.0)
guile> (define forms.dt 0.25)
guile> (define forms.medium 2)
guile> (define forms.D 1.0)
guile> (define forms.E_old 0.0)
guile> (define forms.S (make-f64vector 2 0.0))
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(0.0 0.0)
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(0.363636363636364 0.0)
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(0.760330578512397 0.363636363636364)
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(1.02779864763336 0.760330578512397)
```

```
guile> forms.E_old
−0.0138993238166791
```

Alternatively, we may choose to use the library, in which case our script looks as follows:

```
;;
;; Load the required Forms library module
;;
(set! %load−path (cons "/home/gustav/src/Forms/lib" %load−path))
(load−from−path "media")
;;
;; Two S field slots are needed for the Drude model
;;
(define forms.media.number_of_auxiliary_fields 2)
;;
;; Physics parameters, faked
;;
(define epsilon_1 1.0)
(define epsilon_infty 2.0)
(define nu 3.0)
(define omega_0 4.0)
(define forms.dt 0.25)
(define forms.medium 2)
;;
;; Evaluate inverses and squares
;;
(define one_by_epsilon (/ 1.0 epsilon_1))
(define one_by_epsilon_infty (/ 1.0 epsilon_infty))
(define omega_p_square (* omega_0 omega_0))
;;
;; Define fake values for forms.D, forms.E and forms.S
;;
(define forms.D 1.0)
(define forms.E_old 0.0)
(define forms.S (make−f64vector forms.media.number_of_auxiliary_fields 0.0))
;;
;; Define the test function
;;
(define test.e_lambda
  (lambda ()
    (case forms.medium
      ((1) (dielectric one_by_epsilon))
      ((2) (drude one_by_epsilon_infty omega_p_square nu)))))
```

This time the script defines the required values so all we need to do is to run the iteration:

```
guile> (load "drude−lib.scm")
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(0.0 0.0)
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(−0.363636363636364 0.0)
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(−0.760330578512397 −0.363636363636364)
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(−1.02779864763336 −0.760330578512397)
guile> forms.E_old
−0.0138993238166791
```

The values of the $\boldsymbol{S}$ coefficients come out negative compared to the previous run, but the value of `forms.E_old` at the end comes out the same. This is because the library module builds minuses into the definition of $\boldsymbol{S}$.

Now we redefine `forms.E_old` and `forms.S`, load the DLL extension, and run the same test against the C-implemented `forms.media.model.e_lambda`.

```
guile> (define forms.E_old 0.0)
guile> (define forms.S (make-f64vector 2 0.0))
guile> (load-extension "libguile-drude.dll" "init_e_lambda")
guile> (set! forms.E_old (forms.media.model.e_lambda))
guile> forms.S
#f64(0.0 0.0)
guile> (set! forms.E_old (forms.media.model.e_lambda))
guile> forms.S
#f64(0.363636363636364 0.0)
guile> (set! forms.E_old (forms.media.model.e_lambda))
guile> forms.S
#f64(0.760330578512397 0.363636363636364)
guile> (set! forms.E_old (forms.media.model.e_lambda))
guile> forms.S
#f64(1.02779864763336 0.760330578512397)
guile> forms.E_old
-0.0138993238166792
guile>
```

We get the same numbers (up to the rounding error, due to `forms.media.model.e_lambda` doing its computations in C), which implies that the C-code and the two Scheme codes shown above do the same thing. Depending on whether one trusts C or Scheme more, one can use this technique to test one against the other. What is important though is to observe that the $\boldsymbol{S}$ vector in all three cases is updated according to the formula.

For us, however, what matters here is the execution speed, and to test this we will need to invoke both functions a very large number of times. This is best done from within a FORMS run, because FORMS times its various operations.

But before we go there, we can make one more improvement. Signal injection is carried out by three lambdas. These we can replace by C-functions, as well. The following listing shows an example file. The injected signal is the same as described in Section 2.2.3, page 16.

```
#include <libguile.h>
#include <math.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

/* Required Scheme globals: wavelength, alpha, delay. */

SCM ex_lambda(SCM zeta_scm)
{
  static int first = TRUE;
  static double wavelength, alpha, delay, pi, k;
  double zeta, Ex;

  if (first)
    {
      wavelength = scm_to_double(scm_variable_ref(scm_c_lookup("wavelength")));
      alpha = scm_to_double(scm_variable_ref(scm_c_lookup("alpha")));
      delay = scm_to_double(scm_variable_ref(scm_c_lookup("delay")));
      pi = 2.0 * asin(1.0);
```

```
      k = 2.0 * pi / wavelength;
      first = FALSE;
    }

  zeta = scm_to_double(zeta_scm);
  Ex = sin(k * (zeta + delay)) * (1 − tanh (alpha * (zeta + delay))) * 0.5;
  return scm_from_double(Ex);
}

SCM ey_lambda(SCM zeta_scm)
{
  static int first = TRUE;
  static double wavelength, alpha, delay, pi, k;
  double zeta, Ey;

  if (first)
    {
      wavelength = scm_to_double(scm_variable_ref(scm_c_lookup("wavelength")));
      alpha = scm_to_double(scm_variable_ref(scm_c_lookup("alpha")));
      delay = scm_to_double(scm_variable_ref(scm_c_lookup("delay")));
      pi = 2.0 * asin(1.0);
      k = 2.0 * pi / wavelength;
      first = FALSE;
    }

  zeta = scm_to_double(zeta_scm);
  Ey = −1.0 * sin(k * (zeta + delay)) * (1 − tanh (alpha * (zeta + delay))) * 0.5;
  return scm_from_double(Ey);
}

SCM ez_lambda(SCM zeta_scm)
{
  return scm_from_double(0.0);
}

void init_signal_lambdas()
{
  scm_c_define_gsubr("forms.signal.ex_lambda", 1, 0, 0, (SCM(*)()) ex_lambda);
  scm_c_define_gsubr("forms.signal.ey_lambda", 1, 0, 0, (SCM(*)()) ey_lambda);
  scm_c_define_gsubr("forms.signal.ez_lambda", 1, 0, 0, (SCM(*)()) ez_lambda);
}
```

The file is compiled as above, that is

```
$ gcc `guile−config compile` −O2 −shared −o libguile−inject.dll inject.c \
`guile−config link`
```

and then loaded into Scheme with

```
guile> (load−extension "libguile−inject" "init_signal_lambdas")
```

Before using it in the FORMS run, we can test it against the Scheme script, as we have demonstrated above for the media model.

A full input file that loads both DLL libraries (in Cygwin, or `.so` libraries in Linux) looks as follows:

```
;; Forms input file
;;
;; $Id: TestBallC.scm,v 1.3 2008/04/03 18:53:29 gustav Exp $
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Grid group
```

```
;;
;; Level 0 parameters
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Signal group
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 1.0 1.0 1.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define wavelength 30.0)
(define alpha 0.2)
(define delay 20.0)
;;
(load-extension "Examples/libguile-inject" "init_signal_lambdas")
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Media group
;;
(define forms.media.garbage_collect 0)
;;
;; The media distribution lambda
;;
(define forms.media.distribution.lambda
  (lambda ()
    (draw-ball (f64vector 64 64 64) 10 1)
    (draw-ball (f64vector 64 64 64) 6 2)))
;;
;; The media model lambda
;;
(define epsilon_1 2.7)
(define epsilon_infty 2.36461)
(define nu 0.004286)
```

```
(define omega_0 1.106)
;;
(define forms.media.number_of_auxiliary_fields 2)
;;
(load-extension "Examples/libguile-drude" "init_e_lambda")
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
(define forms.iterate.number_of_steps 1028)
(define forms.iterate.stride 4)
(define forms.iterate.image_frequency 16)
(define forms.iterate.t0 0.0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Output group
;;
(define forms.output.precompute '("Ex" "Ey" "Ez" "Bx" "By" "Bz" "En"))
(define forms.output.write '("En"))
(define forms.output.filename.root "En")
(define forms.output.filename.number_of_digits 3)
(define forms.output.report_outliers 1)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Watch group. This is just for debugging.
;;
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
(define forms.watch.fill_PML_arrays 1)
;;
;; end
;;
```

Instead of providing explicit definitions of various lambdas, here we load precompiled code instead. However various physical parameters are still defined and bound to Scheme symbols. These are accessed by the C-code, but once only—on the first invocation. Whatever parameters need to be used, are stored on static variables after they have been evaluated.

When writing C-code it is important to minimize calls to the *scm_c_lookup* function, as these are bound to slow down the code significantly.

For comparison, here is a listing of an input file that implements the same physics, but this time in Scheme

entirely.

```
;; Forms input file
;;
;; $Id: TestBall.scm,v 1.5 2008/04/03 15:08:24 gustav Exp $
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Grid group
;;
;; Level 0 parameters
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Signal group
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 1.0 1.0 1.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define wavelength 30.0)
(define alpha 0.2)
(define delay 20.0)
(define pi (* 2.0 (asin 1.0)))
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (* 0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
    (* -0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ez_lambda
```

```
  (lambda (zeta)
    0.0))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Media group
;;
(define forms.media.garbage_collect 0)
;;
;; The media distribution lambda
;;
(define forms.media.distribution.lambda
  (lambda ()
    (draw-ball (f64vector 64 64 64) 10 1)
    (draw-ball (f64vector 64 64 64) 6 2)))
;;
;; The media model lambda
;;
(define epsilon_1 2.7)
(define epsilon_infty 2.36461)
(define nu 0.004286)
(define omega_0 1.106)
;;
(define forms.media.number_of_auxiliary_fields 2)
;;
(define forms.media.model.e_lambda
  (lambda ()
    (case forms.medium
      ((1) (let ((one_by_epsilon_1 (/ 1.0 epsilon_1)))
             (* one_by_epsilon_1 forms.D)))
      ((2) (let* ((one_by_epsilon_infty (/ 1.0 epsilon_infty))
                  (factor (* nu forms.dt))
                  (denom (+ 2.0 factor))
                  (C1 (/ 4.0 denom))
                  (C2 (/ (- 2.0 factor) denom))
                  (C3 (/ (* 2.0 omega_0 omega_0 forms.dt forms.dt) denom))
                  (S (f64vector-ref forms.S 0))
                  (S_old (f64vector-ref forms.S 1))
                  (S_tmp (+ (* C1 S) (* -1.0 C2 S_old) (* C3 forms.E_old)))
                  (E (* one_by_epsilon_infty (- forms.D S_tmp))))
             (f64vector-set! forms.S 1 S)
             (f64vector-set! forms.S 0 S_tmp)
             E)))))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
(define forms.iterate.number_of_steps 1028)
(define forms.iterate.stride 4)
(define forms.iterate.image_frequency 16)
(define forms.iterate.t0 0.0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Output group
;;
(define forms.output.precompute '("Ex" "Ey" "Ez" "Bx" "By" "Bz" "En"))
(define forms.output.write '("En"))
```

Table 3: Timing in seconds for 128 iterations for signal and media characterized using precompiled C extensions versus Scheme lambdas.

	C coded functions	Scheme lambdas
advancing fields	24	24
signal injection	17	74
material physics	34	36
output generation	2	2
total	79	138

```
(define forms.output.filename.root "En")
(define forms.output.filename.number_of_digits 3)
(define forms.output.report_outliers 1)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Watch group. This is just for debugging.
;;
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
(define forms.watch.fill_PML_arrays 1)
;;
;; end
;;
```

One would expect that the Scheme version would run much more slowly than the version that loads precompiled C code. But it is not necessarily so. We ought to remember that the user provided media code is invoked only within media other than vacuum and PMLs. If we model scattering on a small grain, then the provided code is invoked only on a small subset of the computational domain. On the other hand, the signal injection code is invoked on all cells of the total/scattered field boundary. So, one may expect more savings here.

This is indeed the case for the examples listed above. Table 3 shows Forms timing for 128 iterations for the two input configurations discussed above. We can see here that there is significant speed up of the signal injection procedure, whereas the material physics evaluation runs only a little faster … because there is little material in the system to do it for.

Of course, if the whole total field region is filled with materials of complex properties, the situation may well be different.

2.5 Input for Parallel Execution

This section discusses computational domain partitioning, mentioned in Section 2.1.1, page 12.

Chombo parallelizes its execution by dividing the computational domain into boxes, following user's hints when provided or defaults, and distributing the boxes among MPI processes. This is accomplished by providing the `max_box_size` parameter in the Grid group. For example,

```
(define forms.grid.level0.cells '(256 256 256))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
```

Here we specify that the level 0 computational domain comprises $256 \times 256 \times 256$ cells. But the largest box can have only up to 128 cells in each direction. The domain will therefore be subdivided into eight $128 \times 128 \times 128$ boxes.

When Forms runs sequentially, it manages all boxes and all data exchanges between them on a single CPU. But when it runs on, say, eight CPUs, it distributes the eight boxes amongst them, so that each CPU is responsible for one $128 \times 128 \times 128$ box and data flows between them over the supercomputer's network.

All this is accomplished by the following five Chombo statements that are discussed in more detail in Section 6.1.3, page 206, module ⟨ *build_levels*: build level 0 90 ⟩:

```
level_0_ptr = new level;
level_0_ptr->domain.define(IntVect::Zero,
                           (n_cells[0] - 1) * BASISV(0)
                           + (n_cells[1] - 1) * BASISV(1)
                           + (n_cells[2] - 1) * BASISV(2));
domainSplit(level_0_ptr->domain, level_0_ptr->vector_of_boxes, max_box_size);
LoadBalance(level_0_ptr->vector_of_processes, level_0_ptr->vector_of_boxes);
level_0_ptr->box_layout.define(level_0_ptr->vector_of_boxes,
                               level_0_ptr->vector_of_processes);
```

The C++ parameters *n_cells*[0], *n_cells*[1], *n_cells*[2] and *max_box_size*, are what is read from the Scheme input shown above.

For a single level run data between boxes is exchanged in two places only. We exchange the $\boldsymbol{H}$ field before the $\boldsymbol{D}$ update (see Section 7.1.1, page 220):

```
level_ptr->H.exchange(current_component);
```

and then we exchange the $\boldsymbol{E}$ field before the $\boldsymbol{B}$ update (see Section 7.4.1, page 238):

```
level_ptr->E.exchange(current_component);
```

> Because data exchanges between CPUs over the supercomputer network are *extremely* expensive, the code exchanges only the data that is going to be spatially differentiated, that is, $\boldsymbol{H}$ and $\boldsymbol{E}$. This means that other data, $\boldsymbol{D}$, $\boldsymbol{B}$ and $\boldsymbol{S}$ is not instantiated within the ghost regions at all, which may impact postprocessing.

What would happen if we were to define

```
(define forms.grid.level0.cells '(256 256 256))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 256)
```

and the job was submitted for parallel execution? The MPI compiled Forms binary would dutifully spawn as many processes as has been requested on the PBS script, but the whole computational domain would fit into a single box only and would be processed in entirety on a single CPU. The job would run sequentially, all MPI Forms processes but one remaining idle.

2.6 Tests

The code can be tested in a variety of ways, some of which are described in this section.

2.6.1 Plane Wave Propagation

File: `Examples/PlaneWaveB.scm`

It is worth pointing out that in a sense the code self-tests. This is because the injected signal is defined analytically. Hence, the discrepancy between the injected signal at various points along the scattered/total field boundary, and the numerically propagated signal *within* the total field region, provides us with the measure of the code's accuracy. We have already discussed this in Section 2.2.7, page 27, and illustrated how to use FRAMES built-in function `interpolate` to carry out point measurements of computed fields.

But we may also wish to investigate if the injected signal propagation is identical in every direction and not dependent on polarization, if there is no media distributed within the total field region. Such tests can pick up possible coding errors that mix directions and that do not show for propagation and polarization along a specific direction due to some serendipitous circumstances.

To this extent we have run six jobs, each with an input file almost identical to the listing in Section 2.2.7, but differing from each other as follows:

1. The signal propagates in the $\boldsymbol{e}_z$ direction, and is $\boldsymbol{e}_x$ polarized.
2. The signal propagates in the $\boldsymbol{e}_z$ direction, and is $\boldsymbol{e}_y$ polarized.
3. The signal propagates in the $\boldsymbol{e}_y$ direction, and is $\boldsymbol{e}_x$ polarized.
4. The signal propagates in the $\boldsymbol{e}_y$ direction, and is $\boldsymbol{e}_z$ polarized.
5. The signal propagates in the $\boldsymbol{e}_x$ direction, and is $\boldsymbol{e}_z$ polarized.
6. The signal propagates in the $\boldsymbol{e}_x$ direction, and is $\boldsymbol{e}_y$ polarized.
7. As above, but *stride* is reset to 8 (it is 4 in all other tests but this one and the next).
8. As above, but *stride* is reset to 2.

The computational domain in all cases was defined by low/high corners of $(0, 0, 0)$ and $(128, 128, 128)$, and the signal was injected into a box given by $(10, 10, 10)$ and $(117, 117, 117)$. The signal was propagated on a grid with grid constant of 0.5 (which is $256^3 = 16,777,216$ cells, plus some on the periphery due to face and edge field mounts, which stretches the grid by one depending on the mount and direction) until $t_e = 128$. The solution was then probed along the center line in the direction of the propagation at x, y or z (depending on how the signal was configured to propagate) equal to 15, 20, 30, 40, 50, and 60, and compared against the expected analytical solution for that location and time. The solution was also probed on the side within the scattered field region, 15 units of length away from the edge of the computational domain.

Table 4 shows the results of the tests for just one point, the one that's 60 units in the direction of signal propagation and positioned in the middle of the computational domain, at $(64, 64)$, in the remaining two directions.

Several things are apparent. First, the computed value of the propagated field, for every direction and every polarization, is always exactly the same, down to all 16 significant digits. It only varies, if the length of the time step changes by changing stride or if the grid constant is different, as is the case for test 0. The observed errors are 0.0017 for stride 4, 0.0025 for stride 8 (yes, a shorter time step does not improve the accuracy here; why? this is because we need to make more of them to get to $t_e = 128$), and 0.019 for stride 2. Second, the computed values of the field in the two directions orthogonal to the main one are again exactly the same (for a given stride), down to 10 or 11 significant digits and an exponent. The latter proves that what we observe here is the artifact of the computational method (most likely caused by the discrepancy between the analytically injected signal and its numerical propagation) and not noise.

A comparison betweens tests 0 and 1 shows that significant improvements in accuracy are attained primarily by higher spacial resolution, accompanied by identical reduction in Δt, meaning that the stride remains the same, and not by reducing the time step alone.

Table 4: Signal propagation in the $\boldsymbol{e}_z$ (tests 1 and 2), $\boldsymbol{e}_y$ (tests 3 and 4) and $\boldsymbol{e}_x$ (tests 5–8) directions ($\boldsymbol{k}/k$) for two mutually orthogonal polarizations ($\boldsymbol{E}/E$) each. Tests 1–6 were run with the stride ($\Delta x/\Delta t$) of 4, tests 7 and 8—otherwise identical to test 6—were run with the strides of 8 and 2 respectively. Test 0 is identical to test 1, but run on a 128^3 grid with grid spacing (Δx) of 1.

test	$\frac{\boldsymbol{k}}{k}$	$\frac{\boldsymbol{E}}{E}$	Δx	$\frac{\Delta x}{\Delta t}$	location	field	computed	expected	error
0	$\boldsymbol{e}_z$	$\boldsymbol{e}_x$	1.0	4	$(64, 64, 60)$	E_x	0.986933627467093	0.994521895368273	0.0076
						E_y	$9.12358218421598 \times 10^{-6}$	0.	
						E_z	$-1.04007380852946 \times 10^{-3}$	0.	
1			0.5			E_x	0.992894816374051	0.994521895368273	0.0017
						E_y	$-9.06036290963048 \times 10^{-6}$	0.	
						E_z	$3.27713712045805 \times 10^{-5}$	0.	
2		$\boldsymbol{e}_y$				E_x	$-9.06036290983193 \times 10^{-6}$	0.	
						E_y	0.992894816374051	0.994521895368273	
						E_z	$3.27713712049292 \times 10^{-5}$	0.	
3	$\boldsymbol{e}_y$	$\boldsymbol{e}_x$			$(64, 60, 64)$	E_x	0.992894816374051	0.994521895368273	
						E_y	$3.27713712046310 \times 10^{-5}$	0.	
						E_z	$-9.06036290970850 \times 10^{-6}$	0.	
4		$\boldsymbol{e}_z$				E_x	$-9.06036290949395 \times 10^{-6}$	0.	
						E_y	$3.27713712047039 \times 10^{-5}$	0.	
						E_z	0.992894816374051	0.994521895368273	
5	$\boldsymbol{e}_x$	$\boldsymbol{e}_z$			$(60, 64, 64)$	E_x	$3.27713712049623 \times 10^{-5}$	0.	
						E_y	$-9.06036290990221 \times 10^{-6}$	0.	
						E_z	0.992894816374051	0.994521895368273	
6		$\boldsymbol{e}_y$				E_x	$3.27713712051991 \times 10^{-5}$	0.	
						E_y	0.992894816374051	0.994521895368273	
						E_z	$-9.06036290935525 \times 10^{-6}$	0.	
7				8		E_x	$2.87198079636099 \times 10^{-4}$	0.	
						E_y	0.992052104193310	0.994521895368273	0.0025
						E_z	$-3.23276481852910 \times 10^{-6}$	0.	
8				2		E_x	$-1.67175408157672 \times 10^{-4}$	0.	
						E_y	0.992615077960128	0.994521895368273	0.0019
						E_z	$7.05581781212037 \times 10^{-6}$	0.	

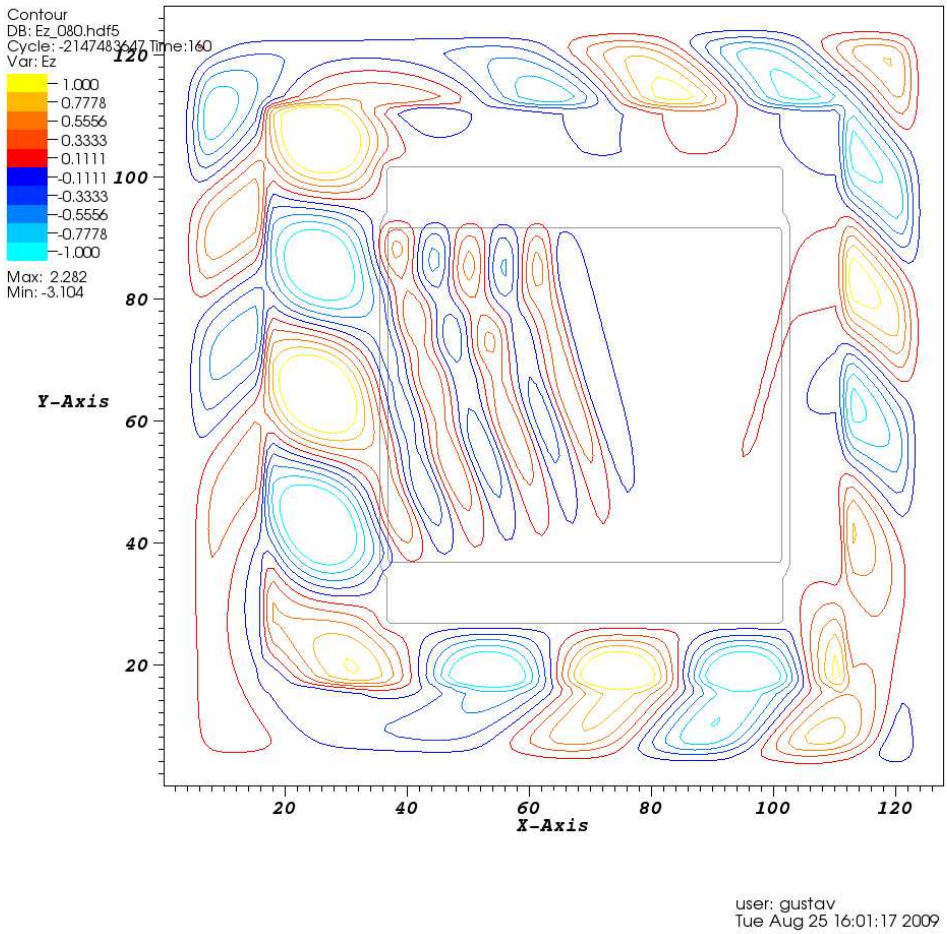


Figure 4: Plane wave polarized in the $\boldsymbol{e}_z$ direction propagates diagonally in the $\boldsymbol{e}_x + \boldsymbol{e}_y$ direction and is incident on the exposed side of a silicon nitride box, wrapped in an ideal conductor. The contour plot of data on the $z = 64$ slice was generated by VisIt from `Ez_080.hdf5`, which was dumped at $t = 160$.

2.6.2 Refraction

File: `Examples/Snell.scm`

In this test we have attempted to simulate the Snell's law of refraction. This is somewhat difficult to do using Forms because of reflections of the incident signal from the dielectric's sides, edges and corners, which interfere with the propagating signal. Nevertheless, we can reproduce the basic physics here and even measure the refraction index.

An $\boldsymbol{e}_z$ polarized harmonic wave of $\lambda = 30$ propagates in the $\boldsymbol{e}_x + \boldsymbol{e}_y$ direction. A box with its low corner at $[35, 35, 35]$ and its high corner at $[102, 92, 92]$ is filled with silicon nitride. The box is additionally wrapped in a 10 units thick layer of an ideal conductor, but so that its two end sides, the one at $x = 35$ and the one at $x = 102$ are exposed. The purpose of the wrapping is to stop the incident signal from entering the dielectric box through its side walls.

As the signal reaches the exposed side of the dielectric box, it enters the medium and refracts. This is shown in Figure 4, which presents a contour graph of E_z on the $z = 64$ slice at $t = 160$. The wave fronts indeed refract in accordance with the Snell's law, as the wave becomes shorter within the medium.

With the figure displayed on the screen, we can resort to measuring the refracted wavelength with a plastic

ruler (!) and compare it against the incident wavelength. The measurements yields, in the case of my display, $\lambda_i = 49\,\mathrm{mm}$ (for the 30 units of length, which is the incident λ) and $\lambda_r = 18\,\mathrm{mm}$ for the refracted wavelength. The ratio is therefore

$$\frac{\lambda_i}{\lambda_r} \approx 2.72. \tag{31}$$

This compares well with the silicon nitride's refraction index, which in our computations is

$$\sqrt{\epsilon_{\mathrm{Si_3N_4}}} = \sqrt{7.5} = 2.74. \tag{32}$$

The direction of the refracted wave is more difficult to measure, because the wavefronts quickly become deformed due to reflections, but here we can probe the solution itself, using `interpolate` and `forms.post.iteration.lambda` to evaluate $\boldsymbol{E} \times \boldsymbol{H}$ at selected points within the dielectric. The refraction index for our geometry should then be, following the Snell's law, n_{y_i}/n_{y_r}, where $\boldsymbol{n}_i$ is the normalized direction of the incident wave's Poynting vector (which is $[1/\sqrt{2}, 1/\sqrt{2}, 0]$), and $\boldsymbol{n}_r$ is the normalized direction of the refracted wave's Poynting vector (which is to be sampled).

Sampling within a radius of 10 around $[55, 55, 64]$ shows a degree of scatter for times greater than about 100 arising because of reflections. But until then, we find that $n_{y_r} \approx 0.27 \pm 0.02$, which yields the refraction index of

$$\frac{1}{\sqrt{2.0}} \frac{1}{0.27} = 2.62, \tag{33}$$

which is fairly close to the expected 2.74, especially given the difficulty of this measurement and low resolution of the model.

The input file for this test is discussed in detail in Section 2.8.1, page 128.

2.6.3 Mie Scattering

File: `Examples/Mie-1.scm`

The Mie scattering test is a yet another dielectric refraction test, but it is more thorough than the test discussed in the previous section. This time we no longer have to do anything about reflections, because they are all taken into account by the Mie's analytical solution. The numerical solution and the analytical solution can be fully compared and differences flagged and analyzed.

The input for the numerical simulation lives on `Examples/Mie-6.scm`, and is discussed in detail in Section 2.8.3, page 132.

In short, an $\boldsymbol{e}_x$ polarized harmonic wave of amplitude 1 and length $\lambda = 60$ propagates in the $\boldsymbol{e}_z$ direction and is incident on a $\mathrm{Si\,O_2}$ ($\epsilon = 3.8$) ball of radius $r = 27$ and centered on $(95, 95, 95)$. This is exactly the center of the computational domain, which stretches from 0 through 190 in each of the three principal directions. The space resolution is $\Delta x = \Delta y = \Delta z = 1$, and the time resolution is $1/4^{\mathrm{th}}$. The whole domain fits in a single box, and the total number of cells, including the ghost ones, is $195^3 = 7,414,875$.

The incident wave is Heaviside windowed, as it enters the total field region at the commencement of the computation. We wait some 900 time units (because $c = 1$ they are the same as the units of length) for the system to stabilize, and for the signal turn on effects to be flushed out. Eventually, at time $t = 907$, we commence field dumps.

The values of E_x are interpolated for points on the $x = x_c = 95$ slice for $y \in [y_c - 80.5, y_c + 80.5]$ and for $z \in [z_c - 80.5, z_c + 80.5]$, where $y_c = z_c = x_c$, and $\Delta y = \Delta z = 1$, and written out on files labeled with time t in human readable text format, for further processing with Gnuplot. We then repeat the same operation, but collect data on a perpendicular slice $y = y_c$, on which we vary $x \in [x_c - 80.5, x_c + 85]$ and z as before, with $\Delta x = \Delta z = 1$.

The results are shown in the left column in Figure 5. The Gnuplot script used to draw the field distribution is

set contour
unset surface
set view map
set cntrparam bspline
set cntrparam points 7

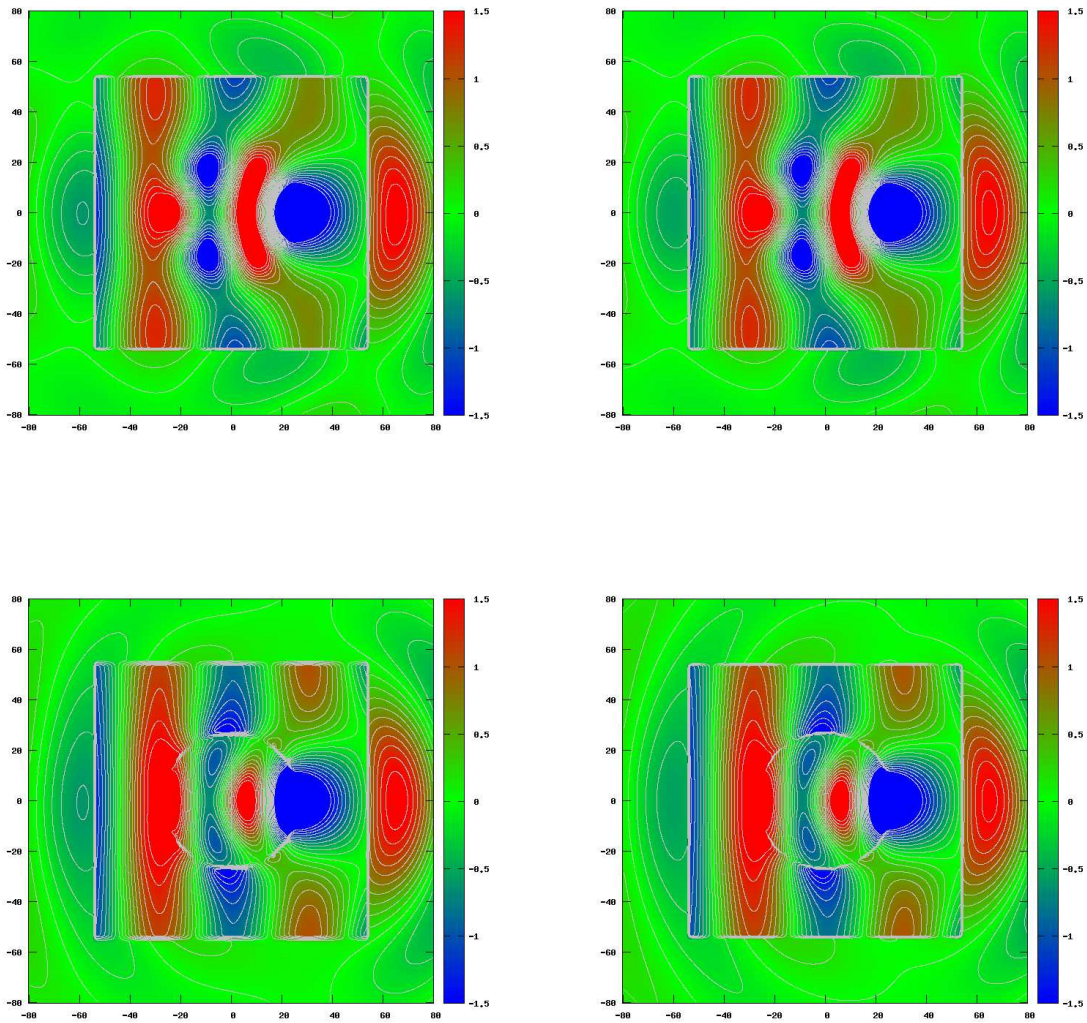


Figure 5: Comparison between numerically and analytically generated data for scattering of an incident harmonic signal on a dielectric ball. The data has been collected on two perpendicular slices that cut through the center of the ball, $x = x_c$ on the top and $y = y_c$ on the bottom. The two panels on the left show a numerically generated solution at $t = 911$, the two panels on the right show an analytically generated solution at $t = 0$.

set cntrparam order 10
set cntrparam levels incremental −1.5,0.15,1.5
set size square
set xrange[−80:80]
set yrange[−80:80]
set cbrange[−1.5:1.5]
set pm3d
set palette defined (−1.5 "blue", 0 "green", 1.5 "red")
unset clabel
unset key
set terminal gif **size** 800,800
set output "Ex_0911.gif"; **splot** "Ex_0911.dat" **with** lines lc rgb 'gray'

Exactly the same script is used to draw analytically generated data for the same field configuration, which is shown in the right column in Figure 5. Thus, the top row shows data on the $x = x_c$ slice for the numerical solution on the left and for the analytical solution on the right, and the bottom row shows data on the $y = y_c = x_c$ slice for the numerical solution on the left and for the analytical solution on the right. In all four panels, the abscissa corresponds to the z direction, with the wave moving from left to right. The colour scale and the contour scale are the same in all four panels, too.

The boundary condition requests that the tangent of the total field outside the ball be continuous across the media interface with the internal field. Therefore we would expect contour lines to be mostly continuous across the interface in the top row, where the incident field is perpendicular to the page, and less so in the bottom one, where the incident field is in the direction of the ordinate. Indeed, this is the case. The interface is sharply delineated in the bottom row. But even here the field is continuous across the interface in the vicinity of $(x, z) = (x_c, z_c \pm r)$.[5]

The numerical simulation reproduces the basic topology of the field with surprising accuracy. The numerical solution within the ball is affected by resolution limits more than the solution outside, because the wave inside the ball is nearly two times shorter. So, whereas the incident wave resolves on 60 grid segments, the wave inside the ball resolves on 30 segments only. Because of internal reflections, and focusing by the internally concave surface of the interface, the field configuration inside the ball can be quite complicated, and may assume relatively high values and steep gradients in spots. To properly compute these, calls for high resolution, if the ball's refraction index is high.

How accurate are we in terms of the actual numbers? We can probe this directly (because we have produced human readable data files). For example, there is an elongated vertical red "sausage" a little to the right of $(0, 0)$ in the top row panels, and then, even further to the right is an intense blue spot. The numerical solution tells us that the read peak value of $E_x = 2.012$ is attained in the vicinity of $(y = \pm 0.5, z = 6.5)$.[6] The analytical solution here reaches a peak of $E_x = 2.029$ at $(y = \pm 0.5, z = 5.5)$. The relative error in the numerical solution is 0.84% regarding the value and 0.63% regarding the location. The peak of -3.729 in the blue blot to the right of the red sausage occurs at $(y = \pm 0.5, z = 25.5)$. For comparison, the equivalent peak in the analytical solution of -3.687 occurs at $(y = \pm 0.5, z = 24.5)$. This yields the relative error of 1.14% in value and 0.63%, as before, in location.

The above suggests that perhaps we should have matched `mie_0000.dat` against `Ex_0910.dat` instead of `Ex_0911.dat`. The match we have used was based on comparing positions of the total field on the total field boundary. In this case, indeed, the field inside the ball appears to lag by one time step behind. Using data from `Ex_0910.dat` instead yields $E_x(0, \pm 0.5, 24.5) = -3.63$ (relative value error of 1.5%), and the minimum is indeed here, and $E_x(0, \pm 0.5, 5.5) = 1.93$ (relative value error of 5%), with the minimum in this location exactly. So, this time we have no location error at all, but our value errors are somewhat larger.

The analytical formulae used in this comparison were obtained from Bohren and Huffman [3].

For the internal field we used

$$\boldsymbol{E}_l = \sum_{n=1}^{\infty} E_n \left(c_n \boldsymbol{M}_{o1n}^{(1)} - \mathrm{i} d_n \boldsymbol{N}_{e1n}^{(1)} \right), \tag{34}$$

where

$$E_n = E_0 \mathrm{i}^n \frac{2n+1}{n(n+1)}, \quad E_0 = 1, \tag{35}$$

[5] In these pannels z runs horizontally and x runs vertically.
[6] In the diagrams z runs horizontally and y runs vertically

$$\boldsymbol{M}^{(1)}_{o1n} = \cos\phi \frac{P^1_n(\cos\theta)}{\sin\theta} j_n(\rho_l)\boldsymbol{e}_\theta - \sin\phi \frac{\mathrm{d}P^1_n(\cos\theta)}{\mathrm{d}\theta} j_n(\rho_l)\boldsymbol{e}_\phi, \quad \rho_l = k_l r, \tag{36}$$

$$\boldsymbol{N}^{(1)}_{e1n} = \frac{n(n+1)j_n(\rho_l)}{\rho_l} \cos\phi\, P^1_n(\cos\theta)\boldsymbol{e}_r \tag{37}$$

$$+\cos\phi \frac{\mathrm{d}P^1_n(\cos\theta)}{\mathrm{d}\theta} \frac{1}{\rho_l} \frac{\mathrm{d}}{\mathrm{d}\rho_l} \left(\rho_l j_n(\rho_l)\right) \boldsymbol{e}_\theta \tag{38}$$

$$-\sin\phi \frac{P^1_n(\cos\theta)}{\sin\theta} \frac{1}{\rho_l} \frac{\mathrm{d}}{\mathrm{d}\rho_l} \left(\rho_l j_n(\rho_l)\right) \boldsymbol{e}_\phi, \quad \rho_l = k_l r, \tag{39}$$

$$c_n = \frac{j_n(\rho_a)\left[\rho_a h^{(1)}_n(\rho_a)\right]' - h^{(1)}_n(\rho_a)\left[\rho_a j_n(\rho_a)\right]'}{j_n(m\rho_a)\left[\rho_a h^{(1)}_n(\rho_a)\right]' - h^{(1)}_n(\rho_a)\left[m\rho_a j_n(m\rho_a)\right]'}, \quad \rho_a = ka, m = \frac{k_l}{k}, \tag{40}$$

$$d_n = \frac{m j_n(\rho_a)\left[\rho_a h^{(1)}_n(\rho_a)\right]' - m h^{(1)}_n(\rho_a)\left[\rho_a j_n(\rho_a)\right]'}{m^2 j_n(m\rho_a)\left[\rho_a h^{(1)}_n(\rho_a)\right]' - h^{(1)}_n(\rho_a)\left[m\rho_a j_n(m\rho_a)\right]'}, \quad \rho_a = ka, m = \frac{k_l}{k}, \tag{41}$$

where k_l is the wave number inside the ball, a is the radius of the ball, r is the radial coordinate, $m = k_l/k$ is the refraction index of the ball relative to the medium it is immersed in, $P^1_n(\cos\theta)$ is an associated Legendre function, $j_n(\rho)$ is a regular spherical Bessel function, and $h^{(1)}_n(\rho) = j_n(\rho) + \mathrm{i}y_n(\rho)$, where $y_n(\rho)$ is an irregular spherical Bessel function, is a spherical Hankel function. An important feature of these expressions is that the Bessels in the vector harmonics $\boldsymbol{M}^{(1)}_{o1n}$ and $\boldsymbol{N}^{(1)}_{e1n}$ are all functions of $\rho_l = k_l r$, not of r or of kr. On the other hand, the Bessels in the coefficients c_n and d_n are functions of ka or of $mka = k_l a$, that is, they mix the internal and the external scales. This is to be expected, because they derive from the boundary conditions imposed on the field. The derivatives of the Bessel functions are to be taken with respect to ρ, that is

$$\left[\rho_a h^{(1)}_n(\rho_a)\right]' = \frac{\mathrm{d}}{\mathrm{d}\rho}\left(\rho h^{(1)}_n(\rho)\right) \quad \text{at} \quad \rho = \rho_a, \tag{42}$$

where ρ may be kr or $kmr = k_l r$, depending on whether the function refers to the internal or the external field.

All sums were cut off at $n = 10$.

For the scattered field we used the following:

$$\boldsymbol{E}_s = \sum_{n=1}^{\infty} E_n \left(\mathrm{i}a_n \boldsymbol{N}^{(3)}_{e1n} - b_n \boldsymbol{M}^{(3)}_{o1n}\right), \tag{43}$$

where E_n is as above, and

$$\boldsymbol{M}^{(3)}_{o1n} = \cos\phi \frac{P^1_n(\cos\theta)}{\sin\theta} h^{(1)}_n(\rho)\boldsymbol{e}_\theta - \sin\phi \frac{\mathrm{d}P^1_n(\cos\theta)}{\mathrm{d}\theta} h^{(1)}_n(\rho)\boldsymbol{e}_\phi, \quad \rho = kr, \tag{44}$$

$$\boldsymbol{N}^{(3)}_{e1n} = \frac{n(n+1)h^{(1)}_n(\rho)}{\rho} \cos\phi\, P^1_n(\cos\theta)\boldsymbol{e}_r \tag{45}$$

$$+\cos\phi \frac{\mathrm{d}P^1_n(\cos\theta)}{\mathrm{d}\theta} \frac{1}{\rho} \frac{\mathrm{d}}{\mathrm{d}\rho} \left(\rho h^{(1)}_n(\rho)\right) \boldsymbol{e}_\theta \tag{46}$$

$$-\sin\phi \frac{P^1_n(\cos\theta)}{\sin\theta} \frac{1}{\rho} \frac{\mathrm{d}}{\mathrm{d}\rho} \left(\rho h^{(1)}_n(\rho)\right) \boldsymbol{e}_\phi, \quad \rho = kr, \tag{47}$$

$$a_n = \frac{m^2 j_n(m\rho_a)\left[\rho_a j_n(\rho_a)\right]' - j_n(\rho_a)\left[m\rho_a j_n(m\rho_a)\right]'}{m^2 j_n(m\rho_a)\left[\rho_a h^{(1)}_n(\rho_a)\right]' - h^{(1)}_n(\rho_a)\left[m\rho_a j_n(m\rho_a)\right]'}, \quad \rho_a = ka, m = \frac{k_l}{k}, \tag{48}$$

$$b_n = \frac{j_n(m\rho_a)\left[\rho_a j_n(\rho_a)\right]' - j_n(\rho_a)\left[m\rho_a j_n(m\rho_a)\right]'}{j_n(m\rho_a)\left[\rho_a h^{(1)}_n(\rho_a)\right]' - h^{(1)}_n(\rho_a)\left[m\rho_a j_n(m\rho_a)\right]'}, \quad \rho_a = ka, m = \frac{k_l}{k}. \tag{49}$$

The implementation of these expressions is discussed in more detail in Section 2.7.7, page 125. Special functions were taken from the GNU Scientific Library, GSL. This is discussed in Section 2.7.6, page 122. A simple script that computes analytical data for the animations can be found on `Examples/Mie-ball-analytical.scm`.

The expressions for the internal and external fields return complex *phasors,* which are functions of position, but not time. We should really write them as $\hat{\boldsymbol{E}}_{l,s}$. To make them into physical fields we need to split them as follows:

$$\boldsymbol{E}(\boldsymbol{r},t) = \Re\left(\hat{\boldsymbol{E}}(\boldsymbol{r})\right)\cos\omega t + \Im\left(\hat{\boldsymbol{E}}(\boldsymbol{r})\right)\sin\omega t, \tag{50}$$

where ω is the frequency of the incident wave. The total field phasor is computed by

$$\hat{\boldsymbol{E}}_{\text{total}} = \hat{\boldsymbol{E}}_s - e^{\mathrm{i}kz}\boldsymbol{e}_x, \tag{51}$$

which we can then split into its real and imaginary parts as above.

2.6.4 Mie Scattering on Silver

File: `Examples/Drude-Input.scm`

Numerical scattering on metals in the Drude regime can be difficult, because signal propagation within the metal in this regime may exceed the speed of light in vacuum. The Courant criterion therefore forces us to shorten the time step in proportion to the value of the refractive index within the metal. Additionally, the incident signal becomes exponentially absorbed within the medium, which puts further demands on spatial resolution of the grid. On the other hand, for higher energies in the Lorentz regime metals behave more like dielectrics, still with larger extinction, but with the refractive index greater than 1.

Because the dielectric constant of metals is complex, auxiliary differential equations must be used to model the transmission of electromagnetic waves through the medium, which adds to the computational cost of the simulation, though not so much in CPU terms as it turns out. The cost is mainly in additional memory requirements.

For this reason, when simulating scattering on a metal ball, we had to lower the resolution from 191 boxes in the x (also y and z) direction to 180 boxes only. This is related to the particular 32-bit testing environment we have employed for the software development (Cygwin), and should not be the case on true 64-bit systems—even for sequential runs. It can be overcome within the Cygwin environment too, when running the job in parallel under MPI. In any case, we have re-tested scattering on a dielectric ball in this 180 box grid configuration, before attempting the new run with the silver parameters.

The input file for this run is discussed in detail in Section 2.8.4. So, here we concentrate on the physics instead.

As in the dielectric example above, we inject an x-polarized Heaviside-windowed harmonic signal propagating in the z-direction into the computational domain. The signal's full wavelength is resolved over $60\,\mathrm{d}x$, with $\mathrm{d}x = 1$. We interpret the signal as an incident wave of 0.89 eV energy. Following the data from Johnson and Christy [12], and assuming the scattering ball to be made of silver, the refractive index for this energy should be $n = 0.13 \pm 0.06 + \mathrm{i}\,(10.10 \pm 0.17)$.

The Forms library file `"constants.scm"` contains auxiliary utilities for exploring the accuracy of models provided within any specific energy range, for example, for the Baida and Labeke model [2] we find:

```
guile> (load-from-path "constants")
guile> (define energies (list 0.64 0.77 0.89 1.02))
guile> (tabulate_n_BL_Ag_450_800 energies)
0.64 0.23 14.089
0.77 0.16 11.699
0.89 0.12 10.110
1.02 0.09 8.808
guile>
```

The printed table is formatted to look like the table in Johnson and Christy, allowing for quick comparison. The fit is good delivering numbers within the tabulated error range. When comparing the FDTD results against the Mie results we have used $n(0.89\,\mathrm{eV})$ as returned by the fit, not as printed in Johnson and Christy.

Because the real part of the refractive index for 0.89 eV is 0.12, the signal propagation within the ball is more than 8 times faster than in the vacuum. For this reason we have shortened the time step to $\mathrm{d}x/16$. This extends the simulation time, even though we have used a C-coded signal injection and a C-coded ADE solver to speed things up.

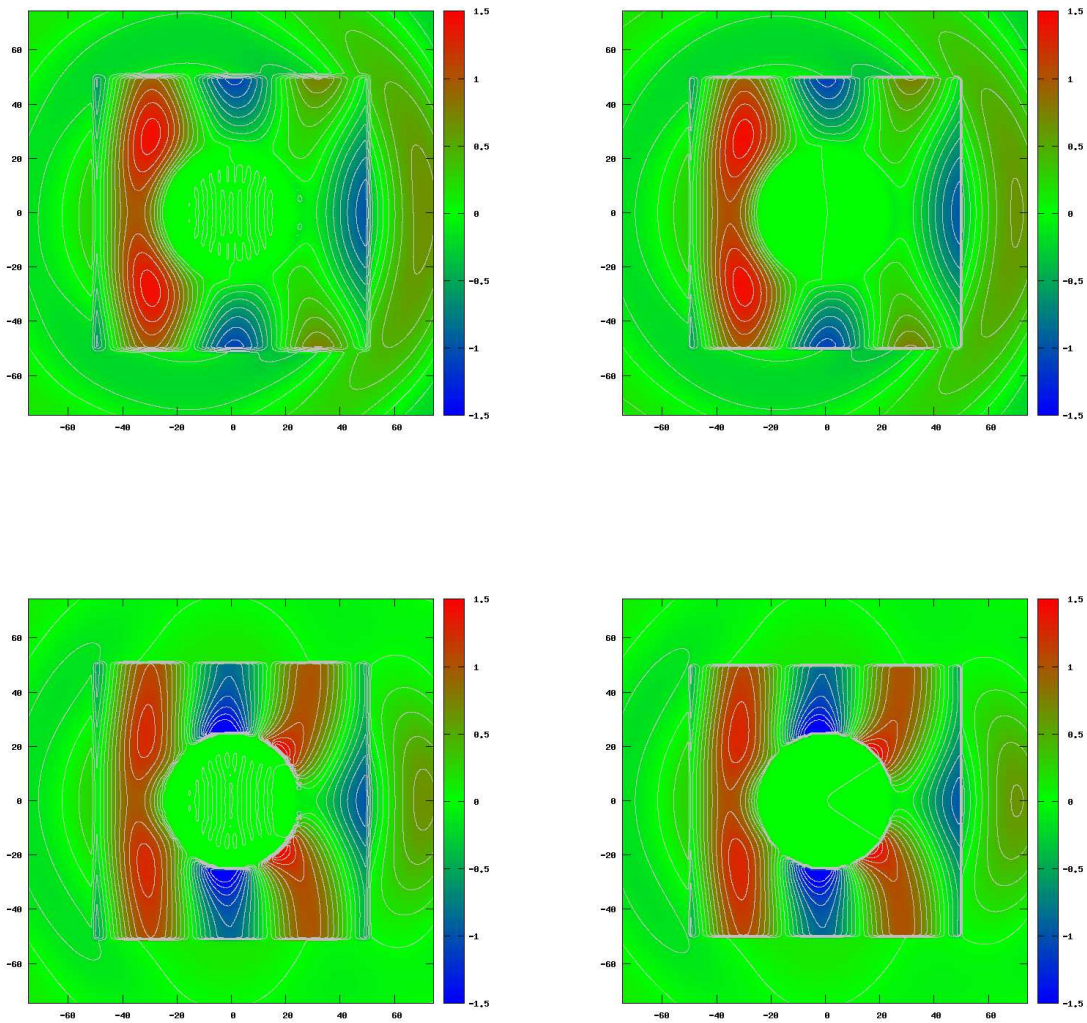


Figure 6: Comparison between numerically and analytically generated data for scattering of an incident harmonic signal on a silver ball at 0.89 eV. The data has been collected on two perpendicular slices that cut through the center of the ball, $x = x_c$ on the top and $y = y_c$ on the bottom. The two panels on the left show a numerically generated solution at $t = 907$, the two panels on the right show an analytically generated solution at $t = 59$.

Table 5: Values of E_x in the vicinity of $(0,0,0)$ in the zy slice, for $y = 0.5$, for FDTD and Mie solutions. The FDTD and ADE resolution does not allow for the accurate reproduction of the very small numbers returned by the analytical solution here.

z	E_x	
	FDTD	Mie
-10.5	3.38×10^{-5}	5.92×10^{-8}
-9.5	-3.53×10^{-5}	2.29×10^{-8}
-8.5	2.75×10^{-5}	8.99×10^{-9}
-7.5	3.83×10^{-5}	3.55×10^{-9}
-6.5	-5.02×10^{-5}	1.42×10^{-9}
-5.5	8.08×10^{-6}	5.77×10^{-10}
-4.5	4.08×10^{-5}	2.38×10^{-10}
-3.5	-4.13×10^{-5}	9.99×10^{-11}
-2.5	2.16×10^{-6}	4.27×10^{-11}
-1.5	6.88×10^{-5}	1.85×10^{-11}
-0.5	-3.31×10^{-5}	7.72×10^{-12}
0.5	-2.34×10^{-5}	1.91×10^{-12}
1.5	6.95×10^{-5}	-3.60×10^{-12}
2.5	4.64×10^{-6}	-1.40×10^{-11}
3.5	-3.99×10^{-5}	-4.05×10^{-11}
4.5	5.30×10^{-5}	-1.11×10^{-10}
5.5	1.40×10^{-5}	-3.03×10^{-10}
6.5	-4.39×10^{-5}	-8.23×10^{-10}
7.5	5.91×10^{-5}	-2.23×10^{-9}
8.5	4.90×10^{-5}	-6.08×10^{-9}
9.5	-4.19×10^{-5}	-1.65×10^{-8}
10.5	5.03×10^{-5}	-4.52×10^{-8}

Figure 6 shows selected snapshots from FDTD and Mie runs that correspond to roughly the same phase. The numerically obtained solution (the two panels on the left) overlaps with the analytically generated one (the two panels on the right) within the expected error bounds—not only for the snapshots presented here, but for all of them, thus demonstrating again that the code solves the problems correctly, this time for a more difficult case that requires the use of a non-trivial ADE. We see that the field configuration at the ball's boundary is the same in both cases, as is the field's general topology within the total and scattered field regions. We also see that the field is expelled from within the inside of the ball.

The field iso-line that can be seen within the ball corresponds to $E_x = 0$. This line is nice and smooth in the analytic case and decays into noise in the FDTD case. However, it's attachments to the external fields at the ball's boundary are the same in the analytical and FDTD cases, as are the angles it subtends with the boundary. The fluctuations in the FDTD solution are less than 10^{-4} and are caused by numerical inaccuracies in the ADE and FDTD procedures. Table 5 illustrates this in detail. Here we have taken the E_x values from the zy-slices (FDTD and Mie) shown in Figure 6 for $y = 0.5$ and z varying between -10.5 and 10.5. The FDTD oscillation about zero is within the accuracy limit of the method itself for the resolution used. It changes from time-slice to time-slice and does not accumulate, which would lead to instability.

2.6.5 Fluxes

File: `Examples/WaveFlux.scm`

In the absence of media and any "hard" or "soft" sources, fluxes of $\boldsymbol{E}$ and $\boldsymbol{B}$ through any enclosed surface should be zero at all iteration steps.

A simple way to test this condition is illustrated by the following listing:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Postprocessing group
;;
(define my-rank (proc-id))
(define root-rank (unique-proc))
(define pool-size (num-proc))
(define my-log (open-file (format #f "scm_out.~a" my-rank) "a"))
(format my-log "Guile log: process number: ~a~%" my-rank)
(format my-log " pool size: ~a~%" pool-size)
;;
(define n_phi 360)
(define n_theta 180)
(define center #f64(60 60 60))
(define radius 25.0)
;;
(format my-log "Computing flux of E through Sphere(~a,~a)~%" center radius)
(force-output my-log)
;;
(define phi_min 0)
(define phi_max (* 2 pi))
(define theta_min 0)
(define theta_max pi)
(define d_phi (/ (- phi_max phi_min) n_phi))
(define d_theta (/ (- theta_max theta_min) n_theta))
;;
(define fluxes (make-vector pool-size 0.0))
;;
(define compute-local-flux
  (lambda (center radius)
    (let* ((x0 (f64vector-ref center 0))
           (y0 (f64vector-ref center 1))
           (z0 (f64vector-ref center 2))
           (flux 0.0))
      (do ((phi (+ phi_min (/ d_phi 2)) (+ phi d_phi)))
          ((> phi phi_max) flux)
        (do ((theta (+ theta_min (/ d_theta 2)) (+ theta d_theta)))
            ((> theta theta_max))
          (let* ((sin_theta (sin theta))
                 (nx (* (cos phi) sin_theta))
                 (ny (* (sin phi) sin_theta))
                 (nz (* (cos theta)))
                 (x (+ x0 (* radius nx)))
                 (y (+ y0 (* radius ny)))
                 (z (+ z0 (* radius nz)))
                 (d2s (* radius radius sin_theta d_phi d_theta))
                 (Ex (interpolate "Ex" x y z))
                 (Ey (interpolate "Ey" x y z))
                 (Ez (interpolate "Ez" x y z))
                 (En (+ (* Ex nx) (* Ey ny) (* Ez nz))))
            (set! flux (+ flux (* d2s En)))))))))
;;
(define compute-flux
  (lambda (center radius)
    (let* ((local-flux (compute-local-flux center radius))
           (status (gather fluxes local-flux root-rank))
           (flux (if (= my-rank root-rank)
                     (do ((count 0 (1+ count))
                          (sum 0.0))
```

```
                    ((>= count pool-size) sum)
                  (set! sum (+ sum (vector-ref fluxes count))))
                0)))
      (broadcast flux root-rank))))
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format my-log "post.iteration: t = ~a, flux = ~a~%"
            forms.time_e (compute-flux center radius))
    (force-output my-log)))
;;
(define forms.post.all.lambda
  (lambda ()
    (format my-log "post.all: finished~%")
    (close-port my-log)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

These are definitions for a parallel job, and we will have to perform some interprocess communication on the Scheme level, so the first group of defines looks at the pool size and structure. Each process opens its own log file, too.

The `forms.post.iteration.lambda` computes the flux of $\boldsymbol{E}$ through a sphere of radius 25 centered on $(60, 60, 60)$. The detailed mathematics for this computation is discussed in Section 3.5.2, page 169, and a listing similar to the one above is provided there, too. The main difference here is that we calculate the flux of $\boldsymbol{E}$, not the flux of $\boldsymbol{n}$, and we do so over the whole sphere, not a hemisphere. We make a modest attempt to optimize the computation by evaluating certain quantities, such as $\phi_{\min}$, $\phi_{\max}$, $\Delta\phi$, $\theta_{\min}$, $\theta_{\max}$, and $\Delta\theta$, up front, and storing the results on globals.

Function `forms.post.iteration.lambda` is merely a wrapper, which writes the result of the computation on the log file, invoking the computation at the same time. The computation itself is deferred to function `compute-flux`, which is itself a wrapper, taking care of collecting (by calling `gather`) locally evaluated fluxes from the participating MPI processes, and adding them up. The result is then broadcast back to participating processes, but this is not strictly necessary. The `root-rank` process can be made the only one writing the answer on the log file.

> The FORMS function `interpolate` is designed to return zero, if a particular data item is not found within any of the grid boxes a given process looks after. This is specially done to facilitate and speed up the evaluation of fluxes—a script that calls `interpolate` does not have to check function returns for misses.

In the end, it is the `compute-local-flux` function that performs the actual flux computation on the portion of data that a given process looks after, and it is much the same as the function discussed in Section 3.5.2.

The output produced on the log files is all identical in this case, with the exception of the preamble, because all processes write the same total flux number. The following listing shows a portion of the log:

```
Guile log: process number: 2
                pool size: 8
Computing flux of E through Sphere(#f64(60.0 60.0 60.0),25.0)
post.iteration: t = 0.125, flux = 0.0
post.iteration: t = 0.25, flux = 0.0
...
post.iteration: t = 126.875, flux = -0.00136514061473036
post.iteration: t = 127.0, flux = -0.00151591753150626
post.iteration: t = 127.125, flux = -0.001369643245134
post.iteration: t = 127.25, flux = -9.16757729662265e-4
post.iteration: t = 127.375, flux = -2.15663970372759e-4
post.iteration: t = 127.5, flux = 5.90951132799944e-4
post.iteration: t = 127.625, flux = 0.00129535512972012
post.iteration: t = 127.75, flux = 0.00168322203256821
post.iteration: t = 127.875, flux = 0.00161249794277296
```

```
post.iteration: t = 128.0, flux = 0.00107854314621392
post.all: finished
```

The flux begins with a true zero, because this is the initial condition for the whole computational domain. As the waves pass through, it remains zero within the standard deviation of about 0.00042 (as evaluated on the sample shown above).

The computation presented here is inefficient on account of two singularities of the coordinate system we have used to cover the sphere with. The polar regions are vastly "overcomputed". But this does not take away from the accuracy of the result, because their contribution to the flux carries less weight—this is encoded in d^2S, which shrinks towards the poles.

If the computation is carried out once only, there is no need to make it more efficient, as long as the job terminates in reasonable time. If the computation is to be repeated frequently in the production context, it can be further optimized by covering the sphere with coordinate patches, and hand-coding the flux computation itself in C.

File: `Examples/WaveFlux-halfLib.scm`

The library functions provided automate computation of fluxes for various surfaces, allowing the user to define arbitrary surfaces as well. How to do this is discussed in Section 2.7.5, page 106. Module `3D-vectors.scm` can be invoked by itself to facilitate operations on three-dimensional vectors. The following listing shows the effect:

```
(define compute−flux
  (lambda (center radius)
    (do ((flux 0.0)
         (phi (+ phi_min (/ d_phi 2)) (+ phi d_phi)))
        ((> phi phi_max) flux)
      (do ((theta (+ theta_min (/ d_theta 2)) (+ theta d_theta)))
          ((> theta theta_max))
        (let* ((sin_theta (sin theta))
               (n (3D−vector (* (cos phi) sin_theta)
                             (* (sin phi) sin_theta)
                             (cos theta)))
               (r (+ center (* radius n)))
               (d2s (* radius radius sin_theta d_phi d_theta))
               (E (interpolate "E" r)))
          (set! flux (+ flux (* d2s (* E n)))))))))
```

The computation unfolds pretty much as above, but this time we handle vectors as whole entities, making use of overloads for arithmetic operations that carry over to vectors. And so, (* E n) encodes the scalar product of $\boldsymbol{E}$ and $\boldsymbol{n}$, whereas (r (+ center (* radius n))) encodes $\boldsymbol{r} = \boldsymbol{r}_c + r\boldsymbol{n}$. Function `interpolate` is overloaded in `measurements.scm` to interpolate the whole vector field in one go, as in (E (**interpolate** "E"r)).

File: `Examples/WaveFlux-Lib.scm`

The following listing, on the other hand, shows how to use predefined surface construction methods and a generic function `compute-flux` to accomplish the same as above by even simpler means:

```
(define my−log (open−file "scm_out_7.dat" "w"))
(define r_0 (3D−vector 60 60 60))
(define r 25)
(define sphere (make−sphere r_0 r))
(define north−cap (make−cap (+ r_0 (* r e_z)) e_z r (* 5 degree)))
(define south−cap (make−cap (− r_0 (* r e_z)) (− e_z) r (* 5 degree)))
(define forms.post.all.lambda
  (lambda ()
    (format my−log "post.all: t = ~a, flux = ~a~%"
            forms.time_e
            (− (+ (compute−flux "E" north−cap)
                  (compute−flux "E" south−cap))
               (compute−flux "E" sphere)))
    (format my−log "post.all: finished~%")
    (close−port my−log)))
```

Table 6: Comparison between numerical and analytical results for the time-average of power flux through the measurement sphere. The two right-most columns yield the absolute and the relative error.

ϕ	θ	$\frac{1}{T}\int_0^T \boldsymbol{n}_p \cdot (\boldsymbol{E}_p \times \boldsymbol{H}_p)\ \mathrm{d}t$			
		FDTD	Mie	$\lvert\Delta\rvert$	%
153.5	21.5	0.545	0.554	0.009	1.6
115.5	30.5	0.259	0.261	0.002	0.8
0.5	6.5	0.881	0.868	0.013	1.5
0.5	90.5	0.027	0.026	0.001	3.8
0.5	170.5	0.095	0.093	0.002	2.2
−65.5	7.5	0.860	0.846	0.014	1.6
−65.5	35.5	0.157	0.153	0.004	2.6
−65.5	172.5	0.096	0.091	0.005	5.5
−97.5	106.5	0.003	0.003	0.000	5.9
−150.5	21.5	0.545	0.550	0.005	0.9

Here we define three surfaces: a sphere, which by default cuts the polar regions out, and two caps to fill the polar regions with coordinate systems that are not singular. Then, when invoking `format` at the end of the program execution, we call `compute-flux` from `measurements.scm` (not the one listed above, the library version is more general) to evaluate fluxes of $\boldsymbol{E}$ through the sphere and the caps, and add them up. The sphere flux is added with a minus, because the sphere's normal vector (when it is constructed with `make-sphere`) points inwards.

2.6.6 Mie Flux

File: `Examples/fdtd-power-map.scm`

This test is very similar to the one described in Section 2.6.3. In fact, it can be run simultaneously with the latter. But here, instead of dumping images of the the fields' selected components, to show their progression with time, we accumulate

$$\frac{1}{T}\int_0^T \boldsymbol{n}_p \cdot (\boldsymbol{E}(\boldsymbol{r}_p, t) \times \boldsymbol{H}(\boldsymbol{r}_p, t))\ \mathrm{d}t \tag{52}$$

for each plaquette of a sphere of radius 75 co-centric with the scattering glass sphere of radius 27 and refraction index

$$n_r = \sqrt{\epsilon_{\mathrm{SiO_2}}} = \sqrt{3.8} \approx 1.95. \tag{53}$$

The measurement sphere is thoroughly embedded within the scattered field region, not touching the total field region's corners or the UPML region's sides. The total field region is made slighly smaller for this occasion extending between its low corner at $(53, 53, 53)$ and its high corner at $(137, 137, 137)$. The UPML layer is also made a little narrower, extending between 0 and 14 at its low side and 176 and 190 at its high side. The center of the system, that is, the center point of both spheres, is located at $(95, 95, 95)$.

Inspection of Figure 7, in which we compare numerically obtained result, shown in the top panel, with the analytically obtained one, shown in the bottom panel, demonstrates that Forms can correctly reproduce the basic physics expected in this case.

We can compare the data even more precisely, because we have numerical outputs for both runs. Data presented in Table 6 shows agreement within the relative error of between 1 and 6%. The relative error is larger for flux values near zero, which points to the presence of numerical noise in the system. For larger flux values, towards the top of the figures, which corresponds to forward scattering, the error drops to about 1%. This is consistent with the expected accuracy of our numerical procedures for the Δx and Δt used in the computation.

This measurement is highly sensitive to numerical imperfections, because it accumulates data. If there is any systematic error in the system, it is going to accumulate too, and show in the final map. Looking at it closely

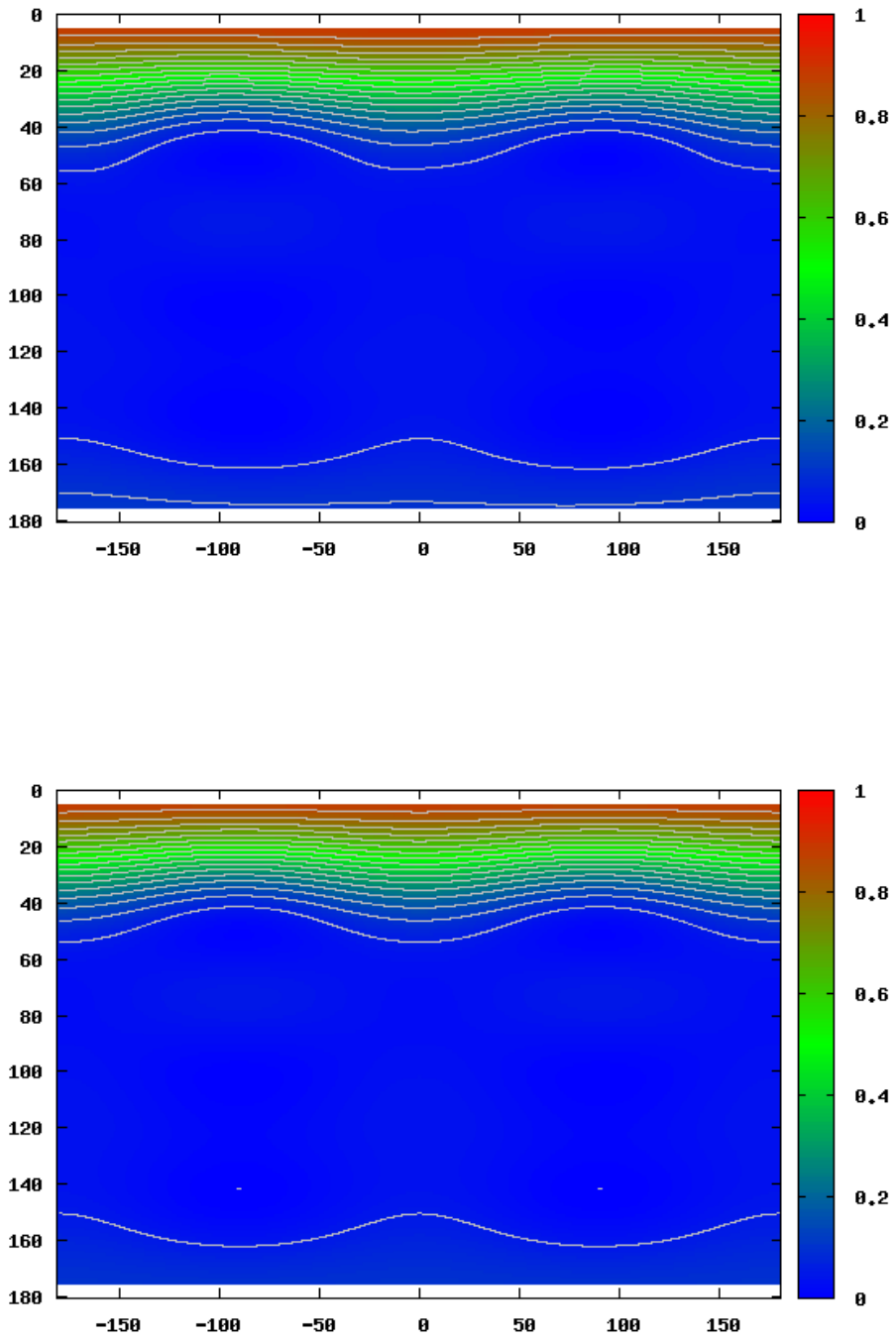


Figure 7: Normal component of the average power flux through a sphere of radius 75 co-centric with a scattering glass sphere of radius 27, for x-polarized incident wave of length 60, moving up in the z-direction. Top figure, FDTD computation with Forms. Bottom figure, analytical computation obtained with Mie solution. The abscissa corresponds to the azimuth angle, ϕ, from $-180°$ to $+180°$. The ordinate corresponds to the inclination angle, θ, from $0°$ at the top, to $+180°$ at the bottom.

we can discern such imperfections. There is a slight loss of symmetry and the backscattering flow is a little wider in the numerical solution than it ought to be.

This sensitivity of the procedure can be used to optimize various numerical parameters. In our case, we used it to optimize the value of $\sigma_{\max}$, that characterizes the UPML boundary, cf. Section 2.2.2, page 16, and Section 3.4, page 160.

We found that setting $\sigma_{\max} \in [0.5, 0.6]$ produced best results and minimized numerical noise effects.

The code for this test is discussed in Section 2.8.6, page 139.

File: `Examples/fdtd-fourier-map.scm`

The same result can be obtained by exercising the Fourier utilities in the `measurements` library. File `fdtd-fourier-map.scm` shows how to do this. The computation here is nearly identical to the one discussed above, but instead of accumulating $\boldsymbol{n}_p \cdot \boldsymbol{E}_p(t) \times \boldsymbol{H}_p(t)\,\mathrm{d}t$ on each plaquette, so as to reconstruct equation (52), here we accumulate Fourier integrals of $\boldsymbol{E}_p(t)$ and $\boldsymbol{H}_p(t)$, namely

$$\hat{\boldsymbol{E}}_p(\omega) = \int_0^{2T} \boldsymbol{E}(\boldsymbol{r}_p, t) e^{\mathrm{i}\omega t}\,\mathrm{d}t \quad \text{and} \quad \hat{\boldsymbol{H}}_p(\omega) = \int_0^{2T} \boldsymbol{H}(\boldsymbol{r}_p, t) e^{\mathrm{i}\omega t}\,\mathrm{d}t. \tag{54}$$

This can be done for more than one frequency, but in this case, we do just the one, $\omega = 2\pi/T$, where T is the wave's period (in our units, in which $c = 1$, $T = \lambda$).

On having completed the accumulation over the duration of two periods, as before, we open a file and write

$$\phi_p \quad \theta_p \quad \frac{1}{2T^2} \boldsymbol{n}_p \cdot \Re\left(\hat{\boldsymbol{E}}_p(\omega) \times \hat{\boldsymbol{H}}_p(\omega)\right) \tag{55}$$

on it, for each plaquette p of the measurement sphere. As the equation (379), page 168, in Section 3.5.1, page 167, demonstrates, this should be the same as (52). Also, cf. equation (373), page 168, which explains where the $1/T^2$ factor comes from. It's due to the difference between the Fourier accumulation, and the phasor.

The result of this test is shown in Figure 8. This figure is indistinguishable from the top panel of Figure 7.

Numerically, the agreement between fluxes generated using this method, and fluxes generated by the direct method described previously is between five and eight significant digits, depending on location. It is generally better towards the south pole of the sphere, $\theta \approx 180°$, and worse towards the north pole, $\theta \approx 0°$.

The input file for this test is discussed in Section 2.8.6, page 139.

2.6.7 Virtual Measurements

File: `Examples/PowerFlux.scm`

In this example we are going to measure total energy received by two tubular "photomultipliers" inserted into the total field region. The "photomultipliers" apertures are circular and the tubes can be positioned under various angles. A simple mathematical model of the device and the measurement is implemented here by the computation of a flux of $\boldsymbol{E} \times \boldsymbol{H}$ across a disk that represents the device's aperture and that can be positioned within the computational domain in any location and oriented in any direction.

The disk parameters are its center, its radius and the direction of its normal. With the help of simple vector algebra, this is enough to give us two vectors perpendicular to the normal, which span the plane of its disk. In this test they are defined as follows:

```
;;
;; Define the detectors
;;
(define center_1 #(40.0 64.0 40.0))
(define center_2 #(88.0 64.0 40.0))
(define normal_1 #(-1 0 1))
(define normal_2 #(1 0 1))
(define radius 10.0)
(define dx (/ (* 2.0 radius) 40.0))
(define dy dx)
;;
;; Normalize the detectors and define their
```

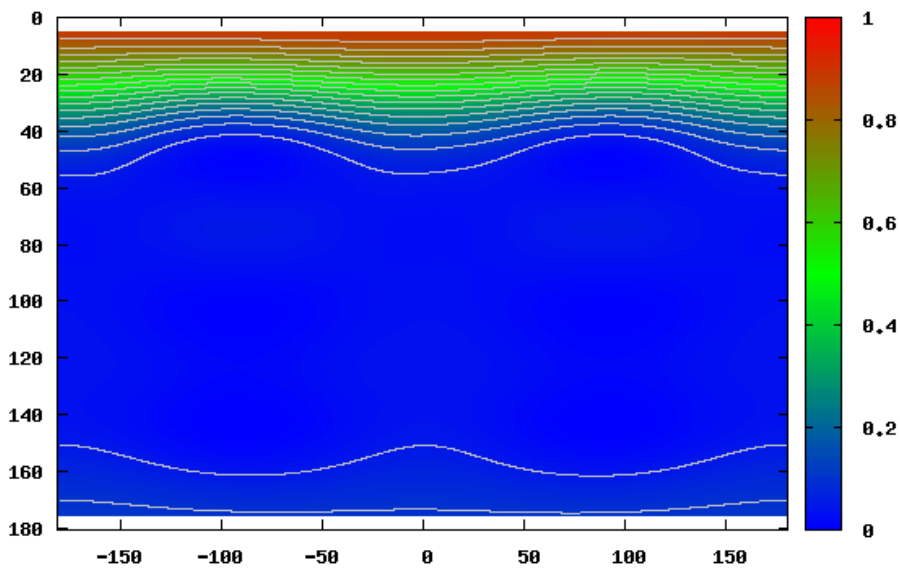


Figure 8: Normal component of the average power flux through a sphere of radius 75 co-centric with a scattering glass sphere of radius 27, for x-polarized incident wave of length 60, moving up in the z-direction—computed by using numerically accumulated Fourier transforms of $\boldsymbol{E}_p(t)$ and $\boldsymbol{H}_p(t)$. The abscissa corresponds to the azimuth angle, ϕ, from $-180°$ to $+180°$. The ordinate corresponds to the inclination angle, θ, from $0°$ at the top, to $+180°$ at the bottom.

```
;; target surfaces
;;
(define n1 (vector−normalize normal_1))
(define n2 (vector−normalize normal_2))
(define ex1 (first−perpendicular n1))
(define ey1 (cross−product ex1 n1))
(define ex2 (first−perpendicular n2))
(define ey2 (cross−product ex2 n2))
;;
```

The detectors are positioned symmetrically with respect to the center of the total field region and the direction of the signal propagation, which is $\boldsymbol{e}_z$. The signal is absorbed by the detectors obliquely, since their disks make a 45° angle with the direction of signal propagation.

The function that computes the flux using data available to a given MPI process is

```
;;
;; Define the function that computes local flux
;;
(define compute−local−flux
  (lambda (center n ex ey radius dx dy)
    (let* ((x−lo (− (− radius (/ dx 2))))
           (y−lo x−lo)
           (x−hi (− radius (/ dx 2)))
           (y−hi x−hi)
           (radius−square (* radius radius))
           (d2s (* dx dy))
           (nx (vector−ref n 0))
           (ny (vector−ref n 1))
           (nz (vector−ref n 2))
           (flux 0))
      (do ((x x−lo (+ x dx)))
          ((> x x−hi) flux)
        (do ((y y−lo (+ y dy)))
            ((> y y−hi))
          (if (<= (+ (* x x) (* y y)) radius−square)
              (let* ((position (vector−add
                                  center (vector−add (vector−mult x ex)
                                                     (vector−mult y ey))))
                     (x0 (vector−ref position 0))
                     (y0 (vector−ref position 1))
                     (z0 (vector−ref position 2))
                     (E (vector
                         (interpolate "Ex" x0 y0 z0)
                         (interpolate "Ey" x0 y0 z0)
                         (interpolate "Ez" x0 y0 z0)))
                     (H (vector
                         (interpolate "Hx" x0 y0 z0)
                         (interpolate "Hy" x0 y0 z0)
                         (interpolate "Hz" x0 y0 z0)))
                     (P (cross−product E H))
                     (Pn (dot−product P n)))
                (set! flux (+ flux (* Pn d2s)))))))))
;;
```

It takes seven arguments, the first four being vectors and the last three floats:

center This is the center $\boldsymbol{r}_0$ of the disk.

n This is vector $\boldsymbol{n}$ that's normal to the disk; it must be normalized.

ex The first vector $\boldsymbol{e}_{x'}$ in the disk plane; must be normalized.

ey The second vector $\boldsymbol{e}_{y'}$ in the disk plane; must be normalized, too—the two vectors must be perpendicular to each other and to $\boldsymbol{n}$, and they define a local orthonormal coordinate system (x', y') on the disk surface.

radius The radius R of the disk.

dx Step length in the local $\boldsymbol{e}_{x'}$ direction, used to integrate the flux.

dy Step length in the local $\boldsymbol{e}_{y'}$ direction.

Given $\boldsymbol{n}$, the user can find $\boldsymbol{e}_{x'}$ by calling function `first-perpendicular`. Then $\boldsymbol{e}_{y'}$ can be obtained by taking $\boldsymbol{e}_{x'} \times \boldsymbol{n}$.

The flux function uses vector algebra to find points on the disk surface,

$$\boldsymbol{r} = \boldsymbol{r}_0 + x'\boldsymbol{e}_{x'} + y'\boldsymbol{e}_{y'}, \tag{56}$$

for x' and y' running between $\pm(R - \Delta x'/2)$ and $\pm(R - \Delta y'/2)$ respectively. The computation is triggered by (x', y') being within the radius of the disk. For such locations we find $\boldsymbol{E}$ and $\boldsymbol{H}$—function `interpolate` not only interpolates $\boldsymbol{H}$ in space, but in time, too, so both $\boldsymbol{E}$ and $\boldsymbol{H}$ are returned for the same time slice and space point. Next, the function evalutes $\boldsymbol{P} = \boldsymbol{E} \times \boldsymbol{H}$, and then takes its dot product with the normal, $\boldsymbol{P} \cdot \boldsymbol{n}$. Finally, it adds the result to the already accumulated flux with the $\mathrm{d}x\,\mathrm{d}y$ weight, which encodes the surface integral, as given by equation (392) on page 170.

Locally computed fluxes are collected from the MPI processes by `compute-flux`, listed below:

```
;;
;; Function that collects local fluxes and adds them up is
;; quite the same as before.
;;
(define compute−flux
  (lambda (center n ex ey radius dx dy)
    (let∗ ((local−flux (compute−local−flux center n ex ey radius dx dy))
           (fluxes (make−vector pool−size 0))
           (status (gather fluxes local−flux root−rank))
           (flux (if (= my−rank root−rank)
                     (do ((count 0 (1+ count))
                          (sum 0.0))
                         ((>= count pool−size) sum)
                       (set! sum (+ sum (vector−ref fluxes count))))
                     0)))
      (broadcast flux root−rank))))
;;
```

The `forms.post.iteration.lambda` thunk does a little more work here than was the case in the previous example, because here we carry out integration over time too, to obtain the total energy absorbed by the detector. The function looks as follows:

```
;;
(define energy_1 0)
(define energy_2 0)
;;
;; Now the post−iteration function
;;
(define forms.post.iteration.lambda
  (lambda ()
    (let∗ ((current−flux−1 (compute−flux center_1 n1 ex1 ey1 radius dx dy))
           (current−flux−2 (compute−flux center_2 n2 ex2 ey2 radius dx dy)))
      (set! energy_1 (+ energy_1 (∗ current−flux−1 forms.dt)))
      (set! energy_2 (+ energy_2 (∗ current−flux−2 forms.dt)))
      (format my−log "t = ~a~%" forms.time_e)
      (format my−log " current flux on detector 1 = ~a, total absorbed = ~a~%"
              current−flux−1 energy_1)
      (format my−log " current flux on detector 2 = ~a, total absorbed = ~a~%"
```

```
                current−flux−2 energy_2))))
;;
```

The output generated by a test run on a single CPU for $\Delta x = 1$ on a 128^3 grid looks as follows:

```
Guile log:
   process number: 0
   pool size: 1
Computing flux of E x H through two photomultiplier tubes
   tube 1: radius = 10.0, center = #(40.0 64.0 40.0), normal = #(−0.707106781186547 0.0 0.707106781186547)
                ex = #(0.0 1.0 0), ey = #(0.707106781186547 0.0 0.707106781186547)
   tube 2: radius = 10.0, center = #(88.0 64.0 40.0), normal = #(0.707106781186547 0.0 0.707106781186547)
                ex = #(0.0 −1.0 0), ey = #(−0.707106781186547 0.0 0.707106781186547)
t = 0.25
   current flux on detector 1 = 0.0, total absorbed = 0.0
   current flux on detector 2 = 0.0, total absorbed = 0.0
t = 0.5
   current flux on detector 1 = 0.0, total absorbed = 0.0
   current flux on detector 2 = 0.0, total absorbed = 0.0
...
t = 127.5
   current flux on detector 1 = 108.585969016262, total absorbed = 11319.7390109556
   current flux on detector 2 = 108.593816242442, total absorbed = 11320.7787778787
t = 127.75
   current flux on detector 1 = 105.898295724077, total absorbed = 11346.2135848866
   current flux on detector 2 = 105.908488324884, total absorbed = 11347.2558999599
t = 128.0
   current flux on detector 1 = 103.265772252444, total absorbed = 11372.0300279497
   current flux on detector 2 = 103.277403693938, total absorbed = 11373.0752508834
post.all: finished
```

The total energy absorbed by both detectors is zero at first, then grows gradually reaching 11372.0300279497 for the first detector and 11373.0752508834 for the second one at $t = 128$. There is no reason why the detectors should absorb different amounts of energy. In our case the discrepancy in the accumulated energy is 1.045 which yields the relative error of less than 1×10^{-4}. This demonstrates the code's ability to maintain left-right symmetry for the injected signal in the absence of media.

When the same job is run on eight processors and on the 256^3 grid with $\Delta x = 0.5$, the corresponding numbers are 11437.7088342658 for the first detector and 11438.009158956 for the second one. This time, the discrepancy is 0.3003, which yields the relative error of 2.7×10^{-5}. We also observe some discrepancy between the $\Delta x = 1$ and $\Delta x = 0.5$ results, which is about 65.7 for the first detector and 64.9 for the second one. This represents about 0.6% of the accumulated value in both cases, easily explained by the improved accuracy of the higher resolution run.

That the latter is the case can be tested additionally by running the job in parallel, but on the same 128^3 grid and with the same parameters as the sequential job, the output of which is listed above. The detector configuration in this case is identical to that of the next example, that talks about Fourier analysis. In that example we find that detector 1 is split between processes 5 and 7, and detector 2 is split between processes 1 and 3. Here, the total energy absorbed by detector 1 in the parallel run is 11372.0300279571, versus 11372.0300279497 for the sequential job. We obtain an agreement down to 12 significant digits between both results, even though they come from different machines and different operating systems. The total energy absorbed by detector 2 in the parallel run is 11373.0752508975, versus 11373.0752508834 for the sequential job. Here the agreement is again down to 12 significant digits. This demonstrates that there is no data overlap in the `interpolate` routine that would double contributions from such overlapping locations. The way `interpolate` handles this, by appropriately shrinking the boxes from which data is extracted, prevents this.

File: `Examples/PowerFlux-Lib.scm`

The code on `Examples/PowerFlux.scm` implements the required vector algebra and then the computation of the flux explicitly, which is laborious to code and prone to mistakes. If we were to make use of the library provided, the post-processing portion of the code would look as follows:

```
(define my−log (open−file "scm_out_2.dat" "w"))
```

```
;;
(define radius 10.0)
(define dx (/ (* 2.0 radius) 40.0))
(define dy dx)
;;
(define center_1 (3D-vector 40.0 64.0 40.0))
(define normal_1 (- e_z e_x))
(define disk_1 (make-disk center_1 normal_1 radius))
;;
(slot-set! disk_1 'increments (vector dx dy))
(iterator disk_1)
;;
(define energy_1 0)
;;
(define center_2 (3D-vector 88.0 64.0 40.0))
(define normal_2 (+ e_z e_x))
(define disk_2 (make-disk center_2 normal_2 radius))
;;
(slot-set! disk_2 'increments (vector dx dy))
(iterator disk_2)
;;
(define energy_2 0)
;;
(define forms.post.iteration.lambda
  (lambda ()
    (set! energy_1 (+ energy_1 (* (compute-flux "E" "H" disk_1) forms.dt)))
    (set! energy_2 (+ energy_2 (* (compute-flux "E" "H" disk_2) forms.dt)))
    (format my-log "t = ~a~%" forms.time_e)
    (format my-log " energy absorbed by detector 1 = ~a~%" energy_1)
    (format my-log " energy absorbed by detector 2 = ~a~%" energy_2)
    (force-output my-log)))
;;
(define forms.post.all.lambda
  (lambda ()
    (format my-log "post.all: finished~%")
    (close-port my-log)))
```

The default increment for the disk is radius/100. Here we change it with the (**slot-set**! disk 'increments (**vector** dx dy)) command to match what's been used in `PowerFlux.scm`. The clause (**iterator** disk) recomputes the plaquettes for the new disk partitioning. Invoking `compute-flux` with two vector field names, as in (**compute-flux** "E""H"disk) evaluates the flux of their vector product, that is, $\boldsymbol{E} \times \boldsymbol{H}$. The results obtained are identical down to all significant digits, which in effect yields an additional library test.

2.6.8 Fourier Analysis

File: `Examples/FourierFlux.scm`

For the same set up as discussed in the previous section, that is, two angled cylindrical detectors, we can ask about power received through filters at specific frequencies, where filters act not on power itself, but on $\boldsymbol{E}$ and $\boldsymbol{H}$ separately, filtering $\boldsymbol{E}(\omega)$ and $\boldsymbol{H}(\omega)$ only. This case is discussed in Section 3.5.1, page 167, where we demonstrate that the average energy flux, due to $\boldsymbol{E}(\boldsymbol{r},\omega)$ and $\boldsymbol{H}(\boldsymbol{r},\omega)$ (this is the so called "monochromatic formula") over the period of vibration, $T = 2\pi/\omega$, is given by (cf. equation (379))

$$\frac{1}{2}\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right). \tag{57}$$

To evaluate this quantity, we have to calculate Fourier transform of the incident signal for $\boldsymbol{E}(t)$ and $\boldsymbol{H}(t)$ at every point of the detector's surface and for every frequency of interest for $t \in [t_{\text{begin}}, t_{\text{end}}]$, and only after this computation completes, can we evaluate (57). If a flux of (57) needs to be computed, we can only do so after we have evaluated $\hat{\boldsymbol{E}}(\omega)$ and $\hat{\boldsymbol{H}}(\omega)$, for every point of interest, first.

To handle the Fourier transform accumulations for the detector surface we first define the range of frequencies and the number of frequencies we want to look at within the range, then allocate space for the arrays that will hold accumulted quantities. This part of the input file code is shown below:

```
;; Prepare data structures for Fourier analysis
;;
(define omega_1 (/ (* 2 pi) wavelength))
(define omega_0 (/ omega_1 2.0))
(define n_frequencies 10)
(define d_omega (/ (- omega_1 omega_0) (- n_frequencies 1)))
;;
(define frequencies (make-vector n_frequencies 0.0))
(define flux-vector-1 (make-vector n_frequencies 0.0))
(define flux-vector-2 (make-vector n_frequencies 0.0))
;;
(do ((count 0 (1+ count)))
    ((>= count n_frequencies))
  (vector-set! frequencies count (+ omega_0 (* count d_omega))))
;;
(define E-array-1 (make-array #(0 0 0) n_points n_points n_frequencies))
(define H-array-1 (make-array #(0 0 0) n_points n_points n_frequencies))
(define E-array-2 (make-array #(0 0 0) n_points n_points n_frequencies))
(define H-array-2 (make-array #(0 0 0) n_points n_points n_frequencies))
;;
```

Each detector has two arrays of vectors associated with it: `E-array` and `H-array`. The entries in both arrays are complex valued vectors obtained by accumulating

$$\hat{\boldsymbol{E}}(\boldsymbol{r},\omega) \quad \leftarrow \quad \hat{\boldsymbol{E}}(\boldsymbol{r},\omega) + \boldsymbol{E}(\boldsymbol{r},t)e^{-\mathrm{i}\omega t}\,\mathrm{d}t \quad \text{and} \tag{58}$$

$$\hat{\boldsymbol{H}}(\boldsymbol{r},\omega) \quad \leftarrow \quad \hat{\boldsymbol{H}}(\boldsymbol{r},\omega) + \boldsymbol{H}(\boldsymbol{r},t)e^{-\mathrm{i}\omega t}\,\mathrm{d}t \tag{59}$$

throughout the whole duration of signal propagation through the computational domain.

The corresponding part of Scheme code, encapsulated in function `accumulate-fourier-transform`, is invoked by `forms.post.iteration.lambda`:

```
;; Now the post-iteration function
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format my-log "t = ~a, accumulating Fourier transforms ... " forms.time_e)
    (accumulate-fourier-transform center_1 n1 ex1 ey1 E-array-1 H-array-1)
    (accumulate-fourier-transform center_2 n2 ex2 ey2 E-array-2 H-array-2)
    (format my-log "done.~%")
    (force-output my-log)))
;;
```

after every iteration, and its own listing looks as follows:

```
;; Define the function that accumulates the Fourier transform.
;; Similar to compute-local-flux.
;;
(define accumulate-fourier-transform
  (lambda (center n ex ey E-array H-array)
    (do ((x x-lo (+ x dx))
         (i 0 (1+ i)))
        ((> x x-hi))
      (do ((y y-lo (+ y dy))
           (j 0 (1+ j)))
          ((> y y-hi))
        (if (<= (+ (* x x) (* y y)) radius-square)
            (let* ((position (vector-add
```

```
                    center (vector−add (vector−mult x ex)
                                       (vector−mult y ey))))
            (x0 (vector−ref position 0))
            (y0 (vector−ref position 1))
            (z0 (vector−ref position 2))
            (E (vector
                (interpolate "Ex" x0 y0 z0)
                (interpolate "Ey" x0 y0 z0)
                (interpolate "Ez" x0 y0 z0)))
            (H (vector
                (interpolate "Hx" x0 y0 z0)
                (interpolate "Hy" x0 y0 z0)
                (interpolate "Hz" x0 y0 z0))))
        (do ((k 0 (1+ k)))
            ((>= k n_frequencies))
          (let ((factor (* (exp (* 0−i (vector−ref frequencies k) forms.time_e)) forms.dt)))
            (array−set! E−array
                        (vector−add (array−ref E−array i j k)
                                    (vector−mult factor E))
                        i j k)
            (array−set! H−array
                        (vector−add (array−ref H−array i j k)
                                    (vector−mult factor H))
                        i j k)))))))))
;;
```

In this code we make use of the Scheme's ability to handle complex number arithmetic, which extends to all basic functions, including `exp`. The code shows how the above two updates are carried out for every point of the detector surface and for every frequency, for $\boldsymbol{E}$ and $\boldsymbol{H}$.

The flux of (57) through the detector's surface is computed only after the iterations have completed, by function `forms.post.all.lambda`, which in this case is somewhat more elaborate. Before we proceed to discuss the function, we first have a look at how the local flux of (57) is computed on the node. This is done by function `compute-local-flux`, which is listed below.

```
;; Define the function that computes local flux
;;
(define compute−local−flux
  (lambda (n E−array H−array flux−vector)
    (do ((k 0 (1+ k)))
        ((>= k n_frequencies))
      (do ((x x−lo (+ x dx))
           (i 0 (1+ i)))
          ((> x x−hi))
        (do ((y y−lo (+ y dy))
             (j 0 (1+ j)))
            ((> y y−hi))
          (if (<= (+ (* x x) (* y y)) radius−square)
              (vector−set!
               flux−vector k
               (+ (vector−ref flux−vector k)
                  (* d2s 0.5
                     (dot−product
                      n
                      (vector−real−part
                       (cross−product (array−ref E−array i j k)
                                      (vector−complex−conjugate (array−ref H−array i j k))))))))))))
;;
```

For every frequency ω_k and for every entry in the arrays of $\boldsymbol{E}$ and $\boldsymbol{H}$ vectors, that correspond to a given (x', y') location on the detector's surface, we

1. evaluate (57),
2. take the dot product of the resulting vector with the normal for the given detector,
3. multiply the resulting number by d^2S,
4. and add it to whatever has been already accumulated in the k-th slot of the corresponding `flux-vector`, which has been prepared and initialized to zero beforehand.

The combination of complex arithmetic and vector arithmetic used by the code make this part of the code rather easy to read.

The `forms.post.all.lambda` functions is listed below:

```
;;
(define forms.post.all.lambda
  (lambda ()
    (format my-log "Computing local fluxes ... ")
    (compute-local-flux n1 E-array-1 H-array-1 flux-vector-1)
    (compute-local-flux n2 E-array-2 H-array-2 flux-vector-2)
    (format my-log "done~%")
    (force-output my-log)
    (format my-log "Collecting local fluxes from MPI processes ... ")
    (let* ((buffer-1 (make-vector pool-size 0))
           (buffer-2 (make-vector pool-size 0)))
      (do ((k 0 (1+ k)))
          ((>= k n_frequencies))
        (gather buffer-1 (vector-ref flux-vector-1 k) root-rank)
        (gather buffer-2 (vector-ref flux-vector-2 k) root-rank)
        (if (= my-rank root-rank)
            (begin
              (vector-set! flux-vector-1 k
                           (do ((count 0 (1+ count))
                                (sum 0))
                               ((>= count pool-size) sum)
                             (set! sum (+ sum (vector-ref buffer-1 count)))))
              (vector-set! flux-vector-2 k
                           (do ((count 0 (1+ count))
                                (sum 0))
                               ((>= count pool-size) sum)
                             (set! sum (+ sum (vector-ref buffer-2 count))))))))
    (format my-log "done.~%")
    (force-output my-log)
    (format my-log " frequencies = ~a~%" frequencies)
    (format my-log " flux-vector-1 = ~a~%" flux-vector-1)
    (format my-log " flux-vector-2 = ~a~%" flux-vector-2)
    (close-port my-log)))
;;
```

The function calls `compute-local-flux` on both detectors. But the fluxes are local to each MPI process, that is, collected only on the data available to this process. We still have to add whatever the processes collected. And so, for every frequency ω_k we `gather` the corresponding entry of the `flux-vector` for a given detector on the `buffer` array, which is then received by the `root-rank` process. The `root-rank` process is the only one now that adds all the entries received in the `buffer` and replaces its own value of `flux-vector(k)` with the sum. This is repeated for both detectors.

Finally, *all* processes print their values of both `flux-vectors` on their log, but only the `root-rank` process will print the sum of fluxes collected from the MPI pool.

The Fourier analysis should be carried out on a specially conditioned signal. The signal should be windowed at both sides, so that the system is fully quiescent at the beginning and then at the end of the computational process. For these tests we have used a *tanh* trimmed signal defined as follows:

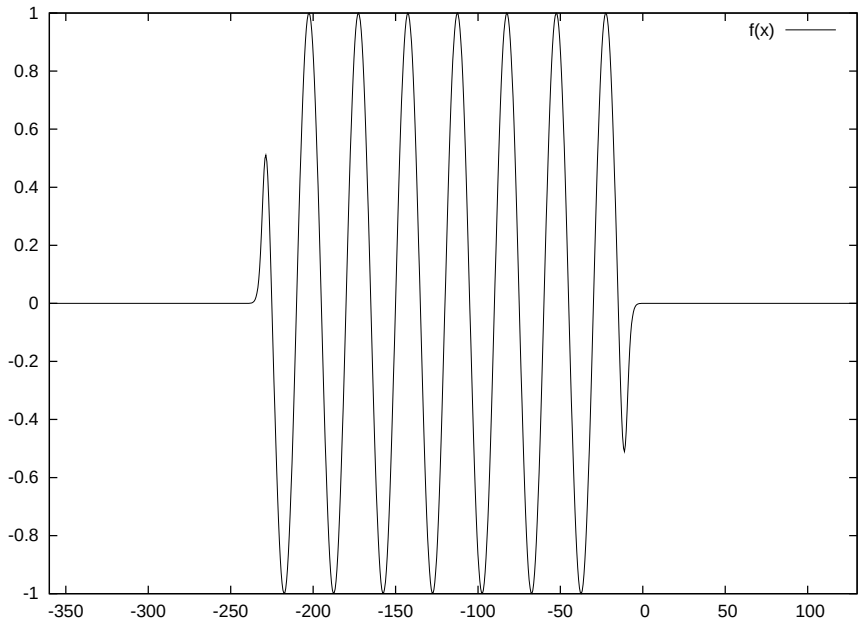


Figure 9: Incident signal used in the Fourier transform test.

```
;;
(define pi (* 2.0 (asin 1.0)))
;;
(define wavelength 30.0)
(define half-width 110.0)
(define delay 120.0)
(define slope 0.4)
(define t_begin 0.0)
(define t_end 360.0)
;;
(define window
  (lambda (zeta)
    (* 0.25
       (+ 1.0 (tanh (* slope (+ zeta half-width delay))))
       (- 1.0 (tanh (* slope (+ zeta (- half-width) delay)))))))
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (* (window zeta)
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
```

This signal is shown in Figure 9. The following shows a sample of output for the `root-rank` process:

Guile **log**:
  process number: 0

```
    pool size: 8
Computing flux of E(omega) x H(omega) through two photomultiplier tubes
    tube 1: radius = 10.0, center = #(40.0 64.0 40.0), normal = #(−0.707106781186547 0.0 0.707106781186547)
                ex = #(0.0 1.0 0), ey = #(0.707106781186547 0.0 0.707106781186547)
    tube 2: radius = 10.0, center = #(88.0 64.0 40.0), normal = #(0.707106781186547 0.0 0.707106781186547)
                ex = #(0.0 −1.0 0), ey = #(−0.707106781186547 0.0 0.707106781186547)
    frequencies = #(0.10471975511966 0.116355283466289 0.127990811812917 0.139626340159546
                    0.151261868506175 0.162897396852804 0.174532925199433 0.186168453546062
                    0.197803981892691 0.20943951023932)
Expected number of iterations: 1440.0
t = 0.25, accumulating Fourier transforms ... done.
t = 0.5, accumulating Fourier transforms ... done.
t = 0.75, accumulating Fourier transforms ... done.
...
t = 359.5, accumulating Fourier transforms ... done.
t = 359.75, accumulating Fourier transforms ... done.
t = 360.0, accumulating Fourier transforms ... done.
Computing local fluxes ... done
Collecting local fluxes from MPI processes ... done.
    frequencies = #(0.10471975511966 0.116355283466289 0.127990811812917 0.139626340159546
                    0.151261868506175 0.162897396852804 0.174532925199433 0.186168453546062
                    0.197803981892691 0.20943951023932)
    flux−vector−1 = #(7212.4253914238 3218.78477254804 4930.02164635976 17566.8698170516
                    1.56298336505291 41839.1687806656 30477.597762414 65929.8700247953
                    730070.875643088 1308953.06264268)
    flux−vector−2 = #(7212.63190597293 3218.74515307828 4929.91929567294 17567.4260835826
                    1.57433269583193 41842.6879076376 30478.5812841156 65935.1828632348
                    730119.249193959 1309092.17962795)
```

The injected signal is almost monochromatic. This shows in the presence of the peak at $\omega_{10} = 2\pi/\lambda$ with other values (apart from ω_9) strongly quenched. In this case, the `root-rank` process does not have the detectors within its own domain at all, so all data is collected from other processes. Detector 1 is split evenly between processes 5 and 7, and detector 2 is split between processes 1 and 3. It is easy to see, by looking at these processes' logs that the `root-rank` processes has evaluated the sums correctly.

As before, we notice that both detectors absorb nearly identical amounts of energy in each spectral range of interest with the relative discrepancy of about 1.1×10^{-4} for the peak and 7.3×10^{-3} for the trough.

The computation was carried out on a 128^3 grid, with $\Delta x = 1$, split amongst eight MPI processes, in a time-sharing mode with other jobs running on the same CPUs.

Repeating the same job on a 256^3 grid, with $\Delta x = 0.5$ returned the following results:

```
Computing local fluxes ... done
Collecting local fluxes from MPI processes ... done.
    frequencies = #(0.10471975511966 0.116355283466289 0.127990811812917 0.139626340159546
                    0.151261868506175 0.162897396852804 0.174532925199433 0.186168453546062
                    0.197803981892691 0.20943951023932)
    flux−vector−1 = #(7221.09970854857 3224.64660958954 4939.71464575871 17612.9565624663
                    1.59833748280165 41982.8524065345 30585.4335921414 66205.9090791842
                    733648.293765427 1316166.67823206)
    flux−vector−2 = #(7221.12530500186 3224.65534848003 4939.71956259091 17613.0258855159
                    1.5983396938775 41983.3071312828 30585.8309692985 66207.1433095112
                    733666.428876275 1316208.40070714)
```

This time the discrepancy between the left and the right detector is less still, the relative difference being 3.2×10^{-5} for the peak and 1.4×10^{-6} for the trough.

The code discussed in this section is quite involved, because it implements all required utilities, including vector operations, explicitly, which bloats its size. There are 310 code lines on the file (i.e., not counting comments and white space). The Forms auxiliary library, discussed in the next section, implements and wraps concepts and operations required. By calling library operations, the source file can be shrunk to 97 lines of code only, and made easier to parse, too. Apart from vector operations, the library also captures computations of fluxes, Fourier accumulations on surfaces, and the computation of the monochromatic power flux presented

here. A file that performs the same computation, but calls library utilities instead, is listed and discussed in Section 2.8, page 128.

Is the above result correct, at least as to the order of magnitude?

We can check it by considering the harmonic component of the signal only and evaluating the integral between 0 and $210 = 7\lambda$—the total width of the packet, minus the two smaller peaks at the beginning and the end. The harmonic component is $E_x(t) = \sin \omega t$, where $\omega = 2\pi/\lambda$ and $\lambda = 30$, and its Fourier transform is approximately

$$\hat{E}_x(\omega) = \int_0^{7\lambda} \sin \omega t \, e^{-\mathrm{i}\omega t} \, \mathrm{d}t = \int_0^{7\lambda} \sin \omega t \, (\cos \omega t - \mathrm{i} \sin \omega t) \, \mathrm{d}t. \tag{60}$$

The first integral, of $\sin \omega t \cos \omega t$, over the full seven periods vanishes, leaving the second integral only:

$$\hat{E}_x(\omega) = -\mathrm{i} \int_0^{7\lambda} \sin^2 \omega t \, \mathrm{d}t = -\frac{\mathrm{i}}{\omega} \int_0^{7\times 2\pi} \sin^2 x \, \mathrm{d}x = -\mathrm{i}\frac{7}{\omega} \int_0^{2\pi} \sin^2 x \, \mathrm{d}x = -\mathrm{i}\frac{7\pi}{\omega} = -\mathrm{i}\tfrac{7}{2}\lambda. \tag{61}$$

For the injected signal $\boldsymbol{H} = \boldsymbol{n} \times \boldsymbol{E}$, where $\boldsymbol{n}$ is the direction of the signal's propagation, see Section 3.3, page 149. Since here $\boldsymbol{n} = \boldsymbol{e}_z$, we find that $\boldsymbol{H} = H_y \boldsymbol{e}_y$ and $H_y = E_x$. Hence

$$\hat{H}_y(\omega) = \hat{E}_x(\omega) = -\mathrm{i}\tfrac{7}{2}\lambda. \tag{62}$$

And so,

$$\tfrac{1}{2}\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right) = \tfrac{1}{2}\left(\tfrac{7}{2}\right)^2 \lambda^2 \boldsymbol{e}_z. \tag{63}$$

The normal to the detectors, each being a disk of radius 10, is $(\boldsymbol{e}_z \pm \boldsymbol{e}_x)/\sqrt{2}$, therefore the flux evaluates to

$$\tfrac{1}{2}\left(\tfrac{7}{2}\right)^2 \lambda^2 \pi 10^2/\sqrt{2} \approx 1,224,569.6, \tag{64}$$

which is close enough to $1,316,208.4$ obtained from our numerical calculation (the discrepancy is about 7%), especially, given that we have neglected the two smaller peaks at the beginning and the end of the packet.

2.7 The Library

The FORMS library is a set of predefined Scheme files that can be placed anywhere on the system—their location is not hardwired in the FORMS code. To point Guile to the right library location for `load-from-path` or `primitive-load-path` users should either `cons` or `append` the location to `%load-path`, which is a list of strings, each string a pathname that is going to be searched for the target file. This should be done before the first load, for example,

```
guile> %load−path
("/usr/share/guile/site" "/usr/share/guile/1.8" "/usr/share/guile")
guile> (set! %load−path (cons "/home/gustav/src/Forms/lib" %load−path))
guile> %load−path
("/home/gustav/src/Forms/lib" "/usr/share/guile/site" "/usr/share/guile/1.8"
"/usr/share/guile")
guile>
```

Having done so, we can now load library modules without much ado, for example,

```
guile> (load−from−path "constants.scm")
guile>
```

or

```
guile> (load−from−path "constants")
guile>
```

because `".scm"` is a default file extension for loads. A list of default extensions is bound to `%load-extensions`.

If the library is to be used during interactive Guile sessions, then one can automate the above procedure by inserting the evaluations in the `~/.guile` file, which Guile reads on startup. On this occasion it is also a good idea to load the `readline` module from `ice-9`. For example:

```
(use−modules (ice−9 readline))
(activate−readline)
(set! %load−path (cons "/home/gustav/src/Forms/lib" %load−path))
```

2.7.1 Constants

Synopsis:

```
guile> (load−from−path "constants")
```

This file defines mathematical, physics and fitted constants of use in nano-photonics. The constants are as shown in Tables 7 and 8. The way the constants appear in specific formulas is as follows:

- For the Drude model (Baida and Van Labeke [2]):

$$\epsilon(\omega) = 1 - \frac{\omega_p^2}{\omega(\omega + \mathrm{i}\gamma)}. \tag{65}$$

- For the Drude model with ϵ_∞ (Gray and Kupka [10]):

$$\epsilon(\omega) = \epsilon_\infty - \frac{\omega_p^2}{\omega(\omega + \mathrm{i}\gamma)}. \tag{66}$$

- For the two pole Lorentz/Drude model (Lee and Gray [13]):

$$\epsilon(\omega) = \epsilon_\infty - \frac{\omega_D^2}{\omega(\omega + \mathrm{i}\gamma_D)} - \frac{\omega_1^2 g_1 \Delta\epsilon}{\omega^2 - \omega_1^2 + 2\mathrm{i}\gamma_1\omega} - \frac{\omega_2^2 g_2 \Delta\epsilon}{\omega^2 - \omega_2^2 + 2\mathrm{i}\gamma_2\omega}, \tag{67}$$

  where $g_1 + g_2 = 1$. We prefer to use $\epsilon_1 = g_1\Delta\epsilon$ and $\epsilon_2 = g_2\Delta\epsilon$ instead.

Table 7: Constants defined on `constants.scm`

symbol	value	comment
mathematical constants		
pi	$\pi = 2\arcsin(1)$	
two_pi	2π	
half_pi	$\pi/2$	
degree	$1^\circ = \pi/180$	
e	$e = \exp(1)$	
physics constants		
speed_of_light	$c = 2.997925 \times 10^8\,\mathrm{m/s}$	*speed of light in vacuum*
c	c	*as above*
epsilon_0	$\epsilon_0 = 8.854185 \times 10^{-12}\,\mathrm{C/(Vm)}$	*permittivity of vacuum*
eps_0	ϵ_0	*as above*
mu_0	$\mu_0 = 4\pi \times 10^{-7}\,\mathrm{Vs/(Am)}$	*permeability of vacuum*
q_e	$q_e = 1.60210 \times 10^{-19}\,\mathrm{C}$	*electron charge*
eV	$q_e \times 1\mathrm{V} = 1.60210 \times 10^{-19}\mathrm{J}$	*electron charge × volt*
gauss	$10^{-4}\,\mathrm{T}$	*one gauss*
nm	$10^{-9}\,\mathrm{m}$	*one nanometer*
m_e	$m_e = 9.1091 \times 10^{-31}\,\mathrm{kg}$	*electron mass*
m_p	$m_p = 1.67252 \times 10^{-27}\,\mathrm{kg}$	*proton mass*
m_n	$m_n = 1.67482 \times 10^{-27}\,\mathrm{kg}$	*neutron mass*
k_B	$k_B = 1.38054 \times 10^{-23}\,\mathrm{J/K}$	*Boltzmann constant*
N_A	$N_A = 6.02252 \times 10^{23}\,1/\mathrm{mol}$	*Avogadro number*
h_planck	$h = 6.6256 \times 10^{-34}\,\mathrm{Js}$	*Planck's constant*
hbar_planck	$\hbar = h/(2\pi)$	
material constants		
epsilon_air	$\epsilon_{\mathrm{air}} = 1.00054$	*dielectric constant of air*
epsilon_Si	$\epsilon_{\mathrm{Si}} = 11.7$	*dielectric constant of silicon*
epsilon_SiO2	$\epsilon_{\mathrm{SiO_2}} = 3.8$	*dielectric constant of silicon dioxide*
epsilon_Si3N4	$\epsilon_{\mathrm{Si_3N_4}} = 7.5$	*dielectric constant of silicon nitride*
epsilon_H2O	$\epsilon_{\mathrm{H_2O}} = 80.1$	*dielectric constant of water*
metal constants *Drude model, from Baida and Van Labeke[2]*		
omega_BL_Ag_450_800	$\omega_p = 1.374 \times 10^{16}\,\mathrm{rad/s}$	*silver plasma frequency for* $\lambda \in [450\,\mathrm{nm}, 800\,\mathrm{nm}]$
gamma_BL_Ag_450_800	$\gamma = 3.210 \times 10^{13}\,\mathrm{rad/s}$	*silver attenuation for* $\lambda \in [450\,\mathrm{nm}, 800\,\mathrm{nm}]$
omega_BL_Au_700_1800	$\omega_p = 1.236 \times 10^{16}\,\mathrm{rad/s}$	*gold plasma frequency for* $\lambda \in [700\,\mathrm{nm}, 1800\,\mathrm{nm}]$
gamma_BL_Au_700_1800	$\gamma = 1.300 \times 10^{14}\,\mathrm{rad/s}$	*gold attenuation for* $\lambda \in [700\,\mathrm{nm}, 1800\,\mathrm{nm}]$
Drude model with ϵ_∞*, from Gray and Kupka [10], based on Lynch and Hunter [14]*		
omega_LH_Ag_300_400	$\omega_p = 11.585\,\mathrm{eV}/\hbar$	*silver plasma frequency for* $\lambda \in [300\,\mathrm{nm}, 400\,\mathrm{nm}]$
gamma_LH_Ag_300_400	$\gamma = 0.203\,\mathrm{eV}/\hbar$	*silver attenuation for* $\lambda \in [300\,\mathrm{nm}, 400\,\mathrm{nm}]$
epsilon_LH_Ag_300_400	$\epsilon_\infty = 8.926$	*silver dielectric constant for* $\lambda \in [300\,\mathrm{nm}, 400\,\mathrm{nm}]$
omega_LH_Ag_400_500	$\omega_p = 9.812\,\mathrm{eV}/\hbar$	*silver plasma frequency for* $\lambda \in [400\,\mathrm{nm}, 500\,\mathrm{nm}]$
gamma_LH_Ag_400_500	$\gamma = 0.259\,\mathrm{eV}/\hbar$	*silver attenuation for* $\lambda \in [400\,\mathrm{nm}, 500\,\mathrm{nm}]$
epsilon_LH_Ag_400_500	$\epsilon_\infty = 5.976$	*silver dielectric constant for* $\lambda \in [400\,\mathrm{nm}, 500\,\mathrm{nm}]$

Table 8: Constants defined on `constants.scm`, continued

symbol	value	comment
A two-pole Lorentz/Drude model with ϵ_∞, from Lee and Gray [13], based on Johnson and Christy [12] *All parameters apply to silver for $\lambda \in [250\,\mathrm{nm}, 1000\,\mathrm{nm}]$*		
epsilon_LG_Ag_D_250_1000	$\epsilon_\infty = 2.3646$	
epsilon_LG_Ag_1_250_1000	$\epsilon_1 = g_1 \Delta\epsilon = 0.2663 \times 1.1831$	
epsilon_LG_Ag_2_250_1000	$\epsilon_2 = g_2 \Delta\epsilon = 0.7337 \times 1.1831$	$g_1 + g_2 = 1$
omega_LG_Ag_D_250_1000	$\omega_D = 8.73770\,\mathrm{eV}/\hbar$	
omega_LG_Ag_1_250_1000	$\omega_1 = 4.38020\,\mathrm{eV}/\hbar$	
omega_LG_Ag_2_250_1000	$\omega_2 = 5.18300\,\mathrm{eV}/\hbar$	
gamma_LG_Ag_D_250_1000	$\gamma_D = 0.07489\,\mathrm{eV}/\hbar$	
gamma_LG_Ag_1_250_1000	$\gamma_1 = 0.28000\,\mathrm{eV}/\hbar$	
gamma_LG_Ag_2_250_1000	$\gamma_2 = 0.54820\,\mathrm{eV}/\hbar$	
A two-pole Lorentz/Drude model with ϵ_∞, from McMahon, et al. [17], also based on [12] *All parameters apply to gold for $\lambda \in [250\,\mathrm{nm}, 1000\,\mathrm{nm}]$*		
epsilon_JMM_Au_D_250_1000	$\epsilon_\infty = 5.39833498$	
epsilon_JMM_Au_1_250_1000	$\epsilon_1 = g_1 \Delta\epsilon = 0.267871678 \times 2.541747093$	
epsilon_JMM_Au_2_250_1000	$\epsilon_2 = g_2 \Delta\epsilon = 0.73212832 \times 2.541747093$	$g_1 + g_2 = 1$
omega_JMM_Au_D_250_1000	$\omega_D = 9.2006880\,\mathrm{eV}/\hbar$	
omega_JMM_Au_1_250_1000	$\omega_1 = 2.8131406\,\mathrm{eV}/\hbar$	
omega_JMM_Au_2_250_1000	$\omega_2 = 3.4394149\,\mathrm{eV}/\hbar$	
gamma_JMM_Au_D_250_1000	$\gamma_D = 0.068017714\,\mathrm{eV}/\hbar$	
gamma_JMM_Au_1_250_1000	$\gamma_1 = 0.286542200\,\mathrm{eV}/\hbar$	
gamma_JMM_Au_1_250_1000	$\gamma_2 = 0.434928310\,\mathrm{eV}/\hbar$	

All quantities are expressed in SI units. For example, where we write $\omega_p = 11.585\,\mathrm{eV}/\hbar$, this really means

$$\omega_p = 11.585 \times 1.60210 \times 10^{-19}\,\mathrm{J} \times 2 \times \pi/(6.6256 \times 10^{-34}\,\mathrm{Js}) \approx 1.76 \times 10^{16}\,\mathrm{rad/s}. \tag{68}$$

All frequencies, including the gammas, are expressed in radians per second, not in Hertz.

This file additionally contains several unit conversion functions and other utilities, namely:

`eV->m` convert energy from eV to meters

`eV->um`, `eV->micron` convert energy from eV to microns

`eV->nm` convert energy from eV to nanometers

`eV->Hz` convert energy from eV to Hertz

`eV->radian-per-sec`, `eV->omega` convert energy from eV to radians per second

`eV->GHz` convert energy from eV to GHz

`eV->THz` convert energy from eV to THz

`THz->eV` convert energy from THz to eV

`nm->eV` convert energy from nanometers to eV

`Angstrom->eV` convert energy Angstroms to eV

`um->eV`, `micron->eV` convert energy from microns to eV

`(tabulate_n_BL_Ag_450_800 energies)` tabulate n for energies provided in the list for this model and substance

`(tabulate_n_BL_Au_700_1800 energies)`

`(tabulate_n_GAWL_Ag_750_900 energies)`

`(tabulate_n_BAF_Ag_700_1200 energies)`

`(tabulate_n_ZS_Ag_700_1200 energies)`

`(tabulate_n_LH_Ag_300_400 energies)`

`(tabulate_n_LH_Ag_400_500 energies)`

`(tabulate_n_LG_Ag_250_1000 energies)`

`(tabulate_n_JMM_Au_250_1000 energies)`

`(tabulate_n_RDEM_Ag energies)`

2.7.2 Vectors

Synopsis:

guile> (**load−from−path** "3D−vectors")

This library file implements operations on three-dimensional vectors. The design utilizes Guile's GOOPS module that's similar to CLOS. An object of class `<3D-vector>` has three slots, which are initialized by default to zero. Slot accessors are called `x-component`, `y-component` and `z-component`, and initialization flags are `#:x`, `#:y`, and `#:z`. For example,

```
guile> (set! %load-path (cons "/home/gustav/src/Forms/lib" %load-path))
guile> (load-from-path "3D-vectors")
guile> (define v (make <3D-vector> #:x 1 #:y 2 #:z 3))
guile> v
<1, 2, 3>
guile> (x-component v)
1
guile> (y-component v)
2
guile> (z-component v)
3
guile> (describe v)
<1, 2, 3> is an instance of class <3D-vector>
Slots are:
x = 1
y = 2
z = 3
guile>
```

The library defines the three basis vectors, $\boldsymbol{e}_x$, $\boldsymbol{e}_y$, and $\boldsymbol{e}_z$, referred to as `e_x`, `e_y`, and `e_z` respectively, as well as the zero vector $\mathbf{0}$, which is here called `e_0`, and overloads the basic arithmetic functions `+`, `-`, `*`, and `/`, to enable addition and subtraction of two vectors, as well as their multiplication and division by a number. The number in question *may* be complex. An auxiliary function `3D-vector` can be used to create a new vector, instead of invoking `make`, and another one `3D-vector?` can be used to check if a given object is a 3D-vector for example

```
guile> (3D-vector 1 2 3)
<1, 2, 3>
guile> (+ (* 1 e_x) (* 2 e_y) (* 3 e_z))
<1, 2, 3>
guile> (3D-vector? e_x)
#t
guile>
```

Users should take care not to confuse `e_x` with `E_x`, etc., in their scripts. Lisps tend to be case insensitive. But... Guile *is* case sensitive (all standard Lisp functions have lower case names only). For example, we can say in Guile

```
guile> (define E_x (+ (* 3 e_x) (* 2 e_y) e_z))
guile> E_x
<3, 2, 1>
guile> e_x
<1, 0, 0>
guile>
```

Nevertheless, it may be a good idea to differentiate between the two more, by some convention. For example, one may refer to the electric field by `Ex` without the underscore.

The two methods `real-part` and `imag-part` have been extended to return real or imaginary part of a complex valued three-dimensional vector. The `=` operator has been overloaded to return true if its two vector arguments are equal. Here is an example of how these operations work:

```
guile> (define v (make <3D-vector> #:x 1+2i #:y 1-2i #:z 2+1i))
guile> v
<1.0+2.0i, 1.0-2.0i, 2.0+1.0i>
guile> (real-part v)
<1.0, 1.0, 2.0>
guile> (imag-part v)
<2.0, -2.0, 1.0>
guile> (* 2 v)
<2.0+4.0i, 2.0-4.0i, 4.0+2.0i>
guile> (/ v 2)
```

```
<0.5+1.0i, 0.5−1.0i, 1.0+0.5i>
guile>
```

Here we demonstrate the `=` operator as applied to two vectors:

```
guile> (define w (make <3D−vector> #:x 1+2i #:y 1−2i #:z 2+1i))
guile> (= v w)
#t
guile> (define z (make <3D−vector>))
guile> z
<0, 0, 0>
guile> (= v z)
#f
guile>
```

Now, let us redefine $\boldsymbol{w}$ and demonstrate addition and subtraction of vectors:

```
guile> (define w (make <3D−vector> #:x 1 #:y 1 #:z 1))
guile> w
<1, 1, 1>
guile> (+ w v)
<2.0+2.0i, 2.0−2.0i, 3.0+1.0i>
guile> (− v w)
<0.0+2.0i, 0.0−2.0i, 1.0+1.0i>
guile>
```

We can also add a number to a vector or subtract one, in which case the number is automaticaly upgraded to number $\times\,[1,1,1]$. For example,

```
guile> (+ 2 v)
<3.0+2.0i, 3.0−2.0i, 4.0+1.0i>
guile> (− v 2)
<−1.0+2.0i, −1.0−2.0i, 0.0+1.0i>
guile>
```

The library adds a method for taking the complex conjugate both of a single complex number and of a complex valued vector. For example,

```
guile> (define a 1+3i)
guile> a
1.0+3.0i
guile> (complex−conjugate a)
1.0−3.0i
guile> v
<1.0+2.0i, 1.0−2.0i, 2.0+1.0i>
guile> (complex−conjugate v)
<1.0−2.0i, 1.0+2.0i, 2.0−1.0i>
guile>
```

The `*` symbol can be used on a complex-valued vector (but not on a complex scalar) to complex-conjugate it, too:

```
guile> v
<1.0+2.0i, 1.0−2.0i, 2.0+1.0i>
guile> (* v)
<1.0−2.0i, 1.0+2.0i, 2.0−1.0i>
guile>
```

This is an obvious analogue of $\boldsymbol{v}^*$.

When applied to two vectors, the `*` operator implements the scalar (dot) product, as follows:

$$(*\,\boldsymbol{u}\,\boldsymbol{v}) = \boldsymbol{u}\cdot\boldsymbol{v} = \sum_{i\in\{x,y,z\}} u_i v_i^*. \tag{69}$$

Here we follow a mathematical notation convention, which applies complex conjugate to the *second* complex-valued vector in the product. Switching to Scheme:

```
guile> (define u (make <3D−vector> #:x 1+1i #:y 2+2i #:z 3+3i))
guile> (define v (make <3D−vector> #:x 1−3i #:y 2−2i #:z 3−1i))
guile> (∗ u u)
28.0
guile> (∗ v v)
28.0
guile> (∗ u v)
4.0+24.0i
guile> (∗ v u)
4.0−24.0i
guile> (+ (∗ (x−component u) (complex−conjugate (x−component v)))
... (∗ (y−component u) (complex−conjugate (y−component v)))
... (∗ (z−component u) (complex−conjugate (z−component v))))
4.0+24.0i
guile>
```

A `norm` of a vector is implemented as $\sqrt{\boldsymbol{v}\cdot\boldsymbol{v}}$ (the second occurrence of $\boldsymbol{v}$ is internally complex-conjugated):

```
guile> (norm u)
5.29150262212918
guile> (sqrt (∗ u u))
5.29150262212918
guile>
```

The library also provides a query function `normalized?` and a function `normalize` that normalizes vectors. Here's how they can be used:

```
guile> (normalized? u)
#f
guile> (normalized? (normalize u))
#t
guile>
```

The cross-product of two vectors is defined by

$$(\times\, \boldsymbol{u}\, \boldsymbol{v}) = \boldsymbol{u}\times\boldsymbol{v} = \sum_i \left(\sum_{j,k} \epsilon_{ijk} u_j v_k^*\right) \boldsymbol{e}_i \tag{70}$$

As above, we apply complex conjugate to the second vector in the product. This is so that we can use the formula to render expressions such as $\hat{\boldsymbol{E}}(f)\times\hat{\boldsymbol{H}}^*(f)$ that appear in the Parseval's theorem and other similar expressions, cf. equations (369), (371), and (379) in Section 3.5.1, page 167.

Here is a demonstration that shows how this works in Scheme:

```
guile> (x e_x e_y)
<0, 0.0, 1.0>
guile> (x e_y e_x)
<0.0, 0, −1.0>
guile>
```

Last, we have a method in the library that returns a normalized vector perpendicular to its argument following this prescription:

$$\begin{aligned}
&(\text{perpendicular-to}\,\boldsymbol{v}):\\
&\quad \text{if}\quad \boldsymbol{v}=\boldsymbol{0}\quad\text{then}\quad \boldsymbol{0}\\
&\quad \text{else if}\quad v_x=v_y=0\quad\text{then}\quad [v_z,0,0]/\,|v_z|\\
&\quad \text{else}\quad [-v_y^*, v_x^*, 0]/\sqrt{v_x v_x^* + v_y v_y^*}.
\end{aligned}$$

This somewhat contrived operation works both for real and for complex-valued vectors. Here is a demonstration:

```
guile> a
<1, 0, 0>
guile> b
<0, 1, 0>
guile> c
<0, 0, 1>
guile> (perpendicular-to a)
<0.0, 1.0, 0.0>
guile> (perpendicular-to b)
<-1.0, 0.0, 0.0>
guile> (perpendicular-to c)
<1.0, 0.0, 0.0>
guile> (load-from-path "constants")
guile> (define w (make <3D-vector>
... #:x (exp (* 0+i (/ pi 4)))
... #:y (exp (* 0+i (* 3 (/ pi 4))))
... #:z (exp (* 0+i (* 5 (/ pi 4))))))
guile> w
<0.707106781186548+0.707106781186547i,
-0.707106781186547+0.707106781186548i,
-0.707106781186548-0.707106781186547i>
guile> (perpendicular-to w)
<0.5+0.5i, 0.5-0.5i, 0.0>
guile> (* w (perpendicular-to w))
0.0
guile> (norm (perpendicular-to w))
1.0
guile>
```

Below we list a brief summary of vector utilies provided.

vector constants

`e_x = <1,0,0>`, `e_y = <0,1,0>`, `e_z = <0,0,1>`, `e_0 = <0,0,0>`

vector operations

`3D-vector` make a 3D-vector, takes three coordinates as arguments;

`3D-vector?` checks if an object is a 3D-vector;

`=` compares two vectors;

`+` adds two or more vectors, adds a number to a vector, or a vector to a number;

`-` subtracts two or more vectors, subracts a number from a vector, or a vector from a number, negates a vector;

`*` evaluates a dot product of two vectors, multiplies a vector by a number, or a number by a vector; produces a complex conjugate of a complex-valued vector, when provided with one vector argument only;

`/` divides a vector by a number;

`real-part` extracts a real part from a complex number or a complex valued vector;

`imag-part` extracts an imaginary part from a complex number or a complex valued vector;

`complex-conjugate` takes a complex conjugate of a number or of a complex valued vector;

`x` evaluates a cross product of two vectors;

`norm` computes a norm (length) of a vector;

`normalize` normalizes a vector;

`normalized?` checks if a vector is normalized;

`perpendicular-to` finds a vector perpendicular to a given one;

`volume` evaluates the volume subtended by three vectors, can be used to check for linear independence;

spherical coordinates

`spherical->cartesian` Converts (r, θ, ϕ) given to it as separate arguments or as a vector to (x, y, z), which is returned as a vector;

`cartesian->spherical` Converts (x, y, z) given to it as separate arguments or as a vector to (r, θ, ϕ), which is returned as a vector;

`spherical->e_r` Returns $\boldsymbol{e}_r$ for a point (r, θ, ϕ) given as three separate arguments or a vector, in Cartesian coordinates;

`cartesian->e_r` Returns $\boldsymbol{e}_r$ for a point (x, y, z) given as three separate arguments or a vector, in Cartesian coordinates;

`spherical->e_theta` Returns $\boldsymbol{e}_\theta$ for a point (r, θ, ϕ) given as three separate arguments or a vector, in Cartesian coordinates;

`cartesian->e_theta` Returns $\boldsymbol{e}_\theta$ for a point (x, y, z) given as three separate arguments or a vector, in Cartesian coordinates;

`spherical->e_phi` Returns $\boldsymbol{e}_\phi$ for a point (r, θ, ϕ) given as three separate arguments or a vector, in Cartesian coordinates;

`cartesian->e_phi` Returns $\boldsymbol{e}_\phi$ for a point (x, y, z) given as three separate arguments or a vector, in Cartesian coordinates.

2.7.3 Signals

Synopsis:

guile> (**load−from−path** "chirps")
guile> (**load−from−path** "windows")

There are two files that provide functions for defining signals. The first one, `"chirps"`, defines frequency modulated signals of constant amplitude. These can then be multiplied by envelopes taken from `"windows"`, to produce finite pulses of various forms.

Both files carry extensive comments that explain the signals defined, providing their analytical form in TEX, example synopsis in Scheme, and Gnuplot commands to graph them. They also provide functions for numerical evaluation of the signals. The functions write their output on `"test.out"`. It can be then fed into Gnuplot for display. This is different from Gnuplot displaying a function according to an analytical formula provided.

Additionally, all functions are source-documented. Invoking `procedure-documentation` on a function prints its synopsis.

The signals and the windows are defined to unfold from right to left, because this is how they are injected into the total field region of FORMS.

Chirps

test-chirp

synopsis (test-chirp chirp width ω_0 α)

documentation

guile> (**procedure−documentation** test−chirp)
"Synopsis: (test−chirp chirp width omega_0 alpha)
Test the chirp function for x in [−6/5 ∗ width, width/5]."

linear-chirp

formula $f(x) = -\sin\left((\omega_0 - \alpha x)\, x\right)$

synopsis (linear-chirp x ω_0 α)

documentation

guile> (**procedure−documentation linear−chirp**)
"Synopsis: (linear−chirp x omega_0 alpha)
Return linear chirp of the form −sin((omega_0 − alpha ∗ x) ∗ x)."

test

guile> (**define** omega_0 (/ (∗ 2 pi) 1800.0))
guile> (test−chirp **linear−chirp** 60 omega_0 0.02)

exponential-chirp-1

formula $f(x) = \sin\left(\omega_0\left(e^{-\alpha x} - 1\right)\right)$

synopsis (exponential-chirp-1 x ω_0 α)

documentation

guile> (**procedure−documentation exponential−chirp−1**)
"Synopsis: (exponential−chirp−1 x omega_0 alpha)
Return exponential chirp of the form sin(omega_0 ∗ (exp (−alpha∗x) − 1))."

test

guile> (**define** omega_0 (/ (∗ 2 pi) 20.0))
guile> (test−chirp **exponential−chirp−1** 60 omega_0 0.1)

exponential-chirp-2

formula $f(x) = -\sin\left(\omega_0 e^{-\alpha x} x\right)$

synopsis (exponential-chirp-2 x ω_0 α)

documentation

guile> (**procedure−documentation exponential−chirp−2**)
"Synopsis: (exponential−chirp−2 x omega_0 alpha)
Return exponential chirp of the form −sin(omega_0 ∗ exp (−alpha ∗ x) ∗ x)."

test

guile> (**define** omega_0 (/ (∗ 2 pi) 20.0))
guile> (test−chirp **exponential−chirp−2** 500 omega_0 −0.002)

Procedure `test-chirp` writes two columns of numbers on file `"test.out"`. To display them in Gnuplot, one just has to issue the command:

gnuplot> **plot** "test.out" **with** lines

Windows

test-window

synopsis (test-window window width #:optional σ)

documentation

guile> (**procedure−documentation test−window**)
"Synopsis: (test−window window width #:optional sigma)
Test the window function for x in [−6/5 ∗ width, width/5]."

double-tanh No compact support.

formula $w(x, \text{width}, \sigma) = \frac{1}{4}\left(1 - \tanh(\sigma x)\right)\left(1 + \tanh(\sigma(x + \text{width}))\right)$

synopsis (double-tanh x width σ)

documentation

guile> (**procedure−documentation double−tanh**)
"Synopsis: (double−tanh x width sigma)
Return a window of the form
0.25 ∗ (1 − tanh (sigma ∗ x)) ∗ (1 + tanh (sigma ∗ (x + width))).
The window does not have a compact support."

test

guile> (**test−window double−tanh** 100 0.2)

gauss No compact support.

formula $w(x, \text{width}, \sigma) = \exp\left(-\frac{1}{2}\left(\frac{x+\text{width}/2}{\sigma\cdot\text{width}/2}\right)^2\right)$

synopsis (gauss x width σ)

documentation

guile> (**procedure−documentation gauss**)
"Synopsis: (gauss x width sigma)
Return a window of the form
exp (−0.5 ∗ ((x + width / 2) / (sigma ∗ width / 2))∗∗2).
The window does not have a compact support."

test

guile> (**test−window gauss** 100 0.3)

blackman-harris Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.35875 - 0.48829\cos\frac{2\pi x}{\text{width}} + 0.14128\cos\frac{4\pi x}{\text{width}} - 0.01168\cos\frac{6\pi x}{\text{width}}$

synopsis (blackman-harris x width)

documentation

guile> (**procedure−documentation blackman−harris**)
"Synopsis: (blackman−harris x width)
Return a window of the form
0.35875 − 0.48829 ∗ cos (2 ∗ pi ∗ x / width)
+ 0.14128 ∗ cos (4 ∗ pi ∗ x / width)
− 0.01168 ∗ cos (6 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window blackman−harris** 100)

blackman-nuttall Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.3635819 - 0.4891775\cos\frac{2\pi x}{\text{width}} + 0.1365995\cos\frac{4\pi x}{\text{width}} - 0.0106411\cos\frac{6\pi x}{\text{width}}$

synopsis (blackman-nuttall x width)

documentation

guile> (**procedure−documentation blackman−nuttall**)
"Synopsis: (blackman−nuttall x width)
Return a window of the form
0.3635819 − 0.4891775 ∗ cos (2 ∗ pi ∗ x / width)
+ 0.1365995 ∗ cos (4 ∗ pi ∗ x / width)
− 0.0106411 ∗ cos (6 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window blackman−nuttall** 100)

nuttall Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.355768 - 0.487396 \cos \frac{2\pi x}{\text{width}} + 0.144232 \cos \frac{4\pi x}{\text{width}} - 0.012604 \cos \frac{6\pi x}{\text{width}}$

synopsis (nuttall x width)

documentation

guile> (**procedure−documentation nuttall**)
"Synopsis: (nuttall x width)
Return a window of the form
0.355768 − 0.487396 ∗ cos (2 ∗ pi ∗ x / width)
+ 0.144232 ∗ cos (4 ∗ pi ∗ x / width)
− 0.012604 ∗ cos (6 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window nuttall** 100)

blackman Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{1-\alpha}{2} - \frac{1}{2} \cos \frac{2\pi x}{\text{width}} + \frac{\alpha}{2} \cos \frac{4\pi x}{\text{width}}$, where $\alpha = 0.16$.

synopsis (blackman x width)

documentation

guile> (**procedure−documentation blackman**)
"Synopsis: (blackman x width)
Return a window of the form
(1 − alpha) / 2 − 0.5 ∗ cos (2 ∗ pi ∗ x / width)
+ alpha / 2 ∗ cos (4 ∗ pi ∗ x / width),
where alpha = 0.16, between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window blackman** 100)

lanczos Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{\sin\left(\pi\left(\frac{2x}{\text{width}}+1\right)\right)}{\left(\pi\left(\frac{2x}{\text{width}}+1\right)\right)}$

synopsis (lanczos x width)

documentation

guile> (**procedure−documentation lanczos**)
"Synopsis: (lanczos x width)
Return a window of the form
sin (pi ∗ (2 ∗ x / width + 1)) / (pi ∗ (2 ∗ x / width + 1))
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window lanczos** 100)

hamming Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.53836 - 0.46164 \cos\left(\frac{2\pi x}{\text{width}}\right)$

synopsis (hamming x *width*)

documentation

guile> (**procedure−documentation hamming**)
"Synopsis: (hamming x width)
Return a window of the form
0.53836 − 0.46164 ∗ cos (2 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window hamming** 100)

hann Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi x}{\text{width}}\right)\right)$

synopsis (hann x width)

documentation

guile> (**procedure−documentation hann**)
"Synopsis: (hann x width)
Return a window of the form
0.5 ∗ (1 − cos (2 ∗ pi ∗ x / width))
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window hann** 100)

bartlett Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{2}{\text{width}}\left(\frac{\text{width}}{2} - \left|x + \frac{\text{width}}{2}\right|\right)$

synopsis (bartlett x width)

documentation

guile> (**procedure−documentation bartlett**)
"Synopsis: (bartlett x width)
Return a window of the form
2 / width ∗ (width / 2 − abs (x + width / 2))
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window bartlett** 100)

bartlett-hann Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.62 - 0.48\left|\frac{x}{\text{width}} + \frac{1}{2}\right| - 0.38\cos\left(\frac{2\pi x}{\text{width}}\right)$

synopsis (bartlett-hann x width)

documentation

guile> (**procedure−documentation bartlett−hann**)
"Synopsis: (bartlett−hann x width)
Return a window of the form
0.62 − 0.48 ∗ abs (x / width + 0.5) − 0.38 ∗ cos (2 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."

test

guile> (**test−window bartlett−hann** 100)

The `test-window` procedure invoked on a window function generates a two column output on `"test.out"`, which can be then displayed with gnuplot by issuing the command

gnuplot> **plot** "test.out" **with** lines

Signals

Synopsis:

guile> (**load−from−path** "signals")

This file loads `"windows"` and `"chirps"`, and then it defines procedure `test-chirp-in-window`, which combines chirps and windows, evaluating the resulting signal at 2,000 points within the window. The signal is printed out on `"test.out"` in two columns, ready to be displayed with gnuplot.

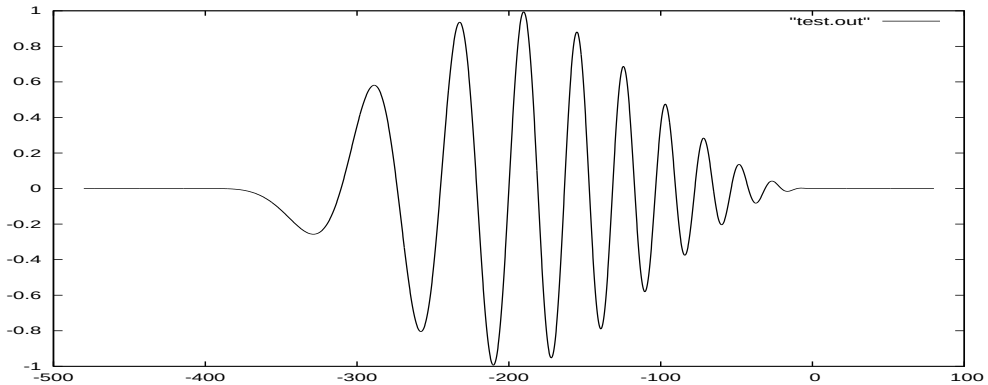
test-chirp-in-window

documentation

guile> (**procedure−documentation test−chirp−in−window**)
"Synopsis: (test−chirp−in−window chirp omega_0 alpha window width #:optional sigma)
Test the chirp in the window for x in [−6/5 ∗ width, width/5]."

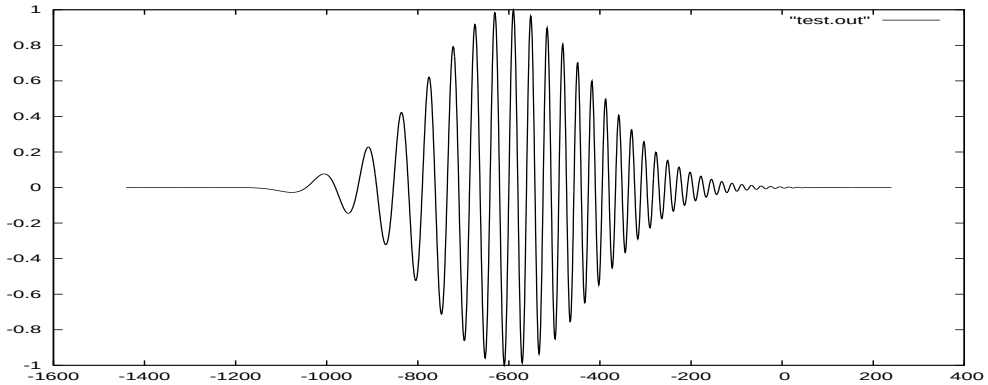
examples

1. A stretching linear chirp in Hann window:



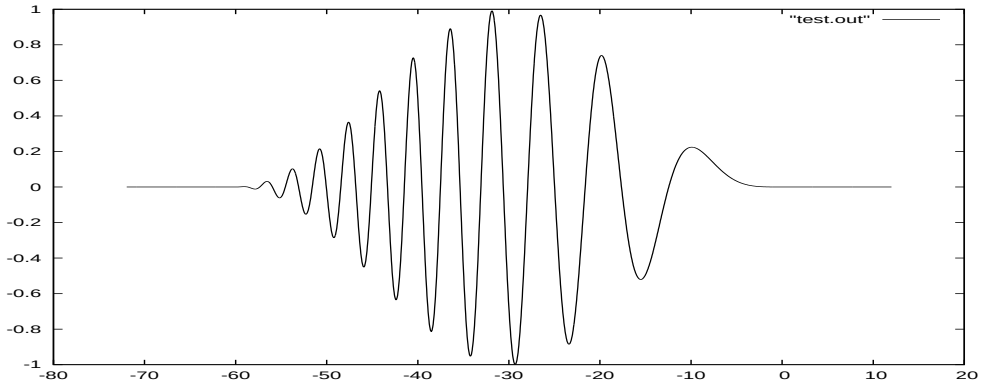
guile> (**define** omega_0 (/ (∗ 2 pi) 20.0))
guile> (**define** width 400.0)
guile> (**test−chirp−in−window linear−chirp** omega_0 (− (/ omega_0 (∗ 2.0 width)))
hann width)

2. A stretching linear chirp in Gauss window:



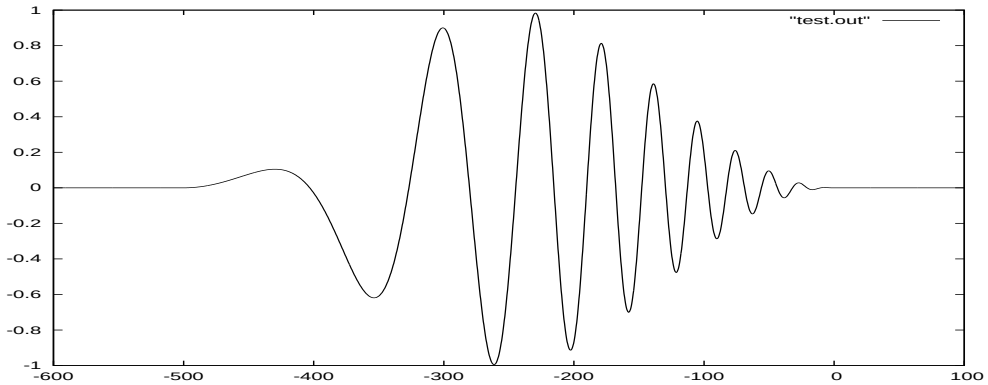
guile> (**define** omega_0 (/ (∗ 2 pi) 20.0))
guile> (**define** width 1200.0)
guile> (**test−chirp−in−window linear−chirp** omega_0 (− (/ omega_0 (∗ 2.0 width)))
gauss width 0.3)

3. A shrinking linear chirp in Hann window:



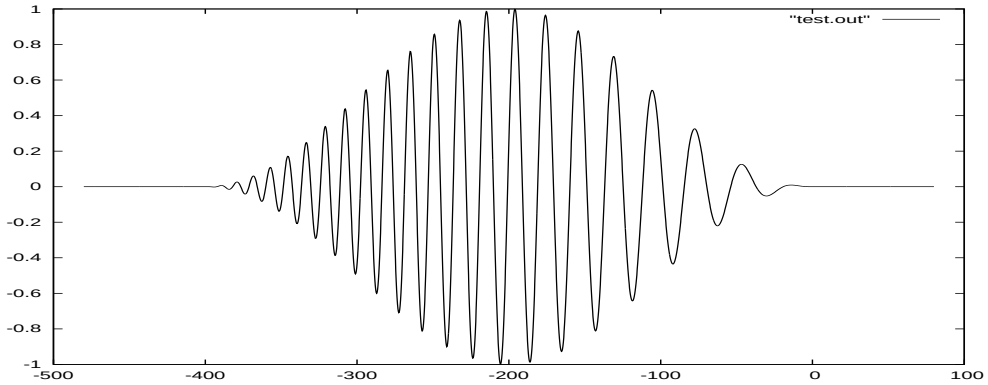
guile> (**define** omega_0 (/ (∗ 2 pi) 1800.0))
guile> (**define** width 60.0)
guile> (**test−chirp−in−window linear−chirp** omega_0 0.02 **hann** width)

4. A stretching exponential chirp in Hann window:



guile> (**define** omega_0 (/ (∗ 2 pi) 20.0))
guile> (**define** width 500.0)
guile> (**test−chirp−in−window exponential−chirp−2** omega_0 −0.002 **hann** width)

5. A shrinking exponential chirp in Hann window:



guile> (**define** omega_0 (/ (∗ 2 pi) 40.0))
guile> (**define** width 400.0)
guile> (**test−chirp−in−window exponential−chirp−2** omega_0 0.002 **hann** width)

2.7.4 Media

Synopsis:

guile> (**load−from−path** "media")

This file contains precooked auxiliary differential equation (ADE) solvers for various media types. The solvers are not "stand-alone". They are designed to slot into the FORMS input file, and utilize global objects defined and filled by FORMS, such as `forms.D`, `forms.media.number_of_auxiliary_fields`, `forms.S`, `forms.dt`, and `forms.E_old`. In effect, the solvers deliver

$$\boldsymbol{E}(t_n) \leftarrow \mathcal{E}\left(\boldsymbol{D}(t_n), \boldsymbol{S}(t_n), \boldsymbol{S}(t_{n-1}), \ldots, \boldsymbol{E}(t_{n-1}), t_n, \mathrm{d}t\right) \tag{71}$$

As a side effect, the solvers also update the auxiliary $\boldsymbol{S}$ field.

The media model implemented at present is isotropic. Minor changes to the code itself will be needed to add a tensor material model. This will be implemented in future releases ◀

Users should understand that the commonly used Courant-Friedrichs-Lewy time step criterion as applied to Maxwell equations does not cover stability of the ADE. It may happen that the Maxwell equations solver time step is too long for the ADE used. ◀

The solvers provided at present are

dielectric Frequency independent scalar dielectric characterized by ϵ.

formula $\epsilon(\omega) = \epsilon = \text{const}$

ADE $\boldsymbol{E}(t_n) = \epsilon^{-1}\boldsymbol{D}(t_n)$

synopsis (set! E (dielectric ϵ^{-1}))

documentation

guile> (**procedure−documentation dielectric**)
"Synopsis:

(dielectric one_by_epsilon)

Return the updated value for E that corresponds to this one_by_epsilon
and forms.D."

drude Drude metal characterized by ω_p, γ and ϵ_∞.

formula

$$\epsilon(\omega) = \epsilon_\infty - \frac{\omega_p^2}{\omega^2 + \mathrm{i}\gamma\omega}$$

ADE

$$\begin{aligned}
\boldsymbol{S}(t_n) &= \frac{1}{2+\gamma\Delta t}\left(4\boldsymbol{S}(t_{n-1}) - (2-\gamma\Delta t)\,\boldsymbol{S}(t_{n-2}) - 2\omega_p^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_\infty}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}(t_n)\right)\\
\boldsymbol{S}(t_{n-2}) &\leftarrow \boldsymbol{S}(t_{n-1})\\
\boldsymbol{S}(t_{n-1}) &\leftarrow \boldsymbol{S}(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 2)
(set! E (drude ϵ_∞^{-1} ω_p^2 γ))

documentation

guile> (**procedure−documentation drude**)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 2)
(drude one_by_epsilon_D omega_p_square gamma)

Return the updated value for E that corresponds to these three parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, forms.S[0], and forms.S[1]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

lorentz_1 A one-pole Lorentz medium characterized by ω_p, γ, ω_t and ϵ_L.

formula

$$\epsilon(\omega) = \epsilon_L - \frac{\omega_p^2}{\omega^2 + \mathrm{i}\gamma\omega - \omega_t^2}$$

ADE

$$\begin{aligned}
\boldsymbol{S}(t_n) &= \frac{1}{2+\gamma\Delta t}\left(\left(4 - 2\omega_t^2\Delta t^2\right)\boldsymbol{S}(t_{n-1}) - (2-\gamma\Delta t)\,\boldsymbol{S}(t_{n-2}) - 2\omega_p^2\Delta t^2\boldsymbol{E}(t_{n-1})\right) \\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_L}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}(t_n)\right) \\
\boldsymbol{S}(t_{n-2}) &\leftarrow \boldsymbol{S}(t_{n-1}) \\
\boldsymbol{S}(t_{n-1}) &\leftarrow \boldsymbol{S}(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 2)
(set! E (lorentz_1 ϵ_L^{-1} ω_p^2 ω_t^2 γ))

documentation

guile> (**procedure−documentation lorentz_1**)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 2)
(lorentz_1 one_by_epsilon_L omega_p_square omega_t_square gamma)

Return the updated value for E that corresponds to these four parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, forms.S[0], and forms.S[1]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

lorentz_2 A two pole Lorentz medium characterized by ϵ_L, then for the first pole ω_{p1}, ω_{t1}, and γ_1, and for the second pole ω_{p2}, ω_{t2}, and γ_2.

formula

$$\epsilon(\omega) = \epsilon_L - \frac{\omega_{p1}^2}{\omega^2 + \mathrm{i}\gamma_1\omega - \omega_{t1}^2} - \frac{\omega_{p2}^2}{\omega^2 + \mathrm{i}\gamma_2\omega - \omega_{t2}^2}$$

ADE

$$
\begin{aligned}
\boldsymbol{S}_1(t_n) &= \frac{1}{2+\gamma_1\Delta t}\left(\left(4-2\omega_{t1}^2\Delta t^2\right)\boldsymbol{S}_1(t_{n-1})-(2-\gamma_1\Delta t)\,\boldsymbol{S}_1(t_{n-2})-2\omega_{p1}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_2(t_n) &= \frac{1}{2+\gamma_2\Delta t}\left(\left(4-2\omega_{t2}^2\Delta t^2\right)\boldsymbol{S}_2(t_{n-1})-(2-\gamma_2\Delta t)\,\boldsymbol{S}_2(t_{n-2})-2\omega_{p2}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_L}\left(\boldsymbol{D}(t_n)+\boldsymbol{S}_1(t_n)+\boldsymbol{S}_2(t_n)\right)\\
\boldsymbol{S}_1(t_{n-2}) &\leftarrow \boldsymbol{S}_1(t_{n-1})\\
\boldsymbol{S}_1(t_{n-1}) &\leftarrow \boldsymbol{S}_1(t_n)\\
\boldsymbol{S}_2(t_{n-2}) &\leftarrow \boldsymbol{S}_2(t_{n-1})\\
\boldsymbol{S}_2(t_{n-1}) &\leftarrow \boldsymbol{S}_2(t_n)
\end{aligned}
$$

synopsis

(define forms.media.number_of_auxiliary_fields 4)
(set! E (lorentz_2 ϵ_L^{-1} ω_{p1}^2 ω_{t1}^2 γ_1 ω_{p2}^2 ω_{t2}^2 γ_2))

documentation

guile> (**procedure−documentation lorentz_2**)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 4)
(lorentz_2 one_by_epsilon_L omega_p_square_1 omega_t_square_1 gamma_1 omega_p_square_2 omega_t_square_2 gamma_2)

Return the updated value for E that corresponds to these seven parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, and forms.S[n], for n in [0..3]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

drude_lorentz_1 A mixed Drude-one-pole-Lorentz medium characterized by the following Drude parameters, ω_D and γ_D; then by the following Lorentz parameters, ω_{pL}, ω_{tL} and γ_L; and by ϵ_∞ here referred to as ϵ_{DL}.

formula

$$\epsilon(\omega)=\epsilon_{DL}-\frac{\omega_D^2}{\omega^2+\mathrm{i}\gamma_D\omega}-\frac{\omega_{pL}^2}{\omega^2+\mathrm{i}\gamma_L\omega-\omega_{tL}^2}$$

ADE

$$
\begin{aligned}
\boldsymbol{S}_D(t_n) &= \frac{1}{2+\gamma_D\Delta t}\left(4\boldsymbol{S}_D(t_{n-1})-(2-\gamma_D\Delta t)\boldsymbol{S}_D(t_{n-2})-2\omega_D^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_L(t_n) &= \frac{1}{2+\gamma_L\Delta t}\left((4-2\omega_{tL}^2\Delta t^2)\boldsymbol{S}_L(t_{n-1})-(2-\gamma_L\Delta t)\boldsymbol{S}_L(t_{n-2})-2\omega_{pL}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_{DL}}\left(\boldsymbol{D}(t_n)+\boldsymbol{S}_D(t_n)+\boldsymbol{S}_L(t_n)\right)\\
\boldsymbol{S}_D(t_{n-2}) &\leftarrow \boldsymbol{S}_D(t_{n-1})\\
\boldsymbol{S}_D(t_{n-1}) &\leftarrow \boldsymbol{S}_D(t_n)\\
\boldsymbol{S}_L(t_{n-2}) &\leftarrow \boldsymbol{S}_L(t_{n-1})\\
\boldsymbol{S}_L(t_{n-1}) &\leftarrow \boldsymbol{S}_L(t_n)
\end{aligned}
$$

synopsis

(define forms.media.number_of_auxiliary_fields 4)
(set! E (drude_lorentz_1 ϵ_{DL}^{-1} ω_D^2 γ_D ω_{pL}^2 ω_{tL}^2 γ_L))

documentation

guile> (**procedure−documentation drude_lorentz_1**)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 4)
(drude_lorentz_1 one_by_epsilon_DL
omega_p_square_D gamma_D
omega_p_square_L omega_t_square_L gamma_L)

Return the updated value for E that corresponds to these six parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, and forms.S[n], for n in [0..3]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

drude_lorentz_2 A mixed Drude-two-pole-Lorentz medium characterized by the following Drude parameters, ω_D and γ_D; then by the following Lorentz parameters, for the first pole ω_{p1}, ω_{t1} and γ_1, and for the second pole ω_{p2}, ω_{t2} and γ_2; and by ϵ_∞ here referred to as ϵ_{DL}.

formula

$$\epsilon(\omega) = \epsilon_{DL} - \frac{\omega_D^2}{\omega^2 + \mathrm{i}\gamma_D\omega} - \frac{\omega_{p1}^2}{\omega^2 + \mathrm{i}\gamma_1\omega - \omega_{t1}^2} - \frac{\omega_{p2}^2}{\omega^2 + \mathrm{i}\gamma_2\omega - \omega_{t2}^2}$$

ADE

$$\begin{aligned}
\boldsymbol{S}_D(t_n) &= \frac{1}{2+\gamma_D\Delta t}\left(4\boldsymbol{S}_D(t_{n-1}) - (2-\gamma_D\Delta t)\boldsymbol{S}_D(t_{n-2}) - 2\omega_D^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_1(t_n) &= \frac{1}{2+\gamma_1\Delta t}\left((4-2\omega_{t1}^2\Delta t^2)\boldsymbol{S}_1(t_{n-1}) - (2-\gamma_1\Delta t)\boldsymbol{S}_1(t_{n-2}) - 2\omega_{p1}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_2(t_n) &= \frac{1}{2+\gamma_2\Delta t}\left((4-2\omega_{t2}^2\Delta t^2)\boldsymbol{S}_2(t_{n-1}) - (2-\gamma_2\Delta t)\boldsymbol{S}_2(t_{n-2}) - 2\omega_{p2}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_{DL}}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}_D(t_n) + \boldsymbol{S}_1(t_n) + \boldsymbol{S}_2(t_n)\right)\\
\boldsymbol{S}_D(t_{n-2}) &\leftarrow \boldsymbol{S}_D(t_{n-1})\\
\boldsymbol{S}_D(t_{n-1}) &\leftarrow \boldsymbol{S}_D(t_n)\\
\boldsymbol{S}_1(t_{n-2}) &\leftarrow \boldsymbol{S}_1(t_{n-1})\\
\boldsymbol{S}_1(t_{n-1}) &\leftarrow \boldsymbol{S}_1(t_n)\\
\boldsymbol{S}_2(t_{n-2}) &\leftarrow \boldsymbol{S}_2(t_{n-1})\\
\boldsymbol{S}_2(t_{n-1}) &\leftarrow \boldsymbol{S}_2(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 6)
(set! E (drude_lorentz_2 ϵ_{DL}^{-1} ω_D^2 γ_D ω_{p1}^2 ω_{t1}^2 γ_1 ω_{p2}^2 ω_{t2}^2 γ_2))

documentation

guile> (**procedure−documentation drude_lorentz_2**)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 6)
(drude_lorentz_1 one_by_epsilon_DL
omega_p_square_D gamma_D
omega_p_square_1 omega_t_square_1 gamma_1
omega_p_square_2 omega_t_square_2 gamma_2)

Return the updated value for E that corresponds to these nine parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, and forms.S[n], for n in [0..5]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

2.7.5 Measurements

Synopsis:

guile> (**load−from−path** "measurements")

The virtual measurements library utilizes the Forms built-in Scheme function `interpolate`—see Sections 2.2.7 above and 3.5.4, page 183, also Section 8.13, page 295, module ⟨Interpolation 198⟩, and, finally, Section 2.7.5, page 116—and then wraps higher level utilities around it.

The built-in primitive's synopsis is:
Synopsis:

(**interpolate** name x y z)

where `name` is one of `"Ex"`, `"Ey"`, `"Ez"`, `"Dx"`, `"Dy"`, `"Dz"`, `"Hx"`, `"Hy"`, `"Hz"`, `"Bx"`, `"By"`, `"Bz"`, and `x`, `y`, and `z` are the x, y and z coordinates of a point onto which the fields are to be interpolated. The function returns a linear, volume weighted interpolation of a named field at (x, y, z) from the eight field mounting points that define the cell to which (x, y, z) belongs. The function knows how the fields are mounted (face versus edge), and takes this into account.

The interpolation takes place at the time point in the integration at which it is invoked. This point always coincides with $\boldsymbol{E}$ and $\boldsymbol{D}$, but not with $\boldsymbol{H}$ and $\boldsymbol{B}$. Consequently, the $\boldsymbol{H}$ and $\boldsymbol{B}$ fields are additionally interpolated in time.

Function `interpolate` is defined only within the Forms run. It cannot be invoked in stand-alone Guile. It is further generalized to a method, which is described in Section 2.7.5 (Fluxes), page 116.

Surface

The library file `measurements.scm` defines class `<surface>`, that provides general means for defining a *single* surface coordinate patch. If a given surface requires multiple coverings, multiple instances of class `<surface>` must be created. The slots of the class are as follows:

equation This is a lambda of two real arguments that returns a `3D-vector`. It corresponds to an equation that describes a surface, cf. Section 3.5.2 (Fluxes), page 169, where we describe a surface in terms of three equations:

$$x = x_S(u, v), \tag{72}$$
$$y = y_S(u, v), \tag{73}$$
$$z = z_S(u, v). \tag{74}$$

The lambda represents the equations—all three.

accessor `equation`

init-keyword `#:equation`

default (**lambda** (u v) (**make** <3D−vector> #:x u #:y v))

The surface specified by the lambda is then covered with a discrete grid of points beginning with (u_0, v_0) and ending at (u_f, v_f) with increments of $(\Delta u, \Delta v)$, by providing the following:

initial-values This is a two element vector of reals that corresponds to u_0 and v_0 of the coordinate patch.

accessor `initial-values`

init-keyword `#:initial-values`

default (**make−vector** 2 0)

final-values This is a two element vector of reals that corresponds to u_f and v_f of the coordinate patch.

accessor `final-values`

init-keyword `#:final-values`

default (**make−vector** 2 10)

increments This is a two element vector of reals that corresponds to Δu and Δv of the coordinate patch.

accessor `increments`

init-keyword `#:increments`

default (**make−vector** 2 1)

restriction A predicate that tells if a given pair (u, v) belongs to the domain of the surface.

accessor `restriction`

init-keyword `#:restriction`

default (**lambda** (u v) #t)

plaquettes A list of plaquettes covering the surface. Class `<plaquette>` is discussed below. Also see a section about Surface Iterator, page 108, which is an external method that generates a list of plaquettes for a given surface.

accessor `plaquettes`

init-keyword `#:plaquettes`

default '()

other-data An initially empty list, to be used for any other auxiliary data to do with this surface.

accessor `other-data`

init-keyword `#:other-data`

default '()

Inherent in this definition is that the surface boundary is, topologically, a rectangle that can be mapped onto $[u_{\rm lo}, u_{\rm hi}] \times [v_{\rm lo}, v_{\rm hi}]$. Hemispheres and circles can be mapped this way using angular coordinates, at a cost of having to deal with coordinate system singularities. Alternatively, we can use the `restriction` to clip some of the (u, v) pairs off. This way, we can define, for example, a disk without having to resort to angular coordinates and thus avoiding the singularity. To do so, we map a disk of radius r onto Cartesian coordinates (x, y) and impose a `restriction` of $x^2 + y^2 \leq r^2$.

Plaquette

A `<plaquette>` is a class that contains the following three slots:

center A `3D-vector` that describes a center of the plaquette, $\boldsymbol{r}_p$.

accessor `center`

init-keyword `#:center`

default e_0

parameters A pair of two reals (u . v), that provide parametric representation of the plaquette's center.

accessor `parameters`

init-keyword `#:parameters`

default (0 . 0)

surface-element A `3D-vector` that corresponds to the plaquette's surface element, $\boldsymbol{n}_p\,\mathrm{d}^2S$. If either $\boldsymbol{n}_p$ or d^S are needed separately they can be extracted by calling `norm` (for d^2S) or `normalize` (for $\boldsymbol{n}_p$) on this slot.

accessor `surface-element`

init-keyword `#:surface-element`

default e_0

other-data A list of any other data the user may wish to associate with the plaquette. For example, it may be a list of triads comprising a frequency ω_k and Fourier transforms $\hat{\boldsymbol{E}}_p(\omega_k)$ and $\hat{\boldsymbol{H}}_p(\omega_k)$ for the plaquette for some number of frequencies, $k = 1 \ldots n$. Or it may be a list comprising $\boldsymbol{E}_p \cdot \boldsymbol{n}_p\,\mathrm{d}^2S$ and $\boldsymbol{B}_p \cdot \boldsymbol{n}_p\,\mathrm{d}^2S$ for the plaquette. The list could then be dumped on a file, thus creating a map of flux through the designated surface at some time slice.

accessor `other-data`

init-keyword `#:other-data`

default '()

Surface Iterator

The `iterator` is a method that takes an instance of class `<surface>` as its argument and initializes its list of plaquettes, generating $\boldsymbol{r}_p(u_c, v_c)$ for the plaquette's center and , $\boldsymbol{n}_p\,\mathrm{d}^2S$ for the plaquette's surface element. The method leaves the `other-data` slot alone.
Synopsis:

guile> (**define** s (**make** <surface>))
guile> (**iterator** s)

The plaquette's surface is evaluated using Coolidge's formula rather than a cross product, because the latter can introduce a systematic error visible especially in the computation of fluxes. The normal vector is evaluated by taking a cross product of the plaquette's diagonals.

Arbitrary iterative operations over the surface, such as computation of a flux, can be then implemented by employing a `do` and crawling over the list with a `car/cdr` combination.

This is the fastest method to process a list, sic! To access list elements by index may easily take ten times longer. ◀

Each plaquette object returned can be then accessed with the appropriate accessor function, `center` for $\boldsymbol{r}_p$, `surface-element` for $\boldsymbol{n}_p\,\mathrm{d}^2S$ and `other-data` for other data associated with the plaquette.

The library provides convenience methods that make variously oriented hemispheres, rectangular planes, or disks. The methods invoke the `iterator` internally so that the `plaquettes` slot of the returned objects are fully initialized.

If an object itself is modified, for example, by altering its inital or final values or its increments with `slot-set!`, the `plaquettes` slot has to be recomputed by invoking the `iterator` explicitly.

Surface Area

Method `surface-area`, defined on `measurements.scm`, provides a simple example of `<surface>` operations. It looks as follows:

```
(define−method (surface−area (surface <surface>))
  (do ((plaquettes (plaquettes surface) (cdr plaquettes))
       (area 0))
      ((null? plaquettes) area)
    (set! area (+ area (norm (surface−element (car plaquettes)))))))
```

In the following example, we create a hemisphere of radius 1, then evaluate its surface area, while refining the covering mesh.

```
guile> (set! %load−path (cons "/home/gustav/src/Forms/lib" %load−path))
guile> (load−from−path "measurements")
guile> (define hemisphere (make−hemisphere 1))
guile> (surface−area hemisphere)
6.26365782187649
```

The hemisphere is created so that the caps that surround the north and south poles are excluded, so as to avoid singularities of the coordinate system. The caps cover the area from the pole to $\theta_c = 5°$ in the north and from $\theta_c = 175°$ to the pole in the south. The surface area of a north cap is given by

$$S_c = 2\pi r^2 \left(1 - \cos\theta_c\right) \tag{75}$$

It is easy to see that the expression has a correct behaviour for $\theta_c = 0$, in which case $S_c = 0$ too, and for $\theta_c = \pi/2$, in which case $S_c = 2\pi r^2$. For $\theta_c = 5°$ and $r = 1$, $S_c = 0.0239094170393268$. But for a hemisphere only half of a cap is taken away in the north and another half in the south, so the calculated surface of the hemisphere with two half-caps cut out should be $2\pi - S_c = 6.25927589014026$, and this is what `surface-area` should have returned.

The error in the computed result is 0.07%. We can do better by reducing the increments, for example

```
guile> (slot−set! hemisphere 'increments (vector (/ degree 8) (/ degree 8)))
guile> (iterator hemisphere)
guile> (surface−area hemisphere)
6.25986704836324
```

This time the error is 0.01%. We have reduced the error 7 times, but not without some computational cost.

The hemisphere in the example is oriented so that it points in the x direction. It is easy to see this by evaluating

```
guile> ((equation hemisphere) 0 half_pi)
<1.0, 0.0, 6.12323399573677e−17>
```

The following illustrates a simple use of the restriction predicate to define a disk without a coordinate system singularity:

```
guile> (define disk (make <surface>
... #:equation (lambda (x y) (make <3D−vector> #:x x #:y y #:z 0))
... #:initial−values (vector −1 −1)
... #:final−values (vector 1 1)
... #:increments (vector 0.01 0.01)
... #:restriction (lambda (x y) (<= (+ (* x x) (* y y)) 1.0))))
guile> (describe disk)
#<<surface> 7f864560> is an instance of class <surface>
Slots are:
equation = #<procedure #f (x y)>
initial−values = #(−1 −1)
final−values = #(1 1)
increments = #(0.01 0.01)
restriction = #<procedure #f (x y)>
plaquettes = ()
```

It is not a good idea to invoke `describe` on a fully defined `<surface>`, because the procedure would list all plaquettes. But in this case, `make` has initialized `plaquettes` to `'()`, so we're OK.

Now we initialize the plaquettes and then compute the surface area:

guile> (**iterator** disk)
guile> (**surface−area** disk)
3.14280000000221

The relative error in this computation is about 0.04%.

Surface Construction Methods

In the following we list utility methods that generate certain predefined surfaces.

make-plane Makes a plane, parametrized by u and v, following the equation

$$\boldsymbol{r}(u,v) = \boldsymbol{r}_0 + u\boldsymbol{e}_u + v\boldsymbol{e}_v, \tag{76}$$

where $\boldsymbol{r}_0$, $\boldsymbol{e}_u$ and $\boldsymbol{e}_v$ are arbitrary vectors. $\boldsymbol{e}_u$ and $\boldsymbol{e}_v$ do not have to be perpendicular, but must not be parallel. The method calls `iterator` to initialize the object's list of plaquettes.

synopsis

(**make−plane** (e_u <3D−vector>) (e_v <3D−vector>))
(**make−plane** (r_0 <3D−vector>) (e_u <3D−vector>) (e_v <3D−vector>))

In the first case, $\boldsymbol{r}_0$ is assumed to be <0, 0, 0>. In both cases $u \in [-100, 100]$ and $v \in [-100, 100]$ and the increments, Δu and Δv, are both 1.

$\boldsymbol{r}_0$ can be chosen at will to position the plane where it needs to be, and $\boldsymbol{e}_u$ and $\boldsymbol{e}_v$ can be chosen to scale the plane's dimensions as needed, and to orient it arbitrarily.

example Generate an $\boldsymbol{e}_x \wedge \boldsymbol{e}_y$ plane centered at $[0, 0, 0]$. Then evaluate its surface area.

guile> (**define** plane (**make−plane** e_x e_y))
guile> (**surface−area** plane)
40000.0
guile>

We can easily make a plane with a surface area of 4 by requesting

guile> (**define** plane (**make−plane** (/ e_x 100) (/ e_y 100)))
guile> (**surface−area** plane)
4.00000000000402
guile>

The specific parametrization used, $(u, v) \in [-100, 100] \times [-100, 100]$ is not as restrictive as it seems—and, of course, the user may change it manually, if desired.

make-hemisphere Makes a hemisphere, of radius r, and centre at $\boldsymbol{r}_0$, parametrized by ϕ and θ with a 5° cap cut out around the $\theta = 0$ and $\theta = \pi$ singularities. The two coordinates ϕ and θ are variously used depending on the requested hemisphere orientation. The method calls `iterator` to generate a list of plaquettes for the surface.

synopsis

(**make−hemisphere** (r <real>))
(**make−hemisphere** (r_0 <3D−vector>) (r <real>))
(**make−hemisphere** (r <real>) (direction <symbol>))
(**make−hemisphere** (r_0 <3D−vector>) (r <real>) (direction <symbol>))

where `direction` may be one of

'forward or 'ahead or 'x in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\sin\theta\cos\phi\,\boldsymbol{e}_x + \sin\theta\sin\phi\,\boldsymbol{e}_y + \cos\theta\,\boldsymbol{e}_z) \tag{77}$$

'backward or 'back or 'minus-x in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(-\sin\theta\cos\phi\,\boldsymbol{e}_x - \sin\theta\sin\phi\,\boldsymbol{e}_y + \cos\theta\,\boldsymbol{e}_z) \tag{78}$$

'left or 'y in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\sin\theta\sin\phi\,\boldsymbol{e}_x + \sin\theta\cos\phi\,\boldsymbol{e}_y + \cos\theta\,\boldsymbol{e}_z) \tag{79}$$

'right or 'minus-y in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(-\sin\theta\sin\phi\,\boldsymbol{e}_x - \sin\theta\cos\phi\,\boldsymbol{e}_y + \cos\theta\,\boldsymbol{e}_z) \tag{80}$$

'upward or 'up or 'z in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\cos\theta\,\boldsymbol{e}_x + \sin\theta\sin\phi\,\boldsymbol{e}_y + \sin\theta\cos\phi\,\boldsymbol{e}_z) \tag{81}$$

'downward or 'down or 'minus-z in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\cos\theta\,\boldsymbol{e}_x - \sin\theta\sin\phi\,\boldsymbol{e}_y - \sin\theta\cos\phi\,\boldsymbol{e}_z) \tag{82}$$

In all cases listed above $\phi \in [-\pi/2, \pi/2]$ and $\theta \in [5°, 175°]$. The increments, $\Delta\phi$ and $\Delta\theta$, are both 1°. The reason for this solution is that it returns the best computational accuracy for all hemisphere orientations. If $\boldsymbol{r}_0$ is not specified, it is assumed <0, 0, 0>.

example Generate a hemisphere of radius 1, oriented towards y, and centered on $[1, 0, 0]$. Then evaluate its surface area.

```
guile> (define hemisphere (make-hemisphere e_x 1 'left))
guile> (surface-area hemisphere)
6.2636578218765
guile>
```

make-sphere Makes a sphere, of radius r, and centre at $\boldsymbol{r}_0$, parametrized by ϕ and θ with a 5° cap cut out around the $\theta = 0$ and $\theta = \pi$ singularities. The two coordinates ϕ and θ are variously used depending on where the cut-outs are requested. The method calls `iterator` to generate a list of plaquettes for the surface.

synopsis

```
(make-sphere (r <real>))
(make-sphere (r_0 <3D-vector>) (r <real>))
(make-sphere (r <real>) (cutouts <symbol>))
(make-sphere (r_0 <3D-vector>) (r <real>) (cutouts <symbol>))
```

where `cutouts` may be one of

'vertical or 'z in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\sin\theta\cos\phi\,\boldsymbol{e}_x + \sin\theta\sin\phi\,\boldsymbol{e}_y + \cos\theta\,\boldsymbol{e}_z) \tag{83}$$

'longitudinal or 'x in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\cos\theta\,\boldsymbol{e}_x + \sin\theta\cos\phi\,\boldsymbol{e}_y + \sin\theta\sin\phi\,\boldsymbol{e}_z) \tag{84}$$

'transverse or 'y in which case the equation is

$$\boldsymbol{r}(\phi,\theta) = \boldsymbol{r}_0 + r\,(\sin\theta\sin\phi\,\boldsymbol{e}_x + \cos\theta\,\boldsymbol{e}_y + \sin\theta\cos\phi\,\boldsymbol{e}_z) \tag{85}$$

In all cases listed above $\phi \in [-\pi, \pi]$ and $\theta \in [5°, 175°]$. The increments, $\Delta\phi$ and $\Delta\theta$, are both 1°.

If $\boldsymbol{r}_0$ is not specified, it is assumed <0, 0, 0>. If `cutouts` is not specified it is assumed `'z`.

example Make a sphere of radius 1 and centre at $[0, 0, 0]$. Change the increments to $\Delta\phi = \Delta\theta = 1°/4$, and inspect the object. Then evaluate its surface area. Compare with the exact result, and find a relative error.

```
guile> (define sphere (make−sphere 1))
guile> (slot−set! sphere 'increments (vector (/ degree 4) (/ degree 4)))
guile> (slot−set! sphere 'plaquettes '())
guile> (describe sphere)
#<<surface> 707a1490> is an instance of class <surface>
Slots are:
equation = #<procedure #f (phi theta)>
initial−values = #(−3.14159265358979 0.0872664625997165)
final−values = #(3.14159265358979 3.05432619099008)
increments = #(0.00436332312998582 0.00436332312998582)
restriction = #<procedure #f (u v)>
plaquettes = ()
guile> (iterator sphere)
guile> (surface−area sphere)
12.5185022025355
guile> (define cap−area (∗ 2 pi (− 1 (cos (∗ 5 degree)))))
guile> (− (∗ 4 pi) (∗ 2 cap−area))
12.5185517802805
guile>
```

We find that the surface area of the sphere with two caps cut out is evaluated with a relative error of $\approx 0.0004\%$.

make-disk Makes a disk of radius r, centered at $\boldsymbol{r}_0$, the normal of which is $\boldsymbol{n}$, parametrized by x and y of the disk's plane. The disk itself is further characterized by a restriction. The method calls `iterator` to generate a list of plaquettes for the disk.

synopsis

```
(make−disk (r_0 <3D−vector>) (n <3D−vector>) (r <real>))
(make−disk (n <3D−vector>) (r <real>))
(make−disk (r_0 <3D−vector>) (n <3D−vector>))
(make−disk (n <3D−vector>))
```

where $\boldsymbol{r}_0$ and r are as above and $\boldsymbol{n}$ is a normal vector, but it does not have to be normalized. The method normalizes it internally anyway. The plaquette vector will point in the same direction as $\boldsymbol{n}$. If $\boldsymbol{r}_0$ is omitted, it is assumed $[0, 0, 0]$. If r is omitted, it is assumed 1. The Δx and Δy of the disk are $r/100$ by default. If changed by `slot-set! iterator` must be invoked explicitly.

example Make a disk of radius 1 and normal $\boldsymbol{e}_y$. Evaluate its surface area. Inspect its leading plaquette.

```
guile> (define disk (make−disk (3D−vector 0 1 0)))
guile> (define disk−area (surface−area disk))
guile> disk−area
3.14280000000221
guile> (/ (− disk−area pi) pi)
3.84310299120135e−4
guile> (car (plaquettes disk))
#<<plaquette> 6b51b670>
guile> (center (car (plaquettes disk)))
<−0.995000000000001, 0.0, 0.0950000000000007>
guile> (surface−element (car (plaquettes disk)))
<0.0, 9.9999999999999e−5, −0.0>
guile> (other−data (car (plaquettes disk)))
()
guile>
```

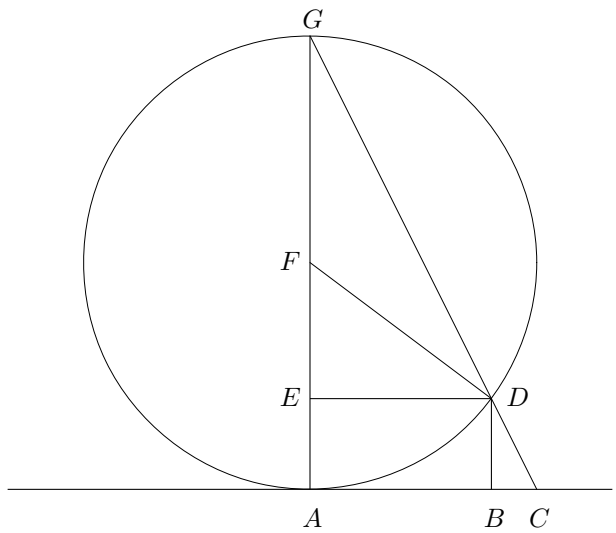


Figure 10: A projection that maps (x, y) coordinates of the plane onto a cap to which the plane is tangential—used by `make-cap`. The figure looks at the projection line from the side, meaning that $\overline{AC} = \sqrt{x_C^2 + y_C^2}$.

The disk area should be π. The error in its evaluation is 0.04%. The plaquette's $\boldsymbol{n}\,\mathrm{d}^2S$ is $0.0001\boldsymbol{e}_y$—up to the machine accuracy.

make-cap The mapping used in the definition of a cap is shown in Figure 10. It is a projection well known in mathematics, as it lets us cover the whole sphere, but its north pole, with a single coordinate patch.

Knowing the (x_C, y_C) coordinates of point C in the plane, we can find the corresponding coordinates (x_D, y_D, z_D) of point D on the sphere as follows. Let

$$\begin{aligned}
\overline{AC} &= r_{xy} = \sqrt{x_C^2 + y_C^2}, && (86)\\
\overline{FD} &= R, && (87)\\
\angle EFD &= \theta, && (88)\\
\angle AGC &= \theta/2, \quad \text{(from the inscribed angle theorem)} && (89)\\
\tan(\theta/2) &= r_{xy}/(2R), && (90)\\
\overline{ED} &= \rho = R\sin\theta, && (91)\\
\overline{EF} &= h = R\cos\theta, && (92)\\
\overline{AE} &= \overline{BD} = z_D = R - h. && (93)
\end{aligned}$$

Wherefrom we find,

$$z_D = R - R\cos\theta = R(1 - \cos\theta). \tag{94}$$

But

$$\cos\theta = \cos 2\frac{\theta}{2} = 1 - 2\sin^2\frac{\theta}{2}. \tag{95}$$

Hence

$$z_D = 2R\sin^2\frac{\theta}{2}. \tag{96}$$

From Figure 10 we find that

$$\sin\frac{\theta}{2} = \frac{\overline{AC}}{\overline{GC}} = \frac{\overline{AC}}{\sqrt{\overline{AC}^2 + \overline{GA}^2}} = \frac{r_{xy}}{\sqrt{r_{xy}^2 + 4R^2}}. \tag{97}$$

Therfore

$$z_D = \frac{2Rr_{xy}^2}{r_{xy}^2 + 4R^2}. \tag{98}$$

It is easy to test this expression. For $r_{xy} = 0$ we get that $z_D = 0$, too, which is correct. For $r_{xy} = 2R$, which corresponds to the situation when D is on the equator, we get that

$$z_D = \frac{2R4R^2}{4R^2 + 4R^2} = R\frac{8R^2}{8R^2} = R, \tag{99}$$

which is, again, correct. Finally, for $r_{xy} = \infty$, which corresponds to point G itself, we find that $z_D = 2R$, which is correct, too.

We can evaluate x_D and y_D similarly. From Figure 10

$$x_D = \rho\cos\phi, \tag{100}$$

$$y_D = \rho\sin\phi, \tag{101}$$

where

$$\rho = R\sin\theta. \tag{102}$$

But

$$\sin\theta = \sin 2\frac{\theta}{2} = 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}. \tag{103}$$

Reasoning as above, we find that

$$\cos\frac{\theta}{2} = \frac{\overline{AG}}{\overline{GC}} = \frac{\overline{AG}}{\sqrt{\overline{AC}^2 + \overline{AG}^2}} = \frac{2R}{\sqrt{r_{xy}^2 + 4R^2}}. \tag{104}$$

And so, combining equations (97), (104), (103), and (102) yields

$$\rho = 2R\sin\frac{\theta}{2}\cos\frac{\theta}{2} = \frac{r_{xy}4R^2}{r_{xy}^2 + 4R^2}. \tag{105}$$

This is also easy to test. For $r_{xy} = 0$ we find that $\rho = 0$, which is correct. For $r_{xy} = 2R$ we find that $\rho = 8R^3/(8R^2) = R$, which is, again, correct. Finally, for $r_{xy} = \infty$ we find $\rho = 0$, which is correct, too.

Angle ϕ is the same as the angle between AC and the major directions in the plane, that is,

$$\cos\phi = \frac{x_C}{r_{xy}}, \tag{106}$$

$$\sin\phi = \frac{y_C}{r_{xy}}. \tag{107}$$

In summary,

$$x_D = \frac{r_{xy}4R^2}{r_{xy}^2 + 4R^2}\frac{x_C}{r_{xy}} = \frac{x_C 4R^2}{r_{xy}^2 + 4R^2}, \tag{108}$$

$$y_D = \frac{r_{xy}4R^2}{r_{xy}^2 + 4R^2}\frac{y_C}{r_{xy}} = \frac{y_C 4R^2}{r_{xy}^2 + 4R^2}. \tag{109}$$

synopsis

(**make−cap** (r_0 <3D−vector>) (e_n <3D−vector>) (r <real>) (theta−max <real>))
(**make−cap** (e_n <3D−vector>) (r <real>) (theta−max <real>))
(**make−cap** (r <real>) (theta−max <real>))
(**make−cap** (theta−max <real>)

In the first, fully specified, form the method creates a cap with its apex at $\boldsymbol{r}_0$ that points with its convexity, so to speak, in the $\boldsymbol{e}_n$ direction. The radius of the cap is r and its span angle is $\theta_{\max}$. Vector $\boldsymbol{e}_n$ does not have to be normalized: the method normalizes it internally. The resulting equation of the surface is

$$\boldsymbol{r}(x,y) = \boldsymbol{r}_0 + \frac{4xr^2}{r_{xy}^2+4r^2}\,\boldsymbol{e}_1 + \frac{4yr^2}{r_{xy}^2+4r^2}\,\boldsymbol{e}_2 - \frac{2r_{xy}^2 r}{r_{xy}^2+4r^2}\,\boldsymbol{e}_3, \tag{110}$$

where $\boldsymbol{e}_3 = \boldsymbol{e}_n/|\boldsymbol{e}_n|$, and $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are such that $\boldsymbol{e}_1 \times \boldsymbol{e}_2 = \boldsymbol{e}_3$. The x coordinate is measured in the $\boldsymbol{e}_1$ direction and the y coordinate is measured in the $\boldsymbol{e}_2$ direction. The reason there is a minus in front of $\boldsymbol{e}_n$ is because we want the cap surface to curve *away* from $\boldsymbol{e}_n$. The plaquette vector $\boldsymbol{n}\,\mathrm{d}^2S$ at $\boldsymbol{r}_0$ points in the same direction as $\boldsymbol{e}_n$.

If $\boldsymbol{r}_0$ is not specified, the cap is assumed to be a fragment of a sphere centered on $[0,0,0]$. In this case $\boldsymbol{r}_0$ is assumed to be $r\boldsymbol{e}_3$.

If neither $\boldsymbol{r}_0$, nor $\boldsymbol{e}_n$ are specified, $\boldsymbol{e}_n$ is assumed to be $\boldsymbol{e}_x$ and $\boldsymbol{r}_0$ is assumed to be $r\boldsymbol{e}_x$.

If neither $\boldsymbol{r}_0$, nor $\boldsymbol{e}_n$, nor r are specified, then r is assumed to be 1 and $\boldsymbol{r}_0$ and $\boldsymbol{e}_n$ are assumed as above.

The cap's angular extent must always be specified.

example Define a cap as shown in Figure 10, that is, the sphere is to sit on point $A = [0,0,0]$, and the cap is to be tangential to the plane on which the sphere sits. The cap's angular span is to be 10°. Evaluate the cap's surface area and compare with the exact formula. Inspect selected plaquettes: does the normal vector point downwards, along $-\boldsymbol{e}_z$?

```
guile> (define cap (make-cap e_0 (- e_z) 1 (* 10 degree)))
guile> (define cap-area (surface-area cap))
guile> (define exact-area (* 2 pi (- 1 (cos (* 10 degree)))))
guile> cap-area
0.0954920371147699
guile> exact-area
0.0954557030567379
guile> (/ (- cap-area exact-area) exact-area)
3.80637896620848e-4
guile>
```

We find that the evaluation error here is $\approx 0.04\%$.

The $\boldsymbol{n}\,\mathrm{d}^2S$ vector is accessed by `surface-element` applied to a plaquette. We select two plaquettes from the list and inspect their $\boldsymbol{n}\,\mathrm{d}^2S$:

```
guile> (define plaquette_1 (car (plaquettes cap)))
guile> (define plaquette_2 (car (cdr (cdr (cdr (cdr (plaquettes cap)))))))
guile> (surface-element plaquette_1)
<-5.2104908833061e-7, 4.97484054185031e-8, -2.96988845700699e-6>
guile> (surface-element plaquette_2)
<-5.21118757051562e-7, 2.88055594350081e-8, -2.97042301976973e-6>
guile>
```

Indeed, the major component of the vector points down, in the direction of $-\boldsymbol{e}_z$.

We can invoke the cap's lambda to see the mapping explicitly. We should have $(0,0)$ map on $(0,0,0)$ and $(0,2)$ map on $(0,1,1)$:

```
guile> ((equation cap) 0 0)
<0, 0, 0>
guile> ((equation cap) 0 2)
<0.0, 1.0, 1.0>
guile>
```

For a situation shown in Figure 10 and for $r = 1$ we have that $\tan(\theta/2) = 0.5$ and $r_{xy} = 1$. This should then yield

$$\rho = \sin\theta = 2\sin(\theta/2)\cos(\theta/2) = 2\frac{1}{1^2+2^2}\frac{2}{1^2+2^2} = \frac{4}{5} = 0.8, \tag{111}$$

and z_D as prescribed by equation (98), namely

$$z_D = \frac{2rr_{xy}^2}{r_{xy}^2 + 4r^2} = \frac{2 \cdot 1 \cdot 1^2}{1^2 + 4 \cdot 1^1} = \frac{2}{5} = 0.4, \tag{112}$$

and this is exactly what we get:

```
guile> ((equation cap) 0 1)
<0.0, 0.8, 0.4>
guile>
```

Fluxes and Fourier Transforms

compute-flux Evaluate flux of a field or of a cross product of two fields through a surface.

synopsis

(**compute−flux** field surface)
(**compute−flux** field_1 field_2 surface)

With the class `<surface>` defined, and with tools for generation of various surfaces provided, computation of fluxes is as trivial as computation of surface area. Surface area itself is an example of a flux. The following listing illustrates this point.

```
(define−method (compute−flux (field−1 <string>) (field−2 <string>) (surface <surface>))
  (do ((plaquettes (plaquettes surface) (cdr plaquettes))
       (flux 0))
      ((null? plaquettes) flux)
    (let* ((plaquette (car plaquettes)))
      (set! flux (+ flux (* (x (interpolate field−1 (center plaquette))
                               (interpolate field−2 (center plaquette)))
                            (surface−element plaquette)))))))
```

For this to work, the `measurements` library redefines function `interpolate` as a generic, and adds the following specification:

interpolate An overload of the internal Forms function `interpolate` that does the whole vector in one call. The name of the vector field is specified by providing one of `"E"`, `"D"`, `"H"`, or `"B"`, to which the method appends "x", "y" and "z" internally, while calling the primitive `interpolate`. The returned value in this case is a `<3D-vector>`.

synopsis

(**interpolate** (name <**string**>) (point <3D−vector>))

In the listing of `compute-flux`, the interpolated `<3D-vector>` is dot-multiplied (the generic `*` does this) by $\boldsymbol{n}\,\mathrm{d}^2S$ which is returned by (surface−element plaquette), and which is a `<3D-vector>`, too. The result of the operation is then added to `flux`. The method returns the value accumulated in `flux` on return.

When invoked with two field names, instead of one, the method evaluates a flux of the cross-product of the two fields. For example,

(**compute−flux** "E" "B" cap)

computes the flux of $\boldsymbol{E} \times \boldsymbol{B}$ through the `cap`.

Support for Fourier transforms of fields on surfaces is provided by the means of several auxiliary functions and methods. These are:

make-frequencies Generate a list of n equally spaced frequencies beginning with the smaller of ω_0 and ω_1 and ending on the larger one, inclusive.

synopsis

(**make−frequencies** omega_0 omega_1 n)

example

```
guile> (make−frequencies 10 20 11)
(10 11 12 13 14 15 16 17 18 19 20)
```

prep-for-fourier This method fills the `other-data` slot of each plaquette covering the `surface` with a list of n zero 3D-vectors (each enclosed in a list of its own), where n is the number of frequencies in the `frequencies` list, and precedes it with a list of field names. If the list of plaquettes is empty, the function invokes the `iterator` to make it. Frequencies themselves are stored on the surface's `other-data` slot (because they are expected to be the same for each plaquette).

synopsis

(**prep−for−fourier** surface frequencies field−name)
(**prep−for−fourier** surface frequencies field−1−name field−2−name)
(**prep−for−fourier** surface frequencies field−1−name field−2−name field−3−name)
(**prep−for−fourier** surface frequencies field−1−name field−2−name field−3−name field−4−name)

In the second, third and fourth forms, the function fills the `other-data` slot of each plaquette covering the `surface` with a list of n pairs, triplets and quadruplets of zero 3D-vectors correspondingly, and precedes it with a list of field names.

example

```
guile> (define disk (make−disk e_z 1))
guile> (slot−set! disk 'increments (vector (/ 1.0 20.0) (/ 1.0 20.0)))
guile> (slot−set! disk 'plaquettes '())
guile> (define frequencies (make−frequencies 10 20 11))
guile> (prep−for−fourier disk frequencies "E")
guile> (describe (other−data disk))
(10 11 12 13 14 15 16 17 18 19 20) is a list.
guile> (describe (other−data (car (plaquettes disk))))
(("E") ((<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>)
(<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>) (<0, 0, 0>))) is a list.
guile> (slot−set! disk 'plaquettes '())
guile> (prep−for−fourier disk frequencies "E" "H")
guile> (describe (other−data disk))
(10 11 12 13 14 15 16 17 18 19 20) is a list.
guile> (describe (other−data (car (plaquettes disk))))
(("E" "H") ((<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>)
(<
0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0,
0
, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0,
0> <0, 0, 0>))) is a list.
guile>
```

We see here that the `other-data` slot of each plaquette becomes effectively a table with a header, ("E""H"), and with a number of rows, each containing (<0, 0, 0> <0, 0, 0>) in the beginning.

accumulate-fourier This function accumulates Fourier contributions of the form $\boldsymbol{F}\exp(-\mathrm{i}\omega t)\,\mathrm{d}t$ for a surface that has been prepared with **prep−for−fourier**, for all fields $\boldsymbol{F}$ specified as arguments to the latter and for all frequencies in the list passed to the latter. The accumulations are stored in the `other-data` table associated with each plaquette, as prepared by the call to **prep−for−fourier**. The method uses `forms.time_e` for t and `forms.dt` for $\mathrm{d}t$. It can be invoked only from within the FORMS run, because it makes use of `interpolate`, as well.

synopsis

(**accumulate−fourier** surface)

example In order to test this function, we must redefine method `interpolate` after loading `"measurements"`, because the original one refers to an internal function of Forms. Also, we must define `forms.time_e` and `forms.dt`, and in the following we set both to 1. The list of frequencies is set to $(0, \pi/2, \pi, 3\pi/2, 2\pi)$. This way $e^{-\mathrm{i}\pi t}$ evaluates to $(1, -\mathrm{i}, -1, \mathrm{i}, 1)$, which makes test interpretation easier. For the same reason, we define `interpolate` to project its argument vector on $\boldsymbol{e}_x$ if the field is `"E"` and to project its argument vector on $\boldsymbol{e}_y$ if the field is `"H"`.

```
guile> (load−from−path "measurements")
guile> (define−method (interpolate (name <string>) (point <3D−vector>))
... (cond
... ((equal? name "E") (* point e_x e_x))
... ((equal? name "H") (* point e_y e_y))
... ((equal? name "B") (* point e_z e_z))))
guile> (define disk (make−disk e_z 1))
guile> (define dx (/ 1.0 20.0))
guile> (define dy dx)
guile> (define forms.time_e 1)
guile> (define forms.dt 1)
guile> (slot−set! disk 'increments (vector dx dy))
guile> (slot−set! disk 'plaquettes '())
guile> (define frequencies (make−frequencies 0 (* 2 pi) 5))
guile> (prep−for−fourier disk frequencies "E" "H")
guile> (describe (other−data disk))
(0.0 1.5707963267949 3.14159265358979 4.71238898038469 6.28318530717959) is a
list.
guile> (describe (car (plaquettes disk)))
#<<plaquette> 7e2089c0> is an instance of class <plaquette>
Slots are:
... center = <0.975000000000001, 0.175, 0.0>
... surface−element = <0.0, −0.0, 0.0025>
... other−data = (("E" "H")
... ((<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>)
... (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0, 0, 0>) (<0, 0, 0> <0,
0, 0>)))
guile> (accumulate−fourier disk)
guile> (describe (other−data (car (plaquettes disk))))
(("E" "H")
... ((<0.975000000000001, 0, 0> <0, 0.175, 0>)
... (<5.97015314584335e−17−0.975000000000001i, 0, 0> <0,
1.07156594925394e−17−0.175i, 0>)
... (<−0.975000000000001−1.19403062916867e−16i, 0, 0> <0,
−0.175−2.14313189850787e−17i, 0>)
... (<−1.79104594375301e−16+0.975000000000001i, 0, 0> <0,
−3.21469784776181e−17+0.175i, 0>)
... (<0.975000000000001+2.38806125833734e−16i, 0, 0> <0,
0.175+4.28626379701574e−17i, 0>)))
is a list.
guile>
```

Translating the above to normal notation we find that after one application of `accumulate-fourier`, the fields stored in the table have been turned into

ω	$\boldsymbol{E}$	$\boldsymbol{H}$
$0.0\,\pi$	$0.975\,\boldsymbol{e}_x$	$0.175\,\boldsymbol{e}_y$
$0.5\,\pi$	$-0.975\,\mathrm{i}\,\boldsymbol{e}_x$	$-0.175\,\mathrm{i}\,\boldsymbol{e}_y$
$1.0\,\pi$	$-0.975\,\boldsymbol{e}_x$	$-0.175\,\boldsymbol{e}_y$
$1.5\,\pi$	$0.975\,\mathrm{i}\,\boldsymbol{e}_x$	$0.175\,\mathrm{i}\,\boldsymbol{e}_y$
$2.0\,\pi$	$0.975\,\boldsymbol{e}_x$	$0.175\,\boldsymbol{e}_y$

for $\boldsymbol{r} = (0.975, 0.175, 0)$—exactly as we would have expected, within the bounds of the machine accuracy.

dump-frequency-power-flux-map This function dumps a map of $P_p(\omega) = \boldsymbol{n}_p \cdot \frac{1}{2}\Re\left(\hat{\boldsymbol{E}}_p(\omega) \times \bar{\hat{\boldsymbol{H}}}_p(\omega)\right)$ using fields $\hat{\boldsymbol{E}}_p(\omega)$ and $\hat{\boldsymbol{H}}_p(\omega)$ accumulated on the surface's plaquettes p by calls to `accumulate-fourier` in the table prepared by `prep-for-fourier` beforehand. The function picks $\hat{\boldsymbol{E}}$ and $\hat{\boldsymbol{H}}$ from the plaquette's table, regardless of the order in which they are written and other fields stored in it. If it can't find one or the other, it does nothing. The data is written out in human readable format, ready for Gnuplot's function `splot` in columns, which correspond to

$$u_p \quad v_p \quad P_p(\omega_1) \quad P_p(\omega_2) \quad \ldots \quad P_p(\omega_n)$$

Whenever u_p changes, an empty row is inserted before the new u_p—Gnuplot's `splot` expects this.

synopsis

(dump−frequency−power−flux−map surface file−name)
(dump−frequency−power−flux−map surface file−name uv−factor)
(dump−frequency−power−flux−map surface file−name u−factor v−factor flux−factor)

In its first form, the function dumps the content using such u, v and field values as it finds. In the second form, u and v are multiplied by the uv−factor provided. This can be used, for example, to convert u and v from radians to degrees. In the third form, u is multiplied by u−factor, v is multiplied by v−factor and the output field is multiplied by flux−factor. This can be used to change, for example, the sign of flux, if $\boldsymbol{n}_p$ points the wrong way, or modify its units, or perform additional scaling related to the duration of accumulation and type of signal.

frequency-power-flux This function evaluates the total flux of $\frac{1}{2}\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right)$ through the surface, assuming that both $\hat{\boldsymbol{E}}(\omega)$ and $\hat{\boldsymbol{H}}(\omega)$ have been accumulated by calls to `accumulate-fourier` in the table prepared by `prep-for-fourier` beforehand. The intention is for this function to be called from `forms.post.all.lambda` after all iterations have completed. The function picks $\boldsymbol{E}$ and $\boldsymbol{H}$ from the plaquette table, regardless of the order in which they are written and other fields stored in it. If it can't find one or the other, it does nothing and returns nothing. Also, it checks if the surface has been prepared for Fourier accumulation in the first place, and does nothing and returns nothing if it hasn't. If the computation completes, the resulting fluxes, in function of ω, are returned in a list ordered the same way as as the frequencies list.

synopsis

(**frequency−power−flux** surface)

example This function is hard to test on a single step operation. In this example we define `forms.time_e` and `forms.dt` to be one, as before, but we make the method `interpolate` even simpler. It returns $\boldsymbol{e}_x$ for $\boldsymbol{E}$ and $\boldsymbol{e}_y$ for $\boldsymbol{H}$. This way $\boldsymbol{E} \times \boldsymbol{H} = \boldsymbol{e}_z$, which is parallel to the direction of the plaquette. The complex exponent factors all cancel, because we multiply $\hat{\boldsymbol{E}}$ by $\bar{\hat{\boldsymbol{H}}}$, which should return $\boldsymbol{e}_z$ for every plaquette and for every frequency. In effect, plaquette fluxes should simply add up to half the surface area of the disk, which is indeed what we see.

```
guile> (load−from−path "measurements")
guile> (define−method (interpolate (name <string>) (point (3D−vector>)))
... (cond
... ((equal? name "E") e_x)
... ((equal? name "H") e_y)))
guile> (define disk (make−disk e_z 1))
guile> (define dx (/ 1.0 20.0))
guile> (define dy dx)
guile> (define forms.time_e 1)
guile> (define forms.dt 1)
```

```
guile> (slot−set! disk 'increments (vector dx dy))
guile> (slot−set! disk 'plaquettes '())
guile> (define frequencies (make−frequencies 0 (∗ 2 pi) 9))
guile> (prep−for−fourier disk frequencies "E" "H")
guile> (accumulate−fourier disk)
guile> (frequency−power−flux disk)
(1.57999999999997 1.57999999999997 1.57999999999997 1.57999999999997
1.57999999999997 1.57999999999997 1.57999999999997 1.57999999999997
1.57999999999997)
guile> (/ (surface−area disk) 2.0)
1.57999999999997
```

prepare-power-flux-map This and the following functions automate the computation of a power flux map through an arbitrary surface. Specifically, what's computed here is

$$\frac{1}{T}\int_0^T \boldsymbol{n}_p \cdot (\boldsymbol{E}(\boldsymbol{r}_p, t) \times \boldsymbol{H}(\boldsymbol{r}_p, t))\, \mathrm{d}t, \tag{113}$$

for the centerpoint $\boldsymbol{r}_p$ and the normal vector $\boldsymbol{n}_p$ of every plaquette. Conceptually the functions are similar to the Fourier ones, but simpler, and more specialized.

synopsis

(**prepare−power−flux−map** surface)

For every plaquette of the surface compute its normal vector $\boldsymbol{n}_p$ (the plaquette stores $\boldsymbol{n}\,\mathrm{d}^2S$, not just the plain $\boldsymbol{n}$ in its surface−element slot), and place it as the first member of a pair in the plaquette's other−data slot. Initialize the second member to zero.

example

```
guile> (load−from−path "measurements")
guile> (define sphere (make−sphere 10 'z))
guile> (prepare−power−flux−map sphere)
guile> (describe (car (plaquettes sphere)))
#<<plaquette> 7dcbd7c0> is an instance of class <plaquette>
Slots are:
center = <−0.958421030076655, 0.00836401361738185, −9.95396198367178>
parameters = (3.1328660073298 . 3.0455995447301)
surface−element = <2.79805585669897e−4, −2.44182635326419e−6,
0.0029058921124222>
other−data = (<0.0958457189924237, −8.36432917959557e−4, 0.99539585016756> .
0.0)
guile>
```

accumulate-power-flux-map This function calculates $\boldsymbol{n}_p \cdot (\boldsymbol{E}(\boldsymbol{r}_p, t) \times \boldsymbol{H}(\boldsymbol{r}_p, t))\, \mathrm{d}t$ for every plaquette of the surface, which must be prepared for the operation by calling **prepare−power−flux−map** beforehand (the function checks if this has been done), and adds it to the number stored in the second slot of the pair that lives in the other−data slot of the plaquette.

synopsis

(**accumulate−power−flux−map** surface)

example

```
(define sphere (make−sphere 10 'z))
(prepare−power−flux−map sphere)
(accumulate−power−flux−map sphere)
```

Because this functions interpolates $\boldsymbol{E}$ and $\boldsymbol{H}$ by going to Forms Chombo data structures, it cannot be called from an interactive Guile session outside of the Forms environemnt.

rescale-power-flux-map This function rescales the values accumulated in the other−data slot of the plaquettes by multiplying them by the factor provided. This can be used to divide the accumulated data by T, the period, for example, or Δt, the time duration of the accumulation, or to change the units of the accumulated data prior to the output.

synopsis

(**rescale−power−flux−map** surface factor)

example

(**let** ((delta−t (− output_time_to output_time_from)))
(**rescale−power−flux−map** sphere (/ delta−t)))

dump-power-flux-map This function writes the content of the accumulated power map on a human and Gnuplot readable file.

synopsis

(**dump−power−flux−map** surface file−name)
(**dump−power−flux−map** surface file−name uv−factor)
(**dump−power−flux−map** surface file−name u−factor v−factor flux−factor)

In the first form, data is written on the output file (file−name is a string) as is: for each plaquette its u and v parameters are written first, followed by the accumulated power flux value. An empty line is issued after every row of data is completed, which is as Gnuplot wants it for the `splot` function.

In the second form, the u and v parameters are multiplied by the uv−factor provided on output. This can be used, for example, to convert the output units from radians to degrees.

In the third form, u is multiplied by u−factor, v is multiplied by v−factor, and the accumulated power flux is multiplied by the flux−factor provided. This way all three items, for each plaquette, can be output using units different from the ones in which they were computed. Additionally, the sign of the power flux scalar can be changed, too. This may be needed sometimes, if $\boldsymbol{n}_p$ points in the opposite direction to the one we want.

example

(**dump−power−flux−map** sphere "power−map.dat" (/ degree) (/ degree) −1)

reset-power-flux-map This function may be called after the data accumulated for some $[t_1, t_2[$ (note that t_2 is not included in the segment, we normally want it this way) has been rescaled and dumped. We may then change the driving frequency and, after the system has stabilized, we may wish to repeat the operation. For this to work we must clear the previously accumulated data, which is what this function does. It replaces the number in the second slot in the other−data pair with zero.

synopsis

(**reset−power−flux−map** surface)

example

(**define** sphere (**make−sphere** center radius 'z))
(**prepare−power−flux−map** sphere)
(**define** forms.post.iteration.labda
(**lambda** ()
...
(**accumulate−power−flux−map** sphere)
...))
(**rescale−power−flux−map** sphere (/ delta−t))
(**dump−power−flux−map** sphere "power−map.dat" (/ degree) (/ degree) −1)
(**reset−power−flux−map** sphere)
...

2.7.6 GSL—GNU Scientific Library

Selected utilities from the GNU Scientific Library are provided as an extension module to be loaded into Guile either for a stand-alone session or in the Forms input file. GSL-to-Scheme wrapper functions, written in C, are defined on `lib/gsl_wrapper.c`, and the `Makefile` in the `lib` subdirectory compiles it and prepares for loading into Scheme. Comments in the preamble of `lib/gsl_wrapper.c` also discuss briefly how to compile and load the module.

In short, to compile and C-load under Cygwin:

```
GSL_CFLAGS = `gsl-config --cflags`
GSL_LIBS = /usr/lib/libgsl.a /usr/lib/libgslcblas.a /usr/lib/libm.a
GUILE_CFLAGS = `guile-config compile`
GUILE_LIBS = `guile-config link`
libguile_gsl.dll: gsl_wrapper.c
        gcc $(GSL_CFLAGS) $(GUILE_CFLAGS) -shared -o libguile_gsl.dll gsl_wrapper.c $(GSL_LIBS) $(GUILE_LIBS)
```

Then to make the functions available to Guile:

```
guile> (load-extension "libguile_gsl" "init_gsl")
guile> (use-modules (gsl bessel))
guile> (use-modules (gsl legendre))
```

This will work only if `"libguile_gsl.dll"` is in the working directory of the Guile session. Otherwise a full pathname must be provided. For this reason it may be convenient to load the extension from the `~/.guile` file, while making sure that it is there in the first place. For example,

```
(define libguile_gsl "/home/gustav/src/Forms/lib/libguile_gsl")
(define libguile_gsl_file (string-join (list libguile_gsl "dll") "." 'infix))
(if (access? libguile_gsl_file (logior R_OK X_OK))
(load-extension libguile_gsl "init_gsl"))
```

The wrappers implement variable range checking, where that is needed, and print error messages, when the variables are not in range. In this case they return either a `NaN` or an empty list.

The primary use of functions in this extension is in code verification. One of the best methods here is to scatter an incident harmonic wave on a dielectric sphere, and compare the numerical outcome against the Mie's analytical solution, which is expressed in terms of Legendre and Bessel functions. This can be done for all (or selected) points of the computational domain, and the result should not be affected by reflections from media interfaces, edges, and corners, as is the case for our simple Snell's Law test, discussed in Sections 2.6.2, page 62, and 2.8.1, page 128. Well, the reflections are all there, but this time they're taken into account by the Mie's solution.

Because the original GSL function names are, first, long, and, second, differentiated by the means of upper (for cylindrical functions) versus lower (for spherical functions) case (Guile is not case-blind, so this would work), we made a decision (perhaps perillous) to rename them. This should also make them easier to call, but carries a risk of confusion, in case user defined functions are similarly named. The general convention adopted here is that the the cylindrical functions J_n and Y_n are translated into `jc_n` and `yc_n` and the spherical functions j_l and y_l are translated into `js_n` and `ys_n`.

The following lists functions made available at this stage.

Module (gsl bessel)

Regular Cylindrical Bessel Functions Also known as Bessel functions of the first kind:

- `(jc_0 x)` where x is a double; returns a double, $J_0(x)$
- `(jc_1 x)` where x is a double; returns a double, $J_1(x)$
- `(jc_2 x)` where x is a double; returns a double, $J_2(x)$
- `(jc_n n x)` where n is an integer and x is a double; returns a double, $J_n(x)$

Regular Modified Cylindrical Bessel Functions These are the extension of $J_n(x)$ into the complex domain, defined by

$$I_n(x) = \mathrm{i}^{-n} J_n(\mathrm{i}x). \tag{114}$$

`(ic_0 x)` where x is a double; returns a double, $I_0(x)$

`(ic_1 x)` where x is a double; returns a double, $I_1(x)$

`(ic_2 x)` where x is a double; returns a double, $I_2(x)$

`(ic_n n x)` where n is an integer and x is a double, returns a double $I_n(x)$

Irregular Cylindrical Bessel Functions Also known as Bessel functions of the second kind. These are the ones that are singular at $x = 0$.

`(yc_0 x)` where x is a double; returns a double, $Y_0(x)$, for $x > 0$, `NaN` otherwise

`(yc_1 x)` where x is a double; returns a double, $Y_1(x)$, for $x > 0$, `NaN` otherwise

`(yc_2 x)` where x is a double; returns a double, $Y_2(x)$, for $x > 0$, `NaN` otherwise

`(yc_n n x)` where n is an integer and x is a double, returns a double, $Y_n(x)$, for $x > 0$, `NaN` otherwise

Irregular Modified Cylindrical Bessel Functions These are the extension of Hankel functions H_n into the complex domain, namely:

$$H_n^+(x) = J_n(x) + \mathrm{i}Y_n(x), \tag{115}$$

$$H_n^-(x) = J_n(x) - \mathrm{i}Y_n(x), \tag{116}$$

$$K_n(x) = \frac{\pi}{2}\mathrm{i}^{n+1}H_n^+(\mathrm{i}x). \tag{117}$$

`(kc_0 x)` where x is a double; returns a double, $K_0(x)$, for $x > 0$, `NaN` otherwise

`(kc_1 x)` where x is a double; returns a double, $K_1(x)$, for $x > 0$, `NaN` otherwise

`(kc_2 x)` where x is a double; returns a double, $K_2(x)$, for $x > 0$, `NaN` otherwise

`(kc_n n x)` where n is an integer and x is a double; returns a double, $K_n(x)$, for $x > 0$, `NaN` otherwise

Regular Spherical Bessel Functions These are the spherical Bessel functions, commonly denoted by $j_n(x)$ and defined by

$$j_n(x) = \sqrt{\frac{\pi}{2x}}J_{n+1/2}(x). \tag{118}$$

In particular we find that

$$j_0(x) = \frac{\sin x}{x}, \tag{119}$$

$$j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}, \tag{120}$$

$$j_2(x) = \left(\frac{3}{x^2} - 1\right)\frac{\sin x}{x} - \frac{3\cos x}{x^2}, \tag{121}$$

$$j_n(x) = (-x)^n\left(\frac{1}{x}\frac{\mathrm{d}}{\mathrm{d}x}\right)^n\frac{\sin x}{x} \tag{122}$$

The wrappers for these functions have been tweaked to call GSL functions when the argument is a double and to call our own hand-coded functions when it's a complex number. The latter use exact trigonometric definitions for j_0, j_1, and j_2 and a recursive formula for j_n. But the recursive formula is unstable for small z and for large n, and the trigonometric definitions contain divisions by small numbers near zero that may generate overflows. To mitigate these problems definitions for j_0 and j_1 additionally resort to series expansion about $z = 0$ for $|z| \leq 0.1$, and the j_n definition resorts to the ascending series expansion for $|z| \leq 3.0$. This works fairly well for n up 10, but it's nowhere near as sophisticated as the GSL implementation for doubles. The observed numerical agreement is of the order of five to 8 significant digits, which is good enough for this application, though not ideal.

`(js_0 x)` where x is a double *or complex;* returns a double *or complex,* $j_0(x)$

`(js_1 x)` where x is a double *or complex;* returns a double *or complex,* $j_1(x)$

`(js_2 x)` where x is a double *or complex;* returns a double *or complex,* $j_2(x)$

`(js_n n x)` where n is an integer and x is a double *or complex;* returns a double *or complex* $j_n(x)$ for $0 \leq n$ and for $0 \leq x$ (if x is a double), `NaN` otherwise

Scaled Regular Modified Spherical Bessel Functions These are related to the modified cylindrical Bessel functions I_n of fractional order by the formula

$$i_n(x) = \sqrt{\frac{\pi}{2x}} I_{n+1/2}(x). \tag{123}$$

Because they like to explode, the functions return them exponentially squashed, namely:

$$i_{n\text{scaled}} = e^{-|x|} i_n(x). \tag{124}$$

`(is_0_scaled x)` where x is a double; returns a double, $e^{-|x|} i_0(x)$

`(is_1_scaled x)` where x is a double; returns a double, $e^{-|x|} i_1(x)$

`(is_2_scaled x)` where x is a double; returns a double, $e^{-|x|} i_2(x)$

`(is_n_scaled n x)` where n is an integer and x is a double; returns a double, $e^{-|x|} i_n(x)$

Irregular Spherical Bessel Functions These functions are commonly denoted by $y_n(x)$ and are defined by

$$y_n(x) = \sqrt{\frac{\pi}{2x}} Y_{n+1/2}(x) \tag{125}$$

In particular

$$y_0(x) = -\frac{\cos x}{x}, \tag{126}$$

$$y_1(x) = -\frac{\cos x}{x^2} - \frac{\sin x}{x}, \tag{127}$$

$$y_2(x) = \left(-\frac{3}{x^2} + 1\right) \frac{\cos x}{x} - \frac{3 \sin x}{x^2}, \tag{128}$$

$$y_n(x) = -(-x)^n \left(\frac{1}{x} \frac{\mathrm{d}}{\mathrm{d}x}\right)^n \frac{\cos x}{x} \tag{129}$$

The wrappers for these functions have been tweaked to call GSL functions when the argument is a double and to call our own hand-coded functions when it's a complex number. No attempt has been made to make our definitions robust near zero, since these functions are all divergent at $z = 0$ anyway, and in our context are used for the scattered field only. Consequently, we have implemented y_0, y_1 and y_2 using the exact trigonometric formulae, and y_n using the recursive formula, that reduces the computation to y_0 and y_1.

`(ys_0 x)` where x is a double *or complex;* returns a double *or complex,* $y_0(x)$

`(ys_1 x)` where x is a double *or complex;* returns a double *or complex,* $y_1(x)$

`(ys_2 x)` where x is a double *or complex;* returns a double *or complex,* $y_2(x)$

`(ys_n n x)` where n is an integer and x is a double *or complex;* returns a double *or complex* $y_n(x)$ for $0 \le n$ and for $0 \le x$ (if x is a double), `NaN` otherwise

Scaled Regular Modified Spherical Bessel Functions These irregular modified spherical Bessel functions, are related to the irregular modified cylindrical Bessel functions $K_n(x)$ by the following formula:

$$k_n(x) = \sqrt{\frac{\pi}{2x}} K_{n+1/2}(x). \tag{130}$$

Because their values tend to be very small, they are returned exponentially enlarged, that is, multiplied by e^x, as in

$$k_{n\text{scaled}}(x) = e^x k_n(x) \tag{131}$$

`(ks_0_scaled x)` where x is a double; returns a double, $e^x k_0(x)$ for $0 < x$, `NaN` otherwise

`(ks_1_scaled x)` where x is a double; returns a double, $e^x k_1(x)$ for $0 < x$, `NaN` otherwise

`(ks_2_scaled x)` where x is a double; returns a double, $e^x k_2(x)$ for $0 < x$, `NaN` otherwise

`(ks_n_scaled n x)` where n is an integer and x is a double; returns a double, $e^x k_n(x)$ for $0 < x$, `NaN` otherwise

Module (gsl legendre)

Legendre polynomials There are two types of these, deriving from associated Legendre functions P_l^m and Q_l^m on setting $m = 0$. Within the module, we refer to them a `p_n` and `q_n`, namely

`(p_0 x)` where x is a double; returns a double, $P_0(x) = 1$

`(p_1 x)` where x is a double; returns a double, $P_1(x) = x$

`(p_2 x)` where x is a double; returns a double, $P_2(x) = \frac{1}{2}\left(3x^2 - 1\right)$

`(p_3 x)` where x is a double; returns a double, $P_3(x) = \frac{1}{2}\left(5x^3 - 3x\right)$

`(p_n n x)` where n is an integer and x is a double; returns a double, $P_n(x)$, if $0 \leq n$ and $|x| \leq 1$, `NaN` otherwise

`(q_0 x)` where x is a double; returns a double, $Q_0(x)$, if $-1 \leq x$ and $x \neq 1$, `NaN` otherwise

`(q_1 x)` where x is a double; returns a double, $Q_1(x)$, if $-1 \leq x$ and $x \neq 1$, `NaN` otherwise

`(q_2 x)` where x is a double; returns a double, $Q_2(x)$, if $-1 \leq x$ and $x \neq 1$, `NaN` otherwise

`(q_3 x)` where x is a double; returns a double, $Q_3(x)$, if $-1 \leq x$ and $x \neq 1$, `NaN` otherwise

`(q_n n x)` where n is an integer and x is a double; returns a double, $Q_n(x)$, if $0 \leq n$ and $-1 \leq x$ and $x \neq 1$, `NaN` otherwise

Associated Legendre Polynomials and Spherical Harmonics

`(p_nm n m x)` where n is an integer, m is an integer and x is a double; returns a double, $P_n^m(x)$, if $0 \leq m$ and $m \leq n$ and $|x| \leq 1$, `NaN` otherwise

`(sph_p_nm n m x)` where n is an integer, m is an integer and x is a double; returns a double,

$$Y_n^m(x) = \sqrt{\frac{2n+1}{4\pi}}\sqrt{\frac{(n-m)!}{(n+m)!}}P_n^m(x), \tag{132}$$

if $0 \leq m$ and $m \leq n$ and $|x| \leq 1$, `NaN` otherwise; observe that there is no $e^{\mathrm{i}m\phi}$ factor in the return, so these are incomplete spherical harmonics

List returns These functions return lists of pairs, each pair is made of a Legendre polynomial and its derivatives, namely

`(p_n_list n x)` where n is an integer and x is a double; returns a list of pairs

$$\left(\left(P_0(x), \frac{\mathrm{d}}{\mathrm{d}x}P_0(x)\right)\left(P_1(x), \frac{\mathrm{d}}{\mathrm{d}x}P_1(x)\right)\ldots\left(P_n(x), \frac{\mathrm{d}}{\mathrm{d}x}P_n(x)\right)\right) \tag{133}$$

if $0 \leq n$ and $|x| \leq 1$, and an empty list otherwise

`(p_nm_list n m x)` where n is an integer, m is and integer and x is a double; returns a list of pairs

$$\left(\left(P_m^m(x), \frac{\mathrm{d}}{\mathrm{d}x}P_m^m(x)\right)\left(P_{m+1}^m(x), \frac{\mathrm{d}}{\mathrm{d}x}P_{m+1}^m(x)\right)\ldots\left(P_n^m(x), \frac{\mathrm{d}}{\mathrm{d}x}P_n^m(x)\right)\right) \tag{134}$$

if $0 \leq m$ and $m \leq n$ and $|x| \leq 1$, and an empty list otherwise

`(sph_p_nm_list n m x)` where n is an integer, m is and integer and x is a double; returns a list of pairs

$$\left(\left(Y_m^m(x), \frac{\mathrm{d}}{\mathrm{d}x}Y_m^m(x)\right)\left(Y_{m+1}^m(x), \frac{\mathrm{d}}{\mathrm{d}x}Y_{m+1}^m(x)\right)\ldots\left(Y_n^m(x), \frac{\mathrm{d}}{\mathrm{d}x}Y_n^m(x)\right)\right) \tag{135}$$

if $0 \leq m$ and $m \leq n$ and $|x| \leq 1$, and an empty list otherwise

2.7.7 Mie

This library file provides functions for evaluation of Mie's vector harmonics and scattering coefficients. It requires the GNU Scientific Library described in the previous section. The library must be loaded into Guile prior to loading this file.
Synopsis:

```
guile> (load-extension "/home/gustav/src/Forms/lib/libguile_gsl" "init_gsl")
guile> (load-from-path "mie")
```

The code here adds a couple of Bessel and Legendre utilities on the Scheme level, namely

Bessel Utilities

`(h1_n n x)` where n is an integer and x is a double; returns a double complex, $h_n^{(1)}(x) = j_n(x) + \mathrm{i}y_n(x)$, where j_n and y_n are spherical Bessel functions

`(h2_n n x)` where n is an integer and x is a double; returns a double complex, $h_n^{(2)}(x) = j_n(x) - \mathrm{i}y_n(x)$, where j_n and y_n are spherical Bessel functions

`(js'_n n x)` where n is an integer and x is a double; returns a double, $\mathrm{d}j_n(x)/\mathrm{d}x$. This and the the other derivatives below are calculated using the following recurrence formula:

$$\frac{n}{x}j_n(x) - \frac{\mathrm{d}}{\mathrm{d}x}j_n(x) = j_{n+1}(x) \tag{136}$$

which holds also for $y_n(x)$, $h_n^{(1)}(x)$ and $h_n^{(2)}(x)$

`(ys'_n n x)` where n is an integer and x is a double; returns a double, $\mathrm{d}y_n(x)/\mathrm{d}x$

`(h1'_n n x)` where n is an integer and x is a double; returns a double complex, $\mathrm{d}h_n^{(1)}(x)/\mathrm{d}x$

`(h2'_n n x)` where n is an integer and x is a double; returns a double complex, $\mathrm{d}h_n^{(2)}(x)/\mathrm{d}x$

Legendre Utilities

`(p'_nm n m x)` where n and m are integers and x is a double; returns a double, $\mathrm{d}P_n^m(x)/\mathrm{d}x$. The function uses the following recurrence formula:

$$(x^2-1)\frac{\mathrm{d}}{\mathrm{d}x}P_n^m(x) = -(n+m)(n-m+1)\sqrt{1-x^2}P_n^{m-1}(x) - mxP_n^m(x) \tag{137}$$

With these and with the vector algebra in hand, we can define the Mie vector harmonics and coefficients following Bohren and Huffman [3]:

Vector Harmonics

`(M_o1n l n k r theta phi)` where l and n are integers, and r, θ and ϕ (spherical coordinates) are all doubles; k is the wave number, which may be double in dielectrics or complex in absorbing media. The function returns a (possibly complex valued) vector $\boldsymbol{M}_{o1n}^{(l)}(r,\theta,\phi)$, where l is either 1 or 3, `NaN` otherwise. This and the other three vectors below are as defined by equations (4.50) on page 95 in [3].

`(M_e1n l n k r theta phi)` where l and n are integers, and r, θ and ϕ are all doubles, as above; k is double in dielectrics and complex in absorbing media. The function returns a (possibly complex valued) vector $\boldsymbol{M}_{e1n}^{(l)}(r,\theta,\phi)$, where l is either 1 or 3, `NaN` otherwise.

`(N_o1n l n k r theta phi)` where l and n are integers, and r, θ and ϕ are spherical coordinates (all doubles); k may be double or complex. The function returns a (possibly complex valued) vector $\boldsymbol{N}_{o1n}^{(l)}(r,\theta,\phi)$, where l is either 1 or 3, `NaN` otherwise.

`(N_e1n l n k r theta phi)` where l and n are integers, and r, θ and ϕ are spherical coordinates (all doubles); k may be double or complex. returns a (possibly complex valued) vector $\boldsymbol{N}_{e1n}^{(l)}(r,\theta,\phi)$, where l is either 1 or 3, `NaN` otherwise.

Scattering Coefficients Coefficients a_n and b_n coefficients are used for the scattered field as follows:

$$\boldsymbol{E}_s = \sum_{n=1}^{\infty} E_n \mathrm{i}\left(a_n\boldsymbol{N}_{e1n}^{(3)} - b_n\boldsymbol{M}_{o1n}^{(3)}\right), \tag{138}$$

$$\boldsymbol{H}_s = \frac{k}{\omega}\sum_{n=1}^{\infty} E_n\left(\mathrm{i}b_n\boldsymbol{N}_{o1n}^{(3)} + a_n\boldsymbol{M}_{e1n}^{(3)}\right), \tag{139}$$

where we have assumed that the permeability of the medium $\mu = 1$, and

$$E_n = \mathrm{i}^n E_0 \frac{2n+1}{n(n+1)}, \tag{140}$$

where E_0 is the amplitude of the incident wave, which (the wave) is assumed to be of the form:

$$\boldsymbol{E}_i = E_0 e^{\mathrm{i}kz} \boldsymbol{e}_x = E_0 e^{\mathrm{i}kr\cos\theta} \boldsymbol{e}_x, \tag{141}$$

that is, it is polarized in the $\boldsymbol{e}_x$ direction and propagates up in the $\boldsymbol{e}_z$ direction. The ball's center is co-incident with the center of the coordinate system.

In turn, coefficients c_n and d_n are used in the expansion of the field inside the ball, namely

$$\boldsymbol{E}_l = \sum_{n=1}^{\infty} E_n \left(c_n \boldsymbol{M}_{o1n}^{(1)} - \mathrm{i} d_n \boldsymbol{N}_{e1n}^{(1)} \right), \tag{142}$$

$$\boldsymbol{H}_l = -\frac{k_l}{\omega} \sum_{n=1}^{\infty} E_n \left(d_n \boldsymbol{M}_{e1n}^{(1)} + \mathrm{i} c_n \boldsymbol{N}_{o1n}^{(1)} \right), \tag{143}$$

where E_n is as above, k_l is the wave number inside the ball, where we have assumed that the ball's permeability is $\mu_1 = 1$.

`(E_n n)` where n is an integer; returns E_n/E_0, as given by equation (140). Because E_0 is a constant that multiplies everything, both $\boldsymbol{E}_s$ and $\boldsymbol{H}_s$, we drop it here. The user can multiply later. In most runs we always assume the amplitude of the incident signal to be 1 anyway.

`(a_n n n_r k r_a)` where n is an integer, n_r is a relative refraction index of a ball with respect to the medium it is immersed in, and it may be complex in absorbing materials, k is the wave number of the incident wave *in the surrounding medium,* and r_a is a ball radius. The function returns a scattering coefficient a_n as defined by equation (4.53), page 100, in [3]. Identical permeability is assumed for the ball and the medium.

`(b_n n n_r k r_a)` where the arguments are as for a_n. Returns a scattering coefficient b_n.

`(c_n n n_r k r_a)` where the arguments are as for a_n. Returns a scattering coefficient c_n.

`(d_n n n_r k r_a)` where the arguments are as for a_n. Returns a scattering coefficient d_n.

Scattered Fields These functions evaluate (138) and (139).

`(E_s n_max wavelength n_r r_a r theta phi)` where $n_{\max}$ is the maximum number of expansion terms to be used, `wavelength` is the wavelength of the incident wave, n_r is the relative refraction index of the ball, that is $n_{\text{ball}}/n_{\text{medium}}$, and it may be complex in absorbing media, r_a is the radius of the ball, r is the radial coordinate of a point *outside* the ball (because this is the *scattered* field), θ is the inclination and ϕ is the azimuth—both in radians. For a monochromatic incident wave, the function returns the scattered component of the electric field at this point as a complex valued vector amplitude

$$\hat{\boldsymbol{E}}_\omega = \boldsymbol{E}_r + \mathrm{i}\boldsymbol{E}_i, \tag{144}$$

where the real value interpretation of the above is $\boldsymbol{E}(t) = \boldsymbol{E}_r \cos\omega t + \boldsymbol{E}_i \sin\omega t$. If the incident wave's amplitude is other than 1, the return should be multiplied by the amplitude.

`(H_s n_max wavelength n_r r_a r theta phi)` all input parameters as above. Returns the magnetic field at (r, θ, ϕ) *outside* the ball as a complex valued vector. If the speed of light in the surrounding medium is other than 1, the result should be multiplied by k/ω. If the amplitude of the incident wave is other than 1, the result should be multiplied by the amplitude, too.

`(E_l n_max wavelength n_r r_a r theta phi)` all input parameters as above. Returns the electric field at (r, θ, ϕ) *inside* the ball as a complex valued vector. The result must be multiplied additionally by the amplitude of the incident signal if it's other than 1.

`(H_l n_max wavelength n_r r_a r theta phi)` all input parameters as above. Returns the magnetic field at (r, θ, ϕ) *inside* the ball as a complex valued vector. On return users should multiply the field by k_l/ω, where $k_l = 2\pi/\lambda_l$, and by E_0, if different from 1. Permeability of both the medium and the ball is assumed to be 1.

2.8 Annotated Examples

2.8.1 Snell's Law

File: `Examples/Snell.scm`

The objective of the input file discussed in this section is to test the Snell's law of refraction. The test is also discussed in Section 2.6.2, page 62. The main difficulty with this test is that it's hard to eliminate reflections from media boundaries, edges and corners. What we do here to make the test clearer is to wrap a block of silicon nitride—that fills a box enclosed between the low corner at $[35, 35, 35]$ and the high corner at $[102, 92, 92]$—in a sleeve, 10 units thick, made of an ideal conductor. The silicon nitride surface is exposed only at $x = 35$ and $x = 102$, and it is covered with the conductor on its remaining four sides for which $x \in [35 \ldots 102]$. The media distribution lambda therefore looks as follows:

```
(define forms.media.distribution.lambda
  (lambda ()
    (draw−box (f64vector 35 25 25) (f64vector 102 102 102) 1)
    (draw−box (f64vector 35 35 35) (f64vector 102 92 92) 2)))
```

Here we first draw a box between the low corner at $[35, 25, 25]$ and the high corner at $[102, 102, 102]$ and fill it with medium number 1. This medium will be later defined as an ideal conductor. Then we draw the next box within the ideal conductor one, and fill it with medium number 2. This is going to be silicon nitride and the drawing overwrites the previously assigned media labels.

The media types themselves are defined by a function that the program is going to use to convert $\boldsymbol{D}$ to $\boldsymbol{E}$:

```
(define one_by_epsilon_dielectric (/ 1.0 epsilon_Si3N4))
;;
(define forms.media.model.e_lambda
  (lambda ()
    (case forms.medium
      ((1) 0.0)
      ((2) (dielectric one_by_epsilon_dielectric)))))
```

An ideal conductor is defined by returning zero for $\boldsymbol{E}$, regardless of the value of $\boldsymbol{D}$. Within the ideal conductor the electric field always vanishes. The dielectric is assigned medium number 2, and its value of $\epsilon_{\mathrm{Si_3N_4}}$ is obtained from the `"constants"` library module, which is loaded at the beginning of `"Snell.scm"`.

And this is how we define media types and their distribution for this simulation.

The signal itself is defined as follows:

```
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 16.0 16.0 16.0))
(define forms.signal.hi '(111.0 111.0 111.0))
;;
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 1.0 1.0 0.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define wavelength 30.0)
(define delay −14.0)
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
```

```
    0.0))
;;
;; We make the delay negative, because we want to see the injected
;; signal right away.
;;
(define delay −15.0)
(define wavelength 30.0)
(define omega (/ (∗ 2.0 pi) wavelength))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    (let ((delayed−zeta (+ zeta delay)))
      (∗ (heaviside delayed−zeta) (sin (∗ omega delayed−zeta))))))
```

The total field region into which the signal is injected is confined to a box with its low corner at $[16, 16, 16]$ and its high corner at $[111, 111, 111]$. The signal propagation direction is $\boldsymbol{e}_x + \boldsymbol{e}_y$. The signal is polarized in the $\boldsymbol{e}_z$ direction and is simply a sine of wavelength 30 windowed by a Heaviside function. The Heaviside function definition is provided on the `"windows"` library module, which is loaded at the beginning of `"Snell.scm"`. The origin of the signal is shifted so that it becomes injected into the total field region as soon as the program commences the execution.

The computational domain of the simulation is surrounded by a 10 units thick UPML layer with $\sigma_{\max} = 20$. This is specified as follows:

```
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 20.0)
(define forms.pml.print_arrays 0)
```

The iterations themselves and the output are conditioned by the value of the `media_only` parameter, which is either #t or #f, and which is defined at the top of the file. When `media_only` is set to #t, the program executes two iterations only (one is not enough) and then outputs an image of the media on `"eMdz_001.hdf5"`. When `media_only` is set to #f, the program executes 1028 iterations and dumps 128 snapshots of E_z on `"Ez_001.hdf5"` through `"Ez_128.hdf5"`.

Here is how the number of iterations is requested:

```
(define t_begin 0)
(define t_end 256)
(define forms.iterate.stride 4)
(define forms.iterate.t0 t_begin)
;;
(if media_only
    (begin
      (define forms.iterate.number_of_steps 2)
      (define forms.iterate.image_frequency 1))
    (begin
      (define forms.iterate.number_of_steps
        (∗ (/ (− t_end t_begin) forms.grid.level0.delta) forms.iterate.stride))
      (define forms.iterate.image_frequency (∗ 2 forms.iterate.stride))))
```

And this code specifies the output of the program:

```
(if media_only
    (begin
      (define forms.output.precompute '("eMdz"))
      (define forms.output.write '("eMdz"))
      (define forms.output.filename.root "eMdz")
      (define forms.output.report_outliers 0))
    (begin
      (define forms.output.precompute '("Ez"))
      (define forms.output.write '("Ez"))
      (define forms.output.filename.root "Ez")
      (define forms.output.report_outliers 1)))
```

Users may request both the media and the electric field to be output at the same time, but this would swell the size of the output files needlessly. Because the media distribution does not change, it is sufficient to dump it just once. The image of the media distribution can be then combined with the evolving image of E_z when postprocessing the data with VisIt. See Section 2.3.2, page 36 for more information on how to do this.

In order to test the direction of signal propagation within the dielectric we sample the Poynting vector at a number of points. This is implemented by the means of `forms.post.iteration.lambda`. To make this work, we must load module `"measurements"`, which overloads `interpolate`, to make it accept the whole position vector as an argument and to make it return one, the whole field, too. Here we construct a list of points explicitly, then iterate over its entries, evaluating $\boldsymbol{P} = \boldsymbol{E} \times \boldsymbol{H}$ at each. The values of $\boldsymbol{P}$, assembled into another list called `directions`, are then printed on standard output as follows:

```
(define forms.post.iteration.lambda
  (lambda ()
    (format #t "Poynting vector directions:~%")
    (let* ((points (list (3D-vector 65 65 64)
                         (3D-vector 60 60 64)
                         (3D-vector 55 55 64)
                         (3D-vector 45 45 64)
                         (3D-vector 45 65 64)
                         (3D-vector 65 45 64)))
           (directions (map (lambda (point)
                              (let* ((E (interpolate "E" point))
                                     (H (interpolate "H" point))
                                     (P (x E H)))
                                (normalize P)))
                            points)))
      (map (lambda (direction)
             (format #t " ~a~%" direction))
           directions))))
```

2.8.2 Fourier Analysis

File: `Examples/FourierFlux-Lib.scm`

This code performs computations identical to the ones discussed in Section 2.6.8, page 80, but it uses library utilities instead. This makes the code shorter, easier to read, and, most importantly, easier to write.

The code begins by loading the required library modules:

```
(set! %load-path (cons "/home/gustav/src/Forms/lib" %load-path))
(load-from-path "windows")
(load-from-path "measurements")
```

We load "windows", because we are going to use a **double−tanh−center** window defined on it. Module `"measurements"` defines surfaces, fluxes and Fourier accumulations. Constants (including π) and 3D-vectors are loaded by these two, as well.

This is followed by definitions of the grid and PMLs:

```
;;
;; The Grid group
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
```

```
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
```

In the Signal group, we define the wavelength as global, because we'll need it in the postprocessing group further down, but other signal parameters, width, delay, and sigma, are pushed inside forms.signal.ex_lambda, where we make use of the precooked **double−tanh−center**, when defining the actuall waveform:

```
(define wavelength 30.0)
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (let* ((width 220.0)
           (delay 120.0)
           (sigma 0.4)
           (delayed−zeta (+ zeta delay)))
      (* (double−tanh−center delayed−zeta width sigma)
         (sin (/ (* 2.0 pi delayed−zeta) wavelength))))))
```

The Media and Iterate groups are not of much interest here, because there is no media and the iteration simply runs 1028 steps, from $t_0 = 0$ to $t_{\text{end}} = 360$:

```
;;
;; The Iterate group
;;
(define t_begin 0.0)
(define t_end 360.0)
;;
(define forms.iterate.stride 4)
(define number_of_steps
  (* (/ (− t_end t_begin) forms.grid.level0.delta) forms.iterate.stride))
;;
(define forms.iterate.number_of_steps number_of_steps)
(define forms.iterate.image_frequency number_of_steps)
(define forms.iterate.t0 t_begin)
```

It is the postprocessing group that we simplify by making use of library modules. Skipping over the various constant definitions, such as ω_1, ω_2, radius, dx, and dy, we begin by making a list of frequencies:

```
(define frequencies (make−frequencies omega_0 omega_1 10))
```

then define the center and the normal for each detector disk:

```
(define center_1 (3D−vector 40.0 64.0 40.0))
(define normal_1 (− e_z e_x))
(define center_2 (3D−vector 88.0 64.0 40.0))
(define normal_2 (+ e_z e_x))
```

where we make use of vector arithmetic defined on "3D−vectors". Then we make the disks themselves, alter their plaquette covers and prepare them for Fourier accumulations on $\boldsymbol{E}$ and $\boldsymbol{H}$:

```
(define disk_1 (make−disk center_1 normal_1 radius))
(slot−set! disk_1 'increments (vector dx dy))
(slot−set! disk_1 'plaquettes '())
(prep−for−fourier disk_1 frequencies "E" "H")
;;
(define disk_2 (make−disk center_2 normal_2 radius))
(slot−set! disk_2 'increments (vector dx dy))
(slot−set! disk_2 'plaquettes '())
(prep−for−fourier disk_2 frequencies "E" "H")
```

The accumulations are carried out when forms.post.iteration.**lambda** is called by evaluating

```
(accumulate−fourier disk_1)
(accumulate−fourier disk_2)
```

Within the same function we log interpolated values of $\boldsymbol{E}$ at the center of both disks on "scm_out_4.dat", so that we can observe the progress of the signal through the disks as the computation unfolds:

```
(define forms.post.iteration.lambda
  (lambda ()
    (accumulate−fourier disk_1)
    (accumulate−fourier disk_2)
    (format my−log "E(~a) = ~a, E(~a) = ~a~%"
            center_1 (interpolate "E" center_1)
            center_2 (interpolate "E" center_2))
    (force−output my−log)))
```

When the iterations are over, we call forms.post.all.**lambda**, where we invoke **frequency−power−flux** on disk_1 and disk_2 to generate lists of power fluxes through both disks in function of frequency:

```
(fluxes_1 (frequency−power−flux disk_1) (cdr fluxes_1))
(fluxes_2 (frequency−power−flux disk_2) (cdr fluxes_2)))
```

We do this in the initialization part of the **do** loop:

```
(do ((frequencies frequencies (cdr frequencies))
     (fluxes_1 (frequency−power−flux disk_1) (cdr fluxes_1))
     (fluxes_2 (frequency−power−flux disk_2) (cdr fluxes_2)))
    ((null? frequencies) (close−port my−log))
  (format my−log "omega = ~a, flux_1 = ~a, flux_2 = ~a~%"
          (car frequencies)
          (car fluxes_1)
          (car fluxes_2)))))
```

the loop that lists the values of ω and the two fluxes on "scm_out_4.dat". The values are exactly as returned by the explicit computation example discussed in Section 2.6.8, page 80, code file `Examples/FourierFlux.scm`, which proves that both codes implement the same computation. Whether the computation is correct, can be ascertained by analyzing one or the other code, whichever one feels more comfortable with, in detail, or comparing the results obtained numerically against analytical computation, if this is feasible. An example of the latter has been provided towards the end of Section 2.6.8.

On return function forms.post.all.**lambda** closes the log file, as is specified by the termination clause of the **do** loop:

```
((null? frequencies) (close−port my−log))
```

2.8.3 Mie Scattering

File: `Examples/Mie-1.scm`

The objective of this example is to test the Forms' FDTD against the analytical solution of Mie for scattering of an incident harmonic wave on a uniform dielectric sphere. The test itself and its outcome are discussed in Section 2.6.3, page 63. Here we are going to focus on the input file instead. The input file shows a number of features that illustrate the power and flexibility gained by providing program users with the extension language.

The three special features that are exercised here are:

1. Introduction of various control parameters that steer the solution in various ways: for example, one may request that only the media distribution be evaluated and dumped on an HDF5 file; or one may request that instead of writing field data on HDF5 files, it be written instead on human readable Gnuplot files.

2. Use of Scheme as a preliminary calculator, to determine the geometry of the region, before the run commences.

3. Specification of an alternative output mode; here we collect field data from a slice that cuts through the computational domain and write it out for postprocessing with Gnuplot.

The input commences by loading a number of library modules discussed in Section 2.7, page 87:

```
(set! %load-path (cons "/home/gustav/src/Forms/lib" %load-path))
(load-from-path "constants")
(load-from-path "media")
(load-from-path "windows")
(load-from-path "measurements")
```

We will obtain material data from `"constants"`, material equations from `"media"`, signal windowing from `"windows"`, and field interpolation procedures from `"measurements"`. Additionally, we also load the `"mie"` module:

```
(load-extension "/home/gustav/src/Forms/lib/libguile_gsl" "init_gsl")
(load-from-path "mie")
```

This is not strictly necessary here, and we do not make any use of it in the example. But we show how to do this, in case the user would like to make direct numerical comparisons between the Mie and numerical solutions.

The script then pulls certain parameter definitions up front, so that they can be easily modifed, namely:

```
(define output_time_from 907)
(define output_time_to 1027)
(define t_begin 0)
(define t_end 1028)
(define HDF5_output #f)
(define media_only #f)
```

We are going to compute without generating any output at all for a certain amount of time, in order to stabilize the system. Then we will generate human readable slices of field data, while the system is between $t = 907$ and $t = 1027$. For this to work, of course, we must keep iterating until $t = 1027$ inclusive. We will not require in this case the HDF5 output and do not need media distribution either. If we wanted to dump the media layout, we'd just set `media_only` to true, this would then automatically activate HDF5 output, as well:

```
(if media_only
    (begin
      (format #t "> Calculating and dumping media layout only.~%")
      (define HDF5_output #t)))
```

Next we commence pre-computation. Here we use Scheme as a calculator in order to evenly divide the computational domain into PML, scattered field and total field regions and position the dielectric ball exactly in the centre of the latter, while leaving sufficiently wide margins on all sides, so that we can observe the scattered field and the total field in its vicinity:

```
(define number_of_cells 191)
(define dx 1)
(define d_pml 15)
(define x_0 0)
(define x_1 (1- number_of_cells))
(define x_0_pml (+ x_0 d_pml))
(define x_1_pml (- x_1 d_pml))
(define d_cpdmn (- x_1_pml x_0_pml))
(define d_sctrd (floor (/ d_cpdmn 6)))
(define x_0_tfr (+ x_0_pml d_sctrd))
(define x_1_tfr (- x_1_pml d_sctrd))
(define d_tfr (- x_1_tfr x_0_tfr))
(define x_c (+ x_0_tfr (/ d_tfr 2)))
(define diameter (/ d_tfr 2))
(define radius (/ diameter 2))
```

Having pre-computed the geometry, we assign it to the Form variables. First for the grid:

```
(define forms.grid.level0.cells (make-list 3 number_of_cells))
(define forms.grid.level0.origin (make-list 3 x_0))
(define forms.grid.level0.delta dx)
(define forms.grid.max_box_size (* 2 number_of_cells))
```

We specify the `max_box_size` to be twice the domain length, to ensure that everthing is confined to a single box.

The PML definitions follow:

```
(define forms.pml.lo (make−list 3 x_0_pml))
(define forms.pml.hi (make−list 3 x_1_pml))
(define forms.pml.sigma_max 10.0)
(define forms.pml.print_arrays 0)
```

And now we get to the Signal group. First, the geometry of the total field region and the direction of the injected signal:

```
(define forms.signal.garbage_collect 0)
(define forms.signal.lo (make−list 3 x_0_tfr))
(define forms.signal.hi (make−list 3 x_1_tfr))
(define forms.signal.direction (list 0 0 1))
(define forms.signal.normalized 1)
```

Next the signal itself. The signal is going to be polarized in the $\boldsymbol{e}_x$ direction, so the E_y and E_z components are zero. The E_x component is

```
(define delay (− 2 x_0_tfr))
(define wavelength 60.0)
(define omega (/ (∗ 2.0 pi) wavelength))
(define forms.signal.ex_lambda
  (lambda (zeta)
    (let ((delayed−zeta (+ zeta delay)))
      (∗ (heaviside delayed−zeta) (sin (∗ omega delayed−zeta))))))
```

The scattering material is just one ball centered on (x_c, x_c, x_c) with a radius of `radius`, filled with a dielectric. We use the parameters of Si_3N_4:

```
(define forms.media.distribution.lambda
  (lambda ()
    (draw−ball (make−f64vector 3 x_c) radius 1)))
(define one_by_epsilon_dielectric (/ 1.0 epsilon_Si3N4))
(define forms.media.model.e_lambda
  (lambda ()
    (case forms.medium
      ((1) (dielectric one_by_epsilon_dielectric)))))
```

The iteration instructions are a little more complicated. If we want to just dump the media distribution, we only need to make two time steps. Otherwise, of course, we need to keep iterating until we reach $t = t_{\text{end}}$:

```
(define forms.iterate.stride 4)
(define forms.iterate.t0 t_begin)
(define forms.iterate.image_frequency forms.iterate.stride)
(if media_only
    (begin
      (define forms.iterate.number_of_steps 2)
      (define forms.iterate.image_frequency 1))
    (define forms.iterate.number_of_steps
        (∗ (/ (− t_end t_begin) forms.grid.level0.delta) forms.iterate.stride)))
```

Regardless of whether we generate HDF5 output or not, when the system finds itself in $t \in [t_{\text{from}}, t_{\text{to}}]$ we want to generate field data from a two dimensional slice for gnuplot. The slice is rectangular, perpendicular to $\boldsymbol{e}_x$, passing through x_c and stretching 80.5 units in the $\pm\boldsymbol{e}_y$ and $\pm\boldsymbol{e}_z$ directions. The data is going to be generated only when `forms.iterate.time_to_dump` is set to 1 by FORMS. This parameter is made available to `forms.post.iteration.lambda`. Within the following script we loop over the requested values of y and z and write the interpolated component of E_x on the output file. A new output file is opened and closed every time `forms.post.iteration.lambda` is invoked, and both conditions (that is "time to dump" and "time in the requested segment") fire up:

```
(define forms.post.iteration.lambda
```

```
(lambda ()
  (if (and (<= output_time_from forms.time_e)
           (<= forms.time_e output_time_to))
      (if (= 1 forms.iterate.time_to_dump)
          (begin
            (format #t "Scheme: writing slice data for time_e = ~a~%"
                    forms.time_e)
            (let* ((y_from (- x_c 80.5))
                   (y_to (+ x_c 80.5))
                   (z_from y_from)
                   (z_to y_to))
              (do ((output-port
                    (open-file
                     (format "Ex_~4,'0d.dat" (inexact->exact (floor forms.time_e)))
                     "w"))
                   (y y_from (1+ y)))
                  ((> y y_to) (close-port output-port))
                (do ((z z_from (1+ z)))
                    ((> z z_to) (format output-port "~%"))
                  (format output-port "~a ~a ~a~%" (- z x_c) (- y x_c)
                          (interpolate "Ex" x_c y z)))))))))
```

This little part of the input code is a nice illustration of how much can be accomplished with the present system, without having to meddle with the FORMS internals.

The Output Group is conditioned by both `HDF5_output` and `media_only`. Depending on what's requested in the input's preamble, we either dump the media, `"eMdz"`, which collects the media distribution data from the E_z mounting points, or E_x:

```
(if HDF5_output
    (if media_only
        (begin
          (define forms.output.precompute '("eMdz"))
          (define forms.output.write '("eMdz"))
          (define forms.output.filename.root "eMdz")
          (define forms.output.report_outliers 0))
        (begin
          (define forms.output.precompute '("Ex"))
          (define forms.output.write '("Ex"))
          (define forms.output.filename.root "Ex")
          (define forms.output.report_outliers 1)))
    (format #t "No HDF5 output.~%"))
```

The remainder of the input file just activates various watchers, so that something is printed on standard output, for the user's enjoyment, as the program goes through the motions.

2.8.4 Mie Scattering on Silver

File: `Examples/Drude-Input.scm`

This input file differs from our previous Mie scattering example in its use of two C-language extensions. The extensions are employed to inject signal and to process the Drude ADE. The extension files are `lib/harmonic.c` and `lib/drude.c`. They must be compiled into either `.dll` files (under Windows/Cygwin) or `.so` files under Linux or MacOS for loading into Guile. File `lib/Makefile` shows how to do this.

The only FORMS library file that's loaded into the run is `"constants"`. This file contains parameters for the Drude model we are going to use.

The other difference is that we reduce the number of cells from 191 to 180, because we will need more memory for the auxiliary fields used by the ADE.

Otherwise the file is identical, until we get to the Signal group. Here we define the wavelength and the delay differently in the way expected by the extension, then load the extension itself:

```
(define forms.signal.wavelength 60.0)
(define forms.signal.delay (- 2 x_0_tfr))
(load-extension "/home/gustav/src/Forms/lib/harmonic.dll" "init_signal_lambdas")
```

The next change is in the Media group. The extension assigns number 2 to the Drude model and number 1 to the dielectric. Both the dielectric and the Drude constants must be defined, even if one only is used. We must define also the number of auxiliary fields as 2. The Drude parameters specified on the `"constants"` file are in radians per second. But these are not the units we want to compute in. We scale them here by specifying the frequency of the incident signal in radians per second and then by specifying it in our computational units. And so, this part of the input file looks as follows:

```
(define forms.media.distribution.lambda
  (lambda ()
    (draw-ball (make-f64vector 3 x_c) radius 2)))
(define forms.media.dielectric.epsilon epsilon_SiO2)
(define forms.media.number_of_auxiliary_fields 2)
(define omega (/ (* 2.0 pi) forms.signal.wavelength))
(define forms.media.drude.gamma (* (/ gamma_BL_Ag_450_800 (eV->omega 0.89)) omega))
(define forms.media.drude.omega (* (/ omega_BL_Ag_450_800 (eV->omega 0.89)) omega))
(define forms.media.drude.epsilon 1.0)
(load-extension "/home/gustav/src/Forms/lib/drude.dll" "init_e_lambda")
```

Finally, there is one more change in the Postprocessing group. Here the lambda writes two data slices every time it is invoked. First for the zy slice, then for the zx slice:

```
(define forms.post.iteration.lambda
  (lambda ()
    (if (and (<= output_time_from forms.time_e)
             (< forms.time_e output_time_to))
        (if (= 1 forms.iterate.time_to_dump)
            (begin
              (format #t "Scheme: writing slice data for time_e = ~a~%"
                      forms.time_e)
              (let* ((y_c x_c)
                     (z_c x_c)
                     (reach (- x_c x_0_pml 1))
                     (y_from (- y_c reach))
                     (y_to (+ y_c reach))
                     (z_from y_from)
                     (z_to y_to)
                     (x_from y_from)
                     (x_to y_to))
                (do ((output-port
                      (open-file
                       (format "Ex_zy_~4,'0d.dat" (inexact->exact (floor forms.time_e)))
                       "w"))
                     (y_i y_from (1+ y_i)))
                    ((> y_i y_to) (close-port output-port))
                  (do ((z_i z_from (1+ z_i)))
                      ((> z_i z_to) (format output-port "~%"))
                    (format output-port "~a ~a ~a~%"
                            (exact->inexact (- z_i z_c))
                            (exact->inexact (- y_i y_c))
                            (interpolate "Ex" x_c y_i z_i))))
                (do ((output-port
                      (open-file
                       (format "Ex_zx_~4,'0d.dat" (inexact->exact (floor forms.time_e)))
                       "w"))
                     (x_i x_from (1+ x_i)))
                    ((> x_i x_to) (close-port output-port))
                  (do ((z_i z_from (1+ z_i)))
```

```
            ((> z_i z_to) (format output-port "~%"))
          (format output-port "~a ~a ~a~%"
                  (exact->inexact (- z_i z_c))
                  (exact->inexact (- x_i x_c))
                  (interpolate "Ex" x_i y_c z_i)))))
(gc))))))
```

2.8.5 Mie Scattering on Silver in Parallel

File: `Examples/Parallel-Drude-Input.scm`

Here we are going to discuss in some detail the input file for a parallel (MPI) job that computes the same as above, that is, scattering on a metal ball. Because it runs in parallel, this is a typical example for a production job to throw on a supercomputer. But the same input can be used to run FORMS in parallel, also under MPI, on a 4-way PC. 6 GB memory will be required for the run.

The input file begins with some simple string manipulations that reconstruct the path to the FORMS library location, and this may work without change both on a desktop PC and on an institutional cluster, if the FORMS source lives in `~/src/Forms` on both systems. `"constants"` is the only library module we'll need here, because we're going to use compiled extensions for the injected signal and material characterization:

```
(define HOME (getenv "HOME"))
(define LIB (string-append HOME "/src/Forms/lib"))
(set! %load-path (cons LIB %load-path))
(load-from-path "constants")
```

Next we evaluate the parallel MPI environment finding about the size of the process pool and the rank number of the process we are currently associated with. This number is then used to construct the name of the output file for the Guile log. Each process logs on its own file here. The file is opened and information about the MPI pool written on it:

```
(define my-rank (proc-id))
(define root-rank (unique-proc))
(define pool-size (num-proc))
(define my-log (open-file (format #f "guile.~a" my-rank) "a"))
(format my-log "Guile log:~%")
(format my-log " process number: ~a~%" my-rank)
(format my-log " pool size: ~a~%" pool-size)
(force-output my-log)
```

What follows are geometry specifications, which are evaluated dynamically by taking the `number_of_cells` as input. This is the number of cells in one direction. The constructed domain will be a cube, with this number of cells in each direction. The domain will then be subdivided into boxes following the instruction:

```
(define forms.grid.max_box_size (/ number_of_cells 2))
```

When the level 0 grid is constructed it will be subdivided into eight boxes, which will then be distributed amongst MPI processes. On a 4-way PC, every process gets two boxes. On an 8-way node of an MPP cluster, every process gets one box. The number of nodes to use is specified on the MPI command line that starts the job.

What follows next are grid, PML, signal, and media specifications, which are as before, but things become a little more complicated, when we define the `forms.post.iteration.lambda`. Here, we have to include the MPI process number in the output file name, because otherwise processes would write on each other's data. In effect, instead of writing a single output file for a given time slice, say, `Ex_zy_0158.dat`, we'll write `Ex_zy_0_0158.dat, Ex_zy_1_0158.dat, ... Ex_zy_7_0158.dat,` if we distribute the job over eight MPI processes. This is how it happens:

```
(open-file
 (format #f "Ex_zy_~d_~4,'0d.dat" my-rank (inexact->exact (floor forms.time_e)))
 "w"))
(open-file
 (format #f "Ex_zx_~d_~4,'0d.dat" my-rank (inexact->exact (floor forms.time_e)))
 "w"))
```

Otherwise the input looks the same as the input for the sequential job.

Now, what are we going to see on the files generated this way? Function `interpolate` returns zero, if a given point does not reside within any of the boxes that a given MPI process is taking care of. To obtain a proper data slice then, we have to add field values from all files generated by the MPI processes for a given time slice. We could do this at the end of the program execution by inserting instructions in `forms.post.all.lambda`, but this is not a big job computationally, and it can be done on a single desktop PC, so it would not be a good idea to waste our supercomputer time allocation by running it within the FORMS job.

And so, we have a separate utility developed to do just this, which is a Scheme script called `merge.scm`.

File: `Examples/merge.scm`

The script defines a single function, called `process-files`, which takes its input, somewhat inelegantly, from a couple of globals, which are defined just ahead of it. It uses a couple of utilities from the SLIB Portable Scheme Library of Aubrey Jaffer, specifically, `fscanf` and `reduce`. As the SLIB library is not distributed with Guile, it has to be separately downloaded and installed. But it has settings for Guile and knows how to work with it.

Function `process-files` is a thunk, meaning that it does not take any arguments. The `lambda` specification begins with a construction of a list of file name roots, such as `Ex_zy_0_`, `Ex_zy_1_`, ... `Ex_zy_7_` if we have seven processes in the MPI pool:

```
(do ((name_root_list '())
     (count 0 (1+ count)))
    ((>= count number_of_cpus) name_root_list)
  (set! name_root_list
        (cons (string-append DATA_DIR name_root
                             (format #f "_~d_" count)) name_root_list)))))
```

As the function enters the main `do` loop that runs over all time slices dumped, for every slice the list of file name roots is mapped onto a list of full file names for the slice:

```
(let ((input_file_names (map (lambda (name)
                               (string-append name (format #f "~4,'0d.dat" n)))
                             name_root_list))
```

The files are then opened for reading and arranged into another list:

```
(do ((input_ports (map (lambda (name)
                         (open-file name "r"))
                       input_file_names))
```

And we open the output file for writing at the same time:

```
(output_port (open-file output_file_name "w"))
```

Now we start looping over all rows in the input files, for each row reading the items from the input files and assembling them into lists:

```
(x (make-list number_of_cpus 0.0))
(y (make-list number_of_cpus 0.0))
(f (make-list number_of_cpus 0.0))
(number_of_items_read (make-list number_of_cpus 0))
(do ((number_of_items_read number_of_items_read (cdr number_of_items_read))
     (input_ports input_ports (cdr input_ports))
     (x x (cdr x))
     (y y (cdr y))
     (f f (cdr f)))
    ((null? input_ports))
  (set! (car number_of_items_read)
        (fscanf (car input_ports)
                " %f %f %f " (car x) (car y) (car f))))
```

When we hit the end of files, the reads return the `eof-objects` in the `number_of_items_read` list. We check for this and set up the `finished` flag, when found:

```
(set! finished (eof-object? (car number_of_items_read)))
```

Otherwise, having the valid data assembled into lists, we write the `x_0` and `y_0` coordinates on the output file, they should be repeated, the same, in their respective lists, followed by the sum of the values read into the `f` list. The sum is computed here by calling `reduce`:

```
(let ((x_0 (car x))
      (y_0 (car y)))
  (if (< x_0 x_last) (format output_port "~%"))
  (set! x_last x_0)
  (format output_port "~a ~a ~a~%" x_0 y_0 (reduce + f))
```

And this is it. The function writes dots on standard output for every 100 rows processed and a newline for every 7000 rows processed, this to provide some feedback to the user that accounts for the work done so far.

Having processed the data in this way for a parallel run, we can now carry out a crucial test: compare the output with data obtained from a sequential computation. Here is a genuine listing:

```
$ pwd
/home/gustav/src/Data/Sequential/Metal/Quarry-Ball-5/DAT
$ for i in *.dat
> do
> diff $i ../../../../Parallel/Metal/Ball-2/DAT_sum/$i
> done
29187c29187
< 17.5 15.5 -9.8564963519246e-6
---
> 17.5 15.5 -9.85649635192459e-6
$
```

The data in the `/Sequential/.../Quarry-Ball-5/DAT` directory was obtained by running a job, otherwise identical to the parallel one, i.e., 256 cells in each direction of the computational domain, on a single core of an IU Quarry Cluster node. The data in the `/Parallel/.../Ball-2/DAT_sum/` directory was obtained by running a job on all 8 cores of an IU Quarry Cluster node under MPI. The generated files were then add-reduced as described in this section. The `diff` finds only one difference for all time slots (in this case 857 through 966) due to rounding on the desktop PC, on which the reduction was carried out. The comparison here runs through both $z \wedge y$ and $z \wedge x$ slices. The fact that we don't see any differences proves that the data is handled correctly on the inter-box boundaries by the computational routines and by the output routines—in this case just `interpolate`—as well.

2.8.6 Power Flux Map

File: `Examples/fdtd-power-map.scm`

This section discusses the codes used in the Mie flux test discussed in Section 2.6.6, page 73.

First, we're going to look at the FORMS input. The file is similar to that discussed in the previous section with minor, but important, changes. The first change is that the geometry of the computational domain is altered somewhat. Here we want to have the scattered field regions sufficiently wide to accommodate the measurement sphere, here referred to as "flux sphere." We shrink the width of the UPML region by one cell to 14, and fix the radius of the scattering sphere at 27. We only request that the scattering sphere be separated from the edge of the total field region by a margin of 15 cells:

```
;; Auxiliary: evaluate the geometry of the computational domain.
;;
(define number_of_cells 191)
;;
;; resolution
;;
(define dx 1)
;;
;; width of the PML region
;;
```

```
(define d_pml 14)
;;
;; evaluate the rest of the domain's geometry
;;
(define x_0 0)
(define x_1 (1− number_of_cells))
(define x_0_pml (+ x_0 d_pml))
(define x_1_pml (− x_1 d_pml))
(define d_cpdmn (− x_1_pml x_0_pml))
;;
;; this time we fix the radius of the ball at 27
;; and seek to minimize the extent of the total field
;; region.
;;
(define radius 27)
(define diameter (∗ 2 radius))
(define x_c (+ x_0_pml (/ d_cpdmn 2)))
;;
(define margin 15)
;;
(define x_0_tfr (− x_c radius margin))
(define x_1_tfr (+ x_c radius margin))
(define d_tfr (− x_1_tfr x_0_tfr))
;;
(define d_sctrd (− x_0_tfr x_0_pml))
;;
(define flux−sphere−radius (ceiling (+ (/ (∗ d_tfr (sqrt 3.0)) 2) 2)))
```

At the end we evaluate the measurement sphere's radius, by making it a little larger, by just two cells, than the distance between the center of the computational domain (given by (x_c, x_c, x_c)) and the corner of the total field region. Because the latter is a cube, this distance is given by $(d_{\mathrm{tfr}}/2)\sqrt{3}$, where d_{tfr} is the width of the total field region box.

The rest proceeds exactly as before, until we get to the postprocessing group of definitions. Here we commence by making the measurement sphere:

```
(define sphere (make−sphere (3D−vector x_c x_c x_c) flux−sphere−radius 'z))
```

The 'z parameter will orient the sphere so that its $\theta = 0°$ inclination angle corresponds to the $+z$ direction, which is the direction in which the injected signal propagates. The polar regions of the sphere are cut out from $\theta = 0°$ to $\theta = 5°$, to avoid coming too close to the singularity of the coordinate system.

Once the sphere has been made, it has to be prepared for the accumulation of power fluxes, $\boldsymbol{n}_p \cdot (\boldsymbol{E}_p \times \boldsymbol{H}_p)\, \mathrm{d}t$ on its plaquettes. We do this by calling

```
(prepare−power−flux−map sphere)
```

Now, we're ready to define our forms.post.iteration.**lambda**:

```
(define dump−map #t)
;;
(define forms.post.iteration.lambda
  (lambda ()
    (if (and (<= output_time_from forms.time_e)
             (< forms.time_e output_time_to))
        (begin
          (format #t "Scheme: accumulating power map~%")
          (accumulate−power−flux−map sphere)
          (if (= 1 forms.iterate.time_to_dump)
              (begin
                (format #t "Scheme: writing slice data for time_e = ~a~%"
                        forms.time_e)
                (let∗ ((y_c x_c)
```

```
                (z_c x_c)
                (y_from (- y_c 80.5))
                (y_to (+ y_c 80.5))
                (z_from y_from)
                (z_to y_to))
            (do ((output-port
                  (open-file
                   (format "Ex_~4,'0d.dat" (inexact->exact (floor forms.time_e)))
                   "w"))
                 (y_i y_from (1+ y_i)))
                ((> y_i y_to) (close-port output-port))
              (do ((z_i z_from (1+ z_i)))
                  ((> z_i z_to) (format output-port "~%"))
                (format output-port "~a ~a ~a~%" (- z_i z_c) (- y_i y_c)
                        (interpolate "Ex" x_c y_i z_i))))))))
  (if (and (> forms.time_e output_time_to) dump-map)
      (let ((delta-t (- output_time_to output_time_from)))
        (format #t "Scheme: rescaling and dumping power map~%")
        (rescale-power-flux-map sphere (/ delta-t))
        (dump-power-flux-map sphere "power-map.dat"
                             (/ degree) (/ degree) -1)
        (reset-power-flux-map sphere)
        (set! dump-map #f))))))
```

The lambda is in the form of a large **if** statement with two clauses. The first one fires up only when $t_e \in [t_{\rm from}, t_{\rm to}[$. These two times are defined by the user at the beginning of the input file. The user should notice that we carefully exclude $t_{\rm to}$ from the measurement this time. This is similar to how we number C-arrays from 0 to some $n-1$, not to n. Here we'll want to take the measurement precisely over two full vibration periods. But if the measurement commences at $t_{\rm from}$, and $t_{\rm to} = t_{\rm from} + T$, then the latter belongs to the *next* period, not to this one. Otherwise we would count the period starting point twice.

The second clause of the primary **if** statement fires up outside the $t_e \in [t_{\rm from}, t_{\rm to}[$ segment. But there is another **if** clause there, which activates it for $t_e > t_{\rm to}$ only, that is, after the data accumulation itself has been completed. In this case we first calculate $\Delta t = t_{\rm to} - t_{\rm from}$, then rescale the accumulated power flux map by dividing it by Δt, which makes it into the average power flux map—averaged over time. Then we dump the map on "power-map.dat" while converting its ϕ and θ coordinates to degrees and reversing the sign of the accumulated flux. Finally we reset the power flux map back to zero.

There is one more little trick in this part of the code. We must make sure that this operation will be performed once only. For this reason we have the logical switch dump-map, which is first set to #t. Once the map has been dumped, we toggle it to #f, to disable these operations on the following iteration cycles.

The first clause of the outermost **if** statement works as in the previous example, that is, it collects E_x data from the $x = x_c$ slice for times in $[t_{\rm from}, t_{\rm to}[$, and writes it on output files numbered by t_e. But before it gets there, it also accumulates data on the measurement sphere:

```
(format #t "Scheme: accumulating power map~%")
(accumulate-power-flux-map sphere)
```

This way we combine both functions in this single test.

File: `Examples/fdtd-fourier-map.scm`

The same result can be obtained not by accumulating $\boldsymbol{n}_p \cdot (\boldsymbol{E}_p(t) \times \boldsymbol{H}_p(t))\,{\rm d}t$ for each plaquette, but by accumulating Fourier transforms of $\boldsymbol{E}_p(t)$ and $\boldsymbol{H}_p(t)$ for $\omega = 2\pi/T$, and then, on completion, evaluating $\frac{1}{2T^2}\boldsymbol{n}_p \cdot \Re\left(\hat{\boldsymbol{E}}_p(\omega) \times \hat{\boldsymbol{H}}_p(\omega)\right)$ for each plaquette. This is discussed in Section 2.6.6, page 75.

File `fdtd-fourier-map.scm` implements just such a computation. It is nearly identical to the one discussed above, but for the following differences. After having defined the sphere, we prepare it for the accumulations by calling

```
(define frequencies (list omega))
(prep-for-fourier sphere frequencies "E" "H")
```

Then, inside the `forms.post.iteration.lambda`, we accumulate the transforms by calling

Table 9: Comparison of the number of operations for two methods of power flux computation. The third method, direct+, is the direct method that skips taking the dot product with $\boldsymbol{n}_p$ until the very end.

operation	number of		
	Fourier	direct	direct+
multiplications	12	9	6
sums/differences	12	12	9

```
(format #t "Scheme: accumulating power map~%")
(accumulate-fourier sphere)
```

The accumulated Fourier transform is *not* the same as the phasor we want. This is easy to see by evaluating

$$\begin{aligned}
&\int_0^{2T} \left(\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t\right) e^{\mathrm{i}\omega t}\,\mathrm{d}t \\
&\quad = \int_0^{2T} \left(\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t\right)\left(\cos\omega t + \mathrm{i}\sin\omega t\right)\,\mathrm{d}t \\
&\quad = \int_0^{2T} \left(\boldsymbol{E}_c \cos^2\omega t + \mathrm{i}\boldsymbol{E}_s \sin^2\omega t\right)\,\mathrm{d}t \\
&\quad = T\left(\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s\right)
\end{aligned} \tag{145}$$

So, we find that it is T times the phasor instead. Consequently, when dumping the power data, we need to divide it by T^2. And so, we call

```
(format #t "Scheme: rescaling and dumping power map~%")
(dump-frequency-power-flux-map
 sphere "power-map.dat" (/ degree) (/ degree) (- (/ (* wavelength wavelength))))
```

This is all the change we need.

Which is a better way?

We can answer this question by looking at the number of arithmetic operations per accumulation per plaquette per frequency. For the Fourier method we just add $\boldsymbol{E}_p(t)e^{\mathrm{i}\omega t}\,\mathrm{d}t$ to the $\hat{\boldsymbol{E}}_p(\omega)$ store, and $\boldsymbol{H}_p(t)e^{\mathrm{i}\omega t}\,\mathrm{d}t$ to the $\hat{\boldsymbol{H}}_p(\omega)$ store. Now, these are complex number operations with expensive function calls to sin and cos inside the exp. But $e^{\mathrm{i}\omega t}$ can be evaluated just once, for every frequency, before looping over the surface's plaquettes, while doing the accumulation, so this is not an issue. However, as the exp returns a complex number here, its multiplication by $\boldsymbol{E}$ implies six real number multiplications and then six for the $\boldsymbol{H}$ field—twelve altogether. Then we have twelve additions on the accumulation.

For comparison, when accumulating $\boldsymbol{n}\cdot(\boldsymbol{E}_p \times \boldsymbol{H}_p)$, we have six real number multiplications in the cross product, three differences, and then three multiplications and sums on taking the scalar product with $\boldsymbol{n}_p$. So, we have nine multiplications and six sums. Then another six sums on the accumulation. This can be further optimized by not taking the dot product with $\boldsymbol{n}_p$, until at the very end, which takes away three multiplications and three additions, but forces us to store the whole vector.

Table 9 compares the two methods, adding the third one, which is "direct-optimized." We see a considerable difference. The direct method with optimization may run nearly twice as fast as the Fourier accumulation.

The Fourier accumulation method is even more expensive, when accumulations are carried out for multiple frequencies and throughout the whole duration of the computation. To accumulate power flux using the direct method for $\Delta t = T$ or even $\Delta t = 2T$ at a time, and to repeat the process a couple of times, while adjusting the incident signal frequency, is a cheaper way to do it.

File: `Examples/mie-power-map.scm`

This file is not a Forms input file. It is a Guile input file, which is used to compute the power flux map for comparison with the FDTD result, using the Mie formula. It illustrates how various facilities developed for Forms can be used in stand-alone Guile scripts.

This time we must load the "mie" library, which must be preceded by loading "libguile_gsl". So the preamble of the script looks as follows:

```
(set! %load-path (cons "/home/gustav/src/Forms/lib" %load-path))
(load-from-path "constants")
(load-from-path "measurements")
;;
;; Mie specific stuff
;;
(load-extension "/home/gustav/src/Forms/lib/libguile_gsl" "init_gsl")
(load-from-path "mie")
```

Then we proceed as before, evaluating the geometry of the computational domain first:

```
(define ball-radius 27)
(define flux-sphere-radius 75)
(define wavelength 60)
(define epsilon epsilon_SiO2)
(define n-ball (sqrt epsilon))
(define number-of-iterations 10)
;;
(define k (/ (* 2 pi) wavelength))
(define k-flux-sphere-radius (* k flux-sphere-radius))
(define k-ball-radius (* k ball-radius))
```

The number-of-iterations parameter tells the Mie functions **E_s** and **H_s** how many sum terms to take, while evaluating the scattered fields. We find that 10 is accurate enough.

Next we create the measurement sphere and prepare it for the power flux accumulation. Here we use the same functions as the FDTD script:

```
(format #t "Preparing a sphere for flux collection... ") (force-output)
(define sphere (make-sphere flux-sphere-radius 'z))
(prepare-power-flux-map sphere)
(format #t "done.~%") (force-output)
```

But we fill the data slots of the plaquettes by calling Mie functions instead:

```
(format #t "Filling the flux sphere with Mie data... ") (force-output)
(do ((plaquettes (plaquettes sphere) (cdr plaquettes)))
    ((null? plaquettes))
  (let* ((plaquette (car plaquettes))
         (phi (car (parameters plaquette)))
         (theta (cdr (parameters plaquette)))
         (normal (car (other-data plaquette)))
         (E (E_s number-of-iterations n-ball k-ball-radius k-flux-sphere-radius theta phi))
         (H (H_s number-of-iterations n-ball k-ball-radius k-flux-sphere-radius theta phi))
         (P (* normal 0.5 (real-part (x E H)))))
    (set-cdr! (other-data plaquette) P)))
(format #t "done.~%") (force-output)
```

Because the Mie solution returns complex phasors, $\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_p)$ and $\hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_p)$, we use equation (379) on page 168, that is

$$P_p(\omega) = \boldsymbol{n}_p \cdot \frac{1}{2}\left(\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_p) \times \hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_p)\right), \tag{146}$$

where subscript p refers to the *plaquette.*

After we have filled the measurement sphere's plaquettes with data, we use the same function as the FDTD input, to dump it on a file:

```
(format #t "Writing the data on mie-power-map.dat... ") (force-output)
(dump-power-flux-map sphere "mie-power-map.dat" (/ degree) (/ degree) -1)
(format #t "done.~%")
```

3 Numerics

Here is a complete summary of numerics, and explicit listing of all formulas used throughout the code.

Mathematics comes first! There is no point, absolutely no point, laying your fingers on the keyboard until you've got the mathematics worked out in all required detail. All the clever coding in the world cannot replace this most basic of all steps.

So, here we are...

3.1 Maxwell Equations

We start from the following form of equations as listed in Feynman [7], Chapter 32, pages 32-4/5.

$$\begin{aligned}
\boldsymbol{\nabla}\cdot\boldsymbol{D} &= \rho_{\text{free}}, && (147)\\
\boldsymbol{\nabla}\cdot\boldsymbol{B} &= 0, && (148)\\
\boldsymbol{\nabla}\times\boldsymbol{H} &= \boldsymbol{j}_{\text{free}} + \partial_t\boldsymbol{D}, && (149)\\
\boldsymbol{\nabla}\times\boldsymbol{E} &= -\partial_t\boldsymbol{B}, && (150)\\
\boldsymbol{D} &= \epsilon\boldsymbol{E}, && (151)\\
\boldsymbol{B} &= \mu\boldsymbol{H}. && (152)
\end{aligned}$$

The $\boldsymbol{D}$ and $\boldsymbol{H}$ are, as Feynman puts it, "hidden ways of *not* paying attention to what is going on inside the material."

In this code we have broadened equations (151) and (152) to cover *any* dependence of $\boldsymbol{E}$ on $\boldsymbol{D}$ and $\boldsymbol{H}$ on $\boldsymbol{B}$, including tensor dependencies, field accumulations, nonlinear dependencies, dependence on location, time, etc.,

$$\begin{aligned}
\boldsymbol{E} &= \boldsymbol{E}\left(\boldsymbol{D},\boldsymbol{r},t,\int_0^t f(\boldsymbol{E},\boldsymbol{r},t'),\mathrm{d}t',\ldots\right), && (153)\\
\boldsymbol{H} &= \boldsymbol{H}\left(\boldsymbol{B},\boldsymbol{r},t,\int_0^t f(\boldsymbol{H},\boldsymbol{r},t'),\mathrm{d}t',\ldots\right). && (154)
\end{aligned}$$

The user defines these by writing a Scheme script. This is discussed in more details in Section 7.3, page 229. Apart from an almost unlimited range of material properties, the user may define moving media, i.e., media whose distribution is a function of time, as well.

On the other hand, FORMS does not allow for the presence of free charges or currents, because it is a strictly scattering code, hence

$$\begin{aligned}
\rho_{\text{free}} &= 0, \quad \text{and} && (155)\\
\boldsymbol{j}_{\text{free}} &= 0, && (156)
\end{aligned}$$

and the resulting two equations

$$\begin{aligned}
\boldsymbol{\nabla}\cdot\boldsymbol{D} &= 0, \quad \text{and} && (157)\\
\boldsymbol{\nabla}\cdot\boldsymbol{B} &= 0, && (158)
\end{aligned}$$

are satisfied automatically by the solution method (FDTD), with the notable exception of multigrid edges, where special action must be taken to ensure the preservation of divergence-free field configurations. The same applies to field averaging procedures that transfer data from finer to coarser grids.

Although there is neither ϵ_0 nor μ_0 used explicitly in equations (147) through (150), we nevertheless eliminate these two pesky constants by the following substitutions

$$\begin{aligned}
\tilde{\boldsymbol{E}} &= \sqrt{\frac{\epsilon_0}{\mu_0}}\boldsymbol{E}, && (159)\\
\tilde{\boldsymbol{D}} &= \sqrt{\frac{1}{\epsilon_0\mu_0}}\boldsymbol{D}, && (160)\\
\tilde{\boldsymbol{B}} &= \frac{1}{\mu_0}\boldsymbol{B}, && (161)\\
\tilde{\boldsymbol{H}} &= \boldsymbol{H}, && (162)\\
c &= \sqrt{\frac{1}{\epsilon_0\mu_0}}. && (163)
\end{aligned}$$

The resulting charge and current free equations for $\tilde{\boldsymbol{E}}$, $\tilde{\boldsymbol{D}}$, $\tilde{\boldsymbol{B}}$, and $\tilde{\boldsymbol{H}}$ look as follows:

$$\boldsymbol{\nabla}\cdot\tilde{\boldsymbol{D}} = 0, \qquad (164)$$

$$\boldsymbol{\nabla} \cdot \tilde{\boldsymbol{B}} = 0, \tag{165}$$

$$\boldsymbol{\nabla} \times \tilde{\boldsymbol{H}} = \frac{\partial}{\partial ct} \tilde{\boldsymbol{D}}, \tag{166}$$

$$\boldsymbol{\nabla} \times \tilde{\boldsymbol{E}} = -\frac{\partial}{\partial ct} \tilde{\boldsymbol{B}}. \tag{167}$$

We are free to choose our unit of time so that $c = 1$, whereupon, dropping the tildas, we get

$$\boldsymbol{\nabla} \times \boldsymbol{H} = \partial_t \boldsymbol{D}, \tag{168}$$

$$\boldsymbol{\nabla} \times \boldsymbol{E} = -\partial_t \boldsymbol{B}, \tag{169}$$

$$\boldsymbol{E} = \boldsymbol{E}\left(\boldsymbol{D}, \boldsymbol{r}, t, \int_0^t f(\boldsymbol{E}, \boldsymbol{r}, t'), \mathrm{d}t', \ldots\right), \tag{170}$$

$$\boldsymbol{H} = \boldsymbol{H}\left(\boldsymbol{B}, \boldsymbol{r}, t, \int_0^t f(\boldsymbol{H}, \boldsymbol{r}, t'), \mathrm{d}t', \ldots\right). \tag{171}$$

Whereas these equations look exactly as (149) and (150), we find that in vacuum

$$\boldsymbol{E} = \boldsymbol{D}, \quad \text{and} \tag{172}$$

$$\boldsymbol{H} = \boldsymbol{B}. \tag{173}$$

On the other hand, because the basic equations are unchanged by this transformation, and because users are free to define $\boldsymbol{E}(\boldsymbol{D})$ and $\boldsymbol{H}(\boldsymbol{B})$ as they wish, they can keep ϵ_0 and μ_0 around.

It is useful to rewrite equations (168) and (169) in an explicit component notation, suitable for direct conversion into computer code:

$$\partial_t D_x = \partial_y H_z - \partial_z H_y, \tag{174}$$

$$\partial_t D_y = \partial_z H_x - \partial_x H_z, \tag{175}$$

$$\partial_t D_z = \partial_x H_y - \partial_y H_x, \tag{176}$$

$$\boldsymbol{D} \rightarrow \boldsymbol{E}, \tag{177}$$

$$\partial_t B_x = \partial_z E_y - \partial_y E_z, \tag{178}$$

$$\partial_t B_y = \partial_x E_z - \partial_z E_x, \tag{179}$$

$$\partial_t B_z = \partial_y E_x - \partial_x E_y, \tag{180}$$

$$\boldsymbol{B} \rightarrow \boldsymbol{H}. \tag{181}$$

3.2 Discretization

We distribute fields $\boldsymbol{E}$ and $\boldsymbol{D}$ on cell edges, and field $\boldsymbol{H}$ on cell faces, as shown in Figure 11, page 150, and as is described in Section 12.2. See also Figure 13, page 398.

The specific locations of the fields within an (i, j, k) cell, where i, j, and k correspond to the cell center, are as follows:

$$\text{Location}\,(E_x(i,j,k)) = \boldsymbol{r}_0 + \left(i, j-\frac{1}{2}, k-\frac{1}{2}\right)\Delta_g, \tag{182}$$

$$\text{Location}\,(E_y(i,j,k)) = \boldsymbol{r}_0 + \left(i-\frac{1}{2}, j, k-\frac{1}{2}\right)\Delta_g, \tag{183}$$

$$\text{Location}\,(E_z(i,j,k)) = \boldsymbol{r}_0 + \left(i-\frac{1}{2}, j-\frac{1}{2}, k\right)\Delta_g, \tag{184}$$

$$\text{Location}\,(H_x(i,j,k)) = \boldsymbol{r}_0 + \left(i-\frac{1}{2}, j, k\right)\Delta_g, \tag{185}$$

$$\text{Location}\,(H_y(i,j,k)) = \boldsymbol{r}_0 + \left(i, j-\frac{1}{2}, k\right)\Delta_g, \tag{186}$$

$$\text{Location}\,(H_z(i,j,k)) = \boldsymbol{r}_0 + \left(i, j, k-\frac{1}{2}\right)\Delta_g, \tag{187}$$

where $\boldsymbol{r}_0$ is the location of the $(0, 0, 0)$ cell center and Δ_g is the grid constant—here assumed to be the same in all three directions.

FDTD is basically a leapfrog in time and space. Assuming $\boldsymbol{E}$ and $\boldsymbol{D}$ are defined at the nth time step $t = t_E = t_0 + n\Delta t$ and $\boldsymbol{H}$ is defined at $t_H = t_E + \Delta t/2$, which we are going to mark as $(n + 1/2)^{\text{th}}$ time step, we can rewrite equations (174) through (180) as follows:

$$D_x^{n+1}(i,j,k) = D_x^n(i,j,k) + \frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right), \tag{188}$$

$$D_y^{n+1}(i,j,k) = D_y^n(i,j,k) + \frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1) - H_z^{n+1/2}(i,j,k) + H_z^{n+1/2}(i-1,j,k)\right), \tag{189}$$

$$D_z^{n+1}(i,j,k) = D_z^n(i,j,k) + \frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k) - H_x^{n+1/2}(i,j,k) + H_x^{n+1/2}(i,j-1,k)\right), \tag{190}$$

$$\boldsymbol{D}^{n+1} \to \boldsymbol{E}^{n+1}, \tag{191}$$

$$B_x^{n+3/2}(i,j,k) = B_x^{n+1/2}(i,j,k) + \frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j,k+1) - E_y^{n+1}(i,j,k) - E_z^{n+1}(i,j+1,k) + E_z^{n+1}(i,j,k)\right), \tag{192}$$

$$B_y^{n+3/2}(i,j,k) = B_y^{n+1/2}(i,j,k) + \frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i+1,j,k) - E_z^{n+1}(i,j,k) - E_x^{n+1}(i,j,k+1) + E_x^{n+1}(i,j,k)\right), \tag{193}$$

$$B_z^{n+3/2}(i,j,k) = B_z^{n+1/2}(i,j,k) + \frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i,j+1,k) - E_x^{n+1}(i,j,k) - E_y^{n+1}(i+1,j,k) + E_y^{n+1}(i,j,k)\right), \tag{194}$$

$$\boldsymbol{B}^{n+3/2} \to \boldsymbol{H}^{n+3/2}. \tag{195}$$

Certain helpful regularities are easily seen here. For example, when evaluating $\boldsymbol{\nabla} \times \boldsymbol{H}$, we reach *down* towards $i - 1$, $j - 1$ and $k - 1$, and the down-reached component is the second and the fourth in the formula. On the other hand, when evaluating $\boldsymbol{\nabla} \times \boldsymbol{E}$ we reach *up* towards $i + 1$, $j + 1$ and $k + 1$, and the up-reached component is the first and the third in the formula.

The way we have arrived at these specific equations was, first, to look at the corresponding differential equation, and then to look at the exact positions of the differentiated fields with respect to the ones being advanced. For example, H_y is attached at $(i, j-1/2, k)$, but the E_z field is attached at $(i-1/2, j-1/2, k)$. The $\partial_x E_z$ derivative has to fall exactly on $(i, j-1/2, k)$ in this case, and it will do so, if we subtract $E_z(i-1/2, j-1/2, k)$ from $E_z(i+1/2, j-1/2, k)$.

3.3 Signal Injection

Signal injection is one of these tricky and immensely confusing parts of FDTD, but it is this that makes it fly. Tremendously useful, together with PML boundaries, which are discussed in the next section. The explicit restating of FDTD equations, (188) through (194) will help us in deriving specifications for the signal injection and extraction algorithm.

We assume the following general signal property (in vacuum, hence $\boldsymbol{E} = \boldsymbol{D}$ and $\boldsymbol{H} = \boldsymbol{B}$)

$$\boldsymbol{E}(\boldsymbol{r}, t) = \boldsymbol{E}(\boldsymbol{n} \cdot \boldsymbol{r} - t), \tag{196}$$

$$\boldsymbol{H}(\boldsymbol{r}, t) = \boldsymbol{H}(\boldsymbol{n} \cdot \boldsymbol{r} - t), \tag{197}$$

where $\boldsymbol{n}$ is a normal vector in the direction of signal propagation. Let us call

$$\zeta = \boldsymbol{n} \cdot \boldsymbol{r} - t \tag{198}$$

then

$$\boldsymbol{E} = \boldsymbol{E}(\zeta), \tag{199}$$

$$\boldsymbol{H} = \boldsymbol{H}(\zeta). \tag{200}$$

It is easy to see that

$$\partial_t \boldsymbol{E} = \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} \partial_t \zeta = -\frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta}, \tag{201}$$

and similarly

$$\partial_t \boldsymbol{H} = -\frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta}. \tag{202}$$

We also have that

$$\boldsymbol{\nabla} \cdot \boldsymbol{E} = \sum_i \partial_i E_i = \sum_i \frac{\mathrm{d}E_i}{\mathrm{d}\zeta} \partial_i \zeta = \sum_i \frac{\mathrm{d}E_i}{\mathrm{d}\zeta} n_i = \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} \cdot \boldsymbol{n}, \tag{203}$$

and similarly

$$\boldsymbol{\nabla} \cdot \boldsymbol{H} = \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta} \cdot \boldsymbol{n}. \tag{204}$$

Since in vacuum $\boldsymbol{\nabla} \cdot \boldsymbol{E} = \boldsymbol{\nabla} \cdot \boldsymbol{H} = 0$, we get

$$\frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} \cdot \boldsymbol{n} = \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta} \cdot \boldsymbol{n} = 0. \tag{205}$$

This can be satisfied easily by making

$$\boldsymbol{E}(\zeta) \perp \boldsymbol{n} \quad \text{and} \quad \boldsymbol{H}(\zeta) \perp \boldsymbol{n}. \tag{206}$$

Curls are similarly evaluated. For example,

$$\boldsymbol{\nabla} \times \boldsymbol{E} = \sum_{ijk} \epsilon_{ijk} \partial_i E_j \boldsymbol{e}_k = \sum_{ijk} \epsilon_{ijk} \frac{\mathrm{d}E_j}{\mathrm{d}\zeta} \partial_i \zeta \boldsymbol{e}_k = \sum_{ijk} \epsilon_{ijk} \frac{\mathrm{d}E_j}{\mathrm{d}\zeta} n_i \boldsymbol{e}_k = \boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta}. \tag{207}$$

Similarly,

$$\boldsymbol{\nabla} \times \boldsymbol{H} = \boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta}. \tag{208}$$

The two Maxwell equations (168) and (169) become

$$\frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} = -\boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta}, \tag{209}$$

$$\frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta} = \boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta}. \tag{210}$$

These two equations are automatically satisfied when $\boldsymbol{n} \cdot \boldsymbol{n} = 1$ and when equations (205) are satisfied as well.

A simple solution of these is

$$\boldsymbol{H}(\zeta) = \boldsymbol{n} \times \boldsymbol{E}(\zeta) \tag{211}$$

Fields $\boldsymbol{H}$ and $\boldsymbol{E}$ are always in-phase. Feynman [7] discusses such a general solution in section 20.1, equations (20.25).

The FORMS user specifies the signal by providing vector $\boldsymbol{n}$ and three arbitrary functions $E_x(\zeta)$, $E_y(\zeta)$ and $E_z(\zeta)$. FORMS will normalize $\boldsymbol{n}$ and will apply

$$\boldsymbol{E}_\perp(\zeta) = \boldsymbol{E}(\zeta) - (\boldsymbol{E}(\zeta) \cdot \boldsymbol{n})\boldsymbol{n} \tag{212}$$

as well as $\boldsymbol{H}(\zeta) = \boldsymbol{n} \times \boldsymbol{E}_\perp(\zeta)$ to the total field region boundary at each time step.

Here is how.

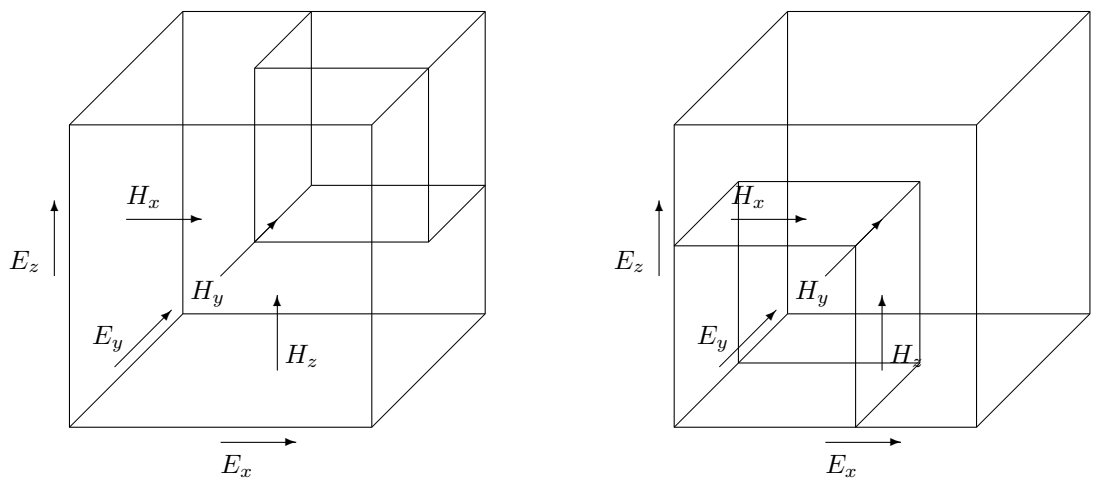


Figure 11: Location of fields in the defining corners of the injection box. The figure on the left shows the low corner, and the figure on the right shows the high corner. The smaller box within the box shows the intersection of the total field box with the corner cell.

Let us suppose, we inject the signal into a box defined by its two corners, $(i_{\rm lo}, j_{\rm lo}, k_{\rm lo})$ and $(i_{\rm hi}, j_{\rm hi}, k_{\rm hi})$. The boundary of the total field region is defined so that the cell centers, also specified by the $(i_{\rm lo}, j_{\rm lo}, k_{\rm lo})$ and $(i_{\rm hi}, j_{\rm hi}, k_{\rm hi})$ **IntVect**s *belong* to the total field box, as shown in Figure 11.

Data on the six faces of the box, as well as data on its edges and in the corners, require special handling. We are going to consider them for each face separately.[7]

west $i = i_{\rm lo}$

[7] The methodology of arriving at the equations is simple, if one thinks about it methodically. In every equation that straddles the face, all terms but one are on one side, and just one term is on the other. The trick is to bring the term that is on the wrong side onto the majority side, by either adding, if it is outside the total signal box, or subtracting, if it is inside the total signal box, the incident signal.

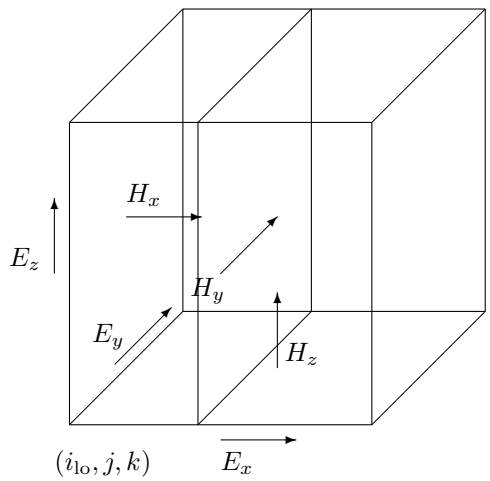


diagram $(i_{\rm lo}, j, k)$

inside $E_x(i_{\rm lo}, j, k)$, $D_x(i_{\rm lo}, j, k)$, $H_y(i_{\rm lo}, j, k)$, $B_y(i_{\rm lo}, j, k)$, $H_z(i_{\rm lo}, j, k)$, $B_z(i_{\rm lo}, j, k)$.

outside $E_y(i_{\rm lo}, j, k)$, $D_y(i_{\rm lo}, j, k)$, $E_z(i_{\rm lo}, j, k)$, $D_z(i_{\rm lo}, j, k)$ $H_x(i_{\rm lo}, j, k)$, $B_x(i_{\rm lo}, j, k)$

equations

1. The $\partial_t D_x$ equation is unaffected, because it involves H_z and H_y, which are both on the same side (but not for $k = k_{\rm lo}$ and not for $j = j_{\rm lo}$, we will have to make sure that these are handled properly, perhaps by formulas for other sides).
2. The $\partial_t D_y$ equation is affected, because one of the two H_zs it needs is inside and the other one is outside. On the other hand, both H_xs it uses are outside. So, here we have

$$\begin{aligned} D_y^{n+1}(i_{\rm lo}, j, k) &= D_y^n(i_{\rm lo}, j, k) \\ &+ \frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i_{\rm lo}, j, k) - H_x^{n+1/2}(i_{\rm lo}, j, k-1) - \underbrace{H_z^{n+1/2}(i_{\rm lo}, j, k)}_{\rm inside} + H_z^{n+1/2}(i_{\rm lo}-1, j, k)\right) \\ &+ \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i_{\rm lo}, j, k). \end{aligned} \tag{213}$$

However, we do not have to implement this correction for $k = k_{\rm lo}$, because here H_z is also outside. On the other hand, for $j = j_{\rm lo}$ and $k > k_{\rm lo}$, the correction must be implemented, because then H_z is inside. At the high end, the corrections are valid for both $k = k_{\rm hi}$ and $j = j_{\rm hi}$. In summary, the summation should run

$$\forall_{j\in[j_{\rm lo}, j_{\rm hi}], k\in[k_{\rm lo}+1, k_{\rm hi}]} \tag{214}$$

3. The $\partial_t D_z$ equation is also affected, because one of the H_ys, it uses, is inside.

$$\begin{aligned} D_z^{n+1}(i_{\rm lo}, j, k) &= D_z^n(i_{\rm lo}, j, k) \\ &+ \frac{\Delta t}{\Delta_g}\left(\underbrace{H_y^{n+1/2}(i_{\rm lo}, j, k)}_{\rm inside} - H_y^{n+1/2}(i_{\rm lo}-1, j, k) - H_x^{n+1/2}(i_{\rm lo}, j, k) + H_x^{n+1/2}(i_{\rm lo}, j-1, k)\right) \\ &- \frac{\Delta t}{\Delta_g}\{H_y^{n+1/2}\}_{\rm incident}(i_{\rm lo}, j, k). \end{aligned} \tag{215}$$

This correction does not apply to the $j = j_{\rm lo}$ case, because here H_y is outside. So, the ranges are

$$\forall_{j\in[j_{\rm lo}+1, j_{\rm hi}], k\in[k_{\rm lo}, k_{\rm hi}]} \tag{216}$$

4. The $\partial_t B_x$ equation is unaffected, because all it needs, E_y and E_z, lives on the same side as B_x.

5. The $\partial_t B_y$ equation is affected, because one of the E_zs it uses, the left one, is on the outside, whereas B_y is inside. So we have to correct this as follows.

$$
\begin{aligned}
B_y^{n+3/2}(i_{\rm lo},j,k) &= B_y^{n+1/2}(i_{\rm lo},j,k) \\
&+\frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i_{\rm lo}+1,j,k) - \underbrace{E_z^{n+1}(i_{\rm lo},j,k)}_{\text{outside}} - E_x^{n+1}(i_{\rm lo},j,k+1) + E_x^{n+1}(i_{\rm lo},j,k)\right) \\
&-\frac{\Delta t}{\Delta_g}\{E_z^{n+1}\}_{\rm incident}(i_{\rm lo},j,k).
\end{aligned} \tag{217}
$$

This correction does not apply at $j = j_{\rm lo}$, because in this case all fields are outside. At $k = k_{\rm lo}$ there is also a correction to E_x, but this should be handled by the bottom face case (to be revisited at the end). In summary, this correction applies

$$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]} \tag{218}$$

6. The $\partial_t B_z$ equation is affected because one of the E_ys is needs is outside.

$$
\begin{aligned}
B_z^{n+3/2}(i_{\rm lo},j,k) &= B_z^{n+1/2}(i_{\rm lo},j,k) \\
&+\frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i_{\rm lo},j+1,k) - E_x^{n+1}(i_{\rm lo},j,k) - E_y^{n+1}(i_{\rm lo}+1,j,k) + \underbrace{E_y^{n+1}(i_{\rm lo},j,k)}_{\text{outside}}\right) \\
&+\frac{\Delta t}{\Delta_g}\{E_y^{n+1}\}_{\rm incident}(i_{\rm lo},j,k).
\end{aligned} \tag{219}
$$

This correction does not apply at $k = k_{\rm lo}$ and at $j = j_{\rm lo}$ there should be another correction added to E_x, but this should be handled by the front face case. In summary, the correction applies

$$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]} \tag{220}$$

east $i = i_{\rm hi}$

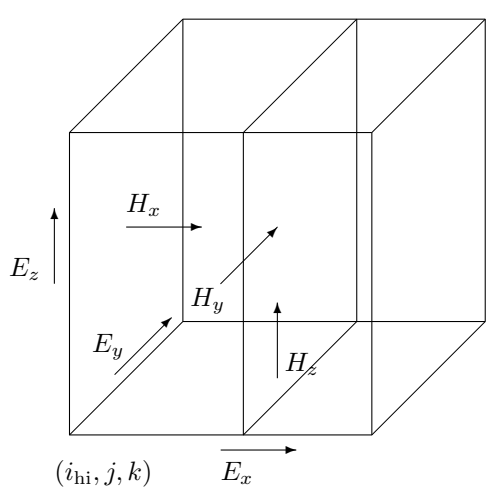


diagram $(i_{\rm hi},j,k)$

inside $E_x(i_{\rm hi},j,k)$, $E_y(i_{\rm hi},j,k)$, $E_z(i_{\rm hi},j,k)$, $H_x(i_{\rm hi},j,k)$, $H_y(i_{\rm hi},j,k)$, $H_z(i_{\rm hi},j,k)$.

outside $E_x(i_{\rm hi}+1,j,k)$, $E_y(i_{\rm hi}+1,j,k)$, $E_z(i_{\rm hi}+1,j,k)$, $H_x(i_{\rm hi}+1,j,k)$, $H_y(i_{\rm hi}+1,j,k)$, $H_z(i_{\rm hi}+1,j,k)$.

equations

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$D_y^{n+1}(i_{\rm hi}+1,j,k) = D_y^{n}(i_{\rm hi}+1,j,k)$

$$+\frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i_{\rm hi}+1,j,k)-H_x^{n+1/2}(i_{\rm hi}+1,j,k-1)-H_z^{n+1/2}(i_{\rm hi}+1,j,k)+\underbrace{H_z^{n+1/2}(i_{\rm hi},j,k)}_{\rm inside}\right)$$

$$-\frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i_{\rm hi},j,k), \tag{221}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i_{\rm hi}+1,j,k)=D_z^{n}(i_{\rm hi}+1,j,k)$$

$$+\frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i_{\rm hi}+1,j,k)-\underbrace{H_y^{n+1/2}(i_{\rm hi},j,k)}_{\rm inside}-H_x^{n+1/2}(i_{\rm hi}+1,j,k)+H_x^{n+1/2}(i_{\rm hi}+1,j-1,k)\right)$$

$$+\frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i_{\rm hi},j,k), \tag{222}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_y^{n+3/2}(i_{\rm hi},j,k)=B_y^{n+1/2}(i_{\rm hi},j,k)$$

$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{E_z^{n+1}(i_{\rm hi}+1,j,k)}_{\rm outside}-E_z^{n+1}(i_{\rm hi},j,k)-E_x^{n+1}(i_{\rm hi},j,k+1)+E_x^{n+1}(i_{\rm hi},j,k)\right)$$

$$+\frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i_{\rm hi}+1,j,k), \tag{223}$$

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i_{\rm hi},j,k)=B_z^{n+1/2}(i_{\rm hi},j,k)$$

$$+\frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i_{\rm hi},j+1,k)-E_x^{n+1}(i_{\rm hi},j,k)-\underbrace{E_y^{n+1}(i_{\rm hi}+1,j,k)}_{\rm outside}+E_y^{n+1}(i_{\rm hi},j,k)\right)$$

$$-\frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i_{\rm hi}+1,j,k). \tag{224}$$

front $j=j_{\rm lo}$

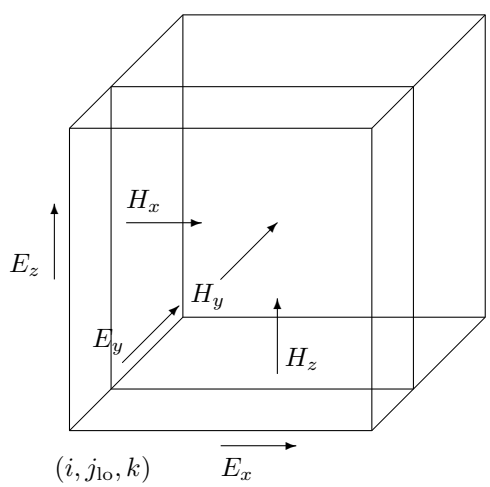


diagram $(i,j_{\rm lo},k)$

inside $E_y(i,j_{\rm lo},k)$, $D_y(i,j_{\rm lo},k)$, $H_x(i,j_{\rm lo},k)$, $B_x(i,j_{\rm lo},k)$, $H_z(i,j_{\rm lo},k)$, $B_z(i,j_{\rm lo},k)$.

outside $E_x(i,j_{\rm lo},k)$, $D_x(i,j_{\rm lo},k)$, $E_z(i,j_{\rm lo},k)$, $D_z(i,j_{\rm lo},k)$, $H_y(i,j_{\rm lo},k)$, $B_y(i,j_{\rm lo},k)$.

equations

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$D_x^{n+1}(i,j_{\rm lo},k) = D_x^n(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{H_z^{n+1/2}(i,j_{\rm lo},k)}_{\rm inside} - H_z^{n+1/2}(i,j_{\rm lo}-1,k) - H_y^{n+1/2}(i,j_{\rm lo},k) + H_y^{n+1/2}(i,j_{\rm lo},k-1)\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{225}$$
$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$$
$$D_z^{n+1}(i,j_{\rm lo},k) = D_z^n(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i,j_{\rm lo},k) - H_y^{n+1/2}(i-1,j_{\rm lo},k) - \underbrace{H_x^{n+1/2}(i,j_{\rm lo},k)}_{\rm inside} + H_x^{n+1/2}(i,j_{\rm lo}-1,k)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{226}$$
$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$$
$$B_x^{n+3/2}(i,j_{\rm lo},k) = B_x^{n+1/2}(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j_{\rm lo},k+1) - E_y^{n+1}(i,j_{\rm lo},k) - E_z^{n+1}(i,j_{\rm lo}+1,k) + \underbrace{E_z^{n+1}(i,j_{\rm lo},k)}_{\rm outside}\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{227}$$
$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$$
$$B_z^{n+3/2}(i,j_{\rm lo},k) = B_z^{n+1/2}(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i,j_{\rm lo}+1,k) - \underbrace{E_x^{n+1}(i,j_{\rm lo},k)}_{\rm outside} - E_y^{n+1}(i+1,j_{\rm lo},k) + E_y^{n+1}(i,j_{\rm lo},k)\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j_{\rm lo},k) \tag{228}$$

back $j = j_{\rm hi}$

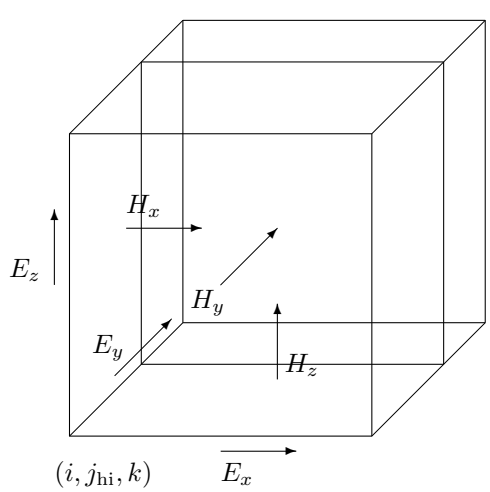


diagram $(i,j_{\rm hi},k)$

inside $E_x(i,j_{\rm hi},k)$, $E_y(i,j_{\rm hi},k)$, $E_y(i,j_{\rm hi},k)$, $H_x(i,j_{\rm hi},k)$, $H_y(i,j_{\rm hi},k)$, $H_y(i,j_{\rm hi},k)$.

outside $E_x(i,j_{\rm hi}+1,k)$, $E_y(i,j_{\rm hi}+1,k)$, $E_y(i,j_{\rm hi}+1,k)$, $H_x(i,j_{\rm hi}+1,k)$, $H_y(i,j_{\rm hi}+1,k)$, $H_y(i,j_{\rm hi}+1,k)$.

equations

$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$$

$$D_x^{n+1}(i,j_{\rm hi}+1,k) = D_x^n(i,j_{\rm hi}+1,k)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j_{\rm hi}+1,k) - \underbrace{H_z^{n+1/2}(i,j_{\rm hi},k)}_{\text{inside}} - H_y^{n+1/2}(i,j_{\rm hi}+1,k) + H_y^{n+1/2}(i,j_{\rm hi}+1,k-1)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\text{incident}}(i,j_{\rm hi},k), \tag{229}$$
$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$$
$$D_z^{n+1}(i,j_{\rm hi}+1,k) = D_z^n(i,j_{\rm hi}+1,k)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i,j_{\rm hi}+1,k) - H_y^{n+1/2}(i-1,j_{\rm hi}+1,k) - H_x^{n+1/2}(i,j_{\rm hi}+1,k) + \underbrace{H_x^{n+1/2}(i,j_{\rm hi},k)}_{\text{inside}}\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\text{incident}}(i,j_{\rm hi},k), \tag{230}$$
$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$$
$$B_x^{n+3/2}(i,j_{\rm hi},k) = B_x^{n+1/2}(i,j_{\rm hi},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j_{\rm hi},k+1) - E_y^{n+1}(i,j_{\rm hi},k) - \underbrace{E_z^{n+1}(i,j_{\rm hi}+1,k)}_{\text{outside}} + E_z^{n+1}(i,j_{\rm hi},k)\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\text{incident}}(i,j_{\rm hi}+1,k), \tag{231}$$
$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$$
$$B_z^{n+3/2}(i,j_{\rm hi},k) = B_z^{n+1/2}(i,j_{\rm hi},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{E_x^{n+1}(i,j_{\rm hi}+1,k)}_{\text{outside}} - E_x^{n+1}(i,j_{\rm hi},k) - E_y^{n+1}(i+1,j_{\rm hi},k) + E_y^{n+1}(i,j_{\rm hi},k)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\text{incident}}(i,j_{\rm hi}+1,k) \tag{232}$$

bottom $k = k_{\rm lo}$

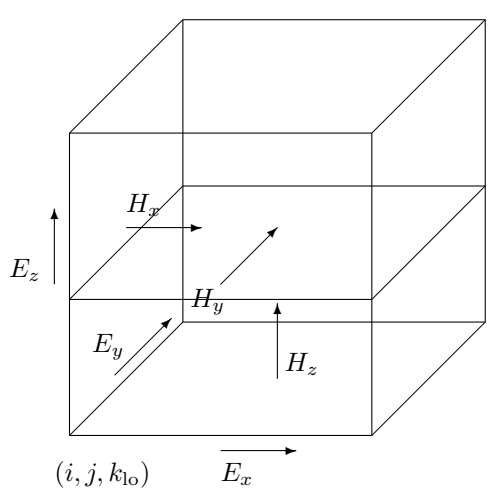


diagram $(i,j,k_{\rm lo})$

inside $E_z(i,j,k_{\rm lo})$, $D_z(i,j,k_{\rm lo})$, $H_x(i,j,k_{\rm lo})$, $B_x(i,j,k_{\rm lo})$, $H_y(i,j,k_{\rm lo})$, $B_y(i,j,k_{\rm lo})$.

outside $E_x(i,j,k_{\rm lo})$, $D_x(i,j,k_{\rm lo})$, $E_y(i,j,k_{\rm lo})$, $D_y(i,j,k_{\rm lo})$, $H_z(i,j,k_{\rm lo})$, $B_z(i,j,k_{\rm lo})$.

equations

$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$

$$D_x^{n+1}(i,j,k_{\rm lo}) = D_x^n(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j,k_{\rm lo} - H_z^{n+1/2}(i,j-1,k_{\rm lo}) - \underbrace{H_y^{n+1/2}(i,j,k_{\rm lo})}_{\text{inside}} + H_y^{n+1/2}(i,j,k_{\rm lo}-1)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\text{incident}}(i,j,k_{\rm lo}), \tag{233}$$
$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$
$$D_y^{n+1}(i,j,k_{\rm lo}) = D_y^n(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{H_x^{n+1/2}(i,j,k_{\rm lo})}_{\text{inside}} - H_x^{n+1/2}(i,j,k_{\rm lo}-1) - H_z^{n+1/2}(i,j,k_{\rm lo}) + H_z^{n+1/2}(i-1,j,k_{\rm lo})\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\text{incident}}(i,j,k_{\rm lo}), \tag{234}$$
$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$
$$B_x^{n+3/2}(i,j,k_{\rm lo}) = B_x^{n+1/2}(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j,k_{\rm lo}+1) - \underbrace{E_y^{n+1}(i,j,k_{\rm lo})}_{\text{outside}} - E_z^{n+1}(i,j+1,k_{\rm lo}) + E_z^{n+1}(i,j,k_{\rm lo})\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\text{incident}}(i,j,k_{\rm lo}), \tag{235}$$
$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$
$$B_y^{n+3/2}(i,j,k_{\rm lo}) = B_y^{n+1/2}(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i+1,j,k_{\rm lo}) - E_z^{n+1}(i,j,k_{\rm lo}) - E_x^{n+1}(i,j,k_{\rm lo}+1) + \underbrace{E_x^{n+1}(i,j,k_{\rm lo})}_{\text{outside}}\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\text{incident}}(i,j,k_{\rm lo}). \tag{236}$$

top $k = k_{\rm hi}$

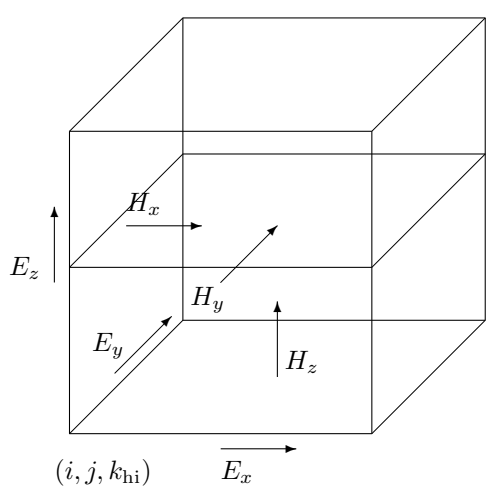


diagram $(i,j,k_{\rm hi})$

inside $E_x(i,j,k_{\rm hi})$, $E_y(i,j,k_{\rm hi})$, $E_z(i,j,k_{\rm hi})$, $H_x(i,j,k_{\rm hi})$, $H_y(i,j,k_{\rm hi})$, $H_z(i,j,k_{\rm hi})$.

outside $E_x(i,j,k_{\rm hi}+1)$, $E_y(i,j,k_{\rm hi}+1)$, $E_z(i,j,k_{\rm hi}+1)$, $H_x(i,j,k_{\rm hi}+1)$, $H_y(i,j,k_{\rm hi}+1)$, $H_z(i,j,k_{\rm hi}+1)$.

equations

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$

$$
\begin{aligned}
&D_x^{n+1}(i,j,k_{\rm hi}+1) = D_x^{n}(i,j,k_{\rm hi}+1) \\
&\quad + \frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j,k_{\rm hi}+1) - H_z^{n+1/2}(i,j-1,k_{\rm hi}+1) - H_y^{n+1/2}(i,j,k_{\rm hi}+1) + \underbrace{H_y^{n+1/2}(i,j,k_{\rm hi})}_{\text{inside}} \right) \\
&\quad - \frac{\Delta t}{\Delta_g}\left\{ H_y^{n+1/2} \right\}_{\text{incident}}(i,j,k_{\rm hi}) \qquad (237)
\end{aligned}
$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$

$$
\begin{aligned}
&D_y^{n+1}(i,j,k_{\rm hi}+1) = D_y^{n}(i,j,k_{\rm hi}+1) \\
&\quad + \frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i,j,k_{\rm hi}+1) - \underbrace{H_x^{n+1/2}(i,j,k_{\rm hi})}_{\text{inside}} - H_z^{n+1/2}(i,j,k_{\rm hi}+1) + H_z^{n+1/2}(i-1,j,k_{\rm hi}+1) \right) \\
&\quad + \frac{\Delta t}{\Delta_g}\left\{ H_x^{n+1/2} \right\}_{\text{incident}}(i,j,k_{\rm hi}) \qquad (238)
\end{aligned}
$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$

$$
\begin{aligned}
&B_x^{n+3/2}(i,j,k_{\rm hi}) = B_x^{n+1/2}(i,j,k_{\rm hi}) \\
&\quad + \frac{\Delta t}{\Delta_g}\left(\underbrace{E_y^{n+1}(i,j,k_{\rm hi}+1)}_{\text{outside}} - E_y^{n+1}(i,j,k_{\rm hi}) - E_z^{n+1}(i,j+1,k_{\rm hi}) + E_z^{n+1}(i,j,k_{\rm hi}) \right) \\
&\quad + \frac{\Delta t}{\Delta_g}\left\{ E_y^{n+1} \right\}_{\text{incident}}(i,j,k_{\rm hi}+1) \qquad (239)
\end{aligned}
$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$

$$
\begin{aligned}
&B_y^{n+3/2}(i,j,k_{\rm hi}) = B_y^{n+1/2}(i,j,k_{\rm hi}) \\
&\quad + \frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i+1,j,k_{\rm hi}) - E_z^{n+1}(i,j,k_{\rm hi}) - \underbrace{E_x^{n+1}(i,j,k_{\rm hi}+1)}_{\text{outside}} + E_x^{n+1}(i,j,k_{\rm hi}) \right) \\
&\quad - \frac{\Delta t}{\Delta_g}\left\{ E_x^{n+1} \right\}_{\text{incident}}(i,j,k_{\rm hi}+1) \qquad (240)
\end{aligned}
$$

summary Signal injection equations can be thought of as an additional correction that is put on top of normal curl computation. This leads to the following final injection formulæ:

west

$$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$$

$$D_y^{n+1}(i_{\rm lo},j,k) \leftarrow D_y^{n+1}(i_{\rm lo},j,k) + \frac{\Delta t}{\Delta_g}\left\{ H_z^{n+1/2} \right\}_{\text{incident}}(i_{\rm lo},j,k), \qquad (241)$$

$$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$$

$$D_z^{n+1}(i_{\rm lo},j,k) \leftarrow D_z^{n+1}(i_{\rm lo},j,k) - \frac{\Delta t}{\Delta_g}\left\{ H_y^{n+1/2} \right\}_{\text{incident}}(i_{\rm lo},j,k), \qquad (242)$$

$$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$$

$$B_y^{n+3/2}(i_{\rm lo},j,k) \leftarrow B_y^{n+3/2}(i_{\rm lo},j,k) - \frac{\Delta t}{\Delta_g}\left\{ E_z^{n+1} \right\}_{\text{incident}}(i_{\rm lo},j,k), \qquad (243)$$

$$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$$

$$B_z^{n+3/2}(i_{\rm lo},j,k) \leftarrow B_z^{n+3/2}(i_{\rm lo},j,k) + \frac{\Delta t}{\Delta_g}\left\{ E_y^{n+1} \right\}_{\text{incident}}(i_{\rm lo},j,k). \qquad (244)$$

east

$$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$$

$$D_y^{n+1}(i_{\rm hi}+1,j,k) \leftarrow D_y^{n+1}(i_{\rm hi}+1,j,k) - \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i_{\rm hi},j,k), \tag{245}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i_{\rm hi}+1,j,k) \leftarrow D_z^{n+1}(i_{\rm hi}+1,j,k) + \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i_{\rm hi},j,k), \tag{246}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_y^{n+3/2}(i_{\rm hi},j,k) \leftarrow B_y^{n+3/2}(i_{\rm hi},j,k) + \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i_{\rm hi}+1,j,k), \tag{247}$$

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i_{\rm hi},j,k) \leftarrow B_z^{n+3/2}(i_{\rm hi},j,k) - \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i_{\rm hi}+1,j,k). \tag{248}$$

front

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$D_x^{n+1}(i,j_{\rm lo},k) \leftarrow D_x^{n+1}(i,j_{\rm lo},k) - \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{249}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i,j_{\rm lo},k) \leftarrow D_z^{n+1}(i,j_{\rm lo},k) + \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{250}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_x^{n+3/2}(i,j_{\rm lo},k) \leftarrow B_x^{n+3/2}(i,j_{\rm lo},k) + \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{251}$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i,j_{\rm lo},k) \leftarrow B_z^{n+3/2}(i,j_{\rm lo},k) - \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j_{\rm lo},k) \tag{252}$$

back

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$D_x^{n+1}(i,j_{\rm hi}+1,k) \leftarrow D_x^{n+1}(i,j_{\rm hi}+1,k) + \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i,j_{\rm hi},k), \tag{253}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i,j_{\rm hi}+1,k) \leftarrow D_z^{n+1}(i,j_{\rm hi}+1,k) - \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j_{\rm hi},k), \tag{254}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_x^{n+3/2}(i,j_{\rm hi},k) \leftarrow B_x^{n+3/2}(i,j_{\rm hi},k) - \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i,j_{\rm hi}+1,k), \tag{255}$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i,j_{\rm hi},k) \leftarrow B_z^{n+3/2}(i,j_{\rm hi},k) + \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j_{\rm hi}+1,k). \tag{256}$$

bottom

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$

$$D_x^{n+1}(i,j,k_{\rm lo}) \leftarrow D_x^{n+1}(i,j,k_{\rm lo}) + \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{257}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$

$$D_y^{n+1}(i,j,k_{\rm lo}) \leftarrow D_y^{n+1}(i,j,k_{\rm lo}) - \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{258}$$

$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$$

$$B_x^{n+3/2}(i,j,k_{\rm lo}) \leftarrow B_x^{n+3/2}(i,j,k_{\rm lo}) - \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{259}$$

$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$

$$B_y^{n+3/2}(i,j,k_{\rm lo}) \leftarrow B_y^{n+3/2}(i,j,k_{\rm lo}) + \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j,k_{\rm lo}) \tag{260}$$

top

$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$

$$D_x^{n+1}(i,j,k_{\rm hi}+1) \leftarrow D_x^{n+1}(i,j,k_{\rm hi}+1) - \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm hi}), \tag{261}$$

$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$$

$$D_y^{n+1}(i,j,k_{\rm hi}+1) \leftarrow D_y^{n+1}(i,j,k_{\rm hi}+1) + \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm hi}), \tag{262}$$

$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$$

$$B_x^{n+3/2}(i,j,k_{\rm hi}) \leftarrow B_x^{n+3/2}(i,j,k_{\rm hi}) + \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i,j,k_{\rm hi}+1), \tag{263}$$

$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$

$$B_y^{n+3/2}(i,j,k_{\rm hi}) \leftarrow B_y^{n+3/2}(i,j,k_{\rm hi}) - \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j,k_{\rm hi}+1) \tag{264}$$

3.3.1 Relationship to Formulæ in Taflove and Hagness

One can compare the above formulæ to those in [21], section 5.8.1, pages 216–221.

First, we observe that the signs of the corrections are in agreement. But, should they be?

Second, we have to contend with different notation regarding indexes. Whereas ours are computational, theirs are logical, organized so that (1) they match ours for the electric field, and (2) $+1/2$ and $-1/2$ shifts are used for the magnetic field, with the $+1/2$ shifts placing the field in the same cell as our electric field. Or, to put it in other words, their field placements are reversed with respect to ours—electric fields are face centered and magnetic fields are edge centered.

Finally, we have to make a correction for the positioning of the total field region.

Whereas ours is defined in terms of cell centers, with cell centers incorporated *into* the total field box, theirs is defined in terms of cell corners (in our specific field mounting system). This affects the location of the correction field, as well as ranges of the complementary indexes. Briefly speaking, for the electric fields, they correct internal (with respect to the total field box) electric fields with external incident magnetic fields. We correct external electric fields with internal magnetic fields. For the magnetic fields it's the other way round. They correct external magnetic fields with internal electric fields and we correct internal magnetic fields with external electric fields.

Because of this one might expect opposite signs of their corrections compared to ours. But their procedure swaps the outside/inside marker too, so this is why the correction signs remain the same.

But what stays reversed, because of different box positioning, are index ranges. For example, their D_y correction on the west face runs for $j \in [j_{\rm lo}, j_{\rm hi}-1], k \in [k_{\rm lo}, k_{\rm hi}]$, which corresponds to the available placements for D_y in the total field box that stretches, in our notation, from $(i_{\rm lo}-1/2, j_{\rm lo}-1/2, k_{\rm lo}-1/2)$ to $(i_{\rm hi}+1/2, j_{\rm hi}+1/2, k_{\rm hi}+1/2)$. Index ranges for our correction to D_y on the same face, in turn, can be understood in terms of the available placements for H_z within the total field box that stretches from $(i_{\rm lo}, j_{\rm lo}, k_{\rm lo})$ to $(i_{\rm hi}, j_{\rm hi}, k_{\rm hi})$. Such a box placement will always miss $H_z(i,j,k_{\rm lo})$, but will capture $H_z(i,j,k_{\rm hi})$.

3.4 UPML Boundary

An Anisotropic Perfectly Matched Layer absorbing medium for truncation of FDTD lattices was first proposed by Stephen Gedney in 1996 [8, 21]. The technique has since been referred to as UPML, and this is what we're going to use in the code.

The method derives from Gedney's observation that a wave incident on a plane perpendicular to $\boldsymbol{e}_z$, and whose material properties are defined by

$$\hat{\boldsymbol{B}} = \boldsymbol{\mu} \cdot \hat{\boldsymbol{H}}, \tag{265}$$

$$\hat{\boldsymbol{D}} = \boldsymbol{\epsilon} \cdot \hat{\boldsymbol{E}}, \tag{266}$$

where the hats mark phasor vectors, and $\boldsymbol{\mu}$ and $\boldsymbol{\epsilon}$ are two identical tensors given by

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & \frac{1}{a} \end{pmatrix}, \tag{267}$$

where a is an arbitrary, possibly complex, number, that this wave does not reflect from the interface, regardless of its frequency and direction. So, if we make the medium of the plane absorbing, for example, by setting

$$a = 1 + \frac{\sigma}{\mathrm{i}\omega}, \tag{268}$$

and the plane sufficiently thick, we may produce a perfect absorbing truncation for an FDTD lattice.

Equation (267) will work on the top and bottom boundaries. On the front/back boundaries the $1/a$ term goes in the middle and on the west/east boundaries it goes into the upper left corner of the matrix.

In summary,

west/east boundary

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} \frac{1}{1+\frac{\sigma}{\mathrm{i}\omega}} & 0 & 0 \\ 0 & 1+\frac{\sigma}{\mathrm{i}\omega} & 0 \\ 0 & 0 & 1+\frac{\sigma}{\mathrm{i}\omega} \end{pmatrix} \tag{269}$$

front/back boundary

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} 1+\frac{\sigma}{\mathrm{i}\omega} & 0 & 0 \\ 0 & \frac{1}{1+\frac{\sigma}{\mathrm{i}\omega}} & 0 \\ 0 & 0 & 1+\frac{\sigma}{\mathrm{i}\omega} \end{pmatrix} \tag{270}$$

bottom/top boundary

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} 1+\frac{\sigma}{\mathrm{i}\omega} & 0 & 0 \\ 0 & 1+\frac{\sigma}{\mathrm{i}\omega} & 0 \\ 0 & 0 & \frac{1}{1+\frac{\sigma}{\mathrm{i}\omega}} \end{pmatrix} \tag{271}$$

The UPML layer is thus specified by two parameters only: the width of the layer and its σ, which should be chosen so that all incident signal becomes attenuated before it can reach the back side of the UPML boundary, where all fields are fixed at zero values. Gedney recommends to vary σ with depth into the UPML layer as follows

$$\sigma(z) = \frac{\sigma_{\max} \left| z - z_0 \right|^m}{d^m}, \tag{272}$$

where z_0 is the position of the UPML boundary, d is the thickness of the UPML layer, and $m = 4$ seems to give the best result following extensive numerical tests. $\sigma_{\max}$, the maximum value of $\sigma(z)$ within the UPML layer, is another phenomenological parameter, found to produce best results when set to

$$\sigma_{\max} \approx \frac{m+1}{150\pi\Delta_g} \tag{273}$$

where Δ_g is the grid constant *in meters*.

The reason for this fuss is that the behaviour of the incident signal on a *discrete* UPML medium is not exactly the same as on a continuous one, so some tweaking is required to make it work best. Taflove and Hagness discuss various other forms of $\sigma(z)$ in [21], Chapter 7.

Edges and corners of UPML region require special treatment. This is accomplished by expressing tensor $\boldsymbol{\mu} = \boldsymbol{\epsilon}$ as a product of (269), (270) and (271), which evaluates to:

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} s_y s_z / s_x & 0 & 0 \\ 0 & s_z s_x / s_y & 0 \\ 0 & 0 & s_x s_y / s_z \end{pmatrix}, \tag{274}$$

where

$$s_x = 1 + \frac{\sigma_x(x)}{\mathrm{i}\omega}, \tag{275}$$

$$s_y = 1 + \frac{\sigma_y(y)}{\mathrm{i}\omega}, \tag{276}$$

$$s_z = 1 + \frac{\sigma_z(z)}{\mathrm{i}\omega}, \tag{277}$$

where $\sigma_x(x)$, $\sigma_y(y)$ and $\sigma_z(z)$ are zero without[8] their *respective*, i.e., perpendicular to $\boldsymbol{e}_x$, $\boldsymbol{e}_y$, and $\boldsymbol{e}_z$, UPML regions, and given, for example, by (272) within.

The resulting Maxwell equations in the frequency domain look as follows

$$\partial_y \hat{H}_z - \partial_z \hat{H}_y = \mathrm{i}\omega \frac{s_y s_z}{s_x} \hat{E}_x, \tag{278}$$

$$\partial_z \hat{H}_x - \partial_x \hat{H}_z = \mathrm{i}\omega \frac{s_z s_x}{s_y} \hat{E}_y, \tag{279}$$

$$\partial_x \hat{H}_y - \partial_y \hat{H}_x = \mathrm{i}\omega \frac{s_x s_y}{s_z} \hat{E}_z, \tag{280}$$

$$\partial_z \hat{E}_y - \partial_y \hat{E}_z = \mathrm{i}\omega \frac{s_y s_z}{s_x} \hat{H}_x, \tag{281}$$

$$\partial_x \hat{E}_z - \partial_z \hat{E}_x = \mathrm{i}\omega \frac{s_z s_x}{s_y} \hat{H}_y, \tag{282}$$

$$\partial_y \hat{E}_x - \partial_x \hat{E}_y = \mathrm{i}\omega \frac{s_x s_y}{s_z} \hat{H}_z. \tag{283}$$

Their conversion to the time domain is non trivial, because the $1/(1 + \sigma/\mathrm{i}\omega)$ term results in a nonlinear dependence on ω. It is dispersive and characterized by a negative conductivity along the $\boldsymbol{e}_z$ direction.

Gedney (see also [21], Section 7.7.1) proposes to deal with the problem by redefining $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{B}}$ in the UPML medium as follows

$$\hat{D}_x = \frac{s_z}{s_x} \hat{E}_x, \tag{284}$$

$$\hat{D}_y = \frac{s_x}{s_y} \hat{E}_y, \tag{285}$$

$$\hat{D}_z = \frac{s_y}{s_z} \hat{E}_z, \tag{286}$$

$$\hat{B}_x = \frac{s_z}{s_x} \hat{H}_x, \tag{287}$$

$$\hat{B}_y = \frac{s_x}{s_y} \hat{H}_y, \tag{288}$$

$$\hat{B}_z = \frac{s_y}{s_z} \hat{H}_z. \tag{289}$$

This simple substitution changes equations (278) to (283) into the following simpler group of differential equations that are combined with material equations.

$$\partial_y \hat{H}_z - \partial_z \hat{H}_y = \mathrm{i}\omega s_y \hat{D}_x, \tag{290}$$

[8] The word "without" is used here in its Shakespearean sense, meaning "outside".

$$\partial_z \hat{H}_x - \partial_x \hat{H}_z = \mathrm{i}\omega s_z \hat{D}_y, \tag{291}$$
$$\partial_x \hat{H}_y - \partial_y \hat{H}_x = \mathrm{i}\omega s_x \hat{D}_z, \tag{292}$$
$$\partial_z \hat{E}_y - \partial_y \hat{E}_z = \mathrm{i}\omega s_y \hat{B}_x, \tag{293}$$
$$\partial_x \hat{E}_z - \partial_z \hat{E}_x = \mathrm{i}\omega s_z \hat{B}_y, \tag{294}$$
$$\partial_y \hat{E}_x - \partial_x \hat{E}_y = \mathrm{i}\omega s_x \hat{B}_z, \tag{295}$$

to be followed by

$$s_x \hat{D}_x = s_z \hat{E}_x, \tag{296}$$
$$s_y \hat{D}_y = s_x \hat{E}_y, \tag{297}$$
$$s_z \hat{D}_z = s_y \hat{E}_z, \tag{298}$$
$$s_x \hat{B}_x = s_z \hat{H}_x, \tag{299}$$
$$s_y \hat{B}_y = s_x \hat{H}_y, \tag{300}$$
$$s_z \hat{B}_z = s_y \hat{H}_z. \tag{301}$$

The term

$$\mathrm{i}\omega s_k = \mathrm{i}\omega \left(1 + \frac{\sigma_k}{\mathrm{i}\omega}\right) = \mathrm{i}\omega + \sigma_k \tag{302}$$

converts into

$$\partial_t + \sigma_k, \tag{303}$$

which yields

$$\partial_y H_z - \partial_z H_y = \partial_t D_x + \sigma_y D_x, \tag{304}$$
$$\partial_z H_x - \partial_x H_z = \partial_t D_y + \sigma_z D_y, \tag{305}$$
$$\partial_x H_y - \partial_y H_x = \partial_t D_z + \sigma_x D_z, \tag{306}$$
$$\partial_z E_y - \partial_y E_z = \partial_t B_x + \sigma_y B_x, \tag{307}$$
$$\partial_x E_z - \partial_z E_x = \partial_t B_y + \sigma_z B_y, \tag{308}$$
$$\partial_y E_x - \partial_x E_y = \partial_t B_z + \sigma_x B_z. \tag{309}$$

These look like Maxwell equations with currents, including a magnetic monopole current even. But the currents vanish everywhere with the exception of edges and corners.

Now, let us turn to the accompanying material equations. Let us consider the first one:

$$\left(1 + \frac{\sigma_x}{\mathrm{i}\omega}\right) \hat{D}_x = \left(1 + \frac{\sigma_z}{\mathrm{i}\omega}\right) \hat{E}_x \tag{310}$$

We multiply both sides by $\mathrm{i}\omega$, which yields

$$(\mathrm{i}\omega + \sigma_x)\, \hat{D}_x = (\mathrm{i}\omega + \sigma_z)\, \hat{E}_x. \tag{311}$$

Again, we convert $\mathrm{i}\omega$ to ∂_t and obtain the following set of equations

$$\partial_t D_x + \sigma_x D_x = \partial_t E_x + \sigma_z E_x, \tag{312}$$
$$\partial_t D_y + \sigma_y D_y = \partial_t E_y + \sigma_x E_y, \tag{313}$$
$$\partial_t D_z + \sigma_z D_z = \partial_t E_z + \sigma_y E_z, \tag{314}$$
$$\partial_t B_x + \sigma_x B_x = \partial_t H_x + \sigma_z H_x, \tag{315}$$
$$\partial_t B_y + \sigma_y B_y = \partial_t H_y + \sigma_x H_y, \tag{316}$$
$$\partial_t B_z + \sigma_z B_z = \partial_t H_z + \sigma_y H_z. \tag{317}$$

Equations (304) to (309) and (312) to (317) constitute a full set of *differential* equations that describe propagation of electromagnetic fields in the UPML medium.

Now we have to discretize them. We take equations (188) through (194) as our starting point. But the current term, $\sigma_k D_j$ should be evaluated at the same point in time as the $\boldsymbol{\nabla} \times \boldsymbol{H}$ term on the other side. This we do by averaging $\boldsymbol{D}$ between $\boldsymbol{D}^{n+1}$ and $\boldsymbol{D}^n$. So, for example, we get

$$\begin{aligned}
&\frac{D_x^{n+1}(i,j,k) - D_x^n(i,j,k)}{\Delta t} + \sigma_y \frac{D_x^{n+1}(i,j,k) + D_x^n(i,j,k)}{2} \\
&= D_x^{n+1}(i,j,k)\left(\frac{1}{\Delta t} + \frac{\sigma_y}{2}\right) - D_x^n(i,j,k)\left(\frac{1}{\Delta t} - \frac{\sigma_y}{2}\right) \\
&= D_x^{n+1}(i,j,k)\frac{2+\sigma_y\Delta t}{2\Delta t} - D_x^n(i,j,k)\frac{2-\sigma_y\Delta t}{2\Delta t} \\
&= \frac{1}{\Delta_g}\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right).
\end{aligned}$$

This trick yields the following update equations for all components of $\boldsymbol{D}$ and $\boldsymbol{B}$ fields

$$\begin{aligned}
D_x^{n+1}(i,j,k) &= \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t} D_x^n(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_y\Delta t)}\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right), \quad (318) \\
D_y^{n+1}(i,j,k) &= \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t} D_y^n(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_z\Delta t)}\left(H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1) - H_z^{n+1/2}(i,j,k) + H_z^{n+1/2}(i-1,j,k)\right), \quad (319) \\
D_z^{n+1}(i,j,k) &= \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t} D_z^n(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_x\Delta t)}\left(H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k) - H_x^{n+1/2}(i,j,k) + H_x^{n+1/2}(i,j-1,k)\right), \quad (320)
\end{aligned}$$

⟨⟨ Insert equations (324) through (326) here. ⟩⟩

$$\begin{aligned}
B_x^{n+3/2}(i,j,k) &= \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t} B_x^{n+1/2}(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_y\Delta t)}\left(E_y^{n+1}(i,j,k+1) - E_y^{n+1}(i,j,k) - E_z^{n+1}(i,j+1,k) + E_z^{n+1}(i,j,k)\right), \quad (321) \\
B_y^{n+3/2}(i,j,k) &= \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t} B_y^{n+1/2}(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_z\Delta t)}\left(E_z^{n+1}(i+1,j,k) - E_z^{n+1}(i,j,k) - E_x^{n+1}(i,j,k+1) + E_x^{n+1}(i,j,k)\right), \quad (322) \\
B_z^{n+3/2}(i,j,k) &= \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t} B_z^{n+1/2}(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_x\Delta t)}\left(E_x^{n+1}(i,j+1,k) - E_x^{n+1}(i,j,k) - E_y^{n+1}(i+1,j,k) + E_y^{n+1}(i,j,k)\right), \quad (323)
\end{aligned}$$

⟨⟨ Insert equations (327) through (329) here. ⟩⟩

It is easy to see that these equations reduce to (188) through (194) for $\sigma_i = 0$.

Equations (318) through (323) must be solved simultaneously with material equations that result from discretization of (312) through (317). Here we also have current terms such as $\sigma_x D_x$ and $\sigma_z E_x$, and these we evaluate at half-time by averaging them between n and $n+1$. For example, equation (318) discretizes as follows

$$\begin{aligned}
&\frac{D_x^{n+1}(i,j,k) - D_x^n(i,j,k)}{\Delta t} + \sigma_x \frac{D_x^{n+1}(i,j,k) + D_x^n(i,j,k)}{2} \\
&= \frac{E_x^{n+1}(i,j,k) - E_x^n(i,j,k)}{\Delta t} + \sigma_z \frac{E_x^{n+1}(i,j,k) + E_x^n(i,j,k)}{2},
\end{aligned}$$

or

$$\frac{2+\sigma_x\Delta t}{2\Delta t} D_x^{n+1}(i,j,k) - \frac{2-\sigma_x\Delta t}{2\Delta t} D_x^n(i,j,k)$$

$$= \frac{2+\sigma_z\Delta t}{2\Delta t}E_x^{n+1}(i,j,k) - \frac{2-\sigma_z\Delta t}{2\Delta t}E_x^{n}(i,j,k),$$

which solves to the following equations—this time for all components of $\boldsymbol{E}$ and $\boldsymbol{H}$.

$$E_x^{n+1}(i,j,k) = \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t}E_x^{n}(i,j,k) + \frac{2+\sigma_x\Delta t}{2+\sigma_z\Delta t}D_x^{n+1}(i,j,k) - \frac{2-\sigma_x\Delta t}{2+\sigma_z\Delta t}D_x^{n}(i,j,k), \quad (324)$$

$$E_y^{n+1}(i,j,k) = \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t}E_y^{n}(i,j,k) + \frac{2+\sigma_y\Delta t}{2+\sigma_x\Delta t}D_y^{n+1}(i,j,k) - \frac{2-\sigma_y\Delta t}{2+\sigma_x\Delta t}D_y^{n}(i,j,k), \quad (325)$$

$$E_z^{n+1}(i,j,k) = \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t}E_z^{n}(i,j,k) + \frac{2+\sigma_z\Delta t}{2+\sigma_y\Delta t}D_z^{n+1}(i,j,k) - \frac{2-\sigma_z\Delta t}{2+\sigma_y\Delta t}D_z^{n}(i,j,k), \quad (326)$$

$$H_x^{n+3/2}(i,j,k) = \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t}H_x^{n+1/2}(i,j,k) + \frac{2+\sigma_x\Delta t}{2+\sigma_z\Delta t}B_x^{n+3/2}(i,j,k) - \frac{2-\sigma_x\Delta t}{2+\sigma_z\Delta t}B_x^{n+1/2}(i,j,k), \quad (327)$$

$$H_y^{n+3/2}(i,j,k) = \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t}H_y^{n+1/2}(i,j,k) + \frac{2+\sigma_y\Delta t}{2+\sigma_x\Delta t}B_y^{n+3/2}(i,j,k) - \frac{2-\sigma_y\Delta t}{2+\sigma_x\Delta t}B_y^{n+1/2}(i,j,k), \quad (328)$$

$$H_z^{n+3/2}(i,j,k) = \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t}H_z^{n+1/2}(i,j,k) + \frac{2+\sigma_z\Delta t}{2+\sigma_y\Delta t}B_z^{n+3/2}(i,j,k) - \frac{2-\sigma_z\Delta t}{2+\sigma_y\Delta t}B_z^{n+1/2}(i,j,k). \quad (329)$$

This method forces us to keep $\boldsymbol{D}_{\text{old}}$ and $\boldsymbol{B}_{\text{old}}$ around, as well as the latest $\boldsymbol{D}$ and $\boldsymbol{B}$. For $\sigma_i = 0$, equation (324) through (329) reduce to

$$\begin{aligned} \boldsymbol{E}^{n+1} - \boldsymbol{E}^{n} &= \boldsymbol{D}^{n+1} - \boldsymbol{D}^{n}, \\ \boldsymbol{H}^{n+3/2} - \boldsymbol{H}^{n+1/2} &= \boldsymbol{B}^{n+3/2} - \boldsymbol{B}^{n+1/2}, \end{aligned}$$

which are satisfied automatically wherever $\boldsymbol{E} = \boldsymbol{D}$ and $\boldsymbol{H} = \boldsymbol{B}$, which is the case in vacuum.

As we have remarked above, the field update equations (318) through (323) reduce to plain Maxwell for $\sigma_k = 0$. This can be checked with **if** statements, but this would be very costly, because **if**s are expensive. It is cheaper computationally to introduce the following one-dimensional arrays:

$$C_{3x}(i) = 2 + \sigma_x(i)\Delta t, \quad (330)$$

$$C_{4x}(i) = 2 - \sigma_x(i)\Delta t, \quad (331)$$

$$C_{0x}(i) = \frac{1}{C_{3x}(i)} = \frac{1}{2+\sigma_x(i)\Delta t}, \quad (332)$$

$$C_{1x}(i) = C_{4x}(i)C_{0x}(i) = \frac{2-\sigma_x(i)\Delta t}{2+\sigma_x(i)\Delta t}, \quad (333)$$

$$C_{2x}(i) = 2\frac{\Delta t}{\Delta_g}C_{0x}(i) = \frac{2\Delta t}{\Delta_g(2+\sigma_x(i)\Delta t)}, \quad (334)$$

$$C_{3y}(j) = 2 + \sigma_y(j)\Delta t, \quad (335)$$

$$C_{4y}(j) = 2 - \sigma_y(j)\Delta t, \quad (336)$$

$$C_{0y}(j) = \frac{1}{C_{3y}(j)} = \frac{1}{2+\sigma_y(j)\Delta t}, \quad (337)$$

$$C_{1y}(j) = C_{4y}(j)C_{0y}(j) = \frac{2-\sigma_y(j)\Delta t}{2+\sigma_y(j)\Delta t}, \quad (338)$$

$$C_{2y}(j) = 2\frac{\Delta t}{\Delta_g}C_{0y}(j) = \frac{2\Delta t}{\Delta_g(2+\sigma_y(j)\Delta t)}, \quad (339)$$

$$C_{3z}(k) = 2 + \sigma_z(k)\Delta t, \quad (340)$$

$$C_{4z}(k) = 2 - \sigma_z(k)\Delta t, \quad (341)$$

$$C_{0z}(k) = \frac{1}{C_{3z}(k)} = \frac{1}{2+\sigma_z(k)\Delta t}, \quad (342)$$

$$C_{1z}(k) = C_{4z}(k)C_{0z}(k) = \frac{2-\sigma_z(k)\Delta t}{2+\sigma_z(k)\Delta t}, \quad (343)$$

$$C_{2z}(k) = 2\frac{\Delta t}{\Delta_g}C_{0z}(k) = \frac{2\Delta t}{\Delta_g(2+\sigma_z(k)\Delta t)} \quad (344)$$

In principle we may consider calculating them separately for $\boldsymbol{D}$ and for $\boldsymbol{B}$ updates, because the $\boldsymbol{D}$ and $\boldsymbol{B}$ fields are mounted differently. But, since UPMLs are not physical for starters and all we want from them is to quench any incident signal, we don't have to be so uptight about it, and we can evaluate the sigmas simply for the center of each cell.

With the help of these, equations (318) through (323) become

$$D_x^{n+1}(i,j,k) = C_{1y}(j)D_x^n(i,j,k) + C_{2y}(j)\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right), \tag{345}$$

$$D_y^{n+1}(i,j,k) = C_{1z}(k)D_y^n(i,j,k) + C_{2z}(k)\left(H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1) - H_z^{n+1/2}(i,j,k) + H_z^{n+1/2}(i-1,j,k)\right), \tag{346}$$

$$D_z^{n+1}(i,j,k) = C_{1x}(i)D_z^n(i,j,k) + C_{2x}(i)\left(H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k) - H_x^{n+1/2}(i,j,k) + H_x^{n+1/2}(i,j-1,k)\right), \tag{347}$$

⟨⟨ Insert equations (351) through (353) here. ⟩⟩

$$B_x^{n+3/2}(i,j,k) = C_{1y}(j)B_x^{n+1/2}(i,j,k) + C_{2y}(j)\left(E_y^{n+1}(i,j,k+1) - E_y^{n+1}(i,j,k) - E_z^{n+1}(i,j+1,k) + E_z^{n+1}(i,j,k)\right), \tag{348}$$

$$B_y^{n+3/2}(i,j,k) = C_{1z}(k)B_y^{n+1/2}(i,j,k) + C_{2z}(k)\left(E_z^{n+1}(i+1,j,k) - E_z^{n+1}(i,j,k) - E_x^{n+1}(i,j,k+1) + E_x^{n+1}(i,j,k)\right), \tag{349}$$

$$B_z^{n+3/2}(i,j,k) = C_{1x}(i)B_z^{n+1/2}(i,j,k) + C_{2x}(i)\left(E_x^{n+1}(i,j+1,k) - E_x^{n+1}(i,j,k) - E_y^{n+1}(i+1,j,k) + E_y^{n+1}(i,j,k)\right), \tag{350}$$

⟨⟨ Insert equations (354) through (356) here. ⟩⟩

And equations (324) through (329) become

$$E_x^{n+1}(i,j,k) = C_{1z}(k)E_x^n(i,j,k) + C_{3x}(i)C_{0z}(k)D_x^{n+1}(i,j,k) - C_{4x}(i)C_{0z}(k)D_x^n(i,j,k), \tag{351}$$

$$E_y^{n+1}(i,j,k) = C_{1x}(i)E_y^n(i,j,k) + C_{3y}(j)C_{0x}(i)D_y^{n+1}(i,j,k) - C_{4y}(j)C_{0x}(i)D_y^n(i,j,k), \tag{352}$$

$$E_z^{n+1}(i,j,k) = C_{1y}(j)E_z^n(i,j,k) + C_{3z}(k)C_{0y}(j)D_z^{n+1}(i,j,k) - C_{4z}(k)C_{0y}(j)D_z^n(i,j,k), \tag{353}$$

$$H_x^{n+3/2}(i,j,k) = C_{1z}(k)H_x^{n+1/2}(i,j,k) + C_{3x}(i)C_{0z}(k)B_x^{n+3/2}(i,j,k) - C_{4x}(i)C_{0z}(k)B_x^{n+1/2}(i,j,k), \tag{354}$$

$$H_y^{n+3/2}(i,j,k) = C_{1x}(i)H_y^{n+1/2}(i,j,k) + C_{3y}(j)C_{0x}(i)B_y^{n+3/2}(i,j,k) - C_{4y}(j)C_{0x}(i)B_y^{n+1/2}(i,j,k), \tag{355}$$

$$H_z^{n+3/2}(i,j,k) = C_{1y}(j)H_z^{n+1/2}(i,j,k) + C_{3z}(k)C_{0y}(j)B_z^{n+3/2}(i,j,k) - C_{4z}(k)C_{0y}(j)B_z^{n+1/2}(i,j,k). \tag{356}$$

Compared to the size of 3-dimensional data arrays of Forms, the one dimensional arrays $C_{0x}(i)$ through $C_{4z}(k)$ are tiny. They can therefore be evaluated and stored beforehand within each MPI process separately and used for local updates of electromagnetic fields within level 0.

For higher levels, we just use plain Maxwell equations, because higher level grids are confined to the total field region and do not touch UPML boundaries.

3.5 Fourier Transforms and Fluxes

The Fourier transform of a time-dependent function $x(t)$, given by

$$\hat{x}(f) = \int_{-\infty}^{\infty} x(t) e^{\mathrm{i}2\pi f t}\,\mathrm{d}t, \tag{357}$$

decomposes the $x(t)$ Hilbert space vector in the frequency basis, so that the reverse holds as well:

$$x(t) = \int_{-\infty}^{\infty} \hat{x}(f) e^{-\mathrm{i}2\pi f t}\,\mathrm{d}f. \tag{358}$$

Here f is the plain Hertz frequency (not an ω) and t is time. This sign and the 2π convention follows "Numerical Recipes" [19].

When expressed in terms of $\omega = 2\pi f$ the formulas become:

$$\hat{x}(\omega) = \int_{-\infty}^{\infty} x(t) e^{\mathrm{i}\omega t}\,\mathrm{d}t, \tag{359}$$

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{x}(\omega) e^{-\mathrm{i}\omega t}\,\mathrm{d}\omega. \tag{360}$$

Every field computed by FORMS, $\boldsymbol{E}$, $\boldsymbol{D}$, $\boldsymbol{H}$, $\boldsymbol{B}$ can be decomposed into its constituent harmonics. For example,

$$\hat{\boldsymbol{E}}(\boldsymbol{r}, f) = \int_{-\infty}^{\infty} \boldsymbol{E}(\boldsymbol{r}, t) e^{\mathrm{i}2\pi f t}\,\mathrm{d}t \approx \sum_{n=0}^{n_{\max}} \boldsymbol{E}^{(n)}(i,j,k) e^{\mathrm{i}2\pi f(t_0 + n\Delta t)}\Delta t, \tag{361}$$

where

$$\boldsymbol{r} = \boldsymbol{r}_0 + i\Delta x \boldsymbol{e}_x + j\Delta y \boldsymbol{e}_y + k\Delta z \boldsymbol{e}_z, \tag{362}$$

and where

$$\boldsymbol{E}(\boldsymbol{r}, t) = 0 \quad \text{for} \quad t < 0 \quad \text{and} \quad t > t_0 + n_{\max}\Delta t \tag{363}$$

within the computational domain covered by the range of (i, j, k). Also, note that i (italic) is and index, whereas $\mathrm{i} = \sqrt{-1}$ (roman).

In particular we may be interested in the flux of $\boldsymbol{E}(\boldsymbol{r}, t)$ or $\hat{\boldsymbol{E}}(\boldsymbol{r}, f)$ through some two-dimensional surface S, for example

$$\int_S \hat{\boldsymbol{E}}(\boldsymbol{r}, f)\cdot \boldsymbol{n}\,\mathrm{d}^2 S = \int_S \left(\int_{-\infty}^{\infty} \boldsymbol{E}(\boldsymbol{r}, t) e^{\mathrm{i}2\pi f t}\,\mathrm{d}t\right)\cdot \boldsymbol{n}\,\mathrm{d}^2 S = \int_{-\infty}^{\infty}\left(\int_S \boldsymbol{E}(\boldsymbol{r}, t)\cdot \boldsymbol{n}\,\mathrm{d}^2 S\right) e^{\mathrm{i}2\pi f t}\,\mathrm{d}t. \tag{364}$$

This yields a simple prescription for such quantities. At every time step we would evaluate the total flux of $\boldsymbol{E}(\boldsymbol{r}, t)$ through S, which is just a single number, and then we would accumulate it for the requested frequencies f_k for $k \in \{1, \ldots, k_{\max}\}$ in an array of $k_{\max}$ entries. When the program completes, the array could be written on a file or on standard output.

In our case, computation of Fourier fields across the whole three-dimensional domain, although it can be done in principle, would be prohibitively expensive in terms of memory—we'd have two add twice the storage for every field, per frequency. Then we would have to do this for every frequency of interest. Assuming, say, 100 such frequencies and our interest in Fourier transforms of $\boldsymbol{E}$ and $\boldsymbol{H}$ throughout the whole computational domain, we would have to allocate space for 400 additional three-dimensional (real) fields. This, clearly, can't be done. Normally, a three-dimensional computation, without the Fourier component, fills the whole available memory of the node.

If only a handful of frequencies is of interest, then the incident signal can be conditioned so as to vibrate at each of the frequencies for a certain amount of time and required characteristics collected this way—this is a memory-cheap way of doing it, because no additional data stores are needed.

Sending a whistle in a Gaussian envelope is a somewhat similar procedure. In this case though Fourier transforms of fields may have to be collected for the specific frequencies of interest, because the whistle frequency changes continuously. Such a procedure may be included in the $\boldsymbol{D}$-to-$\boldsymbol{E}$ routine and its results saved

on an auxiliary $\boldsymbol{S}$ fields—new $\boldsymbol{S}$ fields must be requested for each ω, and for each physical field (like $\boldsymbol{E}$ or $\boldsymbol{H}$) to be processed this way.

But if spectral response of the investigated system is required for frequency continuum throughout the whole domain, then such data can only be harvested at the post-processing stage by re-analyzing the already evolved system for every frequency within the range—or reaching for a spectral code, which would be more appropriate for the task.

On the other hand, if users are interested in spectral dependence of some flux quantities only, then this can be computed easily and at no significant cost, neither to memory nor to CPUs, by using equation (364) above. For this reason we will restrict ourselves in what follows to this simpler situation.

3.5.1 Energy Flux

The flux of Poynting vector across a surface is a special case that requires more consideration.

Given that $\boldsymbol{P} = \boldsymbol{E} \times \boldsymbol{H}$ we can rewrite the expression for $\hat{\boldsymbol{P}}$ as follows:

$$\begin{aligned}\hat{\boldsymbol{P}}(\boldsymbol{r}, f) &= \int_{-\infty}^{\infty} \sum_{ijk} \epsilon_{ijk} E_j(\boldsymbol{r}, t) H_k(\boldsymbol{r}, t) \boldsymbol{e}_i e^{\mathrm{i}2\pi f t}\, \mathrm{d}t \\ &= \sum_{ijk} \epsilon_{ijk} \boldsymbol{e}_i \int_{-\infty}^{\infty} E_j(\boldsymbol{r}, t) H_k(\boldsymbol{r}, t) e^{\mathrm{i}2\pi f t}\, \mathrm{d}t \\ &= \sum_{ijk} \epsilon_{ijk} \boldsymbol{e}_i \widehat{E_j H_k}(\boldsymbol{r}, f). \end{aligned} \tag{365}$$

Alas, this is not the same as

$$\sum_{ijk} \epsilon_{ijk} \boldsymbol{e}_i \hat{E}_j(\boldsymbol{r}, f) \hat{H}_k(\boldsymbol{r}, f), \tag{366}$$

because for any two functions E and H

$$\widehat{EH} \neq \hat{E}\hat{H} \tag{367}$$

in general.

How does then $\boldsymbol{E} \times \boldsymbol{H}$ relate to $\widehat{\boldsymbol{E} \times \boldsymbol{H}}$ and to $\hat{\boldsymbol{E}} \times \hat{\boldsymbol{H}}$ and what may be the physical meaning of the latter two?

From the convolution theorem we know that

$$\widehat{E_j H_k}(\boldsymbol{r}, f) = \left(\hat{E}_j \star \hat{H}_k\right)(\boldsymbol{r}, f) = \int_{-\infty}^{\infty} \hat{E}_j\left(\boldsymbol{r}, f\right) \hat{H}_k\left(\boldsymbol{r}, f - f'\right)\, \mathrm{d}f'. \tag{368}$$

This means that *every* frequency term of H_k contributes to $\widehat{E_j H_k}(\boldsymbol{r}, f)$. The power spectrum mixes all frequencies of its contributing $\boldsymbol{E}$ and $\boldsymbol{H}$.

On the other hand, Parseval's theorem, states that for any two functions E and H,

$$\int_{-\infty}^{\infty} E(t)\bar{H}(t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \hat{E}(f)\bar{\hat{H}}(f)\, \mathrm{d}f. \tag{369}$$

In other words, the scalar product of two Hilbert space vectors, $E(t)$ and $H(t)$, is invariant with respect to the change of basis.

Because in the time domain $H_k(\boldsymbol{r}, t)$ is purely real, $\bar{H}_k(\boldsymbol{r}, t) = H_k(\boldsymbol{r}, t)$ and

$$\int_{-\infty}^{\infty} E_j(\boldsymbol{r}, t) H_k(\boldsymbol{r}, t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \hat{E}_j(f)\bar{\hat{H}}_k(f)\, \mathrm{d}f. \tag{370}$$

Multiplying both sides by $\epsilon_{ijk}\boldsymbol{e}_i$ and adding for all values of i, j, and k yields the cross product,

$$\int_{-\infty}^{\infty} \boldsymbol{P}(\boldsymbol{r}, t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \boldsymbol{E}(\boldsymbol{r}, t) \times \boldsymbol{H}(\boldsymbol{r}, t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \hat{\boldsymbol{E}}(\boldsymbol{r}, f) \times \bar{\hat{\boldsymbol{H}}}(\boldsymbol{r}, f)\, \mathrm{d}f. \tag{371}$$

We can therefore employ the $\hat{\boldsymbol{E}}$ and $\hat{\boldsymbol{H}}$ fields to evaluate the total energy flux across the surface over the whole time of the simulation, although, in our case, this presents no advantage, because we already have $\boldsymbol{E}$ and $\boldsymbol{H}$, so we can evaluate this quantity directly.

But it turns out that $\hat{\boldsymbol{E}} \times \bar{\hat{\boldsymbol{H}}}$ can be used to evaluate the so called monochromatic power flux, and this comes about as follows.

Let us assume a monochromatic electric field of the form

$$\boldsymbol{E}(t) = \boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t \tag{372}$$

Taking a *finite domain* Fourier transform of this field over, say, two full vibration periods yields

$$\begin{aligned}
\hat{\boldsymbol{E}}(\omega) &= \int_0^{2T} \left(\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t\right) e^{\mathrm{i}\omega t}\,\mathrm{d}t \\
&= \int_0^{2T} \left(\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t\right)\left(\cos\omega t + \mathrm{i}\sin\omega t\right)\,\mathrm{d}t \\
&= \int_0^{2T} \left(\boldsymbol{E}_c \cos^2\omega t + \mathrm{i}\boldsymbol{E}_s \sin^2\omega t\right)\,\mathrm{d}t \\
&= T\left(\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s\right) = T\tilde{\boldsymbol{E}}(\omega).
\end{aligned} \tag{373}$$

We'll refer to $\tilde{\boldsymbol{E}}(\omega)$ as the phasor field. We see that

$$\begin{aligned}
\tilde{\boldsymbol{E}} e^{-\mathrm{i}\omega t} &= \left(\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s\right) e^{-\mathrm{i}\omega t} \\
&= \left(\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s\right)\left(\cos\omega t - \mathrm{i}\sin\omega t\right) \\
&= \boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t + \mathrm{i}\left(\boldsymbol{E}_s \cos\omega t - \boldsymbol{E}_c \sin\omega t\right).
\end{aligned} \tag{374}$$

In summary,

$$\Re\left(\tilde{\boldsymbol{E}}(\omega) e^{-\mathrm{i}\omega t}\right) = \boldsymbol{E}(t). \tag{375}$$

It is important to note that the phasor, $\tilde{\boldsymbol{E}}(\omega)$, is not the same as the *finite domain* Fourier transform $\hat{\boldsymbol{E}}(\omega)$, but they are proportional by *half of the accumulation time* on a monochromatic signal

Now, let us evaluate a cross product of $\tilde{\boldsymbol{E}}$ and $\bar{\tilde{\boldsymbol{H}}}$:

$$\begin{aligned}
\tilde{\boldsymbol{E}}(\omega) \times \bar{\tilde{\boldsymbol{H}}}(\omega) &= \left(\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s\right) \times \left(\boldsymbol{H}_c - \mathrm{i}\boldsymbol{H}_s\right) \\
&= \left(\boldsymbol{E}_c \times \boldsymbol{H}_c + \boldsymbol{E}_s \times \boldsymbol{H}_s\right) + \mathrm{i}\left(\boldsymbol{E}_s \times \boldsymbol{H}_c - \boldsymbol{E}_c \times \boldsymbol{H}_s\right).
\end{aligned} \tag{376}$$

Thus,

$$\Re\left(\tilde{\boldsymbol{E}}(\omega) \times \bar{\tilde{\boldsymbol{H}}}(\omega)\right) = \boldsymbol{E}_c \times \boldsymbol{H}_c + \boldsymbol{E}_s \times \boldsymbol{H}_s. \tag{377}$$

How does this relate to $\boldsymbol{E}(t) \times \boldsymbol{H}(t)$?

Let us observe that

$$\begin{aligned}
\boldsymbol{E}(t) \times \boldsymbol{H}(t) &= \left(\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t\right) \times \left(\boldsymbol{H}_c \cos\omega t + \boldsymbol{H}_s \sin\omega t\right) \\
&= \boldsymbol{E}_c \times \boldsymbol{H}_c \cos^2\omega t + \boldsymbol{E}_s \times \boldsymbol{H}_s \sin^2\omega t + \left(\boldsymbol{E}_c \times \boldsymbol{H}_s + \boldsymbol{E}_s \times \boldsymbol{H}_c\right)\cos\omega t \sin\omega t.
\end{aligned} \tag{378}$$

Integrating this over time between $\omega t = 0$ and $\omega t = 2\pi$ kills the $\cos\omega t \sin\omega t$ component, and converts both the $\sin^2\omega t$ and $\cos^2\omega t$ components to $T/2$, where $T = 2\pi/\omega$ is the period of vibration. In summary

$$\frac{1}{T}\int_0^T \boldsymbol{E}(t) \times \boldsymbol{H}(t)\,\mathrm{d}t = \frac{1}{2}\left(\boldsymbol{E}_c \times \boldsymbol{H}_c + \boldsymbol{E}_s \times \boldsymbol{H}_s\right) = \frac{1}{2}\Re\left(\tilde{\boldsymbol{E}}(\omega) \times \bar{\tilde{\boldsymbol{H}}}(\omega)\right). \tag{379}$$

And so we find that the Real part of the cross product of the $\tilde{\boldsymbol{E}}$ and $\tilde{\boldsymbol{H}}$ phasors (not transforms) is twice the average power flux.

Because transforms are proportional to phasors, although transforms can accumulate to very large numbers on long wavetrains, we can use their cross product, too, as long as we scale it following equation (373), or

consider ratios, such as power out to power in. The physical meaning of this operation is that we extract a monochromatic behaviour of the system from its multi-frequency signal response.

Evaluation of the flux of $\Re\left(\hat{\boldsymbol{E}}(\omega)\times\bar{\hat{\boldsymbol{H}}}(\omega)\right)/2$ across a surface is going to be costly, because we will have to accumulate Fourier transforms $\hat{\boldsymbol{E}}(\omega)$ and $\hat{\boldsymbol{H}}(\omega)$ at every plaquette of the surface and for every ω of interest, and store them as the computation unfolds. Only after we have processed the whole signal, can we evaluate $\Re\left(\hat{\boldsymbol{E}}(\omega)\times\bar{\hat{\boldsymbol{H}}}(\omega)\right)/2$ for each plaquette, and for each frequency of interest, using the accumulated quantities, and sum the local fluxes over the whole surface.

We discuss this issue in more detail, and propose a more efficient procedure, based on direct evaluation of power flux in Section 2.8.6, page 139.

3.5.2 Evaluation of Fluxes

The quantity we want to evaluate is

$$\int_S \boldsymbol{P}(\boldsymbol{r},t)\cdot\boldsymbol{n}\,\mathrm{d}^2S \tag{380}$$

for an arbitrary two-dimensional surface S given by parametric equations

$$x = x_S(u,v), \tag{381}$$
$$y = y_S(u,v), \tag{382}$$
$$z = z_S(u,v), \tag{383}$$

where $u\in[u_{\min},u_{\max}]$ and $v\in[v_{\min},v_{\max}]$ are the parameters. For example,

hemisphere

$$x(\phi,\theta) = R\cos\phi\sin\theta, \tag{384}$$
$$y(\phi,\theta) = R\sin\phi\sin\theta, \tag{385}$$
$$z(\phi,\theta) = R\cos\theta, \tag{386}$$

where the parameters are ϕ and θ (instead of u and v), and $\phi\in[-\pi/2,+\pi/2]$ whereas $\theta\in[0,\pi]$.

plane

$$x(u,v) = x_0, \tag{387}$$
$$y(u,v) = u, \tag{388}$$
$$z(u,v) = v, \tag{389}$$

where $u\in[y_{\min},y_{\max}]$ and $v\in[z_{\min},z_{\max}]$.

The first step in evaluating the flux integral (380) is to figure out the way to calculate $\boldsymbol{n}\,\mathrm{d}^2S$ in terms of u and v. On changing u or v by $\mathrm{d}u$ or $\mathrm{d}v$ respectively, the corresponding change in x, y and z is expressed in terms of two infinitesimal vectors,

$$\begin{pmatrix}\partial x(u,v)/\partial u\\ \partial y(u,v)/\partial u\\ \partial z(u,v)/\partial u\end{pmatrix}\mathrm{d}u \quad\text{and}\quad \begin{pmatrix}\partial x(u,v)/\partial v\\ \partial y(u,v)/\partial v\\ \partial z(u,v)/\partial v\end{pmatrix}\mathrm{d}v. \tag{390}$$

The surface area of a parallelogram subtended by the two vectors is equal to the length of their cross product, whereas the latter's direction is perpendicular to the parallelogram. Hence

$$\boldsymbol{n}\,\mathrm{d}^2S = \begin{pmatrix}\partial_u x(u,v)\\ \partial_u y(u,v)\\ \partial_u z(u,v)\end{pmatrix}\mathrm{d}u\times\begin{pmatrix}\partial_v x(u,v)\\ \partial_v y(u,v)\\ \partial_v z(u,v)\end{pmatrix}\mathrm{d}v = \begin{pmatrix}\partial_u y(u,v)\partial_v z(u,v)-\partial_u z(u,v)\partial_v y(u,v)\\ \partial_u z(u,v)\partial_v x(u,v)-\partial_u x(u,v)\partial_v z(u,v)\\ \partial_u x(u,v)\partial_v y(u,v)-\partial_u y(u,v)\partial_v x(u,v)\end{pmatrix}\mathrm{d}u\,\mathrm{d}v, \tag{391}$$

where $\partial_u=\partial/\partial u$ and $\partial_v=\partial/\partial v$.

The flux then is

$$\int_S \boldsymbol{P}(\boldsymbol{r},t)\cdot\boldsymbol{n}\,\mathrm{d}^2S$$
$$=\int_{v_{\min}}^{v_{\max}}\int_{u_{\min}}^{u_{\max}}\begin{pmatrix} P_x(x(u,v),y(u,v),z(u,v),t)\\ P_y(x(u,v),y(u,v),z(u,v),t)\\ P_z(x(u,v),y(u,v),z(u,v),t)\end{pmatrix}\cdot\begin{pmatrix}\partial_u y(u,v)\partial_v z(u,v)-\partial_u z(u,v)\partial_v y(u,v)\\ \partial_u z(u,v)\partial_v x(u,v)-\partial_u x(u,v)\partial_v z(u,v)\\ \partial_u x(u,v)\partial_v y(u,v)-\partial_u y(u,v)\partial_v x(u,v)\end{pmatrix}\mathrm{d}u\,\mathrm{d}v. \tag{392}$$

We can check the formula by evaluating, for example,

$$\int_S \boldsymbol{n}\cdot\boldsymbol{n}\,\mathrm{d}^2S=\mathcal{A}(S), \tag{393}$$

which is also a flux—of a kind—and which should return the area of surface S. We'll do it for the hemisphere and for the plane, as defined by equations (381–383) and (384–386) above.

hemisphere The parameterizing variables here are ϕ and θ. The two infinitesimal vectors that correspond to infinitesimal changes in ϕ and θ are

$$\begin{pmatrix}-R\sin\phi\sin\theta\\ R\cos\phi\sin\theta\\ 0\end{pmatrix}\mathrm{d}\phi\quad\text{and}\quad\begin{pmatrix}R\cos\phi\cos\theta\\ R\sin\phi\cos\theta\\ -R\sin\theta\end{pmatrix}\mathrm{d}\theta. \tag{394}$$

Their cross product is

$$\begin{pmatrix}-R^2\cos\phi\sin^2\theta\\ -R^2\sin\phi\sin^2\theta\\ -R^2\sin\theta\cos\theta\end{pmatrix}\mathrm{d}\phi\,\mathrm{d}\theta. \tag{395}$$

Let us observe that, because of how $\mathrm{d}\phi$ and $\mathrm{d}\theta$ point, the latter points downward, their cross product points inward, into the ball, not outward.

The length of the vector (and therefore also the parallelogram's area) is

$$R^2\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta=R\sin\theta\,\mathrm{d}\phi\,R\,\mathrm{d}\theta. \tag{396}$$

Vector $\boldsymbol{n}$ pointing in the same direction as vector (395) is

$$\frac{1}{R^2\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta}\begin{pmatrix}-R^2\cos\phi\sin^2\theta\\ -R^2\sin\phi\sin^2\theta\\ -R^2\sin\theta\cos\theta\end{pmatrix}\mathrm{d}\phi\,\mathrm{d}\theta=\begin{pmatrix}-\cos\phi\sin\theta\\ -\sin\phi\sin\theta\\ -\cos\theta\end{pmatrix}. \tag{397}$$

Its dot product with vector (395) is

$$\begin{pmatrix}-\cos\phi\sin\theta\\ -\sin\phi\sin\theta\\ -\cos\theta\end{pmatrix}\cdot\begin{pmatrix}-R^2\cos\phi\sin^2\theta\\ -R^2\sin\phi\sin^2\theta\\ -R^2\sin\theta\cos\theta\end{pmatrix}\mathrm{d}\phi\,\mathrm{d}\theta=R^2\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta, \tag{398}$$

and the flux integral (for $\phi\in[-\pi/2,\pi/2]$ and $\theta\in[0,\pi]$) is

$$\int_0^{\pi}\int_{-\pi/2}^{\pi/2}R\,\mathrm{d}\phi\,R\sin\theta\mathrm{d}\theta=\int_1^{-1}R\pi R(-\mathrm{d}\cos\theta)=2\pi R^2. \tag{399}$$

plane The two infinitesimal vectors that correspond to infinitesimal changes in u and v are

$$\begin{pmatrix}0\\1\\0\end{pmatrix}\mathrm{d}u\quad\text{and}\quad\begin{pmatrix}0\\0\\1\end{pmatrix}\mathrm{d}v. \tag{400}$$

Their cross product is

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \mathrm{d}u\,\mathrm{d}v. \tag{401}$$

The length of the vector (and therefore also the parallelogram's area) is

$$\mathrm{d}u\,\mathrm{d}v. \tag{402}$$

Vector $\boldsymbol{n}$ pointing in the same direction as vector (401) is

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{403}$$

Its dot product with vector (401) is

$$\mathrm{d}u\,\mathrm{d}v. \tag{404}$$

and the flux integral (for $u \in [y_{\min}, y_{\max}]$ and $v \in [z_{\min}, z_{\max}]$ is

$$\int_{z_{\min}}^{z_{\max}} \int_{y_{\min}}^{y_{\max}} \mathrm{d}u\,\mathrm{d}v = (y_{\max} - y_{\min}) \cdot (z_{\max} - z_{\min}). \tag{405}$$

The main philosophy of Forms is to let the user provide required model specifications functionally, in Scheme or in C, while the code attends to the basic machinery of (possibly multigrid) FDTD, that includes signal injection (but not signal definition) and scattered field absorption on the boundary, within its internal computational engine. The user specifies media types and their distribution, as well as the injected signal, by providing lambdas.

Similarly any auxiliary computation on the FDTD generated data is to be specified by the user. Our only task here is to figure out what it is that Forms has to provide to make this possible.

To this end we are now going to express the computation of the flux and the following computation of the flux Fourier transform for an arbitrary number of frequencies in Scheme.

The following listing illustrates the flux computation through a hemispherical surface.

```
;; $Id: Flux_1.scm,v 1.1 2008/04/30 17:43:37 gustav Exp $
;;
;; A simple shot at flux computation through a hemisphere.
;; Functions P_x_fun, P_y_fun and P_z_fun define a sample
;; vector field.
;;

(define P_x_fun
  (lambda (x y z)
    (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
      (/ x length))))

(define P_y_fun
  (lambda (x y z)
    (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
      (/ y length))))

(define P_z_fun
  (lambda (x y z)
    (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
      (/ z length))))

(define compflux
  (lambda (r_0 n_phi n_theta)
    (let* ((r 50)
```

```
        (x_0 (f64vector-ref r_0 0))
        (y_0 (f64vector-ref r_0 1))
        (z_0 (f64vector-ref r_0 2))
        (pi (* 2 (asin 1)))
        (phi_min (* -0.5 pi))
        (phi_max (* 0.5 pi))
        (theta_min 0)
        (theta_max pi)
        (d_phi (/ (- phi_max phi_min) n_phi))
        (d_theta (/ (- theta_max theta_min) n_theta))
        (flux 0))
    (do ((phi (+ phi_min (/ d_phi 2)) (+ phi d_phi)))
        ((> phi phi_max))
      (do ((theta (+ theta_min (/ d_theta 2)) (+ theta d_theta)))
          ((> theta theta_max))
        (let* ((sin_theta (sin theta))
               (n_x (* (cos phi) sin_theta))
               (n_y (* (sin phi) sin_theta))
               (n_z (* (cos theta)))
               (x (+ x_0 (* r n_x)))
               (y (+ y_0 (* r n_y)))
               (z (+ z_0 (* r n_z)))
               (d2s (* r r sin_theta d_phi d_theta))
               (P_x (P_x_fun x y z))
               (P_y (P_y_fun x y z))
               (P_z (P_z_fun x y z))
               (P_dot_n (+ (* P_x n_x) (* P_y n_y) (* P_z n_z))))
          (set! flux (+ flux (* d2s P_dot_n))))))
    flux)))
```

The function that computes the flux is `compflux`. It takes three arguments, the center of the hemisphere $\boldsymbol{r}_0$, which is an `f64vector` of three doubles, number of segments in the ϕ direction n_ϕ, and number of segments in the θ direction n_θ. Within its main `let*` clause, angle ϕ varies from $-\pi/2$ to $\pi/2$ and θ varies from 0 to π. Within the double `do` loop of the function we position ourselves *in the center* of each $\mathbf{d}\phi \wedge \mathbf{d}\theta$ plaquette—not in the corner—and evaluate $\boldsymbol{n}$ and d^2S for this location. The radius of the hemisphere is 50. Then we find its x, y and z coordinates, and call three functions that give us $\boldsymbol{P}(x, y, z)$ (time is not used in the calculation explicitly). We take the dot product of $\boldsymbol{P}$ with $\boldsymbol{n}$, multiply by the corresponding d^2S and accumulate the result on `flux`.

We can test the function as follows. Functions `P_x_fun`, `P_y_fun` and `P_z_fun` define the $\boldsymbol{n}$ field for sphere centered at $\boldsymbol{r}_0 = \mathbf{0}$. So, the result should be the surface area of the hemisphere, that is, $2\pi R^2$. And this is indeed what we get, as the following illustrates:

```
guile> (load "Flux_1.scm")
guile> (compflux (f64vector 0 0 0) 180 180)
15708.1626413537
guile> (define pi (* 2 (asin 1)))
guile> (* 2 pi 50 50)
15707.963267949
guile> (/ (- 15708.1626413537 15707.963267949) 15707.963267949)
1.26925051516877e-5
guile>
```

We find that the function evaluates the flux with 1×10^{-5} accuracy for 180 divisions (one degree each) in both directions. If we were to divide the hemisphere into smaller plaquettes, for example, a $1/8^{\text{th}}$ of a degree in each direction, the resulting accuracy would improve to 2×10^{-7}:

```
guile> (compflux (f64vector 0 0 0) 1440 1440)
15707.9663831536
guile> (/ (- 15707.9663831536 15707.963267949) 15707.963267949)
1.9832008441403e-7
guile>
```

This is a computation that users can express themselves on the FORMS input file assuming that

1. the FORMS code invokes it after every iteration;
2. the FORMS code provides functions for field evaluation at a given position and time for all fields it handles, including the user defined auxiliary fields $\boldsymbol{S}$.

The above will work for the sequential version of the code. For the parallel version, we need to add functions for data reduction on the Scheme level and define the behaviour of field evaluation functions in case the data is not available on the node. We will return to this issue later.

The flux computation presented here is not expensive. For example, to evaluate the above flux on $720 \times 720 = 518,400$ plaquettes takes...

```
guile> (current-time) (compflux (f64vector 0 0 0) 720 720) (current-time)
1209589621
15707.975728672
1209589632
guile> (- 1209589632 1209589621)
11
guile>
```

eleven seconds of unoptimized and interpreted Scheme code. But normally, we would not do it on such a large number of plaquettes, because we wouldn't have this many cells in our three-dimensional system to deliver the required resolution anyway. A $512 \times 512 \times 512$ cell job would normally run on 64 nodes, handling a cube of $128 \times 128 \times 128 = 2,097,152$ cells per node. A smaller job could work with a $256 \times 256 \times 256$ grid on eight nodes. So, a 128×128 evaluation of the flux is more likely and this takes less than a second. If the computation becomes more involved, a C routine can be substituted in place of the interpreted Scheme code. Also, we would not normally compute the flux through the whole hemisphere. A smaller spherical cap, or a small portion of a flat plane, are more likely to be used in this context, because the purpose of such evaluations is to simulate measurements.

The flux Fourier transform can be evaluated easily by making use of the Scheme's complex number arithmetic. And so, having evaluated the flux itself, we would multiply it by $\exp(-\mathrm{i}2\pi f t)\ \mathrm{d}t$ and add to the sum of such terms evaluated so far. The corresponding Scheme code may look as follows:

```
;; Forms input file
;;
;; $Id: FourierFlux.scm,v 1.4 2008/06/08 19:40:30 gustav Exp $
;;
;; Inject a plane wave signal into the total field region. Test the
;; solution by evaluating the flux of E x H across two obliquely
;; oriented circular detectors at each time step and accumulating
;; the total power absorbed.
;;
;; All functions are openly defined in Scheme.
;;
;; This version of the input file is for parallel environment: Scheme opens
;; separate log files, numbered by (proc-id).
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Auxiliary modules
;;
(use-modules (ice-9 format))
;;
;; The Grid group
;;
;; Level 0 parameters
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
```

```
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Signal group
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 0.0 0.0 1.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define pi (* 2.0 (asin 1.0)))
;;
(define wavelength 30.0)
(define half-width 110.0)
(define delay 120.0)
(define slope 0.4)
(define t_begin 0.0)
(define t_end 360.0)
;;
(define window
  (lambda (zeta)
    (* 0.25
       (+ 1.0 (tanh (* slope (+ zeta half-width delay))))
       (- 1.0 (tanh (* slope (+ zeta (- half-width) delay)))))))
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (* (window zeta)
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;; The Media group
;;
;; We do not distribute any media for this run, so
;; forms.media.distribution.lambda is undefined. This will
;; trigger a warning, but the program will run.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
;; Here we define stride only. The other three parameters are
;; calculated from our requests for t_begin and t_end in the
;; signal group.
;;
(define forms.iterate.stride 4)
;;
(define number_of_steps (* (/ (- t_end t_begin) forms.grid.level0.delta)
                           forms.iterate.stride))
;;
(define forms.iterate.number_of_steps number_of_steps)
(define forms.iterate.image_frequency number_of_steps)
(define forms.iterate.t0 t_begin)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Postprocessing group
;;
;; Elementary vector algebra
;;
(define vector-norm
  (lambda (n)
    (do ((length (vector-length n))
         (sum 0)
         (count 0 (1+ count)))
        ((>= count length) (sqrt sum))
      (set! sum (+ sum (expt (vector-ref n count) 2))))))
;;
(define vector-normalize
  (lambda (n)
    (let ((length (vector-length n)))
      (do ((norm (vector-norm n))
           (v (make-vector length 0))
           (count 0 (1+ count)))
          ((>= count length) v)
        (vector-set! v count (/ (vector-ref n count) norm))))))
;;
(define vector-normalized?
  (lambda (n)
    (= (vector-norm n) 1)))
;;
(define first-perpendicular
  (lambda (n)
    (if (= (vector-length n) 3)
        (if (vector-normalized? n)
            (let ((nx (vector-ref n 0))
                  (ny (vector-ref n 1))
                  (nz (vector-ref n 2)))
              (if (and (= nx 0) (= ny 0))
```

```
                    (vector 0 nz 0)
                    (let ((denom (sqrt (+ (* nx nx) (* ny ny)))))
                      (vector (/ ny denom) (/ (- nx) denom) 0))))
              (first-perpendicular (vector-normalize n)))
          #f)))
;;
(define cross-product
  (lambda (u v)
    (if (and (= (vector-length u) 3) (= (vector-length v) 3))
        (let ((ux (vector-ref u 0))
              (uy (vector-ref u 1))
              (uz (vector-ref u 2))
              (vx (vector-ref v 0))
              (vy (vector-ref v 1))
              (vz (vector-ref v 2)))
          (vector
           (- (* uy vz) (* uz vy))
           (- (* uz vx) (* ux vz))
           (- (* ux vy) (* uy vx))))
        #f)))
;;
(define dot-product
  (lambda (u v)
    (let ((length-u (vector-length u))
          (length-v (vector-length v)))
      (if (= length-u length-v)
          (do ((sum 0)
               (count 0 (1+ count)))
              ((>= count length-u) sum)
            (set! sum (+ sum (* (vector-ref u count) (vector-ref v count)))))
          #f))))
;;
(define vector-add
  (lambda (u v)
    (if (and (= (vector-length u) 3) (= (vector-length v) 3))
        (let ((ux (vector-ref u 0))
              (uy (vector-ref u 1))
              (uz (vector-ref u 2))
              (vx (vector-ref v 0))
              (vy (vector-ref v 1))
              (vz (vector-ref v 2)))
          (vector
           (+ ux vx)
           (+ uy vy)
           (+ uz vz)))
        #f)))
;;
(define vector-mult
  (lambda (a u)
    (if (and (complex? a) (vector? u) (= (vector-length u) 3))
        (let ((ux (vector-ref u 0))
              (uy (vector-ref u 1))
              (uz (vector-ref u 2)))
          (vector
           (* a ux)
           (* a uy)
           (* a uz)))
        #f)))
;;
```

```
(define vector-div
  (lambda (u a)
    (if (and (complex? a) (not (= a 0)))
        (vector-mult (/ 1.0 a) u)
        #f)))
;;
(define complex-conjugate
  (lambda (z)
    (make-rectangular (real-part z) (- (imag-part z)))))
;;
(define vector-complex-conjugate
  (lambda (v)
    (vector
     (complex-conjugate (vector-ref v 0))
     (complex-conjugate (vector-ref v 1))
     (complex-conjugate (vector-ref v 2)))))
;;
(define vector-real-part
  (lambda (v)
    (vector
     (real-part (vector-ref v 0))
     (real-part (vector-ref v 1))
     (real-part (vector-ref v 2)))))
;;
(define vector-imag-part
  (lambda (v)
    (vector
     (imag-part (vector-ref v 0))
     (imag-part (vector-ref v 1))
     (imag-part (vector-ref v 2)))))
;;
;; Scan your parallel environment
;;
(define my-rank (proc-id))
(define root-rank (unique-proc))
(define pool-size (num-proc))
(define my-log (open-file (format #f "scm_out.~a" my-rank) "a"))
(format my-log "Guile log:~%")
(format my-log " process number: ~a~%" my-rank)
(format my-log " pool size: ~a~%" pool-size)
;;
;; Define the detectors
;;
(define center_1 #(40.0 64.0 40.0))
(define center_2 #(88.0 64.0 40.0))
(define normal_1 #(-1 0 1))
(define normal_2 #(1 0 1))
(define radius 10.0)
(define n_points 40)
(define dx (/ (* 2.0 radius) n_points))
(define dy dx)
(define d2s (* dx dy))
(define x-lo (- (- radius (/ dx 2))))
(define y-lo x-lo)
(define x-hi (- radius (/ dx 2)))
(define y-hi x-hi)
(define radius-square (* radius radius))
;;
;; Normalize the detectors and define their
```

```
;; target surfaces
;;
(define n1 (vector-normalize normal_1))
(define n2 (vector-normalize normal_2))
(define ex1 (first-perpendicular n1))
(define ey1 (cross-product ex1 n1))
(define ex2 (first-perpendicular n2))
(define ey2 (cross-product ex2 n2))
;; Prepare data structures for Fourier analysis
;;
(define omega_1 (/ (* 2 pi) wavelength))
(define omega_0 (/ omega_1 2.0))
(define n_frequencies 10)
(define d_omega (/ (- omega_1 omega_0) (- n_frequencies 1)))
;;
(define frequencies (make-vector n_frequencies 0.0))
(define flux-vector-1 (make-vector n_frequencies 0.0))
(define flux-vector-2 (make-vector n_frequencies 0.0))
;;
(do ((count 0 (1+ count)))
    ((>= count n_frequencies))
  (vector-set! frequencies count (+ omega_0 (* count d_omega))))
;;
(define E-array-1 (make-array #(0 0 0) n_points n_points n_frequencies))
(define H-array-1 (make-array #(0 0 0) n_points n_points n_frequencies))
(define E-array-2 (make-array #(0 0 0) n_points n_points n_frequencies))
(define H-array-2 (make-array #(0 0 0) n_points n_points n_frequencies))
;;
;; Write on the log what you're going to do
;;
(format my-log "Computing flux of E(omega) x H(omega) through two photomultiplier tubes~%")
(format my-log " tube 1: radius = ~a, center = ~a, normal = ~a~%" radius center_1 n1)
(format my-log " ex = ~a, ey = ~a~%" ex1 ey1)
(format my-log " tube 2: radius = ~a, center = ~a, normal = ~a~%" radius center_2 n2)
(format my-log " ex = ~a, ey = ~a~%" ex2 ey2)
(format my-log " frequencies = ~a~%" frequencies)
(format my-log "Expected number of iterations: ~a~%" number_of_steps)
(force-output my-log)
;;
;; Define the function that accumulates the Fourier transform.
;; Similar to compute-local-flux.
;;
(define accumulate-fourier-transform
  (lambda (center n ex ey E-array H-array)
    (do ((x x-lo (+ x dx))
         (i 0 (1+ i)))
        ((> x x-hi))
      (do ((y y-lo (+ y dy))
           (j 0 (1+ j)))
          ((> y y-hi))
        (if (<= (+ (* x x) (* y y)) radius-square)
            (let* ((position (vector-add
                              center (vector-add (vector-mult x ex)
                                                 (vector-mult y ey))))
                   (x0 (vector-ref position 0))
                   (y0 (vector-ref position 1))
                   (z0 (vector-ref position 2))
                   (E (vector
                       (interpolate "Ex" x0 y0 z0)
```

```
                    (interpolate "Ey" x0 y0 z0)
                    (interpolate "Ez" x0 y0 z0)))
                (H (vector
                    (interpolate "Hx" x0 y0 z0)
                    (interpolate "Hy" x0 y0 z0)
                    (interpolate "Hz" x0 y0 z0))))
            (do ((k 0 (1+ k)))
                ((>= k n_frequencies))
              (let ((factor (* (exp (* 0-i (vector-ref frequencies k) forms.time_e)) forms.dt)))
                (array-set! E-array
                            (vector-add (array-ref E-array i j k)
                                        (vector-mult factor E))
                            i j k)
                (array-set! H-array
                            (vector-add (array-ref H-array i j k)
                                        (vector-mult factor H))
                            i j k)))))))))
;;
;; Define the function that computes local flux
;;
(define compute-local-flux
  (lambda (n E-array H-array flux-vector)
    (do ((k 0 (1+ k)))
        ((>= k n_frequencies))
      (do ((x x-lo (+ x dx))
           (i 0 (1+ i)))
          ((> x x-hi))
        (do ((y y-lo (+ y dy))
             (j 0 (1+ j)))
            ((> y y-hi))
          (if (<= (+ (* x x) (* y y)) radius-square)
              (vector-set!
               flux-vector k
               (+ (vector-ref flux-vector k)
                  (* d2s 0.5
                     (dot-product
                      n
                      (vector-real-part
                       (cross-product (array-ref E-array i j k)
                                      (vector-complex-conjugate (array-ref H-array i j k))))))))))))))
;;
;; Now the post-iteration function
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format my-log "t = ~a, accumulating Fourier transforms ... " forms.time_e)
    (accumulate-fourier-transform center_1 n1 ex1 ey1 E-array-1 H-array-1)
    (accumulate-fourier-transform center_2 n2 ex2 ey2 E-array-2 H-array-2)
    (format my-log "done.~%")
    (force-output my-log)))
;;
(define forms.post.all.lambda
  (lambda ()
    (format my-log "Computing local fluxes ... ")
    (compute-local-flux n1 E-array-1 H-array-1 flux-vector-1)
    (compute-local-flux n2 E-array-2 H-array-2 flux-vector-2)
    (format my-log "done~%")
    (force-output my-log)
    (format my-log "Collecting local fluxes from MPI processes ... ")
```

```
    (let* ((buffer-1 (make-vector pool-size 0))
           (buffer-2 (make-vector pool-size 0)))
      (do ((k 0 (1+ k)))
          ((>= k n_frequencies))
        (gather buffer-1 (vector-ref flux-vector-1 k) root-rank)
        (gather buffer-2 (vector-ref flux-vector-2 k) root-rank)
        (if (= my-rank root-rank)
            (begin
              (vector-set! flux-vector-1 k
                           (do ((count 0 (1+ count))
                                (sum 0))
                               ((>= count pool-size) sum)
                             (set! sum (+ sum (vector-ref buffer-1 count)))))
              (vector-set! flux-vector-2 k
                           (do ((count 0 (1+ count))
                                (sum 0))
                               ((>= count pool-size) sum)
                             (set! sum (+ sum (vector-ref buffer-2 count))))))))))
    (format my-log "done.~%")
    (force-output my-log)
    (format my-log " frequencies = ~a~%" frequencies)
    (format my-log " flux-vector-1 = ~a~%" flux-vector-1)
    (format my-log " flux-vector-2 = ~a~%" flux-vector-2)
    (close-port my-log)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Output group
;;
;; There is nothing here: we don't want any images dumped.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Watch group. This is just for debugging.
;;
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
(define forms.watch.fill_PML_arrays 1)
(define forms.watch.scheme 0)
;;
;; end
;;
```

The user defines `fourier_flux` prior to all other actions. The function `post_iteration` is then invoked by FORMS after every iteration and whatever actions are specified by the user are performed. Here we add to `fourier_flux` whatever `compflux` returns, multiplied by the exponential factor and Δt. The complex number accumulated in the variable bound to the `fourier_flux` atom is the sought after Fourier transform of the flux. It can be processed further, to extract real and imaginary parts, or the magnitude and the angle, or just written on standard output or a file.

> In summary, apart from the need to let users evaluate values of various fields at any point, at a given time, we need to provide users with the current time and time increment, as well.

Because the (x, y, z) point specified by the user may fall at any location within the computational cell, the returned value must be interpolated from the available grid values. So, the next section talks about such interpolations.

3.5.3 Parallel Utilities for Scheme

But before we get there, a few words about computing fluxes within a parallel job. In this case, each process, and therefore each Scheme instance (because there are as many Schemes running now as there are MPI processes), controls a subset of the computational domain. The flux surface is likely to span several such subsets.

There are two ways to handle the computation in this case.

The first one is to nominate a single node for carrying out the computation. The node would then collect required data from other nodes, as needed, and would evaluate the flux. The second one is to make every node evaluate its own portion of the flux, if the flux surface cuts through its computational sub-domain. At the end of the computation, the contributions evaluated by each process would be reduced to produce the total flux.

The second way is better, because it reduces communication between nodes, which is always going to be horrendously expensive, also, because it evaluates the flux in parallel. The strategy for the field value return function here would be to return zero, if a given point is not within the sub-domain of the local process, and an interpolated value otherwise.

To support these and other parallel operations on the Scheme level, FORMS Scheme-wraps basic inter-process communication primitives of Chombo, and makes them available to the user. The wrapper code is provided in module ⟨ MPI Functions 197 ⟩, Section 8.12, page 293, and function loads into Scheme appear in function *initialize_scheme*( ), defined in module ⟨ Initialize Scheme 199 ⟩, Section 8.14, page 8.14. Quoting from the latter:

```
scm_c_define_gsubr("num−proc", 0, 0, 0, (SCM(∗)())SCMnumProc);
scm_c_define_gsubr("proc−id", 0, 0, 0, (SCM(∗)())SCMprocID);
scm_c_define_gsubr("barrier", 0, 0, 0, (SCM(∗)())SCMbarrier);
scm_c_define_gsubr("unique−proc", 0, 0, 0, (SCM(∗)())SCMuniqueProc);
scm_c_define_gsubr("gather", 3, 0, 0, (SCM(∗)())SCMgather);
scm_c_define_gsubr("broadcast", 2, 0, 0, (SCM(∗)())SCMbroadcast);
```

The functions' synopses are the same as those of their Chombo equivalents, around which they wrap: *numProc*, *procID*, *barrier*, *uniqueProc*, *gather*, and *broadcast*.

The following code illustrates how they can be used in a Scheme script

```
(let∗ ((pool−size (num−proc))
       (my−rank (proc−id))
       (my−log (open−file (format #f "scm_out.~a" my−rank) "a"))
       (root−rank (unique−proc))
       (data (make−vector pool−size 0))
       (status (gather data (1+ my−rank) root−rank))
       (sum (if (= my−rank root−rank)
                (do ((count 0)
                     (my−sum 0))
                    ((>= count pool−size) my−sum)
                  (set! my−sum (+ my−sum (vector−ref data count)))
                  (set! count (1+ count)))
```

```
                0))
        (rcvd (broadcast sum root−rank)))
    (format my−log "pool size = ~a~%" pool−size)
    (format my−log "my rank = ~a~%" my−rank)
    (format my−log "root rank = ~a~%" root−rank)
    (format my−log "received = ~a~%" rcvd)
    (close−port my−log)
```

Definitions of `pool-size`, `my-rank` and `root-rank` are trivial and require no additional explanations. Functions `num-proc`, `proc-id` and `unique-proc` return the number of processes in the pool, the rank number of *this* process and the rank number of a root process to be used by the two collective operations supported: broadcast and gather.

We use the rank of *this* MPI process to create a log file for it, here referred to by `my-log`, so that any output generated by *this* Scheme instance can be saved on it. Otherwise it would be either lost or scrambled, because Scheme does not write on the Chombo's `pout.<proc>` files—unless specifically configured to do so: for example, instead of

```
(format #f "scm_out.~a" my−rank)
```

we could use

```
(format #f "pout.~a" my−rank)
```

The `gather` function takes three arguments: a vector of size greater than the pool size, into which the root process is to receive the data, an item to be sent, which must be either an integer or a real (and the vector entries must be of the same type), and the rank of the root process. The vector must be made and initialized before it is used by `gather`.

Finally, function `broadcast` takes two arguments: an item to be broadcast, which must be either a real or an integer, and the rank of the root process.

The semantics of `broadcast` and `gather` are the same as those of the Chombo (and eventually MPI) functions that are invoked by the wrappers. For `gather`, the root process is the only one that ends up with meaningful data in the vector, with data sent by process of rank n in the n-th position. For `broadcast` all processes end up with the same data returned by the function.[9]

When this job is run on four CPUs, and the `watch.scheme` parameter is set to 1, we may see the following on, say `pout.2`:

```
SCMnumProc: procs in pool = 4
SCMprocID: my rank = 2
SCMuniqueProc: unique proc rank = 0
SCMgather: sending 3
SCMbroadcast: received: 10
```

For the root process, the Chombo log would say a little more:

```
SCMnumProc: procs in pool = 4
SCMprocID: my rank = 0
SCMuniqueProc: unique proc rank = 0
SCMgather: sending 1
SCMgather: received 1 2 3 4
SCMbroadcast: sending: 10
SCMbroadcast: received: 10
```

And the Scheme log on, say, `scm_out.1`, looks as follows:

```
pool size = 4
my rank   = 1
root rank = 0
received  = 10
```

[9] We must remember here that Scheme passes its arguments by value, hence, the value of the `item` argument cannot change. The user may do so explicitly by evaluating `(set! item (gather item root-rank))`. On the other hand, if the argument is an array, its value is its address, and so the function *can* change argument array entries by evaluating `vector-set!` forms within its body.

3.5.4 Grid Field Interpolation on an Arbitrary Point within the Cell

Mixed Tri/Bi/Uni-Linear Interpolation

A simple three-dimensional interpolation from the eight corner points of a cube is of the form

$$f(x,y,z) = c_{xyz}xyz + c_{yz}yz + c_{zx}zx + c_{xy}xy + c_x x + c_y y + c_z z + c_0. \tag{406}$$

The eight equations for the eight cube corners are sufficient to determine the eight coefficients uniquely. For any fixed pair, (y_0, z_0), (z_0, x_0), or (x_0, y_0), the remaining dependence on x, y, or z respectively is linear.

For a given grid cell, the cube corners can be specified by using the `lo/hi` notation, with, say, x_1 meaning the high value of x within the cube and x_0 meaning the low value. This lets us write the following eight equations:

$$\begin{aligned}
f_7 &= f_{111} = f(x_1,y_1,z_1) = c_{xyz}x_1y_1z_1 + c_{yz}y_1z_1 + c_{zx}z_1x_1 + c_{xy}x_1y_1 + c_x x_1 + c_y y_1 + c_z z_1 + c_0, && (407)\\
f_6 &= f_{110} = f(x_1,y_1,z_0) = c_{xyz}x_1y_1z_0 + c_{yz}y_1z_0 + c_{zx}z_0x_1 + c_{xy}x_1y_1 + c_x x_1 + c_y y_1 + c_z z_0 + c_0, && (408)\\
f_5 &= f_{101} = f(x_1,y_0,z_1) = c_{xyz}x_1y_0z_1 + c_{yz}y_0z_1 + c_{zx}z_1x_1 + c_{xy}x_1y_0 + c_x x_1 + c_y y_0 + c_z z_1 + c_0, && (409)\\
f_4 &= f_{100} = f(x_1,y_0,z_0) = c_{xyz}x_1y_0z_0 + c_{yz}y_0z_0 + c_{zx}z_0x_1 + c_{xy}x_1y_0 + c_x x_1 + c_y y_0 + c_z z_0 + c_0, && (410)\\
f_3 &= f_{011} = f(x_0,y_1,z_1) = c_{xyz}x_0y_1z_1 + c_{yz}y_1z_1 + c_{zx}z_1x_0 + c_{xy}x_0y_1 + c_x x_0 + c_y y_1 + c_z z_1 + c_0, && (411)\\
f_2 &= f_{010} = f(x_0,y_1,z_0) = c_{xyz}x_0y_1z_0 + c_{yz}y_1z_0 + c_{zx}z_0x_0 + c_{xy}x_0y_1 + c_x x_0 + c_y y_1 + c_z z_0 + c_0, && (412)\\
f_1 &= f_{001} = f(x_0,y_0,z_1) = c_{xyz}x_0y_0z_1 + c_{yz}y_0z_1 + c_{zx}z_1x_0 + c_{xy}x_0y_0 + c_x x_0 + c_y y_0 + c_z z_1 + c_0, && (413)\\
f_0 &= f_{000} = f(x_0,y_0,z_0) = c_{xyz}x_0y_0z_0 + c_{yz}y_0z_0 + c_{zx}z_0x_0 + c_{xy}x_0y_0 + c_x x_0 + c_y y_0 + c_z z_0 + c_0, && (414)
\end{aligned}$$

where we have reduced binary forms such as 011 to their decimal equivalents, in this case 3. The same in the matrix form,

$$\begin{pmatrix}
x_1y_1z_1 & y_1z_1 & z_1x_1 & x_1y_1 & x_1 & y_1 & z_1 & 1\\
x_1y_1z_0 & y_1z_0 & z_0x_1 & x_1y_1 & x_1 & y_1 & z_0 & 1\\
x_1y_0z_1 & y_0z_1 & z_1x_1 & x_1y_0 & x_1 & y_0 & z_1 & 1\\
x_1y_0z_0 & y_0z_0 & z_0x_1 & x_1y_0 & x_1 & y_0 & z_0 & 1\\
x_0y_1z_1 & y_1z_1 & z_1x_0 & x_0y_1 & x_0 & y_1 & z_1 & 1\\
x_0y_1z_0 & y_1z_0 & z_0x_0 & x_0y_1 & x_0 & y_1 & z_0 & 1\\
x_0y_0z_1 & y_0z_1 & z_1x_0 & x_0y_0 & x_0 & y_0 & z_1 & 1\\
x_0y_0z_0 & y_0z_0 & z_0x_0 & x_0y_0 & x_0 & y_0 & z_0 & 1
\end{pmatrix}
\begin{pmatrix} c_{xyz}\\ c_{yz}\\ c_{zx}\\ c_{xy}\\ c_x\\ c_y\\ c_z\\ c_0 \end{pmatrix}
=
\begin{pmatrix} f_7\\ f_6\\ f_5\\ f_4\\ f_3\\ f_2\\ f_1\\ f_0 \end{pmatrix}. \tag{415}$$

The structure of the matrix in this equation is such that its determinant is guaranteed to be different from zero. Furthermore, if

$$\begin{aligned}
x_0 = y_0 = z_0 &= 0 \quad \text{and} && (416)\\
x_1 = y_1 = z_1 &= \Delta, && (417)
\end{aligned}$$

then we can simplify it further, namely

$$\begin{pmatrix}
\Delta^3 & \Delta^2 & \Delta^2 & \Delta^2 & \Delta & \Delta & \Delta & 1\\
0 & 0 & 0 & \Delta^2 & \Delta & \Delta & 0 & 1\\
0 & 0 & \Delta^2 & 0 & \Delta & 0 & \Delta & 1\\
0 & 0 & 0 & 0 & \Delta & 0 & 0 & 1\\
0 & \Delta^2 & 0 & 0 & 0 & \Delta & \Delta & 1\\
0 & 0 & 0 & 0 & 0 & \Delta & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & \Delta & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix} c_{xyz}\\ c_{yz}\\ c_{zx}\\ c_{xy}\\ c_x\\ c_y\\ c_z\\ c_0 \end{pmatrix}
=
\begin{pmatrix} f_7\\ f_6\\ f_5\\ f_4\\ f_3\\ f_2\\ f_1\\ f_0 \end{pmatrix}. \tag{418}$$

In this form the equations are easy to solve. Proceeding back to front (mostly) we obtain

$$\begin{aligned}
c_0 &= f_0 && (419)\\
c_z &= (f_1 - f_0)/\Delta, && (420)\\
c_y &= (f_2 - f_0)/\Delta, && (421)\\
c_x &= (f_4 - f_0)/\Delta, && (422)
\end{aligned}$$

$$
\begin{aligned}
c_{yz} &= (f_3 - f_2 - f_1 + f_0)/\Delta^2, && (423)\\
c_{zx} &= (f_5 - f_4 - f_1 + f_0)/\Delta^2, && (424)\\
c_{xy} &= (f_6 - f_4 - f_2 + f_0)/\Delta^2, && (425)\\
c_{xyz} &= (f_7 + f_1 + f_2 - f_3 + f_4 - f_5 - f_6 - f_0)/\Delta^3. && (426)
\end{aligned}
$$

We can subject this solution to various tests. The first test is to evaluate it for $x = 0$ and then for $x = \Delta$. Without much ado, we see that for $x = 0$ equation (406) becomes

$$
\begin{aligned}
f(0, y, z) &= c_{yz}yz + c_y y + c_z z + c_0\\
&= (f_3 - f_2 - f_1 + f_0)\, yz/\Delta^2\\
&\quad + (f_2 - f_0)\, y/\Delta\\
&\quad + (f_1 - f_0)\, z/\Delta\\
&\quad + f_0. && (427)
\end{aligned}
$$

Hence $f(0, y, z)$ depends on $f_3 = f_{011}$, $f_2 = f_{010}$, $f_1 = f_{001}$ and $f_0 = f_{000}$ only. These are the values of f at the corners of the $x = 0$ cube face. Let us now see if equation (406) combined with solution (419–426) similarly predicts that $f(\Delta, y, z)$ should depend on the values of f at the corners of the $x = \Delta$ cube face only. This time we find that

$$
\begin{aligned}
f(\Delta, y, z) &= c_{xyz}\Delta yz + c_{yz}yz + c_{zx}\Delta z + c_{xy}\Delta y + c_x\Delta + c_y y + c_z z + c_0\\
&= (f_7 + f_1 + f_2 - f_3 + f_4 - f_5 - f_6 - f_0)yz/\Delta^2\\
&\quad + (f_3 - f_2 - f_1 + f_0)yz/\Delta^2\\
&\quad + (f_5 - f_4 - f_1 + f_0)z/\Delta + (f_1 - f_0)z/\Delta\\
&\quad + (f_6 - f_4 - f_2 + f_0)y/\Delta + (f_2 - f_0)y/\Delta\\
&\quad + (f_4 - f_0) + f_0\\
&= (f_7 - f_6 - f_5 + f_4)yz/\Delta^2\\
&\quad + (f_6 - f_4)y/\Delta\\
&\quad + (f_5 - f_4)z/\Delta\\
&\quad + f_4. && (428)
\end{aligned}
$$

We discover that this is the same expression as equation (427) on the following substitutions:

$$
\begin{aligned}
f_0 = f_{000} &\rightarrow f_{100} = f_4, && (429)\\
f_1 = f_{001} &\rightarrow f_{101} = f_5, && (430)\\
f_2 = f_{010} &\rightarrow f_{110} = f_6, && (431)\\
f_3 = f_{011} &\rightarrow f_{111} = f_7, && (432)
\end{aligned}
$$

which corresponds indeed to $(x = 0) \rightarrow (x = \Delta)$.

This property ensures that the field is not only interpolated within the cube, but also that the interpolation can be carried out the same way in the adjacent cube, and the values obtained should match on the interface. The interpolation can be extended continously, though not smoothly, across the whole computational domain.

To ensure that the interpolating formulas are correct, we should repeat the procedure for $y = 0$ and $y = \Delta$ and then for $z = 0$ and $z = \Delta$, as well. Instead we will test the interpolation against a sample of various functional expressions that should test its accuracy in various directions.

Simple Tests

The following Scheme script prototypes the interpolating utility and lets us test it against various sample functions.

```
;;
;; Interpolate
;;
;; the grid
;;
(define x_0 1)
(define y_0 2)
(define z_0 3)
(define Delta 1)
;;
;; sample functions
;;
(define f1
  (lambda (x y z)
    (* x y z)))
;;
(define f2
  (lambda (x y z)
    (+ (* 3 x y z) (* 2 x y) (* 4 y z) (* 5 z x) (* -2 x) (* -3 y) (* -4 z))))
;;
(define f3
  (lambda (x y z)
    (let* ((wavelength 30)
           (pi (* 2 (asin 1)))
           (k (/ (* 2 pi) wavelength)))
      (* (sin (* k x)) (sin (* k y)) (sin (* k z))))))
;;
(define f4
  (lambda (x y z)
    (sqrt (+ (* x x) (* y y) (* z z)))))
;;
;; evaluation of a function on a grid point
;;
(define f_grid
  (lambda (f i j k)
    (let ((x (+ x_0 (* i Delta)))
          (y (+ y_0 (* j Delta)))
          (z (+ z_0 (* k Delta))))
      (f x y z))))
;;
;; interpolate an arbitrary function from its
;; grid point values
;;
(define interpolate
  ;;
  ;; requires: f -- the function and x, y, z the arguments
  ;; external: x_0, y_0, z_0, Delta -- define the grid
  ;; returns: interpolated value of f from the eight grid points
  ;; surrounding x, y, z.
  ;;
  (lambda (f x y z)
    (let* (;;
           ;; the low node of the box to which (x, y, z) belongs
           ;;
           (nx (inexact->exact (floor (/ (- x x_0) Delta))))
           (ny (inexact->exact (floor (/ (- y y_0) Delta))))
           (nz (inexact->exact (floor (/ (- z z_0) Delta))))
           ;;
           ;; function values on the grid
```

```
        ;;
        (f0 (f_grid f nx ny nz))
        (f1 (f_grid f nx ny (1+ nz)))
        (f2 (f_grid f nx (1+ ny) nz))
        (f3 (f_grid f nx (1+ ny) (1+ nz)))
        (f4 (f_grid f (1+ nx) ny nz))
        (f5 (f_grid f (1+ nx) ny (1+ nz)))
        (f6 (f_grid f (1+ nx) (1+ ny) nz))
        (f7 (f_grid f (1+ nx) (1+ ny) (1+ nz)))
        ;;
        ;; interpolation coefficients
        ;;
        (c0 f0)
        (cz (/ (- f1 f0) Delta))
        (cy (/ (- f2 f0) Delta))
        (cx (/ (- f4 f0) Delta))
        (cyz (/ (+ f3 (- f2) (- f1) f0) (* Delta Delta)))
        (czx (/ (+ f5 (- f4) (- f1) f0) (* Delta Delta)))
        (cxy (/ (+ f6 (- f4) (- f2) f0) (* Delta Delta)))
        (cxyz (/ (+ f7 f1 f2 (- f3) f4 (- f5) (- f6) (- f0)) (* Delta Delta Delta)))
        ;;
        ;; cell's low corner
        ;;
        (x0 (+ x_0 (* nx Delta)))
        (y0 (+ y_0 (* ny Delta)))
        (z0 (+ z_0 (* nz Delta)))
        ;;
        ;; dx, dy and dz distances into the cell from the low corner
        ;;
        (dx (- x x0))
        (dy (- y y0))
        (dz (- z z0)))
    ;;
    ;; interpolated value
    ;;
    (+ (* cxyz dx dy dz) (* cyz dy dz) (* czx dz dx) (* cxy dx dy) (* cx dx) (* cy dy) (* cz dz) c0))))
```

Functions `f1`, `f2`, `f3` and `f4` can be used to test the interpolation procedure on. Functions `f1` and `f2` are of the same form as equation (406). In this case we would expect the interpolation to return exactly the same result as `f1` and `f2` at every point of the computational domain. Because $\boldsymbol{r}_0 = (1, 2, 3)$ and because $\Delta = 1$, values interpolated on integer locations, which correspond to grid nodes, must be always the same as those returned by the original functions.

The following log shows that this is indeed the case:

```
guile> (load "interpolate.scm")
guile> (f2 3 2 2)
74
guile> (interpolate f2 3 2 2)
74
guile> (f2 3.2 2.1 2.7)
110.252
guile> (interpolate f2 3.2 2.1 2.7)
110.252
guile>
```

Function `f3` is a three-dimensional sine of length 30. This time, interpolation between the nodes of the grid, the constant of which is one, will return approximate result for points that are inside grid cells, and exact results on grid nodes, for example:

```
guile> (f3 3 2 2)
0.097240075008608
```

```
guile> (interpolate f3 3 2 2)
0.097240075008608
guile> (f3 3.2 2.1 2.7)
0.141711110094061
guile> (interpolate f3 3.2 2.1 2.7)
0.14025808634249
guile>
```

Here, the interpolation is accurate to within 1%. At other points, for example, $(3.5, 2.5, 2.5)$, the accuracy gets a little worse, down to 1.6%, but it's still quite good. This example reminds us that in the finite difference world, we cannot expect better, unless we reduce the grid constant by an order of magnitude, say, from 1.0 to 0.1, but this would increase the computational and memory cost by a thousand times.

Function `f4` returns the distance of the point from $(0, 0, 0)$, namely

$$f_4(\boldsymbol{r}) = r. \tag{433}$$

The interpolated values in this case are also about 1% accurate:

```
guile> (f4 3.3 2.8 1.4)
4.54862616621766
guile> (interpolate f4 3.3 2.8 1.4)
4.59421917486799
guile> (/ (- 4.59421917486799 4.54862616621766) 4.54862616621766)
0.0100234679624687
guile>
```

The interpolation function discussed here is implemented as a C++ function *SCMinterpolate*, that can access any principal field of Forms, in Section 8.13, page 295. The function is then loaded into Scheme in Section 8.14, page 299, as follows

```
scm_c_define_gsubr("interpolate", 4, 0, 0, (SCM (*) ()) SCMinterpolate);
```

The first argument of the function is a two character string that indicates the field to be accessed. The remaining three arguments are x, y and z of the location. Magnetic fields $\boldsymbol{H}$ and $\boldsymbol{B}$ are additionally interpolated in time between `new` and `old` instances.

The function can be invoked from a user defined `forms.post.iteration.lambda`, which does not take any arguments. If the lambda is defined, it is executed after every iteration, as implemented in Section 5.3, page 198, module ⟨ Iterate 80 ⟩. If the function returns `SCM_BOOL_F`, the iterations stop and the program exits.

The following example shows a simple invocation of *interpolate* in a Scheme script:

```
(define forms.post.iteration.lambda
  (lambda ()
    (format #t "post.iteration: time_e = ~a, dt = ~a~%" forms.time_e forms.dt)
    (let ((Ex (interpolate "Ex" 32.17 46.15 22.11))
          (Ey (interpolate "Ey" 32.17 46.15 22.11))
          (Ez (interpolate "Ez" 32.17 46.15 22.11)))
      (format #t "post.iteration: Ex = ~a~%" Ex)
      (format #t "post.iteration: Ey = ~a~%" Ey)
      (format #t "post.iteration: Ez = ~a~%" Ez))))
```

Volume Weighted Interpolation

Another simple scheme interpolates a value within the cube based on the volume of boxes formed between the point and the corners of the cube.

We enumerate field values on the cube corners as before, $f_0, \ldots, f_7$. For a given point (x, y, z) such that $x \in [x_0, x_1]$, $y \in [y_0, y_1]$, and $z \in [z_0, z_1]$, and

$$x_1 - x_0 = y_1 - y_0 = z_1 - z_0 = \Delta, \tag{434}$$

the corresponding box volumes are

$$
\begin{aligned}
V_0 &= (x-x_0)(y-y_0)(z-z_0), && (435)\\
V_1 &= (x-x_0)(y-y_0)(z_1-z), && (436)\\
V_2 &= (x-x_0)(y_1-y)(z-z_0), && (437)\\
V_3 &= (x-x_0)(y_1-y)(z_1-z), && (438)\\
V_4 &= (x_1-x)(y-y_0)(z-z_0), && (439)\\
V_5 &= (x_1-x)(y-y_0)(z_1-z), && (440)\\
V_6 &= (x_1-x)(y_1-y)(z-z_0), && (441)\\
V_7 &= (x_1-x)(y_1-y)(z_1-z). && (442)
\end{aligned}
$$

It is easy to see that

$$\sum_i V_i = (x_1-x_0)(y_1-y_0)(z_1-z_0) = \Delta^3 = V_\Delta. \tag{443}$$

So V_i/V_Δ form a set of weights, but... not for the right corners. For example, if $(x,y,z)=(x_0,y_0,z_0)$, the corresponding $V_0=0$, but we want the weight to be 1. It is easy to see that V_7 would provide us with the right weight in this case, because it is V_Δ. So, the formula we arrive at is that the weight should be determined by the volume against the *opposite* corner:

$$
\begin{aligned}
W_0(x,y,z) &= V_7/V_\Delta = (x_1-x)(y_1-y)(z_1-z)/\Delta^3, && (444)\\
W_1(x,y,z) &= V_6/V_\Delta = (x_1-x)(y_1-y)(z-z_0)/\Delta^3, && (445)\\
W_2(x,y,z) &= V_5/V_\Delta = (x_1-x)(y-y_0)(z_1-z)/\Delta^3, && (446)\\
W_3(x,y,z) &= V_4/V_\Delta = (x_1-x)(y-y_0)(z-z_0)/\Delta^3, && (447)\\
W_4(x,y,z) &= V_3/V_\Delta = (x-x_0)(y_1-y)(z_1-z)/\Delta^3, && (448)\\
W_5(x,y,z) &= V_2/V_\Delta = (x-x_0)(y_1-y)(z-z_0)/\Delta^3, && (449)\\
W_6(x,y,z) &= V_1/V_\Delta = (x-x_0)(y-y_0)(z_1-z)/\Delta^3, && (450)\\
W_7(x,y,z) &= V_0/V_\Delta = (x-x_0)(y-y_0)(z-z_0)/\Delta^3. && (451)
\end{aligned}
$$

And now

$$f(x,y,z) = \sum_i f_i W_i(x,y,z). \tag{452}$$

We test it by loading this script into Scheme:

```
;;
;; Interpolate
;;
;; the grid
;;
(define x_0 1)
(define y_0 2)
(define z_0 3)
(define Delta 1)
;;
;; sample functions
;;
(define f1
  (lambda (x y z)
    (* x y z)))
;;
(define f2
  (lambda (x y z)
    (+ (* 3 x y z) (* 2 x y) (* 4 y z) (* 5 z x) (* -2 x) (* -3 y) (* -4 z))))
;;
```

```
(define f3
  (lambda (x y z)
    (let* ((wavelength 30)
           (pi (* 2 (asin 1)))
           (k (/ (* 2 pi) wavelength)))
      (* (sin (* k x)) (sin (* k y)) (sin (* k z))))))
;;
(define f4
  (lambda (x y z)
    (sqrt (+ (* x x) (* y y) (* z z)))))
;;
;; evaluation of a function on a grid point
;;
(define f_grid
  (lambda (f i j k)
    (let ((x (+ x_0 (* i Delta)))
          (y (+ y_0 (* j Delta)))
          (z (+ z_0 (* k Delta))))
      (f x y z))))
;;
;; interpolate an arbitrary function from its
;; grid point values
;;
(define vinterpolate
  ;;
  ;; requires: f -- the function and x, y, z the arguments
  ;; external: x_0, y_0, z_0, Delta -- define the grid
  ;; returns: interpolated value of f from the eight grid points
  ;; surrounding x, y, z.
  ;;
  (lambda (f x y z)
    (let* (;;
           ;; the low node of the box to which (x, y, z) belongs
           ;;
           (nx (inexact->exact (floor (/ (- x x_0) Delta))))
           (ny (inexact->exact (floor (/ (- y y_0) Delta))))
           (nz (inexact->exact (floor (/ (- z z_0) Delta))))
           ;;
           ;; function values on the grid
           ;;
           (f0 (f_grid f nx ny nz))
           (f1 (f_grid f nx ny (1+ nz)))
           (f2 (f_grid f nx (1+ ny) nz))
           (f3 (f_grid f nx (1+ ny) (1+ nz)))
           (f4 (f_grid f (1+ nx) ny nz))
           (f5 (f_grid f (1+ nx) ny (1+ nz)))
           (f6 (f_grid f (1+ nx) (1+ ny) nz))
           (f7 (f_grid f (1+ nx) (1+ ny) (1+ nz)))
           ;;
           ;; cell's low corner
           ;;
           (x0 (+ x_0 (* nx Delta)))
           (y0 (+ y_0 (* ny Delta)))
           (z0 (+ z_0 (* nz Delta)))
           ;;
           ;; cell's high corner
           ;;
           (x1 (+ x0 Delta))
           (y1 (+ y0 Delta))
```

```
        (z1 (+ z0 Delta))
        ;;
        ;; weights
        ;;
        (one-by-delta-cube (/ 1.0 (* Delta Delta Delta)))
        (w0 (* (- x1 x) (- y1 y) (- z1 z) one-by-delta-cube))
        (w1 (* (- x1 x) (- y1 y) (- z z0) one-by-delta-cube))
        (w2 (* (- x1 x) (- y y0) (- z1 z) one-by-delta-cube))
        (w3 (* (- x1 x) (- y y0) (- z z0) one-by-delta-cube))
        (w4 (* (- x x0) (- y1 y) (- z1 z) one-by-delta-cube))
        (w5 (* (- x x0) (- y1 y) (- z z0) one-by-delta-cube))
        (w6 (* (- x x0) (- y y0) (- z1 z) one-by-delta-cube))
        (w7 (* (- x x0) (- y y0) (- z z0) one-by-delta-cube)))
    ;;
    ;; interpolated value
    ;;
    (+ (* f0 w0) (* f1 w1) (* f2 w2) (* f3 w3) (* f4 w4) (* f5 w5) (* f6 w6) (* f7 w7)))))
```

The script is similar to the one used previously, with the exception that function `interpolate` is now replaced with function `vinterpolate`, which implements the volume weighted scheme discussed in this section.

When we run the script and evaluate `vinterpolate` we find that

```
guile> (load "vinterpolate.scm")
guile> (f2 3 2 2)
74
guile> (vinterpolate f2 3 2 2)
74.0
guile> (f2 3.2 2.1 2.7)
110.252
guile> (vinterpolate f2 3.2 2.1 2.7)
110.252
guile> (f3 3 2 2)
0.097240075008608
guile> (vinterpolate f3 3 2 2)
0.097240075008608
guile> (f3 3.2 2.1 2.7)
0.141711110094061
guile> (vinterpolate f3 3.2 2.1 2.7)
0.14025808634249
guile> (f4 3.3 2.8 1.4)
4.54862616621766
guile> (vinterpolate f4 3.3 2.8 1.4)
4.59421917486799
guile>
```

The results are exactly the same as those returned previously, which tells us that the volume weighted interpolation formula is the same as (406). That it should be so, follows from the fact that equation (406) has its coefficients determined *uniquely* by the eight values $f_0, \ldots, f_7$. The volume weighted equation (452) eventually expands into a formula with terms proportional to xyz, xy, yz, zx, x, y, and z, and with coefficients determined by $f_0, \ldots, f_7$, too, and so it must be identical.

We can see this also by evaluating the term proportional to, say, f_2 using equation (406). According to equations (419–426), f_2 is contributed by c_y, c_{yz}, c_{xy}, and c_{xyz}, as follows:

$$f_2 y/\Delta - f_2 yz/\Delta^2 - f_2 xy/\Delta^2 + f_2 xyz/\Delta^3 = f_2(y\Delta^2 - yz\Delta - xy\Delta + xyz)/\Delta^3. \tag{453}$$

In turn, equations (444–451) tell us that

$$\begin{aligned} W_2(x,y,z) &= V_5/V_\Delta = (x_1 - x)(y - y_0)(z_1 - z)/\Delta^3 \\ &= (\Delta - x)(y)(\Delta - z)/\Delta^3 \\ &= y(\Delta^2 - z\Delta - x\Delta + xz)/\Delta^3 \\ &= (y\Delta^2 - yz\Delta - xy\Delta + xyz)/\Delta^3, \end{aligned} \tag{454}$$

which is indeed the same as (453).

Other Interpolation Schemes

Balsara [1] proposes interpolation formulas, which are different from (406), and which are direction dependent, namely

$$B_x(x,y,z) = a_0 + a_x x + a_y y + a_z z + a_{xx}x^2 + a_{xy}xy + a_{xz}xz, \tag{455}$$
$$B_y(x,y,z) = b_0 + b_x x + b_y y + b_z z + b_{xy}xy + b_{yy}y^2 + b_{yz}yz, \tag{456}$$
$$B_z(x,y,z) = c_0 + c_x x + c_y y + c_z z + c_{xz}xz + c_{yz}yz + c_{zz}z^2. \tag{457}$$

For a fixed $y = y_0$ and $z = z_0$, for example, $B_x(x, y_0, z_0)$ is quadratic in x, whereas our formula (406) always results in linear dependence on each variable for the other two fixed. Interpolation (455–457) allows for divergence free field reconstruction in the Balsara AMR scheme for MHD.

Zakharian *et al.* [22] also resort to quadratic interpolations,[10] but theirs are split into two one-dimensional operations. And so, in order to obtain fine grid ghost cell values for E_z, which in their scheme is mounted in the center of the $\boldsymbol{e}_x \wedge \boldsymbol{e}_y$ face,[11] on the $\boldsymbol{e}_x \wedge \boldsymbol{e}_z$ interface, they carry out piecewise quadratic interpolation along the x-axis first:

$$\tilde{E}_z^f\left(i+\frac{1}{4}, j, k-\frac{1}{2}\right) = \frac{5}{32}E_z^c\left(i-1, j, k-\frac{1}{2}\right) + \frac{15}{16}E_z^c\left(i, j, k-\frac{1}{2}\right) - \frac{3}{32}E_z^c\left(i+1, j, k-\frac{1}{2}\right), \tag{458}$$

where superscript f stands for "fine", and superscript c stands for "coarse". Then they carry out another quadratic interpolation normal to the interface, along the y-axis, using the temporary fine grid value found above:

$$E_z^g\left(i+\frac{1}{4}, j+\frac{1}{4}, k-\frac{1}{2}\right) = \frac{8}{15}\tilde{E}_z^f\left(i+\frac{1}{4}, j, k-\frac{1}{2}\right) + \frac{2}{3}E_z^f\left(i+\frac{1}{4}, j+\frac{3}{2}, k-\frac{1}{2}\right) - \frac{1}{5}E_z^f\left(i+\frac{1}{4}, j+\frac{5}{2}, k-\frac{1}{2}\right), \tag{459}$$

where superscript g stands for "ghost".

This, like other higher order schemes, have the disadvantage that data needs to be collected from more than one cell to carry out the interpolation. For the purpose of flux evaluation across an arbitrary surface, this should not be necessary, but we will revisit the issue, when discussing multigrid operations.

[10] They do not derive the expressions, referring the reader to a future paper instead. But the paper has not been published so far. The methodology is said to be based on the two earlier papers that discussed two-dimensional multigrid FDTD, [23] and [24].

[11] In our scheme $\boldsymbol{B}$ is face center mounted and $\boldsymbol{E}$ is edge center mounted.

4 Guile Wrap

The construct seen in *main* boots Guile, which rides on top of the C++ program. This is supposed to be a more portable way of doing this, compared to a somewhat nicer *scm_init_guile*, given that Guile needs to know about the C++ stack.

Function *scm_boot_guile* never returns. Consequently, we defer everything else to *inner_main*, including the handling of exits and MPI startup.

Because the GH interface is now deprecated, we have to abstain from *gh_enter*.

Function *inner_main* takes three arguments. The first one is a pointer to any additional data that we may wish to provide it with. The data is passed to it through the last argument of *scm_boot_guile*, which here is simply Λ (meaning NULL).

The variable *current_level_ptr* is defined here as global. It is set by C++ Chombo functions and then used by C++ Scheme functions to access Chombo data. We also define here *levels_ptr*, which is a pointer to the **Vector** of levels, for interlevel operations that we may wish to orchestrate from within Scheme.

`EXIT_SUCCESS` is defined on *stdlib.h*. In this program I assume that it is zero. If for some totally pathological reason it is not zero on some future system, it should be redefined in the header file to be so.

```
72 #include "Forms.H"
   static void inner_main(void *data, int argc, char *argv[]);
        /* The following is used to let Scheme draw on level grids. */
   level *current_level_ptr;
   Vector<level *> *levels_ptr;
   int main(int argc, char *argv[])
   {
     scm_boot_guile(argc, argv, inner_main, Λ);
          /* Never gets here */
     return (EXIT_SUCCESS);
   }
   ⟨ Inner Main 73 ⟩
```

5 Code Outline

The code of *inner_main*, discussed in this chunk, is bracketed by a CPP # **ifdef** that checks if the code is going to be linked with MPI libraries. If so, *MPI_Init* is called at the very beginning and *MPI_Finalize* is called at the end. The remainder of the code is enclosed within a new scope defined in between. This is to ensure that all Chombo classes, including some that call MPI, go out of scope before *MPI_Finalize* is called (cf. [4], Section 7, "Parallel Programming with Chombo").

Within this new MPI scope we introduce the program, and check if the code has been linked with the 3D Chombo libraries. If not, we flag a problem and jump to exit. This is because some Chombo-style classes (our own and contributed) used by the code, for example, curl-refluxing, work in 3D only, but do not check for it fully.

Throughout the code we use *pout*( ), declared on `parstream.H`, rather than *cout* to print messages. In the parallel context *pout*( ) writes its output on process specific files.

exit_status is defined outside the MPI scope, because it has to be available before and after MPI has gone away, at the end of the program. We use *exit*( ) instead of **return** ( ), because, as we have mentioned above, *scm_boot_guile* that runs *inner_main* never returns, so we cannot return to *main*. We have to *exit*( ) here and now.

Once we get to it, the actual lines of *inner_main* are very simple.

First we read an input file, then we run additional initializations on Scheme, some of which require data provided on the input file. Then, using information provided, we build the multi-grid structure and fill it with data. Finally, we commence the iterations.

Although the code can run from a *restart* file to continue its computations, this is not visible at the top level. Instead, if a *restart* file exists and the user has requested its use, it is loaded by function *build_levels*, discussed in module ⟨ Function *build_levels* 82 ⟩, which is responsible for generating the multi-grid and filling it with initial data.

Similarly, not only images but checkpoint files as well are generated on request by function *dump_data*, which is discussed in module ⟨ Function *dump_data* 201 ⟩.

```
73 ⟨ Inner Main 73 ⟩ ≡
   static void inner_main(void *data, int argc, char *argv[])
   {
     int exit_status = EXIT_SUCCESS;
#ifdef CH_MPI
     MPI_Init(&argc, &argv);
#endif
     {
       pout( ) << argv[0] << ":" << endl;
       pout( ) << "\tVersion␣" << PROGRAM_VERSION << endl;
       pout( ) << "\tProcess␣" << procID( ) << endl;
       if (SpaceDim ≠ 3) {
         pout( ) << "\tERROR(main):␣this␣program␣works␣in␣3D␣only" << endl;
         exit_status = ERR_BAD_DIMENSION;
         goto exit;
       }
       initialize_scheme( );
       ⟨ Process an Input File 74 ⟩
       if (exit_status = post_initialize_scheme( )) {
         pout( ) << "\tERROR(main):␣failed␣to␣initialize␣Scheme" << endl;
         goto exit;
       }
       ⟨ Build a Multigrid 78 ⟩
       ⟨ Fill the Multigrid 79 ⟩
       ⟨ Iterate 80 ⟩
     }
   exit:
```

```
#ifdef CH_MPI
    MPI_Finalize();
#endif
    pout() << "\tFinished. Exiting..." << endl;
    exit(exit_status);
  }
```

This code is cited in chunks 84 and 199.

This code is used in chunk 72.

5.1 Process an Input File

This is a somewhat clumsy part of the code that could be relegated to a separate function. On the other hand, it doesn't do much, and, since CWEB lets us structure the code by other means, we keep it here for simplicity.

The first thing to do here is to check if an input file name has been given to us on the command line. We exit right away, if there is no name.

Once we have the name, we pass it to the **SCMParmParse** class on the object construction, and the constructor processes the file.

74 ⟨ Process an Input File 74 ⟩ ≡

```
char *input_file_name = Λ;
if (argc > 1) {
  input_file_name = argv[1];
  pout() << "\tReading input from " << argv[1] << endl;
}
else {
  pout() << "\tERROR(main): no input file specified" << endl;
  pout() << "\t\tUsage: " << argv[0] << " <inputfile>" << endl;
  exit_status = ERR_NO_INPUT_FILE;
  goto exit;
}
SCMParmParse parameter_parser(argc − 2, argv + 2, Λ, input_file_name);
```

See also chunks 75 and 76.

This code is used in chunk 73.

¶ The input file has been opened and loaded into the program with the **SCMParmParse** constructor. Now we read verbosity and iteration parameters for *inner_main*.

The meaning of the parameters is as follows

watch_main Scheme name: *forms.watch.main*. This is a verbosity parameter. If set to 0, it makes *main* do the remainder of its work silently. If set to 1 or higher, it makes it chat about the progress of computation with verbosity proportional to its value.

number_of_steps Scheme name: *forms.iterate.number_of_steps*. The total number of iterations to be performed.

image_frequency Scheme name: *forms.iterate.image_frequency*. Images will be produced every *image_frequency* time steps.[12]

initial_label Scheme name: *forms.iterate.initial_label*. The initial label to use with dumped image and checkpoint files. If the job is to restart from a checkpoint file, the initial label should be set to an integer following the number of the last image dumped in the previous run, otherwise the previously produced images will be overwritten. A PBS script running the job should attend to this.

The *query* calls alter the default values for the parameters only if the required values are found on the input file. They do not throw errors if they are absent.

75 ⟨ Process an Input File 74 ⟩ +≡

```
SCMParmParse watch_parameter_parser("forms.watch");
SCMParmParse iterate_parameter_parser("forms.iterate");
SCMParmParse media_parameter_parser("forms.media");
int watch_main = false;
int number_of_steps = NUMBER_OF_STEPS;
int image_frequency = IMAGE_FREQUENCY;
int initial_label = INITIAL_LABEL;
int number_of_auxiliary_fields = N_AUXILIARIES;
```

[12] In our previous program SHAPES the meaning of this parameter was different and somewhat confusing. The images were dumped every *stride* ∗ *image_frequency* time steps. In this program *stride* will appear later, when level 0 is built.

```
watch_parameter_parser.query("main", watch_main);
iterate_parameter_parser.query("number_of_steps", number_of_steps);
iterate_parameter_parser.query("image_frequency", image_frequency);
iterate_parameter_parser.query("initial_label", initial_label);
media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
```

¶ Here function *inner_main* chats about what it's read, if it's been asked to do so.

76 ⟨ Process an Input File 74 ⟩ +≡

```
if (watch_main) {
  pout() << "\t\twatch_main␣␣␣␣␣␣␣=␣" << watch_main << endl;
  pout() << "\t\tnumber_of_steps␣␣=␣" << number_of_steps << endl;
  pout() << "\t\timage_frequency␣␣=␣" << image_frequency << endl;
  pout() << "\t\tinitial_label␣␣␣␣=␣" << initial_label << endl;
  pout() << "\t\tauxiliary_fields␣=␣" << number_of_auxiliary_fields << endl;
}
```

5.2 Build Initial Condition

This is a very short and sweet part of the code, mostly because it's all hidden in *build_levels* and *fill_levels*.

The former builds level 0 grid first and then builds iteratively higher level grids. The latter fills the grids with an initial condition, which in this code is zero everywhere, and fills the *medium* field.

The *medium* field is handled as follows. The user provides a Scheme code to assign an integer medium index to each edge of the multigrid. When $\boldsymbol{D}$ needs to be converted to $\boldsymbol{E}$, another piece of user provided Scheme code is invoked. It is passed the medium index, $\boldsymbol{D}$, $\boldsymbol{E}$ and auxiliary fields at this location in its argument list, and then an updated $\boldsymbol{E}$ comes out.

This way the user has the greatest flexibility possible in defining the number of media, media distribution and media types. Whereas in SHAPES this stuff was hardwired into the code and various "models" provided, here, users co-program the code instead. The co-programming takes place on the interpreted level, which means that the program itself does not have to be recompiled every time the user provides new specifications.

What's inside *build_levels* and *fill_levels* is, unfortunately, somewhat complicated. Both functions return zero if everything has gone well, and a non-zero diagnostic if it hasn't. In this case, the diagnostic is transferred to *exit_status* and we jump to *exit*.

5.2.1 Build a Multigrid

78 ⟨Build a Multigrid 78⟩ ≡

```
Vector⟨level *⟩ levels;
levels_ptr = &levels;
if (exit_status = build_levels(levels)) {
  pout() << "\tERROR(main): Failed to build levels. Exiting..." << endl;
  goto exit;
}
else {
  if (watch_main) {
    int actual_number_of_levels = levels.size();
    pout() << "\tCreated " << actual_number_of_levels;
    if (actual_number_of_levels ≡ 1) pout() << " level." << endl;
    else pout() << " levels." << endl;
  }
}
```

This code is used in chunk 73.

5.2.2 Fill the Multigrid

79 ⟨Fill the Multigrid 79⟩ ≡

```
if (exit_status = fill_levels(levels)) {
  pout() << "\tERROR(main): Failed to fill levels. Exiting..." << endl;
  goto exit;
}
else {
  if (watch_main) {
    int actual_number_of_levels = levels.size();
    pout() << "\tFilled " << actual_number_of_levels;
    if (actual_number_of_levels ≡ 1) pout() << " level." << endl;
    else pout() << " levels." << endl;
  }
}
```

This code is used in chunk 73.

5.3 Iterate

This is the heart of the program. But even here we sweep all complexity into functions that are called within the loop to keep the mainlines clean.

The iteration loop takes care both of advancing the multi-level system and of dumping images and checkpoint files. The loop counter *count* is advanced in the loop header. The logical condition *time_to_dump_data* is calculated in the loop header too. The loop termination condition is such that iterations stop on the last image dumped, not on the actual number of iterations requested. This is so because it's pointless to iterate beyond the last image.

Within the loop itself we advance the electric field first, and then we advance the magnetic field. These are really mutually recursive functions that take care of a multi-level time step, so that advancing $\boldsymbol{E}$ at the bottom level (level 0) triggers advancing $\boldsymbol{H}$ and $\boldsymbol{E}$ at higher levels in a mutually recursive cascade and the same holds for advancing $\boldsymbol{H}$ at the bottom level.

Any problems here are handled by looking at function returns and jumping to *exit* if problems are detected.

When the *time_to_dump_data* condition is detected, a function is called that dumps data (images and checkpoint files) for the whole multi-grid.

> I thought of orchestrating this dance from within Scheme. This probably can be done for a sequential code, because we have the global *levels_ptr*. But I truly don't know if MPI calls would run correctly when invoked from within Scheme. Whatever context MPI sets up may not be visible to Scheme invoked C++ functions, and so the *exchange* methods of **LevelData** may fail to exchange data between MPI processes.
>
> This should not affect the various small Scheme functions that we are going to use in this program, because these are strictly local. It's not Scheme orchestrating C++. It's C++ running everything and only invoking Scheme *calculator* when needed.

80 ⟨Iterate 80⟩ ≡

```
if (watch_main) pout() << "\tIterating..." << endl;
clock_t total_clock1, total_clock2, total_clock = (clock_t) 0;
clock_t inject_clock1, inject_clock2, inject_clock = (clock_t) 0;
clock_t convert_clock1, convert_clock2, convert_clock = (clock_t) 0;
clock_t advance_clock1, advance_clock2, advance_clock = (clock_t) 0;
clock_t output_clock1, output_clock2, output_clock = (clock_t) 0;
clock_t post_iteration_clock1, post_iteration_clock2, post_iteration_clock = (clock_t) 0;
clock_t post_all_clock1, post_all_clock2, post_all_clock = (clock_t) 0;
extern bool have_post_iteration_lambda;
extern bool have_post_all_lambda;
extern SCM post_iteration_lambda;
extern SCM post_all_lambda;
extern SCM t_e_var;
extern SCM dt_var;
extern SCM iterate_count_var;
extern SCM time_to_dump_var;
SCMParmParse post_iteration_parser("forms.post.iterate");
total_clock1 = clock();
for (int label = initial_label, count = 1, time_to_dump_data = false;
(int)((count − 1)/image_frequency) ∗ image_frequency + image_frequency ≤ number_of_steps;
count++, time_to_dump_data = ¬(count % image_frequency)) {
  if (watch_main)
    pout() << "\t\tcount␣=␣" << count << ",␣time_e␣=␣" << levels[0]→time_e << ",␣time_h␣=␣" <<
        levels[0]→time_h << endl;
  advance_clock1 = clock();
  if (exit_status = advance_d(levels[0])) {
    pout() << "\tERROR(main):␣Failed␣to␣advance␣D.␣Exiting..." << endl;
    goto exit;
```

```
}
advance_clock2 = clock();
advance_clock += advance_clock2 − advance_clock1;
inject_clock1 = clock();
if (exit_status = inject_d(levels[0])) {
  pout() ≪ "\tERROR(main): Failed to inject D. Exiting..." ≪ endl;
  goto exit;
}
inject_clock2 = clock();
inject_clock += inject_clock2 − inject_clock1;
convert_clock1 = clock();
if (exit_status = d_to_e(levels[0])) {
  pout() ≪ "\tERROR(main): Failed to convert D to E. Exiting..." ≪ endl;
  goto exit;
}
convert_clock2 = clock();
convert_clock += convert_clock2 − convert_clock1;
advance_clock1 = clock();
if (exit_status = advance_b(levels[0])) {
  pout() ≪ "\tERROR(main): Failed to advance B. Exiting..." ≪ endl;
  goto exit;
}
advance_clock2 = clock();
advance_clock += advance_clock2 − advance_clock1;
inject_clock1 = clock();
if (exit_status = inject_b(levels[0])) {
  pout() ≪ "\tERROR(main): Failed to inject B. Exiting..." ≪ endl;
  goto exit;
}
inject_clock2 = clock();
inject_clock += inject_clock2 − inject_clock1;
convert_clock1 = clock();
if (exit_status = b_to_h(levels[0])) {
  pout() ≪ "\tERROR(main): Failed to convert B to H. Exiting..." ≪ endl;
  goto exit;
}
convert_clock2 = clock();
convert_clock += convert_clock2 − convert_clock1;
if (have_post_iteration_lambda) {
  post_iteration_clock1 = clock();
      /* Update t_e and Δt on forms.time_e and forms.dt. We do this by reference avoiding the symbol
      table lookup. Any SCMParmParse object can be used for this, because the variables are not
      accessed by name. */
  post_iteration_parser.fastput(t_e_var, levels[0]→time_e);
  post_iteration_parser.fastput(dt_var, levels[0]→dt);
  post_iteration_parser.fastput(iterate_count_var, count);
  post_iteration_parser.fastput(time_to_dump_var, time_to_dump_data);
  SCM status = scm_call_0(post_iteration_lambda);
      /* scm_call_0 may return anything that's a valid SCM type. If it's a boolean false, we flag a
      problem and exit. */
  if (scm_is_bool(status)) {
    if (scm_is_false(status)) {
      pout() ≪ "\tERROR(main): forms.post.iteration.lambda returned #f" ≪ endl;
      goto exit;
```

```
        }
      }
      post_iteration_clock2 = clock();
      post_iteration_clock += post_iteration_clock2 - post_iteration_clock1;
    }    /* if (have_post_iteration_lambda) */
    if (time_to_dump_data) {
      if (watch_main) pout() << "\t\tdumping data, file label " << label << endl;
      output_clock1 = clock();
      if (exit_status = dump_data(levels, label++)) {
        pout() << "\tERROR(main): Failed to dump data. Exiting..." << endl;
        goto exit;
      }
      output_clock2 = clock();
      output_clock += output_clock2 - output_clock1;
    }
  }    /* the iteration for loop */
  if (have_post_all_lambda) {
    post_all_clock1 = clock();
    SCM status = scm_call_0(post_all_lambda);
    if (scm_is_bool(status)) {
      if (scm_is_false(status)) {
        pout() << "\tERROR(main): forms.post.all.lambda returned #f" << endl;
        goto exit;
      }
    }
    post_all_clock2 = clock();
    post_all_clock += post_all_clock2 - post_all_clock1;
  }    /* if (have_post_all_lambda) */
  total_clock2 = clock();
  total_clock += total_clock2 - total_clock1;
  pout() << "\tIterations completed in " << (int) total_clock/CLOCKS_PER_SEC << "s" << endl;
  pout() << "\t\tAdvancing fields took " << (int) advance_clock/CLOCKS_PER_SEC << "s" << endl;
  pout() << "\t\tSignal injection took " << (int) inject_clock/CLOCKS_PER_SEC << "s" << endl;
  pout() << "\t\tMaterial physics took " << (int) convert_clock/CLOCKS_PER_SEC << "s" << endl;
  if (have_post_iteration_lambda)
    pout() << "\t\tPost-iteration lambda " << (int) post_iteration_clock/CLOCKS_PER_SEC << "s" << endl;
  if (have_post_all_lambda)
    pout() << "\t\tPost-all lambda " << (int) post_all_clock/CLOCKS_PER_SEC << "s" << endl;
  pout() << "\t\tProducing output took " << (int) output_clock/CLOCKS_PER_SEC << "s" << endl;
```

This code is cited in chunk 69.

This code is used in chunk 73.

6 Initialization

Here we define the content of file *Initialize.c*. Functions used to define the grid and the initial condition are described here.

81 ⟨ `Initialize.c` 81 ⟩ ≡

```
#include "Forms.H"
        /* Functions used to build initial condition */
  ⟨ Function build_levels 82 ⟩
  ⟨ Function fill_levels 94 ⟩
```

6.1 Build Levels

This function, *build_levels*, generates and distributes amongst MPI processes grids for all levels as requested by the user. We used to do this recursively in SHAPES, but this time all levels are constructed iteratively within the single body of this function. The resulting function is somewhat long, but really quite simple. A great deal of flexibility is provided by linking Scheme into the program. Because of this *build_levels* does not have to be as elaborate as it was in SHAPES.

The general layout of the function is very simple, as shown below, and the logic of each module is also simple, if at times tedious.

82 ⟨ Function *build_levels* 82 ⟩ ≡

```
int build_levels(Vector⟨level *⟩ &levels)
{
  ⟨ build_levels: variable definitions 83 ⟩
  ⟨ build_levels: read grid specifications 88 ⟩
  ⟨ build_levels: build level 0 90 ⟩
  ⟨ build_levels: build higher levels 91 ⟩
exit:
  if (watch_build_levels) pout() << "build_levels: returning to caller" << endl;
  return return_status;
}
```

This code is cited in chunk 73.

This code is used in chunk 81.

6.1.1 Variable Definitions

There is a fair number of variables to define here, and it gets pretty tedious to do so. But if this section is chopped into smaller chunks, the variables can be organized and explained in groups.

We begin with scalars. These are mostly integers and there's just one **Real** thrown in.

The meaning of the first two: *return_status* and *watch_build_levels* is obvious. The latter's value is read from the Scheme space, if not found there, it's set to *false*.

Then we have a group of important variables that specify the grid. They are

number_of_levels This is a number of grid levels to be generated altogether including level 0. This number is read from the Scheme space.

max_box_size This is a maximum size of the level 0 box. The level 0 domain will be subdivided into these. The boxes will then be distributed amongst the participating MPI processes. The actual dimensions of the level 0 grid are read into a vector that will be discussed in the next chunk.

refine_block_factor This is a smallest box size in cells of a subgrid to be generated that will be used for the subgrid **DisjointBoxLayout**.

refine_buffer_size This is a number of grid cells that separates its boundary with the coarser grid from its boundary with the finer grid. These two must not get too close. This variable is read from the Scheme space, but if it is too small, it is reset to the default `REFINE_BUFFER_SIZE` by Scheme routines that draw on the level's *tags* field.

refine_max_size This is a maximum box size, in cells, used to generate a subgrid. Similar to *max_box_size* for level 0.

refine_ratio This is a refinement ratio between grid levels. This variable cannot be changed in this version of the program and is preset to `REFINE_RATIO`, which is 2. Note that in 3-dimensions, the refinement ratio of 2 subdivides each cell into eight sub-cells–this is almost an order of magnitude increase in 3-D resolution, with the correspondingly high cost in storage and computation. So 2 is plenty.

The other reason we don't allow for this parameter to be user-adjusted is because we do not have field stitching mathematics developed for refinement ratios other than 2. Yet. ◀

refine_fill_ratio This parameter tells the regriding routine how tightly it should enclose the *tags* field in the newly generated subgrid. 1.0 means *very tightly,* 0.0 means *not tightly at all.* The lower this number, the larger the boxes used to generate the subgrid. If the *tags* field has a very intricate shape and this parameter is set to 1.0, the regriding routine will generate a large number of tiny boxes, with which the complicated boundary of the *tags* fiels will be drawn.

refine_base_level, *refine_top_level* These two parameters are used to tell the regriding routine which levels should be refined.

83 ⟨ *build_levels*: variable definitions 83 ⟩ ≡

```
int return_status = EXIT_SUCCESS;
int watch_build_levels = false;
int number_of_levels = N_LEVELS;
int max_box_size = MAX_BOX_SIZE;
int refine_block_factor = REFINE_BLOCK_FACTOR;
int refine_buffer_size = REFINE_BUFFER_SIZE;
int refine_max_size = REFINE_MAX_SIZE;
int refine_ratio = REFINE_RATIO;
int refine_base_level, refine_top_level;
Real refine_fill_ratio = REFINE_FILL_RATIO;
```

See also chunks 84, 85, 86, and 87.

This code is cited in chunk 91.

This code is used in chunk 82.

¶ Now we get to define various **Vector**s. They are all 3-dimensional, i.e., *SpaceDim* (let us remember that we have already checked if *SpaceDim* is three in module ⟨ Inner Main 73 ⟩, section 5, page 193).

n_cells This is a number of cells of the level 0 grid in each of its principal directions.

delta This used to be a vector, but I have decided to make it a scalar. It is a grid constant of the level 0 grid and it is the same for each principal direction of the grid, meaning that each grid cell is a cube. This is what our Fortran routines assume. This simplifies and speeds up computation, and there is no real advantage in making it different for every direction.

origin These are physical coordinates of the center of the $(0, 0, 0)$ cell of the level 0 grid.

pml_xyz_min These are *physical* coordinates of the low corner of the PML boundary box. The PML box, of course, must be contained entirely within the level 0 domain.

pml_xyz_max These are *physical* coordinates of the high corner of the PML boundary box.

signal_xyz_min These are *physical* coordinates of the low corner of the signal injection box. The signal injection box must be contained entirely within the PML box.

signal_xyz_max These are *physical* coordinates of the high corner of the signal injection box.

84 ⟨ *build_levels*: variable definitions 83 ⟩ +≡

```
Vector⟨int⟩ n_cells(SpaceDim, N_CELLS);
Real delta = DELTA;
Real stride = STRIDE;
Real t0 = T0;
Vector⟨Real⟩ origin(SpaceDim, X0);
Vector⟨Real⟩ pml_xyz_min(SpaceDim, PML_XYZ_MIN);
Vector⟨Real⟩ pml_xyz_max(SpaceDim, PML_XYZ_MAX);
Vector⟨Real⟩ signal_xyz_min(SpaceDim, SIGNAL_XYZ_MIN);
Vector⟨Real⟩ signal_xyz_max(SpaceDim, SIGNAL_XYZ_MAX);
```

¶ These are my **SCMParmParse** parsers for reading parameters from the Scheme space for `"grid"`, `"grid.level0"`, `"watch"`, `"pml"` and `"signal"` groups.

85 ⟨ *build_levels*: variable definitions 83 ⟩ +≡
```
SCMParmParse grid_parameter_parser("forms.grid");
SCMParmParse level0_parameter_parser("forms.grid.level0");
SCMParmParse watch_parameter_parser("forms.watch");
SCMParmParse pml_parameter_parser("forms.pml");
SCMParmParse signal_parameter_parser("forms.signal");
SCMParmParse iterate_parameter_parser("forms.iterate");
```

¶ These are pointers to levels.

current_level_ptr This is a global pointer variable that's been declared in section 4, page 192. Here the code will point it to the last fully defined level. Predefined drawing routines invoked by Scheme know about this variable and use it to draw on its *tags* field.

level_0_ptr This is a pointer to level 0 while it is being constructed and before it becomes *levels*[0].

new_level_ptr This is a pointer to a higher level while it is being constructed and before it is appended to the *levels* array.

86 ⟨ *build_levels*: variable definitions 83 ⟩ +≡
```
extern level *current_level_ptr;
level *level_0_ptr;
level *new_level_ptr;
```

¶ These are special variables that are used by the **BRMeshRefine** machine, which constructs boxes for the next refined level.

vector_of_tag_sets This is a vector of tag sets. Each vector entry corresponds to a level. Within this level we tag cells for refinement. The tags for a given level are assembled into a set, and the set is then appended to the vector. For example, *vector_of_tag_sets*[0] is a set of level 0 cells tagged for refinement.

vector_of_refinements This is a vector of refinement ratios between levels. In out case, it is simply $[2, 2, 2, 2, \ldots]$.

vector_of_vectors_of_boxes The boxes of each level are arranged into a vector, *vector_of_boxes*. Then the level vectors themselves are arranged into a vector, *vector_of_vectors_of_boxes*, that specifies the whole multi-level grid, currently constructed.

new_vector_of_vectors_of_boxes After the **BRMeshRefine** machine has done its job, it returns the new multi-level grid structure on the *new_vector_of_vectors_of_boxes*. From this we can pick up the last *vector_of_boxes*, it will be the newly created sub-grid, and distribute it over the MPI processes to balance the load.

87 ⟨ *build_levels*: variable definitions 83 ⟩ +≡
```
Vector⟨IntVectSet⟩ vector_of_tag_sets;
Vector⟨int⟩ vector_of_refinements;
Vector⟨Vector⟨Box⟩⟩ vector_of_vectors_of_boxes, new_vector_of_vectors_of_boxes;
```

6.1.2 Read Grid Specifications

So, now we get to the tedious part of parsing the Scheme space for the parameters that this function needs. We also check on this occasion if some parameters are sensibly defined.

The first one we read is *watch_build_levels*, which activates the chat. Then we read *number_of_levels*. This number must fit between `N_LEVELS_MIN`, which is 1, and `N_LEVELS_MAX`. Because building and advancing levels

is costly, both computationally and memory-wise, we restrict their numbers. We jump to exit if *number_of_levels* is bad.

Similarly we read *max_box_size* and ensure that it fits between MAX_BOX_SIZE_MIN and MAX_BOX_SIZE_MAX. On this occasion we also ensure that *max_box_size* is a power of 2. All sizes in FORMS should be a power of 2, preferably.

88 ⟨ *build_levels*: read grid specifications 88 ⟩ ≡

```
watch_parameter_parser.query("build_levels", watch_build_levels);
if (watch_build_levels) pout() << "build_levels:" << endl;
if (grid_parameter_parser.query("number_of_levels", number_of_levels)) {
  if (watch_build_levels) pout() << "\tread:␣number_of_levels␣=␣" << number_of_levels << endl;
  if ((number_of_levels < N_LEVELS_MIN) ∨ (number_of_levels > N_LEVELS_MAX)) {
    pout() << "\n\tERROR(build_levels):␣number_of_levels␣out␣of␣range" << endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
if (grid_parameter_parser.query("max_box_size", max_box_size)) {
  if (watch_build_levels) pout() << "\tread:␣max_box_size␣=␣" << max_box_size << endl;
  if ((max_box_size < MAX_BOX_SIZE_MIN) ∨ (max_box_size > MAX_BOX_SIZE_MAX)) {
    pout() << "\n\tERROR(build_levels):␣max_box_size␣out␣of␣range" << endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
  if (¬is_a_pwr_of_two(max_box_size)) {
    pout() << "\n\tERROR(build_levels):␣max_box_size␣not␣a␣power␣of␣2" << endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
```

See also chunk 89.

This code is used in chunk 82.

¶ In this chunk we read array variables.

First we read number of cells in each direction, *n_cells*, which must fall between N_CELLS_MIN and N_CELLS_MAX, and which must be a power of two as well. Then we read grid constant *delta*. It must be a positive real number. Then we read the *origin* coordinates, no checking is done here.

The statements that follow read PML box corners, low and high, and signal injection box corners, low and high. No checking is done here, because we need to construct the level 0 grid first. We check if these parameters are correct in chunk ⟨ *build_levels*: build level 0 90 ⟩, section 6.1.3, page 207.

89 ⟨ *build_levels*: read grid specifications 88 ⟩ +≡

```
if (level0_parameter_parser.queryarr("cells", n_cells, 0, SpaceDim)) {
  if (watch_build_levels) pout() << "\tread:␣n_cells␣=␣" << n_cells << endl;
  for (int i = 0; i < SpaceDim; i++) {
    if ((n_cells[i] < N_CELLS_MIN) ∨ (n_cells[i] > N_CELLS_MAX)) {
      pout() << "\n\tERROR(build_levels):␣n_cells[" << i << "]␣out␣of␣range" << endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
    if (¬is_a_pwr_of_two(n_cells[i])) {
      pout() << "\n\tERROR(build_levels):␣n_cells[" << i << "]␣not␣a␣power␣of␣2" << endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
```

```
    }
  }
  if (level0_parameter_parser.query("delta", delta)) {
    if (watch_build_levels) pout() << "\tread: delta = " << delta << endl;
    if (delta ≤ 0) {
      pout() << "\n\tERROR(build_levels): delta out of range" << endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
  }
  if (iterate_parameter_parser.query("stride", stride)) {
    if (watch_build_levels) pout() << "\tread: stride = " << stride << endl;
    if (stride < STRIDE) {
      pout() << "\n\tERROR(build_levels): stride out of range" << endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
  }
  if (iterate_parameter_parser.query("t0", t0)) {
    if (watch_build_levels) pout() << "\tread: t0 = " << t0 << endl;
  }
        /* Here we allow any three real numbers. If queryarr doesn't crash, it means the input is correct. */
  if (level0_parameter_parser.queryarr("origin", origin, 0, SpaceDim))
    if (watch_build_levels) pout() << "\tread: origin = " << origin << endl;
  if (pml_parameter_parser.queryarr("lo", pml_xyz_min, 0, SpaceDim))
    if (watch_build_levels) pout() << "\tread: pml.lo = (" << pml_xyz_min << ")" << endl;
  if (pml_parameter_parser.queryarr("hi", pml_xyz_max, 0, SpaceDim))
    if (watch_build_levels) pout() << "\tread: pml.hi = (" << pml_xyz_max << ")" << endl;
  if (signal_parameter_parser.queryarr("lo", signal_xyz_min, 0, SpaceDim))
    if (watch_build_levels) pout() << "\tread: signal.lo = (" << signal_xyz_min << ")" << endl;
  if (signal_parameter_parser.queryarr("hi", signal_xyz_max, 0, SpaceDim))
    if (watch_build_levels) pout() << "\tread: signal.hi = (" << signal_xyz_max << ")" << endl;
```

6.1.3 Build Level 0 Grid

Finally, all input read, we get to build the grid.

We begin by building the level 0 grid. This is done a little differently from building grids for higher levels. It is also simpler.

The first step is to allocate space for it with **new level**. Then we define the domain. It is a box in the **IntVect** space with its low corner located at $(0, 0, 0)$ and its high corner located at $(n\_cells[0], n\_cells[1], n\_cells[2])$. The domain's periodicity is default, i.e., non-periodic.

Having defined the domain, we split it into boxes of maximum side length given by *max_box_size*. Chombo function *domainSplit* does the job, and puts the result on *level_0_ptr→vector_of_boxes*. Chombo function *LoadBalance* is then invoked to distribute the boxes amongst participating MPI processes. The output of the function is *level_0_ptr→vector_of_processes*. With these two vectors, of boxes and processes, we can now define the disjoint box layout for level 0. This is the level 0 grid.

Once we're done with the grid, we fill the remaining parameters of the level, the *origin* and the *delta*.

It may happen that the actual grid constructed by *domainSplit* is not exactly as we have requested. The function may chop the grid here and there, while constructing the boxes. So we want to find out the real grid's low and high corners rather than assume them. This is done by function *effective_grid_bounds*, which is defined in module ⟨ Function *effective_grid_bounds* 214 ⟩, page 318, section 10.3. This is a non-trivial parallel function that involves some inter-process communication. The function fills *level_0_ptr→ijk_min*, *level_0_ptr→ijk_max*, *level_0_ptr→xyz_min* and *level_0_ptr→xyz_max* slots of the level structure.

Once we have returned we check if the PML and signal boxes, as read from the Scheme space, are correctly configured, and make appropriate assignments, if they are.

Eventually, having constructed the level, we append it to the *levels* vector. From this point onwards we refer to it as *levels*[0], or (∗*levels_ptr*)[0] (I wonder if this is going to work..., haven't tried this yet.)

90 ⟨ *build_levels*: build level 0 90 ⟩ ≡

```
level_0_ptr = new level;
level_0_ptr→domain.define(IntVect::Zero,
    (n_cells[0] − 1) ∗ BASISV(0) + (n_cells[1] − 1) ∗ BASISV(1) + (n_cells[2] − 1) ∗ BASISV(2));
domainSplit(level_0_ptr→domain, level_0_ptr→vector_of_boxes, max_box_size);
LoadBalance(level_0_ptr→vector_of_processes, level_0_ptr→vector_of_boxes);
level_0_ptr→box_layout.define(level_0_ptr→vector_of_boxes, level_0_ptr→vector_of_processes);
if (level_0_ptr→box_layout.isDisjoint()) {
  if (watch_build_levels > 1)
    pout() ≪ "\tdisjoint_box_layout␣=␣" ≪ endl ≪ level_0_ptr→box_layout ≪ endl;
}
else {
  pout() ≪ "\n\tERROR(build_levels):␣Failed␣to␣construct␣disjoint␣box␣layout␣for␣level␣0" ≪
      endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
level_0_ptr→origin.resize(SpaceDim);
level_0_ptr→origin = origin;
level_0_ptr→delta = delta;
level_0_ptr→dt = delta/stride;
level_0_ptr→time_e = t0;
level_0_ptr→time_h = t0 + level_0_ptr→dt/2;
level_0_ptr→number = 0;
    /∗ Find xyz_min, xyz_max, ijk_min and ijk_max. The domainSplit function may have changed them a
    little. ∗/
if (return_status = effective_grid_bounds(level_0_ptr)) {
  pout() ≪ "\n\tERROR(build_levels):␣Cannot␣find␣grid␣bounds" ≪ endl;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((pml_xyz_min[i] < level_0_ptr→xyz_min[i] + PML_MARGIN) ∨ (pml_xyz_max[i] >
      level_0_ptr→xyz_max[i] − PML_MARGIN)) {
    pout() ≪ "\n\tERROR(build_levels):␣PML␣margin␣too␣narrow" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
  if ((signal_xyz_min[i] < pml_xyz_min[i] + SIGNAL_MARGIN) ∨ (signal_xyz_max[i] >
      pml_xyz_max[i] − SIGNAL_MARGIN)) {
    pout() ≪ "\n\tERROR(build_levels):␣Signal␣margin␣too␣narrow" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
level_0_ptr→pml_xyz_min.resize(SpaceDim);
level_0_ptr→pml_xyz_min = pml_xyz_min;
level_0_ptr→pml_xyz_max.resize(SpaceDim);
level_0_ptr→pml_xyz_max = pml_xyz_max;
level_0_ptr→signal_xyz_min.resize(SpaceDim);
level_0_ptr→signal_xyz_min = signal_xyz_min;
level_0_ptr→signal_xyz_max.resize(SpaceDim);
level_0_ptr→signal_xyz_max = signal_xyz_max;
```

```
    /* Evaluate signal_ijk_lo and signal_ijk_hi. */
  for (int i = 0; i < SpaceDim; i++) {
    level_0_ptr->signal_ijk_lo[i] = (int) floor((signal_xyz_min[i] - origin[i])/delta + 0.5);
    level_0_ptr->signal_ijk_hi[i] = (int) floor((signal_xyz_max[i] - origin[i])/delta + 0.5);
  }
  levels.clear();
  levels.push_back(level_0_ptr);
  if (watch_build_levels) {
    pout() << "\tdomain␣=␣" << "[" << levels[0]->ijk_min[0] << "," << levels[0]->ijk_max[0] << "]" <<
        "␣x␣" << "[" << levels[0]->ijk_min[1] << "," << levels[0]->ijk_max[1] << "]" << "␣x␣" << "[" <<
        levels[0]->ijk_min[2] << "," << levels[0]->ijk_max[2] << "]" << endl;
    pout() << "\tnumber␣of␣boxes␣=␣" << levels[0]->box_layout.size() << endl;
  }
```

This code is cited in chunks 23 and 89.

This code is used in chunk 82.

6.1.4 Build Higher Levels

Higher levels are now built iteratively within the **for** loop, rather than recursively, as was the case in SHAPES. And here is the loop. The loop lives inside an **if** statement, that checks if higher levels need to be built at all, and if we have adequate specifications provided to do so.

The latter check is simple: we just check if there is a bound variable called `"forms.grid.lambda"` defined in the Scheme space and if it's a procedure. Both are accomplished by the one call to the *isProcedure* method of the **SCMParmParse** class.

Assuming that we are to build higher levels and have some non-trivial instructions provided in the lambda, we read the basic refinement parameters from the Scheme space. These are *refine_fill_ratio*, *refine_block_factor*, *refine_buffer_size* and *refine_max_size*, all defined in chunk ⟨ *build_levels*: variable definitions 83 ⟩, page 203, section 6.1.1.

We do not check for their correctness here, which is an omission that will need to be rectified in future.

Then we truncate the three basic vectors that will be used in building subgrids: *vector_of_tag_sets*, *vector_of_refinements* and *vector_of_vectors_of_boxes*, and, at long last, we enter the loop. What happens inside it is deferred to module ⟨ *build_levels*: build next level 92 ⟩, page 209, section 6.1.4.

Since the vectors have not been used so far, they probably don't need to be truncated, because they're made of zero length when defined. So this is a "just in case" precaution.

91 ⟨ *build_levels*: build higher levels 91 ⟩ ≡

```
  if (number_of_levels > 1) {
    if (grid_parameter_parser.isProcedure("lambda")) {
      grid_parameter_parser.query("fill_ratio", refine_fill_ratio);
      grid_parameter_parser.query("block_factor", refine_block_factor);
      grid_parameter_parser.query("buffer_size", refine_buffer_size);
      grid_parameter_parser.query("max_size", refine_max_size);
      vector_of_tag_sets.clear();
      vector_of_refinements.clear();
      vector_of_vectors_of_boxes.clear();
      for (int level_number = 1; level_number < number_of_levels; level_number++) {
        if (watch_build_levels) pout() << "\n\tbuilding␣level␣" << level_number << endl;
        ⟨ build_levels: build next level 92 ⟩
      }
    }
    else {
      pout() << "\n\tERROR(build_levels):␣no␣lambda" << endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
```

```
}
```
This code is used in chunk 82.

Build Next Level

Here the fun begins.

The way a next level grid is made is as follows. First we tag selected cells of a current grid for refinement. The tagged cells are arranged into a set and this set is passed to a special machine, called **BRMeshRefine**, together with the grid specifications up to the current level. The machine makes the next level grid and returns it as a vector of boxes, because this is how a grid is specified in Chombo.

Now, the trick of this chunk is in the tagging. Tagging in SHAPES was a very clumsy process, which apart from being complicated was also inflexible. Tagging in FORMS is done by a Scheme closure, `"forms.grid.lambda"`, which user have to provide. The FORMS code is much simpler and much shorter, and yet even the most complex subgrids may be constructed.

Before we can invoke Scheme though, we have to prepare a few things.

First we set the global pointer *current_level_ptr* to point to the last level fully defined so far. The C++ routines that Scheme will invoke, communicate with the Chombo data structures by the means of this pointer.

Next we define the *tags* field, which the Scheme program will draw on. It is simply a field of integers defined on the last level's disjoint box layout. The *tags* field is then initialized to zero.

And now we simply ask Scheme to execute `"grid.level.lambda"` on the current level. As the Scheme closure is called on every **for** loop iteration *current_level_ptr* is updated to point to the last level fully built. It is on this level that we draw the outline for the next level.

Every Scheme evaluation effected by *scm_call_1* returns some Scheme value, which here we capture on *lambda_return*. These are all of type **SCM** in the C++ program that calls Scheme. The user closure may return anything, but here we check for just one possibility, namely that the returned value is boolean and *false*. This we interpret as a *failure* flag and on having encountered it here, we return to the caller raising an error.

Scheme does its job silently, unless explicitly asked to print something with *display* or *write*. But let us note that anything that Scheme prints will *not* go into the *pout*( ) output stream. To put Scheme output there, it must be translated into C strings first, then printed with *pout*( ).

There have been numerous changes to the Guile interface. This program assumes version 1.8.1 or later of Guile. In this version the general rule is

to Scheme *from* C use *scm_from* functions

from Scheme *to* C use *scm_to* functions

For example, *scm_from_int* takes a C integer and converts it to a Scheme integer. When calling *scm_call_1* we must ensure that the second argument is of the **SCM** type, exactly as it would be in the `(forms.grid.lambda ...)` statement of Scheme. We cannot just say *scm_call_0*(*lambda*, *level_number*).

In earlier versions of Guile, for example, 1.6.X, *scm_int2num* was used in place of *scm_from_int*. These older interfaces are now deprecated.

92 ⟨ *build_levels*: build next level 92 ⟩ ≡

```
    /* current_level_ptr must be a global variable, because this is the simplest way to pass it to Scheme C
       procedures. */
  current_level_ptr = levels[level_number − 1];
  LevelData⟨BaseFab⟨int⟩⟩ &tags = current_level_ptr→tags;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator data_iterator(box_layout);
      /* Create the tag field over the current level's box layout and initialize it to zero */
  if (watch_build_levels) pout( ) ≪ "\tcreating␣tags␣LevelData" ≪ endl;
  tags.define(box_layout, 1, 0 ∗ IntVect :: Unit);
  if (tags.isDefined( )) {
    for (data_iterator.reset( ); data_iterator.ok( ); ++data_iterator) {
      tags[data_iterator( )].setVal(0, 0);
    }
```

```
}
else {
  pout() << "\n\tERROR(build_levels): Failed to define tags field" << endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
      /* Invoke Scheme to draw on the tags field. By now we know that forms.grid.lambda exists and is a
      closure. */
SCM lambda = scm_variable_ref(scm_c_lookup("forms.grid.lambda"));
SCM lambda_return = scm_call_1(lambda, scm_from_int(level_number));
if (scm_is_bool(lambda_return)) {
  if (scm_is_false(lambda_return)) {
    pout() << "\n\tERROR(build_levels): Lambda returned #f" << endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
```

See also chunk 93.

This code is cited in chunk 91.

This code is used in chunk 91.

¶ Once we have returned from Scheme, there should be something drawn on *current_level_ptr→tags*, which is a **LevelData⟨BaseFab⟨int⟩⟩**. "Something drawn", means that there will be ones in the *tags* field in some cells—this means drawing, whereas *tags* in other cells will remain zero.

But **BRMeshRefine** does not work with a **LevelData⟨BaseFab⟨int⟩⟩**. It wants a vector of **IntVectSet**s instead. So we need to convert the data on *tags* to an **IntVectSet** *tag_set*.

This is almost as tricky as finding the effective grid bounds and requires some communication between the MPI processes too. Function *make_tag_set*, defined in section 10.4, page 320, module ⟨ Function *make_tag_set* 222 ⟩, performs the conversion. So here we just call it and then proceed.

When *make_tag_set* returns, there should be something in *current_level_ptr→tag_set*, so we append it to *vector_of_tag_sets*. We append another *refine_ratio* to *vector_of_refinments* and *current_level_ptr→vector_of_boxes* to *vector_of_vectors_of_boxes*.

Now we are ready to bring the **BRMeshRefine** machine, and once it's created we ask it to refine the current level only. There is no need to redo the previously constructed subgrids. This is done by calling a **BRMeshRefine**::*regrid* method.

The method is quite slow. It may take a long while for it to return. From my simple tests just about all in this chunk runs like a rocket, but the code visibly halts on **BRMeshRefine**::*regrid*.

When **BRMeshRefine**::*regrid* returns finally, we must first check that it has done anything for starters. Has it created the next level? We check this by inspecting the size of *new_vector_of_vectors_of_boxes*. This vector should be longer by one entry than *vector_of_vectors_of_boxes*. But then we must also check that the last vector in *new_vector_of_vectors_of_boxes*, the one that would define the new level, is not of length zero.

These checks passed, we may be confident that **BRMeshRefine**::*regrid* has produced a valid new level subgrid, so at this stage we create the new level with *new_level_ptr* = **new level**. We have the vector of boxes for it, waiting in *new_vector_of_vectors_of_boxes*[*level_number*], but we must still distribute it amongst the MPI processes. We call function *LoadBalance* to do this, and then we define the **DisjointBoxLayout** of the new level.

In case this operation bumps out, we still check if the produced box layout is indeed disjoint and exit raising an error flag if it is not.

The new level is almost complete. We give it its level number, its domain, we tell it about the PML and signal boxes. We calculate its *delta* and its *origin*. The origin will have shifted according to the formula

$$\boldsymbol{\Delta}^{\text{new}} = \frac{\boldsymbol{\Delta}^{\text{current}}}{n_r} \tag{460}$$

$$\boldsymbol{r}_0^{\text{new}} = \boldsymbol{r}_0^{\text{current}} - \frac{n_r - 1}{2}\boldsymbol{\Delta}^{\text{new}}, \tag{461}$$

where n_r is the refinement ratio, $\boldsymbol{r}_0$ is the origin vector, and $\boldsymbol{\Delta}$ is the grid constant vector.

Figure 12 shows how the origin of the grid moves for $n_r = 4$. We see that we have to move it 3 halves of the refined lattice constant down and to the left. For $n_r = 2$ it would be only one half.

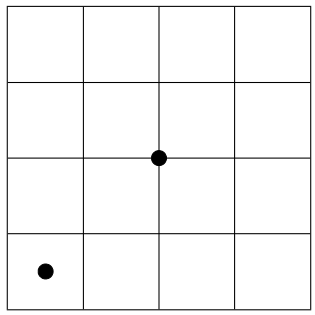


Figure 12: Movement of the grid origin on grid refinement for $n_r = 4$.

Finally, we find the real bounds of the new grid, both in the **IntVect** and in the physical space by calling *effective_grid_bounds*, and append the newly created level to the *levels* vector. From now on, we will refer to it by *levels*[*level_number*].

And this is it. This is how we make the multigrid.

93 ⟨ *build_levels*: build next level 92 ⟩ +≡

```
make_tag_set(current_level_ptr);
if (watch_build_levels) pout() << "\tcreating␣vectors" << endl;
vector_of_tag_sets.push_back(current_level_ptr→tag_set);
vector_of_refinements.push_back(refine_ratio);
vector_of_vectors_of_boxes.push_back(current_level_ptr→vector_of_boxes);
if (watch_build_levels) pout() << "\tcreating␣BRMeshRefine" << endl;
BRMeshRefine mesh_refine(levels[0]→domain, vector_of_refinements, refine_fill_ratio, refine_block_factor,
    refine_buffer_size, refine_max_size);
new_vector_of_vectors_of_boxes.clear();
refine_base_level = current_level_ptr→number;
refine_top_level = current_level_ptr→number;
if (watch_build_levels) pout() << "\tregriding" << endl;
mesh_refine.regrid(new_vector_of_vectors_of_boxes, vector_of_tag_sets, refine_base_level, refine_top_level,
    vector_of_vectors_of_boxes);
if (new_vector_of_vectors_of_boxes.size() ≤ vector_of_vectors_of_boxes.size()) {
  pout() << "\n\tERROR(build_levels):␣failed␣to␣generate␣vector␣of␣boxes␣for␣level␣" <<
      level_number << endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
if (new_vector_of_vectors_of_boxes[level_number].size() ≡ 0) {
  pout() << "\n\tERROR(build_levels):␣empty␣vector␣of␣boxes␣for␣level␣" << level_number << endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
if (watch_build_levels) pout() << "\tcreating␣new␣level" << endl;
new_level_ptr = new level;
new_level_ptr→vector_of_boxes = new_vector_of_vectors_of_boxes[level_number];
if (watch_build_levels) pout() << "\tbalancing␣load" << endl;
LoadBalance(new_level_ptr→vector_of_processes, new_level_ptr→vector_of_boxes);
if (watch_build_levels) pout() << "\tcreating␣disjoint␣box␣layout" << endl;
new_level_ptr→box_layout.define(new_level_ptr→vector_of_boxes, new_level_ptr→vector_of_processes);
```

```
if (¬new_level_ptr→box_layout.isDisjoint()) {
  pout() ≪ "\n\tERROR(build_levels):␣failed␣to␣construct␣disjoint␣box␣layout␣for␣level␣" ≪
      level_number ≪ endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
new_level_ptr→number = level_number;
if (watch_build_levels) pout() ≪ "\trefining␣domain" ≪ endl;
new_level_ptr→domain = refine(current_level_ptr→domain, refine_ratio);
if (watch_build_levels) pout() ≪ "\tsmall␣items" ≪ endl;
new_level_ptr→pml_xyz_min.resize(SpaceDim);
new_level_ptr→pml_xyz_max.resize(SpaceDim);
new_level_ptr→signal_xyz_min.resize(SpaceDim);
new_level_ptr→signal_xyz_max.resize(SpaceDim);
new_level_ptr→pml_xyz_min = current_level_ptr→pml_xyz_min;
new_level_ptr→pml_xyz_max = current_level_ptr→pml_xyz_max;
new_level_ptr→signal_xyz_min = current_level_ptr→signal_xyz_min;
new_level_ptr→signal_xyz_max = current_level_ptr→signal_xyz_max;
new_level_ptr→delta = current_level_ptr→delta/refine_ratio;
new_level_ptr→dt = current_level_ptr→dt/(refine_ratio + 1);
new_level_ptr→time_e = t0;
new_level_ptr→time_h = t0 + new_level_ptr→dt/2;
new_level_ptr→origin.resize(SpaceDim);
for (int i = 0; i < SpaceDim; i++) {
  new_level_ptr→origin[i] = current_level_ptr→origin[i] − (refine_ratio − 1) ∗ new_level_ptr→delta/2.0;
}
effective_grid_bounds(new_level_ptr);
if (watch_build_levels) pout() ≪ "\tappending␣to␣the␣levels␣array" ≪ endl;
levels.push_back(new_level_ptr);
```

6.2 Fill Levels

Fill levels with data: initialize all arrays to zero, then invoke Scheme procedure to draw media distribution.

94 ⟨Function *fill_levels* 94⟩ ≡

```
int fill_levels(Vector⟨level *⟩ &levels) { int return_status = EXIT_SUCCESS;
    int watch_fill_levels = false;
    int print_pml_arrays = false;
    int number_of_levels = N_LEVELS;
    int auxiliaries = N_AUXILIARIES;
    int number_of_precompute_fields = 0;
    int number_of_write_fields = 0;
    IntVect ghost_margin = N_GHOST_CELLS * IntVect::Unit;
        /* Parsers used by the function */
    SCMParmParse watch_parameter_parser("forms.watch");
    SCMParmParse grid_parameter_parser("forms.grid");
    SCMParmParse media_parameter_parser("forms.media");
    SCMParmParse signal_parameter_parser("forms.signal");
    SCMParmParse output_parameter_parser("forms.output");
    SCMParmParse pml_parameter_parser("forms.pml");
        /* Global variables */
    extern level *current_level_ptr;
```

See also chunks 95, 96, 97, 98, 102, and 103.

This code is cited in chunks 22 and 200.

This code is used in chunk 81.

95 ¶⟨Function *fill_levels* 94⟩ +≡

```
        /* Action */
watch_parameter_parser.query("fill_levels", watch_fill_levels);
if (watch_fill_levels) pout() << "fill_levels:" << endl;
if (grid_parameter_parser.query("number_of_levels", number_of_levels)) {
  if (watch_fill_levels) pout() << "\tread:␣number_of_levels␣=␣" << number_of_levels << endl;
  if ((number_of_levels < N_LEVELS_MIN) ∨ (number_of_levels > N_LEVELS_MAX)) {
    pout() << "\n\tERROR(fill_levels):␣number_of_levels␣out␣of␣range" << endl;
    return_status = ERR_FILL_LEVELS;
    goto exit;
  }
}
if (media_parameter_parser.query("number_of_auxiliary_fields", auxiliaries)) {
  if (watch_fill_levels) pout() << "\tread:␣number_of_auxiliary_fields␣=␣" << auxiliaries << endl;
  if ((auxiliaries < N_AUXILIARIES_MIN) ∨ (auxiliaries > N_AUXILIARIES_MAX)) {
    pout() << "\n\tERROR(fill_levels):␣number_of_auxiliary_fields␣out␣of␣range" << endl;
    return_status = ERR_FILL_LEVELS;
    goto exit;
  }
}
```

96 ¶⟨Function *fill_levels* 94⟩ +≡

```
        /* Allocate arrays for the data. */
for (int n = 0; n < number_of_levels; n++) {
  DisjointBoxLayout &level_box_layout = levels[n]→box_layout;
      /* Iterators */
      /* LayoutIterator contains all boxes of the layout on every processor, whether they belong to it or
      not, whereas DataIterator contains only the boxes that are managed by a given processor. */
```

```
levels[n]→data_iterator = level_box_layout.dataIterator();
levels[n]→layout_iterator = level_box_layout.layoutIterator();
    /* CurlBox fields, D and E. Each field is defined with two components. Component 0 is for the new
    field and component 1 is for the old one. */
if (watch_fill_levels) pout() << "\tlevel␣" << n << ":" << endl;
if (watch_fill_levels) pout() << "\t\tdefining␣E" << flush;
levels[n]→E.define(level_box_layout, 2, ghost_margin);
if (watch_fill_levels) pout() << "␣D" << flush;
levels[n]→D.define(level_box_layout, 2, ghost_margin);
    /* Auxiliary fields. The user specifies how many components will be needed. */
if (auxiliaries) {
  if (watch_fill_levels) pout() << "␣S" << flush;
  levels[n]→S.define(level_box_layout, auxiliaries, ghost_margin);
}
if (watch_fill_levels) pout() << "␣...␣done." << endl << flush;
      /* From this point onwards we can find about the number of auxiliary fields by invoking the
      levels[n]→S.nComp() method. This is actually a method inherited from the BoxLayoutData class. */
      /* EdgeFab⟨int⟩ fields */
if (watch_fill_levels) pout() << "\t\tdefining␣medium_E" << flush;
levels[n]→medium_E.define(level_box_layout, 1, ghost_margin);
    /* FaceFab⟨int⟩ fields */
if (watch_fill_levels) pout() << "␣medium_H" << flush;
levels[n]→medium_H.define(level_box_layout, 1, ghost_margin);
if (watch_fill_levels) pout() << "␣...␣done." << endl << flush;
      /* FluxBox fields */
if (watch_fill_levels) pout() << "\t\tdefining␣H" << flush;
levels[n]→H.define(level_box_layout, 2, ghost_margin);
if (watch_fill_levels) pout() << "␣B" << flush;
levels[n]→B.define(level_box_layout, 2, ghost_margin);
if (watch_fill_levels) pout() << "␣...␣done." << endl << flush;
      /* Output fields */
number_of_write_fields = output_parameter_parser.countval("write");
number_of_precompute_fields = output_parameter_parser.countval("precompute");
if (number_of_write_fields ∧ number_of_precompute_fields) {
    /* These fields have no margin, because they are not exchanged between CPUs */
  if (watch_fill_levels) pout() << "\t\tdefining␣output␣fields" << flush;
  levels[n]→Out.define(level_box_layout, number_of_precompute_fields, IntVect::Zero);
  levels[n]→OutWrite.define(level_box_layout, number_of_write_fields, IntVect::Zero);
  if (watch_fill_levels) pout() << "␣...␣done." << endl << flush;
}
if (watch_fill_levels) {
  pout() << "\t\tE:␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→E.nComp() << endl;
  pout() << "\t\tD:␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→D.nComp() << endl;
  pout() << "\t\tH:␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→H.nComp() << endl;
  pout() << "\t\tB:␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→B.nComp() << endl;
  if (auxiliaries) pout() << "\t\tS:␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→S.nComp() << endl;
  pout() << "\t\tmedia␣␣#␣of␣components␣=␣" << levels[n]→medium_E.nComp() << endl;
  pout() << "\t\t␣␣␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→medium_H.nComp() << endl;
  if (number_of_precompute_fields ∧ number_of_write_fields) {
    pout() << "\t\tOut:␣␣␣#␣of␣components␣=␣" << levels[n]→Out.nComp() << endl;
    pout() << "\t\t␣␣␣␣␣␣␣#␣of␣components␣=␣" << levels[n]→OutWrite.nComp() << endl;
  }
}
}    /* first loop over levels–allocations */
```

97 ¶⟨ Function *fill_levels* 94 ⟩ +≡

```
    /* Initialize the arrays */
  for (int n = 0; n < number_of_levels; n++) {
    DataIterator &data_iterator = levels[n]→data_iterator;
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      levels[n]→E[data_iterator()].setVal((Real) ZERO);
      levels[n]→D[data_iterator()].setVal((Real) ZERO);
      levels[n]→H[data_iterator()].setVal((Real) ZERO);
      levels[n]→B[data_iterator()].setVal((Real) ZERO);
      levels[n]→medium_E[data_iterator()].setVal((int) ZERO);
      levels[n]→medium_H[data_iterator()].setVal((int) ZERO);
      if (auxiliaries) levels[n]→S[data_iterator()].setVal((Real) ZERO);
      if (number_of_precompute_fields ∧ number_of_write_fields) {
        levels[n]→Out[data_iterator()].setVal((Real) ZERO);
        levels[n]→OutWrite[data_iterator()].setVal((Real) ZERO);
      }
    }
    for (int count = 0; count < 2; count++) {
      levels[n]→E_idx[count] = count;
      levels[n]→D_idx[count] = count;
      levels[n]→H_idx[count] = count;
      levels[n]→B_idx[count] = count;
    }
    if (watch_fill_levels) {
      pout() ≪ "\tlevel␣" ≪ n ≪ ":" ≪ endl;
      pout() ≪ "\t\tAll␣Fabs␣initialized␣to␣zero." ≪ endl;
    }
  }    /* second loop over levels–initializations */
```

¶ Here we attend to certain structures that are needed for level 0 calculations. Level 0 is different, because it has PMLs and signal injection.

98 ⟨ Function *fill_levels* 94 ⟩ +≡

```
    /* On level 0 we are going to mark UPML and scattered field regions to facilitate D to E and B to H
       conversions. */
  if (return_status = mark_regions(levels[0])) {
    pout() ≪ "\n\tERROR(fill_levels):␣mark_regions␣failed" ≪ endl;
    goto exit;
  }
  if (return_status = mark_TFR_faces(levels[0])) {
    pout() ≪ "\n\tERROR(fill_levels:)␣mark_TFR_faces␣failed" ≪ endl;
    goto exit;
  }
  if (return_status = fill_PML_arrays(levels[0])) {
    pout() ≪ "\n\tERROR(fill_levels):␣fill_PML_arrays␣failed" ≪ endl;
    goto exit;
  }
  pml_parameter_parser.query("print_arrays", print_pml_arrays);
  if (print_pml_arrays) {
    pout() ≪ "\n\tPrinting␣PML␣arrays" ≪ endl;
    ⟨ print Ckx arrays 99 ⟩
    ⟨ print Cky arrays 100 ⟩
    ⟨ print Ckz arrays 101 ⟩
  }
```

99 ¶⟨ print *Ckx* arrays 99 ⟩ ≡

```
pout() << "\tC0x:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C0x.size(); count++) {
  pout() << "␣" << levels[0]->C0x[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC1x:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C1x.size(); count++) {
  pout() << "␣" << levels[0]->C1x[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC2x:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C2x.size(); count++) {
  pout() << "␣" << levels[0]->C2x[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC3x:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C3x.size(); count++) {
  pout() << "␣" << levels[0]->C3x[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC4x:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C4x.size(); count++) {
  pout() << "␣" << levels[0]->C4x[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
```

This code is used in chunk 98.

100 ¶⟨ print *Cky* arrays 100 ⟩ ≡

```
pout() << "\tC0y:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C0y.size(); count++) {
  pout() << "␣" << levels[0]->C0y[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC1y:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C1y.size(); count++) {
  pout() << "␣" << levels[0]->C1y[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC2y:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C2y.size(); count++) {
  pout() << "␣" << levels[0]->C2y[count];
  if (¬(count % 6)) pout() << endl << "\t\t";
}
pout() << endl;
pout() << "\tC3y:" << endl << "\t\t";
for (int count = 0; count < levels[0]->C3y.size(); count++) {
```

```
    pout() << "␣" << levels[0]→C3y[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
  pout() << "\tC4y:" << endl << "\t\t";
  for (int count = 0; count < levels[0]→C4y.size(); count++) {
    pout() << "␣" << levels[0]→C4y[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
```

This code is used in chunk 98.

101 ¶⟨ print *Ckz* arrays 101 ⟩ ≡

```
  pout() << "\tC0z:" << endl << "\t\t";
  for (int count = 0; count < levels[0]→C0z.size(); count++) {
    pout() << "␣" << levels[0]→C0z[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
  pout() << "\tC1z:" << endl << "\t\t";
  for (int count = 0; count < levels[0]→C1z.size(); count++) {
    pout() << "␣" << levels[0]→C1z[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
  pout() << "\tC2z:" << endl << "\t\t";
  for (int count = 0; count < levels[0]→C2z.size(); count++) {
    pout() << "␣" << levels[0]→C2z[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
  pout() << "\tC3z:" << endl << "\t\t";
  for (int count = 0; count < levels[0]→C3z.size(); count++) {
    pout() << "␣" << levels[0]→C3z[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
  pout() << "\tC4z:" << endl << "\t\t";
  for (int count = 0; count < levels[0]→C4z.size(); count++) {
    pout() << "␣" << levels[0]→C4z[count];
    if (¬(count % 6)) pout() << endl << "\t\t";
  }
  pout() << endl;
```

This code is used in chunk 98.

102 ¶⟨ Function *fill_levels* 94 ⟩ +≡

```
      /* Now we have to draw media on levels[n]→medium by invoking the appropriate Scheme procedure. */
  if (media_parameter_parser.isProcedure("distribution.lambda")) {
    SCM lambda = scm_variable_ref(scm_c_lookup("forms.media.distribution.lambda"));
    for (int n = 0; n < number_of_levels; n++) {
      current_level_ptr = levels[n];
      SCM lambda_return = scm_call_0(lambda);
```

```
      if (scm_is_bool(lambda_return)) {
        if (scm_is_false(lambda_return)) {
          pout() << "\n\tERROR(build_levels): Lambda returned #f" << endl;
          return_status = ERR_FILL_LEVELS;
          goto exit;
        }
      }
    }
  }
  else {
    pout() << "\n\tWARNING(build_levels): distribution lambda not provided" << endl;
        /* This is not an error—users may wish to define media within the d_to_e script, or just propagate in
        vacuum. */
  }
```

103 ¶⟨ Function *fill_levels* 94 ⟩ +≡

```
exit:
  if (watch_fill_levels) pout() << "fill_levels: returning to caller" << endl;
  return return_status; }
```

7 Iteration

Here we define the content of file *Iteration.c*. Functions used in the iteration loop are defined here.

104 ⟨ `Iteration.c` 104 ⟩ ≡

```
#include <Forms.H>
    /* Evolving the D and E fields */
  ⟨ Function advance_d 105 ⟩
  ⟨ Function advance_d_0 106 ⟩
  ⟨ Function advance_d_n 107 ⟩
  ⟨ Function inject_d 108 ⟩
  ⟨ Function d_to_e 115 ⟩
  ⟨ Function d_to_e_0 116 ⟩
  ⟨ Function d_to_e_0_0 117 ⟩
  ⟨ Function d_to_e_0_n 118 ⟩
  ⟨ Function d_to_e_n 119 ⟩
  ⟨ Function d_to_e_n_0 120 ⟩
  ⟨ Function d_to_e_n_n 121 ⟩
    /* Evolving the B and H fields */
  ⟨ Function advance_b 125 ⟩
  ⟨ Function advance_b_0 126 ⟩
  ⟨ Function advance_b_n 127 ⟩
  ⟨ Function inject_b 128 ⟩
  ⟨ Function b_to_h 131 ⟩
  ⟨ Function b_to_h_0 132 ⟩
  ⟨ Function b_to_h_n 133 ⟩
```

7.1 Advance the $\boldsymbol{D}$ field

The main function here is just a simple wrapper that checks the level number and calls a more detailed wrapper. Level zero requires a more involved advance than a higher level, because here we need to attend to UPMLs and inject the signal.

Consequently, whereas a higher level advance is given by the relatively simple equations (188) through (190), level zero advance is given by the UPML equations (318) through (320), which have to be combined with the signal injection equations (241) through (264), and the additional UPML $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion equations (324) through (325).

105 ⟨Function *advance_d* 105⟩ ≡

```
int advance_d(level *level_ptr)
{
  int watch_advance_d = false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_d", watch_advance_d);
  if (watch_advance_d) pout() << "advance_d:␣level␣#" << level_number << endl;
  if (level_number == 0)    /* level 0: special advance is needed here */
  {
    if (watch_advance_d) pout() << "\tCalling␣advance_d_0" << endl;
    if (return_status = advance_d_0(level_ptr)) {
      pout() << "\tERROR(advance_d):␣Failed␣to␣advance␣level␣0" << endl;
      goto exit;
    }
  }
  else    /* not level 0: simple advance */
  {
    if (watch_advance_d) pout() << "\tCalling␣advance_d_n" << endl;
    if (return_status = advance_d_n(level_ptr)) {
      pout() << "\tERROR(advance_d):␣Failed␣to␣advance␣level␣" << level_number << endl;
      goto exit;
    }
  }
  level_ptr→time_e += level_ptr→dt;
exit: return return_status;
}
```

This code is used in chunk 104.

7.1.1 Advance Level Zero (with UPML)

106 ⟨Function *advance_d_0* 106⟩ ≡

```
int advance_d_0(level *level_ptr)
{
  int watch_advance_d = false;
  int watch_update_d = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator &data_iterator = level_ptr→data_iterator;
  int Cx_lo = -ghost_margin;
  int Cx_hi = (int) level_ptr→C1x.size() - ghost_margin;
  int Cy_lo = -ghost_margin;
  int Cy_hi = (int) level_ptr→C1y.size() - ghost_margin;
```

```
int Cz_lo = -ghost_margin;
int Cz_hi = (int) level_ptr->C1z.size() - ghost_margin;
Interval current_component;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("advance_d", watch_advance_d);
watch_parameter_parser.query("update_d", watch_update_d);
if (watch_advance_d) pout() << "\tadvance_d_0:" << endl;
if (level_number != 0) {
  pout() << "\t\tERROR(advance_d_0): This is not level zero" << endl;
  return_status = ERR_ADVANCE_D;
  goto exit;
}
current_component.define(level_ptr->H_idx[0], level_ptr->H_idx[0]);
level_ptr->H.exchange(current_component);
swap_idx(level_ptr->D_idx);      /* swap the new/old fields */
if (watch_advance_d > 1) pout() << "\t\tentering the box loop" << endl << flush;
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  FArrayBox &Dx = level_ptr->D[data_index][0];
  FArrayBox &Dy = level_ptr->D[data_index][1];
  FArrayBox &Dz = level_ptr->D[data_index][2];
  FArrayBox &Hx = level_ptr->H[data_index][0];
  FArrayBox &Hy = level_ptr->H[data_index][1];
  FArrayBox &Hz = level_ptr->H[data_index][2];
  Vector<Real> &C1x = level_ptr->C1x;
  Vector<Real> &C2x = level_ptr->C2x;
  Vector<Real> &C1y = level_ptr->C1y;
  Vector<Real> &C2y = level_ptr->C2y;
  Vector<Real> &C1z = level_ptr->C1z;
  Vector<Real> &C2z = level_ptr->C2z;
  if (watch_advance_d > 1)
    pout() << "\t\t\tcalling update_d_upml_ on box " << Dx.box() << " ... " << flush;
  update_d_upml_(
  &SpaceDim, &ghost_margin, &(level_ptr->D_idx[0]), &(level_ptr->H_idx[0]),
  &Cx_lo, &Cx_hi, &C1x[0], &C2x[0],
  &Cy_lo, &Cy_hi, &C1y[0], &C2y[0],
  &Cz_lo, &Cz_hi, &C1z[0], &C2z[0],
  &(Dx.loVect()[0]), &(Dx.loVect()[1]), &(Dx.loVect()[2]), &(Dx.hiVect()[0]), &(Dx.hiVect()[1]),
      &(Dx.hiVect()[2]), Dx.dataPtr(),
  &(Dy.loVect()[0]), &(Dy.loVect()[1]), &(Dy.loVect()[2]), &(Dy.hiVect()[0]), &(Dy.hiVect()[1]),
      &(Dy.hiVect()[2]), Dy.dataPtr(),
  &(Dz.loVect()[0]), &(Dz.loVect()[1]), &(Dz.loVect()[2]), &(Dz.hiVect()[0]), &(Dz.hiVect()[1]),
      &(Dz.hiVect()[2]), Dz.dataPtr(),
  &(Hx.loVect()[0]), &(Hx.loVect()[1]), &(Hx.loVect()[2]), &(Hx.hiVect()[0]), &(Hx.hiVect()[1]),
      &(Hx.hiVect()[2]), Hx.dataPtr(),
  &(Hy.loVect()[0]), &(Hy.loVect()[1]), &(Hy.loVect()[2]), &(Hy.hiVect()[0]), &(Hy.hiVect()[1]),
      &(Hy.hiVect()[2]), Hy.dataPtr(),
  &(Hz.loVect()[0]), &(Hz.loVect()[1]), &(Hz.loVect()[2]), &(Hz.hiVect()[0]), &(Hz.hiVect()[1]),
      &(Hz.hiVect()[2]), Hz.dataPtr(),
  &watch_update_d, &return_status
  );
  if (watch_advance_d > 1) pout() << "done." << endl << flush;
  if (return_status != EXIT_SUCCESS) {
    pout() << "\t\tERROR(advance_d_0): update_d_upml_ returned " << return_status << endl;
```

```
        goto exit;
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 104.

7.1.2 Advance Higher Levels

107 ⟨Function *advance_d_n* 107⟩ ≡

```
  int advance_d_n(level *level_ptr)
  {
    int watch_advance_d = false;
    int watch_update_d = false;
    int return_status = EXIT_SUCCESS;
    int ghost_margin = N_GHOST_CELLS;
    int level_number = level_ptr→number;
    Real dt_by_dg = (level_ptr→dt)/(level_ptr→delta);
    DataIterator &data_iterator = level_ptr→data_iterator;
    Interval current_component;
    SCMParmParse watch_parameter_parser("forms.watch");
    watch_parameter_parser.query("advance_d", watch_advance_d);
    watch_parameter_parser.query("update_d", watch_update_d);
    if (watch_advance_d) pout() ≪ "\tadvance_d_n:" ≪ endl;
    if (level_number ≡ 0) {
      pout() ≪ "\t\tERROR(advance_d_n):␣invoked␣on␣level␣zero" ≪ endl;
      return_status = ERR_ADVANCE_D;
      goto exit;
    }
    current_component.define(level_ptr→H_idx[0], level_ptr→H_idx[0]);
    level_ptr→H.exchange(current_component);
    swap_idx(level_ptr→D_idx);      /* swap new/old fields */
        /* Loop over boxes */
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      FArrayBox &Dx = level_ptr→D[data_index][0];
      FArrayBox &Dy = level_ptr→D[data_index][1];
      FArrayBox &Dz = level_ptr→D[data_index][2];
      FArrayBox &Hx = level_ptr→H[data_index][0];
      FArrayBox &Hy = level_ptr→H[data_index][1];
      FArrayBox &Hz = level_ptr→H[data_index][2];
      update_d_(
      &SpaceDim, &ghost_margin,
      &(level_ptr→D_idx[0]), &(level_ptr→H_idx[0]),
      &dt_by_dg,
      &(Dx.loVect()[0]), &(Dx.loVect()[1]), &(Dx.loVect()[2]), &(Dx.hiVect()[0]), &(Dx.hiVect()[1]),
          &(Dx.hiVect()[2]), Dx.dataPtr(),
      &(Dy.loVect()[0]), &(Dy.loVect()[1]), &(Dy.loVect()[2]), &(Dy.hiVect()[0]), &(Dy.hiVect()[1]),
          &(Dy.hiVect()[2]), Dy.dataPtr(),
      &(Dz.loVect()[0]), &(Dz.loVect()[1]), &(Dz.loVect()[2]), &(Dz.hiVect()[0]), &(Dz.hiVect()[1]),
          &(Dz.hiVect()[2]), Dz.dataPtr(),
      &(Hx.loVect()[0]), &(Hx.loVect()[1]), &(Hx.loVect()[2]), &(Hx.hiVect()[0]), &(Hx.hiVect()[1]),
          &(Hx.hiVect()[2]), Hx.dataPtr(),
```

```
    &(Hy.loVect()[0]), &(Hy.loVect()[1]), &(Hy.loVect()[2]), &(Hy.hiVect()[0]), &(Hy.hiVect()[1]),
        &(Hy.hiVect()[2]), Hy.dataPtr(),
    &(Hz.loVect()[0]), &(Hz.loVect()[1]), &(Hz.loVect()[2]), &(Hz.hiVect()[0]), &(Hz.hiVect()[1]),
        &(Hz.hiVect()[2]), Hz.dataPtr(),
    &watch_update_d, &return_status
    );
    if (return_status ≠ EXIT_SUCCESS) {
      pout() << "\t\tERROR(advance_d_n): update_d_returned " << return_status << endl;
      goto exit;
    }
  }
exit: return return_status;
}
```

This code is used in chunk 104.

7.2 Inject the $\boldsymbol{D}$ field

108 ⟨Function *inject_d* 108⟩ ≡

```
int inject_d(level *level_ptr)
{
    /* Inject the D field on the total/scattered field boundary. */
  int return_status = EXIT_SUCCESS;
  int watch_inject_d = (int) false;
  int garbage_collect = (int) false;
  SideIterator side_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse signal_parameter_parser("forms.signal");
  watch_parameter_parser.query("inject_d", watch_inject_d);
  signal_parameter_parser.query("garbage_collect", garbage_collect);
  if (watch_inject_d) pout() << "inject_d:" << endl;
  if (level_ptr->number ≠ 0) {
    pout() << "\tERROR(inject_d): not level zero" << endl;
    return_status = ERR_INJECT_D;
    goto exit;
  }
  for (int dir = 0; dir < SpaceDim; dir++) {
    for (side_iterator.reset(); side_iterator.ok(); ++side_iterator) {
      Side::LoHiSide side = side_iterator();
      if (return_status = inject_d(level_ptr, dir, side)) {
        pout() << "\tERROR(inject_d): failed on dir = " << dir << ", side = " << side << endl;
        goto exit;
      }
    }
  }
  if (garbage_collect) scm_gc();
exit: return return_status;
}
```

See also chunk 109.

This code is used in chunk 104.

109 ¶⟨Function *inject_d* 108⟩ +≡

```
int inject_d(level *level_ptr, int dir, Side::LoHiSide side)
{
  int return_status = EXIT_SUCCESS;
  int watch_inject_d = (int) false;
  int side_number = (int) side;
  int ghost_margin = N_GHOST_CELLS;
  IntVect ijk_lo = level_ptr->signal_ijk_lo;
  IntVect ijk_hi = level_ptr->signal_ijk_hi;
  Vector<Real> &origin = level_ptr->origin;
  Real &delta = level_ptr->delta;
  LevelData<CurlBox> &D = level_ptr->D;
  int &new_component = level_ptr->D_idx[0];
  Vector<DataIndex> &side_boxes = level_ptr->TFR_face_boxes[dir][side];
  Real dt_by_delta = level_ptr->dt/level_ptr->delta;
  Real time_h = level_ptr->time_h;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("inject_d", watch_inject_d);
  if (watch_inject_d) pout() << "\tinject_d: dir = " << dir << ", side = " << side << endl;
```

```
if (dir < 0 ∨ dir ≥ SpaceDim) {
  pout() << "\t\tERROR(inject_d): bad direction" << endl;
  return_status = ERR_INJECT_D;
  goto exit;
}
for (int count = 0; count < side_boxes.size(); count++) {
    /* This loop picks up only the boxes that cross this side and that live on this processor. */
  DataIndex data_index = side_boxes[count];
  switch (dir) {
  case 0:     /* West–East */
    {
      FArrayBox &Dy = D[data_index][1];
      FArrayBox &Dz = D[data_index][2];
      IntVect Dy_lo = Dy.smallEnd();
      IntVect Dy_hi = Dy.bigEnd();
      IntVect Dz_lo = Dz.smallEnd();
      IntVect Dz_hi = Dz.bigEnd();
      inject_d_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_h, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(Dy_lo[0]), &(Dy_lo[1]), &(Dy_lo[2]), &(Dy_hi[0]), &(Dy_hi[1]),
          &(Dy_hi[2]), Dy.dataPtr(), &(Dz_lo[0]), &(Dz_lo[1]), &(Dz_lo[2]), &(Dz_hi[0]), &(Dz_hi[1]),
          &(Dz_hi[2]), Dz.dataPtr(), &watch_inject_d, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() << "\tERROR(inject_d): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
    }
    break;
  case 1:     /* Front–Back */
    {
      FArrayBox &Dx = D[data_index][0];
      FArrayBox &Dz = D[data_index][2];
      IntVect Dx_lo = Dx.smallEnd();
      IntVect Dx_hi = Dx.bigEnd();
      IntVect Dz_lo = Dz.smallEnd();
      IntVect Dz_hi = Dz.bigEnd();
      inject_d_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_h, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(Dx_lo[0]), &(Dx_lo[1]), &(Dx_lo[2]), &(Dx_hi[0]), &(Dx_hi[1]),
          &(Dx_hi[2]), Dx.dataPtr(), &(Dz_lo[0]), &(Dz_lo[1]), &(Dz_lo[2]), &(Dz_hi[0]), &(Dz_hi[1]),
          &(Dz_hi[2]), Dz.dataPtr(), &watch_inject_d, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() << "\tERROR(inject_d): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
    }
    break;
  case 2:     /* Bottom–Top */
    {
      FArrayBox &Dx = D[data_index][0];
      FArrayBox &Dy = D[data_index][1];
      IntVect Dx_lo = Dx.smallEnd();
      IntVect Dx_hi = Dx.bigEnd();
      IntVect Dy_lo = Dy.smallEnd();
```

```
        IntVect Dy_hi = Dy.bigEnd();
        inject_d_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
            &(origin[2]), &delta, &dt_by_delta, &time_h, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
            &(ijk_hi[1]), &(ijk_hi[2]), &(Dx_lo[0]), &(Dx_lo[1]), &(Dx_lo[2]), &(Dx_hi[0]), &(Dx_hi[1]),
            &(Dx_hi[2]), Dx.dataPtr(), &(Dy_lo[0]), &(Dy_lo[1]), &(Dy_lo[2]), &(Dy_hi[0]), &(Dy_hi[1]),
            &(Dy_hi[2]), Dy.dataPtr(), &watch_inject_d, &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() << "\tERROR(inject_d): Fortran problem, dir = " << dir << endl;
          goto exit;
        }
      }
      break;
    }    /* switch */
  }    /* for (int count = 0; count < side_boxes.size(); count++) */
exit: return return_status;
}
```

7.2.1 Incident Field Functions

These functions must be compiled with plain C, because they're called by Fortran. When they are compiled with C++, the loader fails.

For this to work seamlessly with the Makefile, changes must be made to $(CHOMBO_HOME)/mk/Make.rules, namely, a $(_lib_config) (%. o) : %. c rule must be added in the "rules to build objects for executable programs" section, and, possibly, rules may have to be added to the section that runs *makedepend*. It's rather messy and it is an ommission on the side of Chombo engineers not to have provided them for plain C.

Magnetic Field Injection

These three functions, *hx_inc_*, *hy_inc_*, *hz_inc_*, are called by the Fortran subroutine *inject_d_*. They are basically C wrappers for the Scheme call that implements the injection functions $E_x(\zeta)$, $E_y(\zeta)$ and $E_z(\zeta)$, where $\zeta = \boldsymbol{n} \cdot \boldsymbol{r} - t$. The wrapper implements $\boldsymbol{H} = \boldsymbol{n} \times \boldsymbol{E}(\zeta)$.

In explicit notation this is

$$\begin{aligned} H_x &= n_y E_z - n_z E_y, \\ H_y &= n_z E_x - n_x E_z, \\ H_z &= n_x E_y - n_y E_x. \end{aligned}$$

$\boldsymbol{n}$ is obtained from the main program globals, where it is put by the Scheme initialization routine.

Note that because these three functions are callable from Fortran, the arguments are *addresses* of x, y, z, and t, not their actual values. Same applies to *ex_inc_*, *ey_inc_*, and *ez_inc_*.

111 ⟨ Injection.c_std 111 ⟩ ≡

```
#include <REAL.H>
#include <libguile.h>
#include "./Injection.h"
  Real hx_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
  {
    extern Real ny, nz;
    return (ny * ez_inc_(x, y, z, t) − nz * ey_inc_(x, y, z, t));
  }
  Real hy_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
  {
    extern Real nx, nz;
```

```
  return (nz * ex_inc_(x, y, z, t) - nx * ez_inc_(x, y, z, t));
}
Real hz_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
  extern Real nx, ny;
  return (nx * ey_inc_(x, y, z, t) - ny * ex_inc_(x, y, z, t));
}
```

See also chunk 112.

Electric Field Injection

The signal injection is defined by the user in the $\boldsymbol{E}$ terms. The user specifies the direction of the signal by providing vector $\boldsymbol{n}$. The vector does not have to be normalized, because the program normalizes it.

The user also has to provide three functions that define $E_x(\zeta)$, $E_y(\zeta)$ and $E_z(\zeta)$, where $\zeta = \boldsymbol{n} \cdot \boldsymbol{r} - t$. Vector $\boldsymbol{E}$ should be perpendicular to $\boldsymbol{n}$. In case the user forgets about it, the program enforces this anyway. This, however, is costly, so the user, who's certain that $\forall_\zeta \boldsymbol{E}(\zeta) \perp \boldsymbol{n}$, may tell the program that this is the case, whereupon a global variable *normalized* will be set to *true*, and the perpendicularization of $\boldsymbol{E}$ will be omitted.

112 ⟨ `Injection.c_std` 111 ⟩ +≡

```
Real ex_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
    /* global variables */
  extern Real nx, ny, nz;
  extern SCM ex_lambda, ey_lambda, ez_lambda;
  extern int normalized;
      /* local variables */
  Real zeta;
  Real ex = 0.0, ey, ez;
  Real e_times_n;
  zeta = nx * (*x) + ny * (*y) + nz * (*z) - (*t);
  ex = scm_to_double(scm_call_1(ex_lambda, scm_from_double(zeta)));
  if (¬normalized) {
    ey = scm_to_double(scm_call_1(ey_lambda, scm_from_double(zeta)));
    ez = scm_to_double(scm_call_1(ez_lambda, scm_from_double(zeta)));
    e_times_n = ex * nx + ey * ny + ez * nz;
    ex = ex - e_times_n * nx;
  }
  return ex;
}
Real ey_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
    /* global variables */
  extern Real nx, ny, nz;
  extern SCM ex_lambda, ey_lambda, ez_lambda;
  extern int normalized;
      /* local variables */
  Real zeta;
  Real ex, ey = 0.0, ez;
  Real e_times_n;
  zeta = nx * (*x) + ny * (*y) + nz * (*z) - (*t);
  ey = scm_to_double(scm_call_1(ey_lambda, scm_from_double(zeta)));
  if (¬normalized) {
    ex = scm_to_double(scm_call_1(ex_lambda, scm_from_double(zeta)));
    ez = scm_to_double(scm_call_1(ez_lambda, scm_from_double(zeta)));
```

```
    e_times_n = ex * nx + ey * ny + ez * nz;
    ey = ey - e_times_n * ny;
  }
  return ey;
}
Real ez_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
    /* global variables */
  extern Real nx, ny, nz;
  extern SCM ex_lambda, ey_lambda, ez_lambda;
  extern int normalized;
      /* local variables */
  Real zeta;
  Real ex, ey, ez = 0.0;
  Real e_times_n;
  zeta = nx * (*x) + ny * (*y) + nz * (*z) - (*t);
  ez = scm_to_double(scm_call_1(ez_lambda, scm_from_double(zeta)));
  if (¬normalized) {
    ex = scm_to_double(scm_call_1(ex_lambda, scm_from_double(zeta)));
    ey = scm_to_double(scm_call_1(ey_lambda, scm_from_double(zeta)));
    e_times_n = ex * nx + ey * ny + ez * nz;
    ez = ez - e_times_n * nz;
  }
  return ez;
}
```

Injection Includes

113 ⟨ Injection.h_std 113 ⟩ ≡

```
Real hx_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real hy_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real hz_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real ex_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real ey_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real ez_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
```

7.3 Convert D to E

7.3.1 Function d_to_e

115 ⟨Function *d_to_e* 115⟩ ≡

```
int d_to_e(level *level_ptr)
{
  int watch_d_to_e = (int) false;
  int garbage_collect = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  if (watch_d_to_e) {
    pout() << "d_to_e:" << endl;
    pout() << "\tlevel_number␣=␣" << level_number << endl;
  }
  if (level_number ≡ 0) {
    if (return_status = d_to_e_0(level_ptr)) {
      pout() << "\tERROR(d_to_e):␣failure␣in␣d_to_e_0:␣" << return_status << endl;
      goto exit;
    }
  }
  else {
    if (return_status = d_to_e_n(level_ptr)) {
      pout() << "\tERROR(d_to_e):␣failure␣in␣d_to_e_n:␣" << return_status << endl;
      goto exit;
    }
  }
  media_parameter_parser.query("garbage_collect", garbage_collect);
  if (garbage_collect) scm_gc();
exit:
  if (watch_d_to_e) pout() << "d_to_e:␣returning␣to␣caller" << endl;
  return return_status;
}
```

This code is used in chunk 104.

7.3.2 Function d_to_e_0

116 ⟨Function *d_to_e_0* 116⟩ ≡

```
int d_to_e_0(level *level_ptr)
{
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  int number_of_auxiliary_fields = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
  if (watch_d_to_e) {
    pout() << "\td_to_e_0:" << endl;
    pout() << "\t\tnumber_of_auxiliary_fields␣=␣" << number_of_auxiliary_fields << endl;
```

```
    }
    if (level_number ≠ 0) {
      pout() << "\t\tERROR(d_to_e_0): not level zero" << endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    if (number_of_auxiliary_fields) {
      if (return_status = d_to_e_0_n(level_ptr)) {
        pout() << "\t\tERROR(d_to_e_0): failure in d_to_e_0_n" << endl;
        goto exit;
      }
    }
    else {
      if (return_status = d_to_e_0_0(level_ptr)) {
        pout() << "\t\tERROR(d_to_e_0): failure in d_to_e_0_0" << endl;
        goto exit;
      }
    }
  exit:
    if (watch_d_to_e) pout() << "\td_to_e_0: returning to caller" << endl;
    return return_status;
  }
```

This code is cited in chunk 22.

This code is used in chunk 104.

7.3.3 Function d_to_e_0_0

117 ⟨Function *d_to_e_0_0* 117⟩ ≡

```
  int d_to_e_0_0(level *level_ptr)
  {
      /* Convert D to E in every cell of this level zero with no auxiliary fields. */
    int watch_d_to_e = (int) false;
    int return_status = EXIT_SUCCESS;
    int ghost_margin = N_GHOST_CELLS;
    int level_number = level_ptr→number;
    int number_of_auxiliary_fields = 0;
    DataIterator data_iterator = level_ptr→data_iterator;
    SCMParmParse watch_parameter_parser("forms.watch");
    SCMParmParse media_parameter_parser("forms.media");
    watch_parameter_parser.query("d_to_e", watch_d_to_e);
    media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
    if (watch_d_to_e) pout() << "\t\td_to_e_0_0:" << endl;
    if (level_number) {
      pout() << "\t\t\tERROR(d_to_e_0_0): not level zero" << endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    if (number_of_auxiliary_fields) {
      pout() << "\t\t\tERROR(d_to_e_0_0): auxiliary fields needed" << endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    swap_idx(level_ptr→E_idx);      /* swap the new/old field index */
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
```

```
      DataIndex data_index = data_iterator();
      CurlBox &E = level_ptr→E[data_index];
      CurlBox &D = level_ptr→D[data_index];
      EdgeFab⟨int⟩ &medium_E = level_ptr→medium_E[data_index];
      Real &time_e = level_ptr→time_e;
      Real &dt = level_ptr→dt;
      Vector⟨Real⟩ &C0x = level_ptr→C0x;
      Vector⟨Real⟩ &C1x = level_ptr→C1x;
      Vector⟨Real⟩ &C3x = level_ptr→C3x;
      Vector⟨Real⟩ &C4x = level_ptr→C4x;
      Vector⟨Real⟩ &C0y = level_ptr→C0y;
      Vector⟨Real⟩ &C1y = level_ptr→C1y;
      Vector⟨Real⟩ &C3y = level_ptr→C3y;
      Vector⟨Real⟩ &C4y = level_ptr→C4y;
      Vector⟨Real⟩ &C0z = level_ptr→C0z;
      Vector⟨Real⟩ &C1z = level_ptr→C1z;
      Vector⟨Real⟩ &C3z = level_ptr→C3z;
      Vector⟨Real⟩ &C4z = level_ptr→C4z;
      int Cx_lo = −ghost_margin;
      int Cx_hi = (int) C1x.size() − ghost_margin;
      int Cy_lo = −ghost_margin;
      int Cy_hi = (int) C1y.size() − ghost_margin;
      int Cz_lo = −ghost_margin;
      int Cz_hi = (int) C1z.size() − ghost_margin;
      for (int dir = 0; dir < SpaceDim; dir++) {
        FArrayBox &E_dir = E[dir];
        FArrayBox &D_dir = D[dir];
        BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
        d_to_e_0_0_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→E_idx[0]), &(level_ptr→D_idx[0]), &time_e,
            &dt, &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0], &Cy_lo, &Cy_hi, &C0y[0], &C1y[0],
            &C3y[0], &C4y[0], &Cz_lo, &Cz_hi, &C0z[0], &C1z[0], &C3z[0], &C4z[0], &(D_dir.loVect()[0]),
            &(D_dir.loVect()[1]), &(D_dir.loVect()[2]), &(D_dir.hiVect()[0]), &(D_dir.hiVect()[1]),
            &(D_dir.hiVect()[2]), D_dir.dataPtr(), E_dir.dataPtr(), medium_E_dir.dataPtr(), &watch_d_to_e,
            &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() ≪ "\t\t\tERROR(d_to_e_0_0): failure in d_to_e_0_0_: " ≪ return_status ≪ endl;
          goto exit;
        }
      }    /* for (int dir = 0; dir < SpaceDim; dir++) */
    }    /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  exit:
    if (watch_d_to_e) pout() ≪ "\t\td_to_e_0_0: returning to caller" ≪ endl;
    return return_status;
  }
```

This code is used in chunk 104.

7.3.4 Function d_to_e_0_n

118 ⟨Function *d_to_e_0_n* 118⟩ ≡

```
  int d_to_e_0_n(level *level_ptr)
  {
      /* Convert D to E in every cell of this level zero with auxiliary fields. */
    int watch_d_to_e = (int) false;
    int return_status = EXIT_SUCCESS;
```

```
int ghost_margin = N_GHOST_CELLS;
int level_number = level_ptr→number;
int number_of_auxiliary_fields = 0;
DataIterator data_iterator = level_ptr→data_iterator;
SCMParmParse watch_parameter_parser("forms.watch");
SCMParmParse media_parameter_parser("forms.media");
watch_parameter_parser.query("d_to_e", watch_d_to_e);
media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
if (watch_d_to_e) pout() << "\t\td_to_e_0_n:" << endl;
if (level_number) {
  pout() << "\t\t\tERROR(d_to_e_0_n): not level zero" << endl;
  return_status = ERR_D_TO_E;
  goto exit;
}
if (¬number_of_auxiliary_fields) {
  pout() << "\t\t\tERROR(d_to_e_0_n): no auxiliary fields" << endl;
  return_status = ERR_D_TO_E;
  goto exit;
}
swap_idx(level_ptr→E_idx);      /* swap the new/old field index */
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  CurlBox &E = level_ptr→E[data_index];
  CurlBox &D = level_ptr→D[data_index];
  CurlBox &S = level_ptr→S[data_index];
  EdgeFab<int> &medium_E = level_ptr→medium_E[data_index];
  Real &time_e = level_ptr→time_e;
  Real &dt = level_ptr→dt;
  Vector<Real> &C0x = level_ptr→C0x;
  Vector<Real> &C1x = level_ptr→C1x;
  Vector<Real> &C3x = level_ptr→C3x;
  Vector<Real> &C4x = level_ptr→C4x;
  Vector<Real> &C0y = level_ptr→C0y;
  Vector<Real> &C1y = level_ptr→C1y;
  Vector<Real> &C3y = level_ptr→C3y;
  Vector<Real> &C4y = level_ptr→C4y;
  Vector<Real> &C0z = level_ptr→C0z;
  Vector<Real> &C1z = level_ptr→C1z;
  Vector<Real> &C3z = level_ptr→C3z;
  Vector<Real> &C4z = level_ptr→C4z;
  int Cx_lo = −ghost_margin;
  int Cx_hi = (int) C1x.size() − ghost_margin;
  int Cy_lo = −ghost_margin;
  int Cy_hi = (int) C1y.size() − ghost_margin;
  int Cz_lo = −ghost_margin;
  int Cz_hi = (int) C1z.size() − ghost_margin;
  if (S.nComp() ≠ number_of_auxiliary_fields) {
    pout() << "\t\t\tERROR(d_to_e_0_n): bad number of S components" << endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  for (int dir = 0; dir < SpaceDim; dir++) {
    FArrayBox &E_dir = E[dir];
    FArrayBox &D_dir = D[dir];
```

```
      FArrayBox &S_dir = S[dir];
      BaseFab<int> &medium_E_dir = medium_E[dir];
      d_to_e_0_n_(&SpaceDim, &dir, &ghost_margin, &number_of_auxiliary_fields, &(level_ptr→E_idx[0]),
          &(level_ptr→D_idx[0]), &time_e, &dt, &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0],
          &Cy_lo, &Cy_hi, &C0y[0], &C1y[0], &C3y[0], &C4y[0], &Cz_lo, &Cz_hi, &C0z[0], &C1z[0],
          &C3z[0], &C4z[0], &(D_dir.loVect()[0]), &(D_dir.loVect()[1]), &(D_dir.loVect()[2]),
          &(D_dir.hiVect()[0]), &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]), D_dir.dataPtr(),
          E_dir.dataPtr(), S_dir.dataPtr(), medium_E_dir.dataPtr(), &watch_d_to_e, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() << "\t\t\tERROR(d_to_e_0_n): failure in d_to_e_0_n_: " << return_status << endl;
        goto exit;
      }
    }    /* for (int dir = 0; dir < SpaceDim; dir++) */
  }    /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
exit:
  if (watch_d_to_e) pout() << "\t\td_to_e_0_n: returning to caller" << endl;
  return return_status;
}
```

This code is cited in chunk 22.

This code is used in chunk 104.

7.3.5 Function d_to_e_n

119 ⟨Function d_to_e_n 119⟩ ≡

```
int d_to_e_n(level *level_ptr)
{
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  int number_of_auxiliary_fields = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
  if (watch_d_to_e) {
    pout() << "\td_to_e_n:" << endl;
    pout() << "\t\tnumber_of_auxiliary_fields = " << number_of_auxiliary_fields << endl;
  }
  if (level_number ≡ 0) {
    pout() << "\t\tERROR(d_to_e_n): invoked on level zero" << endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  if (number_of_auxiliary_fields) {
    if (return_status = d_to_e_n_n(level_ptr)) {
      pout() << "\t\tERROR(d_to_e_n): failure in d_to_e_n_n" << endl;
      goto exit;
    }
  }
  else {
    if (return_status = d_to_e_n_0(level_ptr)) {
      pout() << "\t\tERROR(d_to_e_n): failure in d_to_e_n_0" << endl;
      goto exit;
    }
```

```
    }
  exit:
    if (watch_d_to_e) pout() << "\td_to_e_n:␣returning␣to␣caller" << endl;
    return return_status;
  }
```

This code is used in chunk 104.

7.3.6 Function d_to_e_n_0

120 ⟨Function *d_to_e_n_0* 120⟩ ≡

```
int d_to_e_n_0(level *level_ptr)
{
    /* Convert D to E in every cell of this level (other than zero) with no auxiliary fields. */
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  int number_of_auxiliary_fields = 0;
  DataIterator data_iterator = level_ptr→data_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
  if (watch_d_to_e) pout() << "\t\td_to_e_n_0:" << endl;
  if (¬level_number) {
    pout() << "\t\t\tERROR(d_to_e_n_0):␣invoked␣on␣level␣zero" << endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  if (number_of_auxiliary_fields) {
    pout() << "\t\t\tERROR(d_to_e_n_0):␣auxiliary␣fields␣needed" << endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  swap_idx(level_ptr→E_idx);
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    CurlBox &E = level_ptr→E[data_index];
    CurlBox &D = level_ptr→D[data_index];
    EdgeFab⟨int⟩ &medium_E = level_ptr→medium_E[data_index];
    Real &time_e = level_ptr→time_e;
    Real &dt = level_ptr→dt;
    for (int dir = 0; dir < SpaceDim; dir++) {
      FArrayBox &E_dir = E[dir];
      FArrayBox &D_dir = D[dir];
      BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
      d_to_e_n_0_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→E_idx[0]), &(level_ptr→D_idx[0]), &time_e,
          &dt, &(D_dir.loVect()[0]), &(D_dir.loVect()[1]), &(D_dir.loVect()[2]), &(D_dir.hiVect()[0]),
          &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]), D_dir.dataPtr(), E_dir.dataPtr(),
          medium_E_dir.dataPtr(), &watch_d_to_e, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() << "\t\t\tERROR(d_to_e_n_0):␣failure␣in␣d_to_e_n_0_:␣" << return_status << endl;
        goto exit;
```

```
        }
      }     /* for (int dir = 0; dir < SpaceDim; dir++) */
    }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  exit:
    if (watch_d_to_e) pout() << "\t\td_to_e_n_0: returning to caller" << endl;
    return return_status;
  }
```

This code is used in chunk 104.

7.3.7 Function d_to_e_n_n

121 ⟨Function d_to_e_n_n 121⟩ ≡

```
int d_to_e_n_n(level *level_ptr)
{
    /* Convert D to E in every cell of this level (other than zero) with auxiliary fields. */
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  int number_of_auxiliary_fields = 0;
  DataIterator data_iterator = level_ptr→data_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
  if (watch_d_to_e) pout() << "\t\td_to_e_n_n:" << endl;
  if (¬level_number) {
    pout() << "\t\t\tERROR(d_to_e_n_n): invoked on level zero" << endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  if (¬number_of_auxiliary_fields) {
    pout() << "\t\t\tERROR(d_to_e_n_n): no auxiliary fields" << endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  swap_idx(level_ptr→E_idx);
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    CurlBox &E = level_ptr→E[data_index];
    CurlBox &D = level_ptr→D[data_index];
    CurlBox &S = level_ptr→S[data_index];
    EdgeFab⟨int⟩ &medium_E = level_ptr→medium_E[data_index];
    Real &time_e = level_ptr→time_e;
    Real &dt = level_ptr→dt;
    if (S.nComp() ≠ number_of_auxiliary_fields) {
      pout() << "\t\t\tERROR(d_to_e_n_n): bad number of S components" << endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    for (int dir = 0; dir < SpaceDim; dir++) {
      FArrayBox &E_dir = E[dir];
      FArrayBox &D_dir = D[dir];
      FArrayBox &S_dir = S[dir];
```

```
        BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
        d_to_e_n_n_(&SpaceDim, &dir, &ghost_margin, &number_of_auxiliary_fields, &(level_ptr→E_idx[0]),
            &(level_ptr→D_idx[0]), &time_e, &dt, &(D_dir.loVect()[0]), &(D_dir.loVect()[1]),
            &(D_dir.loVect()[2]), &(D_dir.hiVect()[0]), &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]),
            D_dir.dataPtr(), E_dir.dataPtr(), S_dir.dataPtr(), medium_E_dir.dataPtr(), &watch_d_to_e,
            &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() ≪ "\t\t\tERROR(d_to_e_n_n): failure in d_to_e_n_n_: " ≪ return_status ≪ endl;
          goto exit;
        }
      }    /* for (int dir = 0; dir < SpaceDim; dir++) */
    }    /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  exit:
    if (watch_d_to_e) pout() ≪ "\t\td_to_e_n_n: returning to caller" ≪ endl;
    return return_status;
  }
```

This code is used in chunk 104.

7.3.8 Scheme Driven *D*-to-*E* Conversion

The following two plain C functions are called from Fortran in the **case default** clause of the *select* **case** ($medium\_E(i,j,k)$) switch. The functions prepare things for calling Scheme and then call it. Because all that has to be done here consists of side-evaluations on $\boldsymbol{E}$, $\boldsymbol{E}_{\rm old}$, $\boldsymbol{D}$ and $\boldsymbol{D}_{\rm old}$ we have to feed these into Scheme globals that must be defined beforehand. The functions *do not* check for the presence of the definitions though, because there is simply no time for this. This has to run as fast as possible. It is the responsibility of the programmer, meaning *me*, to ensure that a correct Scheme environment has been created before these functions are invoked.

The first function is *scm_d_to_e_0_*. In this form the function does not expect to be passed any auxiliary fields.

122 ⟨Conversion.c_std 122⟩ ≡

```
#include <REAL.H>
#include <libguile.h>
#include "./Conversion.h"    /* This must be plain C, so we cannot call SCMParmParse. */
  Real scm_d_to_e_0_(const int *const direction, const int *const medium, const Real *const E_old, const
          Real *const D, const Real *const D_old, const Real *const t, const Real *const dt)
  {
    extern SCM E_old_var, D_var, D_old_var, medium_var, direction_var, t_e_var, dt_var;
    extern SCM e_lambda;
    Real E_ret;
    scm_variable_set_x(D_var, scm_from_double(*D));
    scm_variable_set_x(E_old_var, scm_from_double(*E_old));
    scm_variable_set_x(D_old_var, scm_from_double(*D_old));
    scm_variable_set_x(medium_var, scm_from_int(*medium));
    scm_variable_set_x(direction_var, scm_from_int(*direction));
    scm_variable_set_x(t_e_var, scm_from_double(*t));
    scm_variable_set_x(dt_var, scm_from_double(*dt));
    E_ret = scm_to_double(scm_call_0(e_lambda));
    return E_ret;
  }
```

See also chunk 123.

This code is cited in chunk 22.

¶ The second function is much like the first one, but it handles additionally auxiliary fields.

123 ⟨ Conversion.c_std 122 ⟩ +≡

```
Real scm_d_to_e_n_(const int *const direction, const int *const n_aux_fields, const int
        *const medium, const Real *const E_old, const Real *const D, const Real *const D_old, Real
        *const S, const Real *const t, const Real *const dt)
{
  extern SCM E_old_var, D_var, D_old_var, S_var_ref, medium_var, direction_var, t_e_var, dt_var;
  extern SCM e_lambda;
  Real E_ret;
  Real *S_ptr;
  int count;
  scm_variable_set_x(D_var, scm_from_double(*D));
  scm_variable_set_x(E_old_var, scm_from_double(*E_old));
  scm_variable_set_x(D_old_var, scm_from_double(*D_old));
  scm_variable_set_x(medium_var, scm_from_int(*medium));
  scm_variable_set_x(direction_var, scm_from_int(*direction));
  scm_variable_set_x(t_e_var, scm_from_double(*t));
  scm_variable_set_x(dt_var, scm_from_double(*dt));
  S_ptr = S;
  for (count = 0; count < *n_aux_fields; count++)
    scm_uniform_vector_set_x(S_var_ref, scm_from_int(count), scm_from_double((double) *S_ptr++));
  E_ret = scm_to_double(scm_call_0(e_lambda));
  S_ptr = S;
  for (count = 0; count < *n_aux_fields; count++)
    *S_ptr++ = (Real) scm_to_double(scm_uniform_vector_ref(S_var_ref, scm_from_int(count)));
  return E_ret;
}
```

¶ And this is a header file for "Conversion.c", just declaring both functions and EXIT_SUCCESS.

124 ⟨ Conversion.h_std 124 ⟩ ≡

```
#ifdef EXIT_SUCCESS
#undef EXIT_SUCCESS
#endif
# define EXIT_SUCCESS 0
  Real scm_d_to_e_0_(const int *const direction, const int *const medium, const Real *const E_old, const
      Real *const D, const Real *const D_old, const Real *const t, const Real *const dt);
  Real scm_d_to_e_n_(const int *const direction, const int *const n_aux_fields, const int
      *const medium, const Real *const E_old, const Real *const D, const Real *const D_old, Real
      *const S, const Real *const t, const Real *const dt);
```

7.4 Advance the B field

125 ⟨Function *advance_b* 125⟩ ≡

```
int advance_b(level *level_ptr)
{
  int watch_advance_b = false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_b", watch_advance_b);
  if (watch_advance_b) pout() << "advance_b:␣level␣#" << level_number << endl;
  if (level_number == 0)    /* level 0: special advance is needed here */
  {
    if (watch_advance_b) pout() << "\tCalling␣advance_b_0" << endl;
    if (return_status = advance_b_0(level_ptr)) {
      pout() << "\tERROR(advance_b):␣Failed␣to␣advance␣level␣0" << endl;
      goto exit;
    }
  }
  else    /* not level 0: simple advance */
  {
    if (watch_advance_b) pout() << "\tCalling␣advance_b_n" << endl;
    if (return_status = advance_b_n(level_ptr)) {
      pout() << "\tERROR(advance_b):␣Failed␣to␣advance␣level␣" << level_number << endl;
      goto exit;
    }
  }
  level_ptr→time_h += level_ptr→dt;
exit: return return_status;
}
```

This code is used in chunk 104.

7.4.1 Advance Level Zero (with UPML)

126 ⟨Function *advance_b_0* 126⟩ ≡

```
int advance_b_0(level *level_ptr)
{
  int watch_advance_b = false;
  int watch_update_b = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator &data_iterator = level_ptr→data_iterator;
  int Cx_lo = -ghost_margin;
  int Cx_hi = (int) level_ptr→C1x.size() - ghost_margin;
  int Cy_lo = -ghost_margin;
  int Cy_hi = (int) level_ptr→C1y.size() - ghost_margin;
  int Cz_lo = -ghost_margin;
  int Cz_hi = (int) level_ptr→C1z.size() - ghost_margin;
  Interval current_component;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_b", watch_advance_b);
  watch_parameter_parser.query("update_b", watch_update_b);
  if (watch_advance_b) pout() << "\tadvance_b_0:" << endl;
```

```
    if (level_number ≠ 0) {
      pout() << "\t\tERROR(advance_b_0): This is not level zero" << endl;
      return_status = ERR_ADVANCE_B;
      goto exit;
    }
    current_component.define(level_ptr→E_idx[0], level_ptr→E_idx[0]);
    level_ptr→E.exchange(current_component);
    swap_idx(level_ptr→B_idx);
    if (watch_advance_b > 1) pout() << "\t\tentering the box loop" << endl << flush;
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      FArrayBox &Bx = level_ptr→B[data_index][0];
      FArrayBox &By = level_ptr→B[data_index][1];
      FArrayBox &Bz = level_ptr→B[data_index][2];
      FArrayBox &Ex = level_ptr→E[data_index][0];
      FArrayBox &Ey = level_ptr→E[data_index][1];
      FArrayBox &Ez = level_ptr→E[data_index][2];
      Vector<Real> &C1x = level_ptr→C1x;
      Vector<Real> &C2x = level_ptr→C2x;
      Vector<Real> &C1y = level_ptr→C1y;
      Vector<Real> &C2y = level_ptr→C2y;
      Vector<Real> &C1z = level_ptr→C1z;
      Vector<Real> &C2z = level_ptr→C2z;
      if (watch_advance_b > 1)
        pout() << "\t\t\tcalling update_b_upml_ on box " << Bx.box() << " ... " << flush;
      update_b_upml_(
      &SpaceDim, &ghost_margin,
      &(level_ptr→B_idx[0]), &(level_ptr→E_idx[0]),
      &Cx_lo, &Cx_hi, &C1x[0], &C2x[0],
      &Cy_lo, &Cy_hi, &C1y[0], &C2y[0],
      &Cz_lo, &Cz_hi, &C1z[0], &C2z[0],
      &(Bx.loVect()[0]), &(Bx.loVect()[1]), &(Bx.loVect()[2]), &(Bx.hiVect()[0]), &(Bx.hiVect()[1]),
          &(Bx.hiVect()[2]), Bx.dataPtr(),
      &(By.loVect()[0]), &(By.loVect()[1]), &(By.loVect()[2]), &(By.hiVect()[0]), &(By.hiVect()[1]),
          &(By.hiVect()[2]), By.dataPtr(),
      &(Bz.loVect()[0]), &(Bz.loVect()[1]), &(Bz.loVect()[2]), &(Bz.hiVect()[0]), &(Bz.hiVect()[1]),
          &(Bz.hiVect()[2]), Bz.dataPtr(),
      &(Ex.loVect()[0]), &(Ex.loVect()[1]), &(Ex.loVect()[2]), &(Ex.hiVect()[0]), &(Ex.hiVect()[1]),
          &(Ex.hiVect()[2]), Ex.dataPtr(),
      &(Ey.loVect()[0]), &(Ey.loVect()[1]), &(Ey.loVect()[2]), &(Ey.hiVect()[0]), &(Ey.hiVect()[1]),
          &(Ey.hiVect()[2]), Ey.dataPtr(),
      &(Ez.loVect()[0]), &(Ez.loVect()[1]), &(Ez.loVect()[2]), &(Ez.hiVect()[0]), &(Ez.hiVect()[1]),
          &(Ez.hiVect()[2]), Ez.dataPtr(),
      &watch_update_b, &return_status
      );
      if (watch_advance_b > 1) pout() << "done." << endl << flush;
      if (return_status ≠ EXIT_SUCCESS) {
        pout() << "\t\tERROR(advance_b_0): update_b_upml_ returned " << return_status << endl;
        goto exit;
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 104.

7.4.2 Advance Higher Levels

127 ⟨Function *advance_b_n* 127⟩ ≡

```
int advance_b_n(level *level_ptr)
{
  int watch_advance_b = false;
  int watch_update_b = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  Real dt_by_dg = (level_ptr→dt)/(level_ptr→delta);
  DataIterator &data_iterator = level_ptr→data_iterator;
  Interval current_component;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_b", watch_advance_b);
  watch_parameter_parser.query("update_b", watch_update_b);
  if (watch_advance_b) pout() << "\tadvance_b_n:" << endl;
  if (level_number == 0) {
    pout() << "\t\tERROR(advance_b_n): invoked on level zero" << endl;
    return_status = ERR_ADVANCE_B;
    goto exit;
  }
  current_component.define(level_ptr→E_idx[0], level_ptr→E_idx[0]);
  level_ptr→E.exchange(current_component);
  swap_idx(level_ptr→B_idx);
      /* Loop over boxes */
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    FArrayBox &Bx = level_ptr→B[data_index][0];
    FArrayBox &By = level_ptr→B[data_index][1];
    FArrayBox &Bz = level_ptr→B[data_index][2];
    FArrayBox &Ex = level_ptr→E[data_index][0];
    FArrayBox &Ey = level_ptr→E[data_index][1];
    FArrayBox &Ez = level_ptr→E[data_index][2];
    update_b_(
    &SpaceDim, &ghost_margin,
    &(level_ptr→B_idx[0]), &(level_ptr→E_idx[0]),
    &dt_by_dg,
    &(Bx.loVect()[0]), &(Bx.loVect()[1]), &(Bx.loVect()[2]), &(Bx.hiVect()[0]), &(Bx.hiVect()[1]),
        &(Bx.hiVect()[2]), Bx.dataPtr(),
    &(By.loVect()[0]), &(By.loVect()[1]), &(By.loVect()[2]), &(By.hiVect()[0]), &(By.hiVect()[1]),
        &(By.hiVect()[2]), By.dataPtr(),
    &(Bz.loVect()[0]), &(Bz.loVect()[1]), &(Bz.loVect()[2]), &(Bz.hiVect()[0]), &(Bz.hiVect()[1]),
        &(Bz.hiVect()[2]), Bz.dataPtr(),
    &(Ex.loVect()[0]), &(Ex.loVect()[1]), &(Ex.loVect()[2]), &(Ex.hiVect()[0]), &(Ex.hiVect()[1]),
        &(Ex.hiVect()[2]), Ex.dataPtr(),
    &(Ey.loVect()[0]), &(Ey.loVect()[1]), &(Ey.loVect()[2]), &(Ey.hiVect()[0]), &(Ey.hiVect()[1]),
        &(Ey.hiVect()[2]), Ey.dataPtr(),
    &(Ez.loVect()[0]), &(Ez.loVect()[1]), &(Ez.loVect()[2]), &(Ez.hiVect()[0]), &(Ez.hiVect()[1]),
        &(Ez.hiVect()[2]), Ez.dataPtr(),
    &watch_update_b, &return_status
    );
    if (return_status ≠ EXIT_SUCCESS) {
      pout() << "\t\tERROR(advance_b_n): update_b_ returned " << return_status << endl;
      goto exit;
```

```
        }
      }
    exit: return return_status;
    }
```

This code is used in chunk 104.

7.5 Inject the $\boldsymbol{B}$ field

128 ⟨Function *inject_b* 128⟩ ≡

```
int inject_b(level *level_ptr)
{
    /* Inject the B field on the total/scattered field boundary. */
  int return_status = EXIT_SUCCESS;
  int watch_inject_b = (int) false;
  int garbage_collect = (int) false;
  SideIterator side_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse signal_parameter_parser("forms.signal");
  watch_parameter_parser.query("inject_b", watch_inject_b);
  signal_parameter_parser.query("garbage_collect", garbage_collect);
  if (watch_inject_b) pout() << "inject_b:" << endl;
  if (level_ptr->number ≠ 0) {
    pout() << "\tERROR(inject_b): not level zero" << endl;
    return_status = ERR_INJECT_B;
    goto exit;
  }
  for (int dir = 0; dir < SpaceDim; dir++) {
    for (side_iterator.reset(); side_iterator.ok(); ++side_iterator) {
      Side::LoHiSide side = side_iterator();
      if (return_status = inject_b(level_ptr, dir, side)) {
        pout() << "\tERROR(inject_b): failed on dir = " << dir << ", side = " << side << endl;
        goto exit;
      }
    }
  }
  if (garbage_collect) scm_gc();
exit: return return_status;
}
```

See also chunk 129.

This code is used in chunk 104.

129 ¶⟨Function *inject_b* 128⟩ +≡

```
int inject_b(level *level_ptr, int dir, Side::LoHiSide side)
{
  int return_status = EXIT_SUCCESS;
  int watch_inject_b = (int) false;
  int side_number = (int) side;
  int ghost_margin = N_GHOST_CELLS;
  IntVect ijk_lo = level_ptr->signal_ijk_lo;
  IntVect ijk_hi = level_ptr->signal_ijk_hi;
  Vector<Real> &origin = level_ptr->origin;
  Real &delta = level_ptr->delta;
  LevelData<FluxBox> &B = level_ptr->B;
  int &new_component = level_ptr->B_idx[0];
  Vector<DataIndex> &side_boxes = level_ptr->TFR_face_boxes[dir][side];
  Real dt_by_delta = level_ptr->dt/level_ptr->delta;
  Real time_e = level_ptr->time_e;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("inject_b", watch_inject_b);
  if (watch_inject_b) pout() << "\tinject_b: dir = " << dir << ", side = " << side << endl;
```

```
if (dir < 0 ∨ dir ≥ SpaceDim) {
  pout() ≪ "\t\tERROR(inject_b): bad direction" ≪ endl;
  return_status = ERR_INJECT_B;
  goto exit;
}
for (int count = 0; count < side_boxes.size(); count++) {
    /* This loop picks up only the boxes that cross this side and that live on this processor. */
  DataIndex data_index = side_boxes[count];
  switch (dir) {
  case 0:     /* West-East */
    {
      FArrayBox &By = B[data_index][1];
      FArrayBox &Bz = B[data_index][2];
      IntVect By_lo = By.smallEnd();
      IntVect By_hi = By.bigEnd();
      IntVect Bz_lo = Bz.smallEnd();
      IntVect Bz_hi = Bz.bigEnd();
      inject_b_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_e, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(By_lo[0]), &(By_lo[1]), &(By_lo[2]), &(By_hi[0]), &(By_hi[1]),
          &(By_hi[2]), By.dataPtr(), &(Bz_lo[0]), &(Bz_lo[1]), &(Bz_lo[2]), &(Bz_hi[0]), &(Bz_hi[1]),
          &(Bz_hi[2]), Bz.dataPtr(), &watch_inject_b, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() ≪ "\tERROR(inject_d): Fortran problem, dir = " ≪ dir ≪ endl;
        goto exit;
      }
    }
    break;
  case 1:     /* Front-Back */
    {
      FArrayBox &Bx = B[data_index][0];
      FArrayBox &Bz = B[data_index][2];
      IntVect Bx_lo = Bx.smallEnd();
      IntVect Bx_hi = Bx.bigEnd();
      IntVect Bz_lo = Bz.smallEnd();
      IntVect Bz_hi = Bz.bigEnd();
      inject_b_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_e, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(Bx_lo[0]), &(Bx_lo[1]), &(Bx_lo[2]), &(Bx_hi[0]), &(Bx_hi[1]),
          &(Bx_hi[2]), Bx.dataPtr(), &(Bz_lo[0]), &(Bz_lo[1]), &(Bz_lo[2]), &(Bz_hi[0]), &(Bz_hi[1]),
          &(Bz_hi[2]), Bz.dataPtr(), &watch_inject_b, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() ≪ "\tERROR(inject_d): Fortran problem, dir = " ≪ dir ≪ endl;
        goto exit;
      }
    }
    break;
  case 2:     /* Bottom-Top */
    {
      FArrayBox &Bx = B[data_index][0];
      FArrayBox &By = B[data_index][1];
      IntVect Bx_lo = Bx.smallEnd();
      IntVect Bx_hi = Bx.bigEnd();
      IntVect By_lo = By.smallEnd();
```

```
        IntVect By_hi = By.bigEnd();
        inject_b_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
            &(origin[2]), &delta, &dt_by_delta, &time_e, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
            &(ijk_hi[1]), &(ijk_hi[2]), &(Bx_lo[0]), &(Bx_lo[1]), &(Bx_lo[2]), &(Bx_hi[0]), &(Bx_hi[1]),
            &(Bx_hi[2]), Bx.dataPtr(), &(By_lo[0]), &(By_lo[1]), &(By_lo[2]), &(By_hi[0]), &(By_hi[1]),
            &(By_hi[2]), By.dataPtr(), &watch_inject_b, &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() << "\tERROR(inject_d): Fortran problem, dir = " << dir << endl;
          goto exit;
        }
      }
      break;
    }    /* switch (dir) */
  }    /* for (int count = 0; count < side_boxes.size(); count++) */
exit: return return_status;
}
```

7.6 Convert B to H

7.6.1 Function b_to_h

131 ⟨Function *b_to_h* 131⟩ ≡

```
int b_to_h(level *level_ptr)
{
  int watch_b_to_h = (int) false;
  int garbage_collect = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("b_to_h", watch_b_to_h);
  if (watch_b_to_h) {
    pout() << "b_to_h:" << endl;
    pout() << "\tlevel_number␣=␣" << level_number << endl;
  }
  if (level_number ≡ 0) {
    if (return_status = b_to_h_0(level_ptr)) {
      pout() << "\tERROR(b_to_h):␣failure␣in␣b_to_h_0:␣" << return_status << endl;
      goto exit;
    }
  }
  else {
    if (return_status = b_to_h_n(level_ptr)) {
      pout() << "\tERROR(b_to_h):␣failure␣in␣b_to_h_n:␣" << return_status << endl;
      goto exit;
    }
  }
  media_parameter_parser.query("garbage_collect", garbage_collect);
  if (garbage_collect) scm_gc();
exit:
  if (watch_b_to_h) pout() << "b_to_h:␣returning␣to␣caller" << endl;
  return return_status;
}
```

This code is used in chunk 104.

7.6.2 Function b_to_h_0

132 ⟨Function *b_to_h_0* 132⟩ ≡

```
int b_to_h_0(level *level_ptr)
{
    /* Convert B to H in every cell of this level zero. */
  int watch_b_to_h = (int) false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator data_iterator = level_ptr→data_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("b_to_h", watch_b_to_h);
  if (watch_b_to_h) pout() << "\tb_to_h_0:" << endl;
  if (level_number) {
    pout() << "\t\tERROR(b_to_h_0):␣not␣level␣zero" << endl;
```

```
      return_status = ERR_B_TO_H;
      goto exit;
    }
    swap_idx(level_ptr→H_idx);
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      FluxBox &H = level_ptr→H[data_index];
      FluxBox &B = level_ptr→B[data_index];
      FaceFab⟨int⟩ &medium_H = level_ptr→medium_H[data_index];
      Real &time_h = level_ptr→time_h;
      Real &dt = level_ptr→dt;
      Vector⟨Real⟩ &C0x = level_ptr→C0x;
      Vector⟨Real⟩ &C1x = level_ptr→C1x;
      Vector⟨Real⟩ &C3x = level_ptr→C3x;
      Vector⟨Real⟩ &C4x = level_ptr→C4x;
      Vector⟨Real⟩ &C0y = level_ptr→C0y;
      Vector⟨Real⟩ &C1y = level_ptr→C1y;
      Vector⟨Real⟩ &C3y = level_ptr→C3y;
      Vector⟨Real⟩ &C4y = level_ptr→C4y;
      Vector⟨Real⟩ &C0z = level_ptr→C0z;
      Vector⟨Real⟩ &C1z = level_ptr→C1z;
      Vector⟨Real⟩ &C3z = level_ptr→C3z;
      Vector⟨Real⟩ &C4z = level_ptr→C4z;
      int Cx_lo = −ghost_margin;
      int Cx_hi = (int) C1x.size() − ghost_margin;
      int Cy_lo = −ghost_margin;
      int Cy_hi = (int) C1y.size() − ghost_margin;
      int Cz_lo = −ghost_margin;
      int Cz_hi = (int) C1z.size() − ghost_margin;
      for (int dir = 0; dir < SpaceDim; dir++) {
        FArrayBox &H_dir = H[dir];
        FArrayBox &B_dir = B[dir];
        BaseFab⟨int⟩ &medium_H_dir = medium_H[dir];
        b_to_h_0_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→H_idx[0]), &(level_ptr→B_idx[0]), &time_h, &dt,
            &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0], &Cy_lo, &Cy_hi, &C0y[0], &C1y[0],
            &C3y[0], &C4y[0], &Cz_lo, &Cz_hi, &C0z[0], &C1z[0], &C3z[0], &C4z[0], &(B_dir.loVect()[0]),
            &(B_dir.loVect()[1]), &(B_dir.loVect()[2]), &(B_dir.hiVect()[0]), &(B_dir.hiVect()[1]),
            &(B_dir.hiVect()[2]), B_dir.dataPtr(), H_dir.dataPtr(), medium_H_dir.dataPtr(), &watch_b_to_h,
            &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() ≪ "\t\t\tERROR(b_to_h_0): failure in b_to_h_0_: " ≪ return_status ≪ endl;
          goto exit;
        }
      }     /* for (int dir = 0; dir < SpaceDim; dir++) */
    }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  exit:
    if (watch_b_to_h) pout() ≪ "\tb_to_h_0: returning to caller" ≪ endl;
    return return_status;
  }
```

This code is used in chunk 104.

7.6.3 Function b_to_h_n

133 ⟨Function *b_to_h_n* 133⟩ ≡

```
int b_to_h_n(level *level_ptr)
{
    /* Convert B to H in every cell of this level (other than zero). */
  int watch_b_to_h = (int) false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator data_iterator = level_ptr→data_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("b_to_h", watch_b_to_h);
  if (watch_b_to_h) pout() << "\tb_to_h_n:" << endl;
  if (¬level_number) {
    pout() << "\t\tERROR(b_to_h_n): invoked on level zero" << endl;
    return_status = ERR_B_TO_H;
    goto exit;
  }
  swap_idx(level_ptr→H_idx);
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    FluxBox &H = level_ptr→H[data_index];
    FluxBox &B = level_ptr→B[data_index];
    FaceFab⟨int⟩ &medium_H = level_ptr→medium_H[data_index];
    Real &time_h = level_ptr→time_h;
    Real &dt = level_ptr→dt;
    for (int dir = 0; dir < SpaceDim; dir++) {
      FArrayBox &H_dir = H[dir];
      FArrayBox &B_dir = B[dir];
      BaseFab⟨int⟩ &medium_H_dir = medium_H[dir];
      b_to_h_n_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→H_idx[0]), &(level_ptr→B_idx[0]), &time_h, &dt,
          &(B_dir.loVect()[0]), &(B_dir.loVect()[1]), &(B_dir.loVect()[2]), &(B_dir.hiVect()[0]),
          &(B_dir.hiVect()[1]), &(B_dir.hiVect()[2]), B_dir.dataPtr(), H_dir.dataPtr(),
          medium_H_dir.dataPtr(), &watch_b_to_h, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() << "\t\t\tERROR(b_to_h_n): failure in b_to_h_n_: " << return_status << endl;
        goto exit;
      }
    }    /* for (int dir = 0; dir < SpaceDim; dir++) */
  }    /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
exit:
  if (watch_b_to_h) pout() << "\tb_to_h_n: returning to caller" << endl;
  return return_status;
}
```

This code is used in chunk 104.

8 Scheme Functions

134 ⟨ Scheme.c 134 ⟩ ≡

```
#include "Forms.H"
    /* Functions for drawing on Chombo data structures. */
  ⟨ Function draw_box 135 ⟩
  ⟨ Function draw_ball 141 ⟩
  ⟨ Function draw_ellipsoid 147 ⟩
  ⟨ Function draw_cylinder 153 ⟩
  ⟨ Function draw_cone 159 ⟩
  ⟨ Function draw_disk 165 ⟩
  ⟨ Function draw_ary_ellipsoid 171 ⟩
  ⟨ Function draw_parallelepiped 177 ⟩
  ⟨ Function draw_torus 183 ⟩
  ⟨ Function draw_lens 189 ⟩
    /* Auxiliary functions for looking at the Scheme name space. */
  ⟨ Auxiliary Scheme Procedures 196 ⟩
    /* Scheme interfaces to some Chombo/MPI communications. */
  ⟨ MPI Functions 197 ⟩
    /* Interpolate any field from its grid locations on any point in 3D-space. */
  ⟨ Interpolation 198 ⟩
    /* Initialize Scheme variables and intrinsics used by Forms. */
  ⟨ Initialize Scheme 199 ⟩
```

8.1 Draw Box

This is just a wrapper, in this new version, calling an appropriate C++ utility. Its arguments must be two *scm_f64vector*s, and an optional integer, the *color*, with which to draw. If the *color* is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following this one.

135 ⟨ Function *draw_box* 135 ⟩ ≡

```
SCM draw_box(SCM lower_corner, SCM upper_corner, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_lower_corner(SpaceDim, 0.0);
  Vector⟨Real⟩ C_upper_corner(SpaceDim, 0.0);
  bool draw_on_media = false;
  int C_color = 0;
  int watch_draw_box = 0;
  ⟨ draw_box: introduce yourself 136 ⟩
  ⟨ draw_box: read, check, and translate arguments 137 ⟩
  ⟨ draw_box: call appropriate utility 138 ⟩
exit:
  if (watch_draw_box) pout() << "draw_box: returning to caller" << endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_box 139 ⟩
⟨ Function draw_tags_box 140 ⟩
```

This code is cited in chunk 311.

This code is used in chunk 134.

136 ¶⟨ *draw_box*: introduce yourself 136 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_box", watch_draw_box);
if (watch_draw_box) pout() << "draw_box:" << endl;
```

This code is used in chunk 135.

137 ¶⟨ *draw_box*: read, check, and translate arguments 137 ⟩ ≡

```
    /* Are the two first arguments vectors of length SpaceDim, and can they be used for the two corners of
    a box? */
if (scm_is_true(scm_f64vector_p(lower_corner)) ∧ scm_is_true(scm_f64vector_p(upper_corner)) ∧
      (scm_to_int(scm_f64vector_length(lower_corner)) ≡ SpaceDim) ∧
      (scm_to_int(scm_f64vector_length(upper_corner)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_lower_corner[i] = scm_to_double(scm_f64vector_ref(lower_corner, scm_from_int(i)));
    C_upper_corner[i] = scm_to_double(scm_f64vector_ref(upper_corner, scm_from_int(i)));
  }
  if (watch_draw_box) pout() << "\tbox corners: #f64(" << C_lower_corner << ") and #f64(" <<
        C_upper_corner << ")" << endl;
}
else {
  pout() << "\tERROR(draw_box): arguments must be two #f64(a b c) vectors" << endl;
  return_status = false;
  goto exit;
}
```

```
for (int i = 0; i < SpaceDim; i++) {
  if (¬(C_lower_corner[i] < C_upper_corner[i])) {
    pout() << "\tERROR(draw_box): corners ill-defined" << endl;
    return_status = false;
    goto exit;
  }
}
        /* Is the third, optional, argument provided, and can it be used for color? */
if (SCM_UNBNDP(color)) {
  draw_on_media = false;
}
else {
  draw_on_media = true;
  if (scm_is_integer(color)) {
    C_color = scm_to_int(color);
    if (C_color < 0) {
      pout() << "\tERROR(draw_box): negative colors are reserved" << endl;
      return_status = false;
      goto exit;
    }
  }
  else {
    pout() << "\tERROR(draw_box): drawing color must be an integer" << endl;
    return_status = false;
    goto exit;
  }
}
```

This code is used in chunk 135.

138 ¶⟨ *draw_box*: call appropriate utility 138 ⟩ ≡

```
if (draw_on_media) return_status = draw_media_box(C_lower_corner, C_upper_corner, C_color);
else return_status = draw_tags_box(C_lower_corner, C_upper_corner);
```

This code is used in chunk 135.

8.1.1 Drawing Box on Media

139 ⟨ Function *draw_media_box* 139 ⟩ ≡

```
int draw_media_box(Vector<Real> &C_lower_corner, Vector<Real> &C_upper_corner, int &C_color)
{
  extern level *current_level_ptr;
  Vector<Real> &signal_xyz_min = current_level_ptr→signal_xyz_min;
  Vector<Real> &signal_xyz_max = current_level_ptr→signal_xyz_max;
  Vector<Real> &origin = current_level_ptr→origin;
  Real &delta = current_level_ptr→delta;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator &data_iterator = current_level_ptr→data_iterator;
  IntVect bounding_box_lo = IntVect::Zero;
  IntVect bounding_box_hi = IntVect::Zero;
  int buffer_size = REFINE_BUFFER_SIZE;
  int ghost_margin = N_GHOST_CELLS;
  int watch_draw_box = false;
  int return_status = true;     /* this is for Scheme handling */
  Box bounding_box;
  SCMParmParse watch_parameter_parser("forms.watch");
```

```
watch_parameter_parser.query("draw_box", watch_draw_box);
if (watch_draw_box) pout() << "\tdraw_media_box:" << endl;
if ((C_lower_corner.size() ≠ SpaceDim) ∨ (C_upper_corner.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_box): badly dimensioned corner vectors" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((signal_xyz_min[i] + buffer_size * delta > C_lower_corner[i]) ∨ (signal_xyz_max[i] − buffer_size * delta <
        C_upper_corner[i])) {
    pout() << "\tERROR(draw_box): bounding box not in total field region" << endl;
    return_status = false;
    goto exit;
  }
}
      /* Find approximate IntVect representation of the box corners and pad it. */
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lower_corner[i] − origin[i])/delta + 0.5 + EPSILON) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_upper_corner[i] − origin[i])/delta − 0.5 − EPSILON) + buffer_size;
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_ellipsoid): bounding box empty" << endl;
  return_status = false;
  goto exit;
}
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (bounding_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_box_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_lower_corner[0]), &(C_upper_corner[0]), &C_color,
          &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
          medium_E_dir.dataPtr(), &watch_draw_box, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_box_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
      f_draw_box_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_lower_corner[0]), &(C_upper_corner[0]), &C_color,
          &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
          medium_H_dir.dataPtr(), &watch_draw_box, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_box_): Fortran problem, dir = " << dir << endl;
```

```
            goto exit;
          }
        }
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 135.

8.1.2 Drawing Box on Tags

This function can be done very similarly to the one that draws on the media, because *f_draw_box_* is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel anyway.

140 ⟨ Function *draw_tags_box* 140 ⟩ ≡

```
int draw_tags_box(Vector⟨Real⟩ &C_lower_corner, Vector⟨Real⟩ &C_upper_corner)
{
  pout() << "ERROR(draw_box): Drawing on tags not implemented yet." << endl;
  return false;
}
```

This code is used in chunk 135.

8.2 Draw Ball

This is just a wrapper, calling an appropriate C++ utility. Its arguments must be an *scm_f64vector* (the center of the ball), a **double** (its radius), and an optional integer, the *color* with which to draw. If the *color* is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following this one.

141 ⟨ Function *draw_ball* 141 ⟩ ≡

```
SCM draw_ball(SCM center, SCM radius, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_center(SpaceDim, 0.0);
  Real C_radius;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_ball = 0;
  ⟨ draw_ball: introduce yourself 142 ⟩
  ⟨ draw_ball: read, check, and translate arguments 143 ⟩
  ⟨ draw_ball: call appropriate utility 144 ⟩
exit:
  if (watch_draw_ball) pout() ≪ "draw_ball:␣returning␣to␣caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_ball 145 ⟩
⟨ Function draw_tags_ball 146 ⟩
```

This code is used in chunk 134.

142 ¶⟨ *draw_ball*: introduce yourself 142 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ball", watch_draw_ball);
if (watch_draw_ball) pout() ≪ "draw_ball:" ≪ endl;
```

This code is used in chunk 141.

143 ¶⟨ *draw_ball*: read, check, and translate arguments 143 ⟩ ≡

```
    /* Is the first argument vector of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(center)) ∧ (scm_to_int(scm_f64vector_length(center)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++)
    C_center[i] = scm_to_double(scm_f64vector_ref(center, scm_from_int(i)));
  if (watch_draw_ball) pout() ≪ "\tball␣center:␣#f64(" ≪ C_center ≪ ")" ≪ endl;
}
else {
  pout() ≪ "\tERROR(draw_ball):␣" ≪ "first␣argument␣must␣be␣an␣#f64(a␣b␣c)␣v\
      ector␣(center)" ≪ endl;
  return_status = false;
  goto exit;
}
if (scm_is_real(radius)) {
  C_radius = scm_to_double(radius);
  if (watch_draw_ball) pout() ≪ "\tradius:␣␣␣␣␣␣" ≪ C_radius ≪ endl;
}
else {
```

```
    pout() << "\tERROR(draw_ball): " << "second argument must be a real (center)" << endl;
    return_status = false;
    goto exit;
  }
        /* Is the third, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_ball): negative colors are reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_ball): drawing color must be an integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 141.

144 ¶⟨ *draw_ball*: call appropriate utility 144 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_ball(C_center, C_radius, C_color);
  else return_status = draw_tags_ball(C_center, C_radius);
```

This code is used in chunk 141.

8.2.1 Drawing Ball on Media

145 ⟨ Function *draw_media_ball* 145 ⟩ ≡

```
  int draw_media_ball(Vector<Real> &C_center, Real &C_radius, int &C_color)
  {
    extern level *current_level_ptr;
    Vector<Real> &signal_xyz_min = current_level_ptr->signal_xyz_min;
    Vector<Real> &signal_xyz_max = current_level_ptr->signal_xyz_max;
    Vector<Real> &origin = current_level_ptr->origin;
    Real &delta = current_level_ptr->delta;
    DisjointBoxLayout &box_layout = current_level_ptr->box_layout;
    DataIterator &data_iterator = current_level_ptr->data_iterator;
    IntVect bounding_box_lo = IntVect::Zero;
    IntVect bounding_box_hi = IntVect::Zero;
    int buffer_size = REFINE_BUFFER_SIZE;
    int ghost_margin = N_GHOST_CELLS;
    int watch_draw_ball = false;
    int return_status = true;     /* this is for Scheme handling */
    Box bounding_box;
    SCMParmParse watch_parameter_parser("forms.watch");
```

```
watch_parameter_parser.query("draw_ball", watch_draw_ball);
if (watch_draw_ball) pout() << "\tdraw_media_ball:" << endl;
if (C_center.size() ≠ SpaceDim) {
  pout() << "\t\tERROR(draw_media_ball): badly sized center vector" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((signal_xyz_min[i] + buffer_size * delta > C_center[i] - C_radius) ∨ (signal_xyz_max[i] - buffer_size * delta <
        C_center[i] + C_radius)) {
    pout() << "\tERROR(draw_ball): ball too close to signal injection boundary" << endl;
    return_status = false;
    goto exit;
  }
}
      /* Find approximate IntVect representation of the ball box corners and pad it. */
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_center[i] - C_radius - origin[i])/delta + 0.5 + EPSILON) - buffer_size;
  bounding_box_hi[i] = (int) ceil((C_center[i] + C_radius - origin[i])/delta - 0.5 - EPSILON) + buffer_size;
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (bounding_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_ball_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_center[0]), &C_radius, &C_color, &(E_box_dir.type()[0]), &(E_lo[0]),
          &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]), medium_E_dir.dataPtr(),
          &watch_draw_ball, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_ball_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
      f_draw_ball_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_center[0]), &C_radius, &C_color, &(H_box_dir.type()[0]), &(H_lo[0]),
          &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]), medium_H_dir.dataPtr(),
          &watch_draw_ball, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_ball_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
    }
  }
}
```

```
exit: return return_status;
}
```

This code is used in chunk 141.

8.2.2 Drawing Ball on Tags

This function can be done very similarly to the one that draws a ball on the media, because *f_draw_ball_* is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel anyway.

146 ⟨Function *draw_tags_ball* 146⟩ ≡

```
int draw_tags_ball(Vector⟨Real⟩ &C_center, Real &C_radius)
{
  pout() << "ERROR(draw_ball): Drawing on tags not implemented yet." << endl;
  return false;
}
```

This code is used in chunk 141.

8.3 Draw Ellipsoid

This is just a wrapper, calling an appropriate C++ utility. Its arguments must be two *scm_f64vector*s (the focal points of the ellipsoid), a **double** (the sum of distances from a surface point to the focal points), and an optional integer, the *color* with which to draw. If the *color* is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following this one.

147 ⟨ Function *draw_ellipsoid* 147 ⟩ ≡

```
SCM draw_ellipsoid(SCM focus_1, SCM focus_2, SCM sum, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_focus_1(SpaceDim, 0.0), C_focus_2(SpaceDim, 0.0);
  Real C_sum;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_ellipsoid = 0;
  ⟨ draw_ellipsoid: introduce yourself 148 ⟩
  ⟨ draw_ellipsoid: read, check, and translate arguments 149 ⟩
  ⟨ draw_ellipsoid: call appropriate utility 150 ⟩
exit:
  if (watch_draw_ellipsoid) pout() << "draw_ellipsoid: returning to caller" << endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_ellipsoid 151 ⟩
⟨ Function draw_tags_ellipsoid 152 ⟩
```

This code is used in chunk 134.

148 ¶⟨ *draw_ellipsoid*: introduce yourself 148 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ellipsoid", watch_draw_ellipsoid);
if (watch_draw_ellipsoid) pout() << "draw_ellipsoid:" << endl;
```

This code is used in chunk 147.

149 ¶⟨ *draw_ellipsoid*: read, check, and translate arguments 149 ⟩ ≡

```
    /* Is the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(focus_1)) ∧ (scm_to_int(scm_f64vector_length(focus_1)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(focus_2)) ∧ (scm_to_int(scm_f64vector_length(focus_2)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_focus_1[i] = scm_to_double(scm_f64vector_ref(focus_1, scm_from_int(i)));
    C_focus_2[i] = scm_to_double(scm_f64vector_ref(focus_2, scm_from_int(i)));
  }
  if (watch_draw_ellipsoid) {
    pout() << "\tellipsoid focus_1: #f64(" << C_focus_1 << ")" << endl;
    pout() << "\t          focus_2: #f64(" << C_focus_2 << ")" << endl;
  }
}
else {
  pout() << "\tERROR(draw_ellipsoid): " << "first two arguments must be #f64(a b c) vectors" <<
      endl;
  return_status = false;
  goto exit;
```

```
}
if (scm_is_real(sum)) {
  C_sum = scm_to_double(sum);
  if (watch_draw_ellipsoid) pout() << "\t␣␣␣␣␣␣␣␣␣␣␣␣␣sum:␣" << C_sum << endl;
}
else {
  pout() << "\tERROR(draw_ellipsoid):␣" << "third␣argument␣must␣be␣a␣real␣(center)" << endl;
  return_status = false;
  goto exit;
}
        /* Is the fourth, optional, argument provided, and can it be used for color? */
if (SCM_UNBNDP(color)) {
  draw_on_media = false;
}
else {
  draw_on_media = true;
  if (scm_is_integer(color)) {
    C_color = scm_to_int(color);
    if (C_color < 0) {
      pout() << "\tERROR(draw_ellipsoid):␣negative␣colors␣are␣reserved" << endl;
      return_status = false;
      goto exit;
    }
  }
  else {
    pout() << "\tERROR(draw_ellipsoid):␣drawing␣color␣must␣be␣an␣integer" << endl;
    return_status = false;
    goto exit;
  }
}
```

This code is used in chunk 147.

150 ¶⟨ *draw_ellipsoid*: call appropriate utility 150 ⟩ ≡

```
if (draw_on_media) return_status = draw_media_ellipsoid(C_focus_1, C_focus_2, C_sum, C_color);
else return_status = draw_tags_ellipsoid(C_focus_1, C_focus_2, C_sum);
```

This code is used in chunk 147.

8.3.1 Drawing Ellipsoid on Media

Here we are going to inscribe the ellipsoid into a cylinder, then find a box that encapsulates the cylinder, following logic that's used in *draw_media_cylinder*, module ⟨ Function *draw_media_cylinder* 157 ⟩.

151 ⟨ Function *draw_media_ellipsoid* 151 ⟩ ≡

```
int draw_media_ellipsoid(Vector<Real> &C_focus_1, Vector<Real> &C_focus_2, Real &C_sum, int
        &C_color)
{
  extern level *current_level_ptr;
  Vector<Real> &signal_xyz_min = current_level_ptr→signal_xyz_min;
  Vector<Real> &signal_xyz_max = current_level_ptr→signal_xyz_max;
  Vector<Real> &origin = current_level_ptr→origin;
  Real &delta = current_level_ptr→delta;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator &data_iterator = current_level_ptr→data_iterator;
  IntVect bounding_box_lo = IntVect::Zero;
```

```
IntVect bounding_box_hi = IntVect::Zero;
IntVect total_field_lo = IntVect::Zero;
IntVect total_field_hi = IntVect::Zero;
int buffer_size = REFINE_BUFFER_SIZE;
int ghost_margin = N_GHOST_CELLS;
int watch_draw_ellipsoid = false;
int return_status = true;      /* this is for Scheme handling */
Vector<Real> C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0), r(SpaceDim, 0.0), n(SpaceDim, 0.0),
    C_focus_1_ext(SpaceDim, 0.0), C_focus_2_ext(SpaceDim, 0.0);
Real r_length, delta_r, radius;
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ellipsoid", watch_draw_ellipsoid);
if (watch_draw_ellipsoid) pout() << "\tdraw_media_ellipsoid:" << endl;
if ((C_focus_1.size() != SpaceDim) || (C_focus_2.size() != SpaceDim)) {
  pout() << "\t\tERROR(draw_media_ellipsoid): badly sized focus vectors" << endl;
  return_status = false;
  goto exit;
}
        /* r = f_2 - f_1 */
for (int i = 0; i < SpaceDim; i++) r[i] = C_focus_2[i] - C_focus_1[i];
        /* r = sqrt(r · r) */
r_length = sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
if (C_sum <= r_length) {
  pout() << "\tERROR(draw_ellipsoid): sum <= distance between foci" << endl;
  return_status = false;
  goto exit;
}
        /* n = r/r */
for (int i = 0; i < SpaceDim; i++) n[i] = r[i]/r_length;
        /* For an ellipse, a perihelion distance, Δr = 1/2 (S − r), where S is the sum of distances from the
        foci for every point of an ellipse—it is always the same. */
delta_r = (C_sum - r_length)/2.0;
      /* The radius of an ellipsoid is given, from Pythagoras theorem, by ρ = 1/2 √(S^2 − r^2). */
radius = sqrt(C_sum*C_sum - r_length*r_length)/2.0;
      /* Here we extend the major axis of the ellipsoid by Δr beyond f_1 and f_2, so that C_focus_1_ext and
      C_focus_2_ext correspond to the ends of the ellipsoid, not to the foci. */
for (int i = 0; i < SpaceDim; i++) {
  C_focus_1_ext[i] = C_focus_1[i] - delta_r*n[i];
  C_focus_2_ext[i] = C_focus_2[i] + delta_r*n[i];
}
        /* And here we find the low and the high corner of a box that encloses the ellipsoid. */
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = min(C_focus_1_ext[i], C_focus_2_ext[i]) - radius;
  C_hi[i] = max(C_focus_1_ext[i], C_focus_2_ext[i]) + radius;
}
        /* Here we convert C_lo and C_hi, to the corresponding grid coordinates and shift them additionally
        by buffer_size, to ensure we do not get too close to the total field region boundary. We find the low
        and the high corners of the total field region, in the grid coordinates, at the same time. */
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] - origin[i])/delta) - buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] - origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] - origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] - origin[i])/delta);
```

```
    }
        /* Now we are ready to define both boxes, the box that encloses the ellipsoid and the total field
        region box, and ensure that the former is enclosed within the latter. */
    bounding_box.define(bounding_box_lo, bounding_box_hi);
    total_field_box.define(total_field_lo, total_field_hi);
    if (bounding_box.isEmpty()) {
      pout() << "\tERROR(draw_ellipsoid): bounding box empty" << endl;
      return_status = false;
      goto exit;
    }
    if (¬total_field_box.contains(bounding_box)) {
      pout() << "\tERROR(draw_ellipsoid): bounding box not in total field region" << endl;
      return_status = false;
      goto exit;
    }
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      Box layout_box = box_layout.get(data_index);
      if (bounding_box.intersects(layout_box)) {
          /* This box may need work. This will be ultimately decided by the Fortran routine. */
        for (int dir = 0; dir < SpaceDim; dir++) {
          BaseFab<int> &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
          Box E_box_dir = medium_E_dir.box();
          IntVect E_lo = E_box_dir.smallEnd();
          IntVect E_hi = E_box_dir.bigEnd();
          f_draw_ellipsoid_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_focus_1[0]), &(C_focus_2[0]), &C_sum, &C_color,
              &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
              medium_E_dir.dataPtr(), &watch_draw_ellipsoid, &return_status);
          if (¬return_status) {
            pout() << "\t\tERROR(f_draw_ellipsoid_): Fortran problem, dir = " << dir << endl;
            goto exit;
          }
          BaseFab<int> &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
          Box H_box_dir = medium_H_dir.box();
          IntVect H_lo = H_box_dir.smallEnd();
          IntVect H_hi = H_box_dir.bigEnd();
          f_draw_ellipsoid_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_focus_1[0]), &(C_focus_2[0]), &C_sum, &C_color,
              &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
              medium_H_dir.dataPtr(), &watch_draw_ellipsoid, &return_status);
          if (¬return_status) {
            pout() << "\t\tERROR(f_draw_ellipsoid_): Fortran problem, dir = " << dir << endl;
            goto exit;
          }
        }
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 147.

8.3.2 Drawing Ellipsoid on Tags

This function can be done very similarly to the one that draws a ellipsoid on the media, because *f_draw_ellipsoid_* is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel anyway.

152 ⟨Function *draw_tags_ellipsoid* 152⟩ ≡

```
int draw_tags_ellipsoid(Vector⟨Real⟩ &C_focus_1, Vector⟨Real⟩ &C_focus_2, Real &C_sum)
{
  pout() << "ERROR(draw_ellipsoid): Drawing on tags not implemented yet." << endl;
  return false;
}
```

This code is used in chunk 147.

8.4 Draw Cylinder

This is just a wrapper, calling an appropriate C++ utility. Its arguments are two *scm_f64vector*s, the beginning and the end points of the cylinder's main axis, and a **double**, the radius of the cylinder. If the *color*, which is optional, is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following the wrapper.

153 ⟨ Function *draw_cylinder* 153 ⟩ ≡

```
SCM draw_cylinder(SCM end_1, SCM end_2, SCM radius, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_end_1(SpaceDim, 0.0), C_end_2(SpaceDim, 0.0);
  Real C_radius;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_cylinder = 0;
  ⟨ draw_cylinder: introduce yourself 154 ⟩
  ⟨ draw_cylinder: read, check, and translate arguments 155 ⟩
  ⟨ draw_cylinder: call appropriate utility 156 ⟩
exit:
  if (watch_draw_cylinder) pout() ≪ "draw_cylinder:␣returning␣to␣caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_cylinder 157 ⟩
⟨ Function draw_tags_cylinder 158 ⟩
```

This code is cited in chunk 157.

This code is used in chunk 134.

154 ¶⟨ *draw_cylinder*: introduce yourself 154 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_cylinder", watch_draw_cylinder);
if (watch_draw_cylinder) pout() ≪ "draw_cylinder:" ≪ endl;
```

This code is used in chunk 153.

¶ Here we inspect the arguments passed first. The first two have to be *scm_f64vector*s of length *SpaceDim*. If they are we transfer them to C++ vectors, *C_end_1* and *C_end_2*, if not, we flag an error and return. Then we deal similarly with the radius, which has to be a **double**, if it is, we transfer its value to a C++ double, called *C_radius*, if not, we flag an error and return.

Finally, we do the same for the fourth, optional, argument, which ought to be an integer, if specified at all. Because it is optional, it may be returned as *unbound,* which we can check for with the `SCM_UNBNDP` predicate. The colour specified must be an integer greater than or equal to zero. Although Forms uses negative media colours, they're reserved for internal operations only, and are not allowed on input.

155 ⟨ *draw_cylinder*: read, check, and translate arguments 155 ⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(end_1)) ∧ (scm_to_int(scm_f64vector_length(end_1)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(end_2)) ∧ (scm_to_int(scm_f64vector_length(end_2)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_end_1[i] = scm_to_double(scm_f64vector_ref(end_1, scm_from_int(i)));
    C_end_2[i] = scm_to_double(scm_f64vector_ref(end_2, scm_from_int(i)));
  }
  if (watch_draw_cylinder) {
```

```
      pout() << "\tcylinder␣end_1:␣#f64(" << C_end_1 << ")" << endl;
      pout() << "\t␣␣␣␣␣␣␣␣␣end_2:␣#f64(" << C_end_2 << ")" << endl;
    }
  }
  else {
    pout() << "\tERROR(draw_cylinder):␣" << "the␣first␣two␣arguments␣must␣be␣#f64(a␣\
        b␣c)␣vectors" << endl;
    return_status = false;
    goto exit;
  }
  if (scm_is_real(radius)) {
    C_radius = scm_to_double(radius);
    if (watch_draw_cylinder) pout() << "\t␣␣␣␣␣␣␣␣␣radius:␣" << C_radius << endl;
  }
  else {
    pout() << "\tERROR(draw_cylinder):␣" << "the␣third␣argument␣must␣be␣a␣real␣(radius)" << endl;
    return_status = false;
    goto exit;
  }
        /* Is the fourth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_cylinder):␣negative␣colors␣are␣reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_cylinder):␣drawing␣color␣must␣be␣an␣integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 153.

156 ¶⟨ *draw_cylinder*: call appropriate utility 156 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_cylinder(C_end_1, C_end_2, C_radius, C_color);
  else return_status = draw_tags_cylinder(C_end_1, C_end_2, C_radius);
```

This code is used in chunk 153.

8.4.1 Drawing Cylinder on Media

This is a longer routine, but it still doesn't draw anything. Instead it does still more checking and wrapping, before it calls *f_draw_cylinder_*, which is a Fortran subroutine that does the actual drawing.

The first check is of the size of *C_end_1* and *C_end_2*. It is superfluous but harmless, because we ensure this already when creating both vectors in module ⟨ Function *draw_cylinder* 153 ⟩. We put it here in expectation that the function may be called from another environment in the future.

Next step is to create a **Box** that is going to encapsulate the cylinder. Within this routine we will use the box to

1. ensure that the cylinder is fully enclosed within the total field region;
2. check if at least a part of the cylinder is enclosed in a *layout_box*, for which we are about to invoke *f_draw_cylinder_*—of course, we won't do so, if the whole cylinder is outside.

The semantics of **Box** . *define* require that we identify low and high corners of the box, which then have to be translated to **IntVect**s and passed as arguments. The cylinder may be oriented any way within the box and the user may define it by specifying its begin and end points in any order, too.

The trick we resort to here is

1. select the three lowest and the three highest values of coordinates from *end_1* and *end_2*—these are our initial values for the box corners, *C_lo* and *C_hi*;
2. subtract the radius from coordinates of the low corner and add the radius to coordinates of the high corner—the box made of the two should fully enclose the cylinder;
3. convert the corners' physical coordinates to grid cell numbers, i.e., *IntVects*, *bounding_box_lo* and *bounding_box_hi*—add extra padding, *buffer_size*, for safety;
4. having constructed corner **IntVect**s we make the *bounding_box*.

While we're at it, we also construct corner **IntVect**s for the total field region, *total_field_lo* and *total_field_hi*, and construct the total field region box, *total_field_box*. Now, we can check if the latter contains the former. We flag an error and exit, if it does not.

At this stage we are ready to draw. This is a complex process, because we have to draw in each box of the layout separately. And so we loop over the layout. We check if the *bounding_box* intersects the *layout_box* returned by the iterator, and only when it does, we call *f_draw_cylinder_*.

We draw for each grid direction separately and we draw twice! We draw first on the $\boldsymbol{E}$ sites, then we draw on the $\boldsymbol{B}$ sites. One of the parameters passed to the *f_draw_cylinder_* is the grid type. It is face centered for $\boldsymbol{B}$ and edge centered for $\boldsymbol{E}$. *f_draw_cylinder_* knows how to make use of this information. It will scan through all sites of a given type within the *bounding_box*, checking for each if it resides within the cylinder itself including its boundary. For every site that does, it'll mark it on *medium_E* and *medium_H* with the *C_color* requested.

157 ⟨ Function *draw_media_cylinder* 157 ⟩ ≡

```
int draw_media_cylinder(Vector⟨Real⟩ &C_end_1, Vector⟨Real⟩ &C_end_2, Real &C_radius, int &C_color)
{
  extern level *current_level_ptr;
  Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
  Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
  Vector⟨Real⟩ &origin = current_level_ptr→origin;
  Real &delta = current_level_ptr→delta;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator &data_iterator = current_level_ptr→data_iterator;
  IntVect bounding_box_lo = IntVect::Zero;
  IntVect bounding_box_hi = IntVect::Zero;
  IntVect total_field_lo = IntVect::Zero;
  IntVect total_field_hi = IntVect::Zero;
  int buffer_size = REFINE_BUFFER_SIZE;
  int ghost_margin = N_GHOST_CELLS;
  int watch_draw_cylinder = false;
  int return_status = true;    /* this is for Scheme handling */
  Vector⟨Real⟩ C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
  Box bounding_box, total_field_box;
  SCMParmParse watch_parameter_parser("forms.watch");
```

```
watch_parameter_parser.query("draw_cylinder", watch_draw_cylinder);
if (watch_draw_cylinder) pout() << "\tdraw_media_cylinder:" << endl;
if ((C_end_1.size() ≠ SpaceDim) ∨ (C_end_2.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_cylinder): badly sized end vectors" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = min(C_end_1[i], C_end_2[i]) − C_radius;
  C_hi[i] = max(C_end_1[i], C_end_2[i]) + C_radius;
}
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_cylinder): bounding box empty" << endl;
  return_status = false;
  goto exit;
}
if (¬total_field_box.contains(bounding_box)) {
  pout() << "\tERROR(draw_cylinder): bounding box not in total field region" << endl;
  return_status = false;
  goto exit;
}
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (bounding_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_cylinder_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_end_1[0]), &(C_end_2[0]), &C_radius, &C_color,
          &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
          medium_E_dir.dataPtr(), &watch_draw_cylinder, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_cylinder_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
      f_draw_cylinder_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_end_1[0]), &(C_end_2[0]), &C_radius, &C_color,
          &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
```

```
            medium_H_dir.dataPtr(), &watch_draw_cylinder, &return_status);
        if (¬return_status) {
          pout() << "\t\tERROR(f_draw_cylinder_): Fortran problem, dir = " << dir << endl;
          goto exit;
        }
      }
    }
  }
exit: return return_status;
}
```

This code is cited in chunk 151.

This code is used in chunk 153.

8.4.2 Drawing Cylinder on Tags

This function can be done similarly to the one that draws a cylinder on the media, because *f_draw_cylinder_* (this is a Fortran code that does things quickly) is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel.

158 ⟨Function *draw_tags_cylinder* 158⟩ ≡

```
int draw_tags_cylinder(Vector⟨Real⟩ &C_cylinder_1, Vector⟨Real⟩ &C_cylinder_2, Real &C_radius)
{
  pout() << "ERROR(draw_cylinder): Drawing on tags not implemented yet." << endl;
  return false;
}
```

This code is used in chunk 153.

8.5 Draw Cone

This is just a wrapper, calling an appropriate C++ utility. Its arguments are two *scm_f64vector*s, the beginning and the end points of the cone's main axis, and a **double**, the radius of the cone at its base. If the *color*, which is optional, is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following the wrapper.

The function is almost identical to *draw_cylinder*, because a cone is fully inscribed within a cylinder. The only difference is at the Fortran level, where a logical function *in_cone* is used to do the checking instead of *in_cylinder*.

The semantics of the function assume that *end_1* corresponds to the base of the cone and *end_2* to its apex.

159 ⟨ Function *draw_cone* 159 ⟩ ≡

```
SCM draw_cone(SCM end_1, SCM end_2, SCM radius, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_end_1(SpaceDim, 0.0), C_end_2(SpaceDim, 0.0);
  Real C_radius;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_cone = 0;
  ⟨ draw_cone: introduce yourself 160 ⟩
  ⟨ draw_cone: read, check, and translate arguments 161 ⟩
  ⟨ draw_cone: call appropriate utility 162 ⟩
exit:
  if (watch_draw_cone) pout() ≪ "draw_cone: returning to caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_cone 163 ⟩
⟨ Function draw_tags_cone 164 ⟩
```

This code is cited in chunk 163.

This code is used in chunk 134.

160 ¶⟨ *draw_cone*: introduce yourself 160 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_cone", watch_draw_cone);
if (watch_draw_cone) pout() ≪ "draw_cone:" ≪ endl;
```

This code is used in chunk 159.

¶ Here we inspect the arguments passed first. The first two have to be *scm_f64vector*s of length *SpaceDim*. If they are we transfer them to C++ vectors, *C_end_1* and *C_end_2*, if not, we flag an error and return. Then we deal similarly with the radius, which has to be a **double**, if it is, we transfer its value to a C++ double, called *C_radius*, if not, we flag an error and return.

Finally, we do the same for the fourth, optional, argument, which ought to be an integer, if specified at all. Because it is optional, it may be returned as *unbound,* which we can check for with the `SCM_UNBNDP` predicate. The colour specified must be an integer greater than or equal to zero. Although FORMS uses negative media colours, they're reserved for internal operations only, and are not allowed on input.

161 ⟨ *draw_cone*: read, check, and translate arguments 161 ⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(end_1)) ∧ (scm_to_int(scm_f64vector_length(end_1)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(end_2)) ∧ (scm_to_int(scm_f64vector_length(end_2)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
```

```
      C_end_1[i] = scm_to_double(scm_f64vector_ref(end_1, scm_from_int(i)));
      C_end_2[i] = scm_to_double(scm_f64vector_ref(end_2, scm_from_int(i)));
    }
    if (watch_draw_cone) {
      pout() << "\tcone␣end_1:␣#f64(" << C_end_1 << ")" << endl;
      pout() << "\t␣␣␣␣␣end_2:␣#f64(" << C_end_2 << ")" << endl;
    }
  }
  else {
    pout() << "\tERROR(draw_cone):␣" << "the␣first␣two␣arguments␣must␣be␣#f64(a␣b␣c)␣vectors" <<
        endl;
    return_status = false;
    goto exit;
  }
  if (scm_is_real(radius)) {
    C_radius = scm_to_double(radius);
    if (watch_draw_cone) pout() << "\t␣␣␣␣␣radius:␣" << C_radius << endl;
  }
  else {
    pout() << "\tERROR(draw_cone):␣" << "the␣third␣argument␣must␣be␣a␣real␣(radius)" << endl;
    return_status = false;
    goto exit;
  }
        /* Is the fourth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_cone):␣negative␣colors␣are␣reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_cone):␣drawing␣color␣must␣be␣an␣integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 159.

162 ¶⟨ *draw_cone*: call appropriate utility 162 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_cone(C_end_1, C_end_2, C_radius, C_color);
  else return_status = draw_tags_cone(C_end_1, C_end_2, C_radius);
```

This code is used in chunk 159.

8.5.1 Drawing Cone on Media

This is a longer routine, but it still doesn't draw anything. Instead it does still more checking and wrapping, before it calls *f_draw_cone_*, which is a Fortran subroutine that does the actual drawing.

The first check is of the size of *C_end_1* and *C_end_2*. It is superfluous but harmless, because we ensure this already when creating both vectors in module ⟨ Function *draw_cone* 159 ⟩. We put it here in expectation that the function may be called from another environment in the future.

Next step is to create a **Box** that is going to encapsulate the cone. Within this routine we will use the box to

1. ensure that the cone is fully enclosed within the total field region;
2. check if at least a part of the cone is enclosed in a *layout_box*, for which we are about to invoke *f_draw_cone_*—of course, we won't do so, if the whole cone is outside.

Because a cone is fully inscribed in a cylinder, the construction of the box proceeds exactly as we would have done with the cylinder.

The semantics of **Box** . *define* require that we identify low and high corners of the box, which then have to be translated to **IntVect**s and passed as arguments. The cone may be oriented any way within the box and the user may define it by specifying its begin and end points in any order, too.

The trick we resort to here is

1. select the three lowest and the three highest values of coordinates from *end_1* and *end_2*—these are our initial values for the box corners, *C_lo* and *C_hi*;
2. subtract the radius from coordinates of the low corner and add the radius to coordinates of the high corner—the box made of the two should fully enclose the cone;
3. convert the corners' physical coordinates to grid cell numbers, i.e., *IntVects*, *cone_box_lo* and *cone_box_hi*—add extra padding, *buffer_size*, for safety;
4. having constructed corner **IntVect**s we make the *cone_box*.

While we're at it, we also construct corner **IntVect**s for the total field region, *total_field_lo* and *total_field_hi*, and construct the total field region box, *total_field_box*. Now, we can check if the latter contains the former. We flag an error and exit, if it does not.

At this stage we are ready to draw. This is a complex process, because we have to draw in each box of the layout separately. And so we loop over the layout. We check if the *cone_box* intersects the *layout_box* returned by the iterator, and only when it does, we call *f_draw_cone_*.

We draw for each grid direction separately and we draw twice! We draw first on the ***E*** sites, then we draw on the ***B*** sites. One of the parameters passed to the *f_draw_cone_* is the grid type. It is face centered for ***B*** and edge centered for ***E***. *f_draw_cone_* knows how to make use of this information. It will scan through all sites of a given type within the *cone_box*, checking for each if it resides within the cone itself including its boundary. For every site that does, it'll mark it on *medium_E* and *medium_H* with the *C_color* requested.

163 ⟨ Function *draw_media_cone* 163 ⟩ ≡

```
int draw_media_cone(Vector⟨Real⟩ &C_end_1, Vector⟨Real⟩ &C_end_2, Real &C_radius, int &C_color)
{
  extern level *current_level_ptr;
  Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
  Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
  Vector⟨Real⟩ &origin = current_level_ptr→origin;
  Real &delta = current_level_ptr→delta;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator &data_iterator = current_level_ptr→data_iterator;
  IntVect bounding_box_lo = IntVect::Zero;
  IntVect bounding_box_hi = IntVect::Zero;
  IntVect total_field_lo = IntVect::Zero;
  IntVect total_field_hi = IntVect::Zero;
  int buffer_size = REFINE_BUFFER_SIZE;
  int ghost_margin = N_GHOST_CELLS;
  int watch_draw_cone = false;
  int return_status = true;    /* this is for Scheme handling */
```

```
Vector⟨Real⟩ C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_cone", watch_draw_cone);
if (watch_draw_cone) pout() << "\tdraw_media_cone:" << endl;
if ((C_end_1.size() ≠ SpaceDim) ∨ (C_end_2.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_cone): badly sized end vectors" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = min(C_end_1[i], C_end_2[i]) − C_radius;
  C_hi[i] = max(C_end_1[i], C_end_2[i]) + C_radius;
}
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_cone): bounding box empty" << endl;
  return_status = false;
  goto exit;
}
if (¬total_field_box.contains(bounding_box)) {
  pout() << "\tERROR(draw_cone): bounding box not in total field region" << endl;
  return_status = false;
  goto exit;
}
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (bounding_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab⟨int⟩ &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_cone_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_end_1[0]), &(C_end_2[0]), &C_radius, &C_color,
          &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
          medium_E_dir.dataPtr(), &watch_draw_cone, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_cone_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab⟨int⟩ &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
```

```
        f_draw_cone_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
            &(bounding_box_hi[0]), &(C_end_1[0]), &(C_end_2[0]), &C_radius, &C_color,
            &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
            medium_H_dir.dataPtr(), &watch_draw_cone, &return_status);
        if (¬return_status) {
          pout() << "\t\tERROR(f_draw_cone_): Fortran problem, dir = " << dir << endl;
          goto exit;
        }
      }
    }
  }
exit: return return_status;
}
```

This code is used in chunk 159.

8.5.2 Drawing Cone on Tags

This function can be done similarly to the one that draws a cone on the media, because *f_draw_cone_* (this is a Fortran code that does things quickly) is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel.

164 ⟨Function *draw_tags_cone* 164⟩ ≡

```
int draw_tags_cone(Vector<Real> &C_cone_1, Vector<Real> &C_cone_2, Real &C_radius)
{
  pout() << "ERROR(draw_cone): Drawing on tags not implemented yet." << endl;
  return false;
}
```

This code is used in chunk 159.

8.6 Draw Disk

165 ⟨ Function *draw_disk* 165 ⟩ ≡

```
SCM draw_disk(SCM center, SCM normal, SCM radius, SCM thickness, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_center(SpaceDim, 0.0), C_normal(SpaceDim, 0.0);
  Real C_radius, C_thickness;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_disk = 0;
  ⟨ draw_disk: introduce yourself 166 ⟩
  ⟨ draw_disk: read, check, and translate arguments 167 ⟩
  ⟨ draw_disk: call appropriate utility 168 ⟩
exit:
  if (watch_draw_disk) pout() << "draw_disk: returning to caller" << endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_disk 169 ⟩
⟨ Function draw_tags_disk 170 ⟩
```

This code is used in chunk 134.

166 ¶⟨ *draw_disk*: introduce yourself 166 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_disk", watch_draw_disk);
if (watch_draw_disk) pout() << "draw_disk:" << endl;
```

This code is used in chunk 165.

167 ¶⟨ *draw_disk*: read, check, and translate arguments 167 ⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(center)) ∧ (scm_to_int(scm_f64vector_length(center)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(normal)) ∧ (scm_to_int(scm_f64vector_length(normal)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_center[i] = scm_to_double(scm_f64vector_ref(center, scm_from_int(i)));
    C_normal[i] = scm_to_double(scm_f64vector_ref(normal, scm_from_int(i)));
  }
  if (watch_draw_disk) {
    pout() << "\tdisk center: #f64(" << C_center << ")" << endl;
    pout() << "\t     normal: #f64(" << C_normal << ")" << endl;
  }
}
else {
  pout() << "\tERROR(draw_disk): " << "the first two arguments must be #f64(a b c) vectors" <<
      endl;
  return_status = false;
  goto exit;
}
if (scm_is_real(radius)) {
  C_radius = scm_to_double(radius);
  if (watch_draw_disk) pout() << "\t     radius: " << C_radius << endl;
}
else {
```

```
    pout() << "\tERROR(draw_disk): " << "the third argument must be a real (radius)" << endl;
    return_status = false;
    goto exit;
  }
  if (scm_is_real(thickness)) {
    C_thickness = scm_to_double(thickness);
    if (watch_draw_disk) pout() << "\t     thickness: " << C_thickness << endl;
  }
  else {
    pout() << "\tERROR(draw_disk): " << "the fourth argument must be a real (thickness)" << endl;
    return_status = false;
    goto exit;
  }
        /* Is the fifth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_disk): negative colors are reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_disk): drawing color must be an integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 165.

168 ¶⟨ *draw_disk*: call appropriate utility 168 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_disk(C_center, C_normal, C_radius, C_thickness, C_color);
  else return_status = draw_tags_disk(C_center, C_normal, C_radius, C_thickness);
```

This code is used in chunk 165.

8.6.1 Drawing Disk on Media

169 ⟨ Function *draw_media_disk* 169 ⟩ ≡

```
  int draw_media_disk(Vector⟨Real⟩ &C_center, Vector⟨Real⟩ &C_normal, Real &C_radius, Real
          &C_thickness, int &C_color)
  {
    extern level *current_level_ptr;
    Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
    Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
    Vector⟨Real⟩ &origin = current_level_ptr→origin;
    Real &delta = current_level_ptr→delta;
    DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
    DataIterator &data_iterator = current_level_ptr→data_iterator;
```

```
IntVect bounding_box_lo = IntVect::Zero;
IntVect bounding_box_hi = IntVect::Zero;
IntVect total_field_lo = IntVect::Zero;
IntVect total_field_hi = IntVect::Zero;
int buffer_size = REFINE_BUFFER_SIZE;
int ghost_margin = N_GHOST_CELLS;
int watch_draw_disk = false;
int return_status = true;      /* this is for Scheme handling */
Vector<Real> C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
Vector<Real> C_end_1(SpaceDim, 0.0), C_end_2(SpaceDim, 0.0);
Vector<Real> n(SpaceDim, 0.0);
Real C_normal_norm;
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_disk", watch_draw_disk);
if (watch_draw_disk) pout() << "\tdraw_media_disk:" << endl;
if ((C_center.size() ≠ SpaceDim) ∨ (C_normal.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_disk): badly sized end vectors" << endl;
  return_status = false;
  goto exit;
}
if ((C_normal[0] ≡ 0.0) ∧ (C_normal[1] ≡ 0.0) ∧ (C_normal[2] ≡ 0.0)) {
  pout() << "\t\tERROR(draw_media_disk): normal vector zero" << endl;
  return_status = false;
  goto exit;
}
C_normal_norm = sqrt(C_normal[0]*C_normal[0]+C_normal[1]*C_normal[1]+C_normal[2]*C_normal[2]);
n[0] = C_normal[0]/C_normal_norm;
n[1] = C_normal[1]/C_normal_norm;
n[2] = C_normal[2]/C_normal_norm;
for (int i = 0; i < SpaceDim; i++) {
  C_end_1[i] = C_center[i] − n[i] * C_thickness/2.0;
  C_end_2[i] = C_center[i] + n[i] * C_thickness/2.0;
}
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = min(C_end_1[i], C_end_2[i]) − C_radius;
  C_hi[i] = max(C_end_1[i], C_end_2[i]) + C_radius;
}
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_disk): bounding box empty" << endl;
  return_status = false;
  goto exit;
}
if (¬total_field_box.contains(bounding_box)) {
  pout() << "\tERROR(draw_disk): bounding box not in total field region" << endl;
  return_status = false;
```

```
      goto exit;
    }
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      Box layout_box = box_layout.get(data_index);
      if (bounding_box.intersects(layout_box)) {
          /* This box may need work. This will be ultimately decided by the Fortran routine. */
        for (int dir = 0; dir < SpaceDim; dir++) {
          BaseFab<int> &medium_E_dir = current_level_ptr->medium_E[data_index][dir];
          Box E_box_dir = medium_E_dir.box();
          IntVect E_lo = E_box_dir.smallEnd();
          IntVect E_hi = E_box_dir.bigEnd();
          f_draw_disk_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_center[0]), &(C_normal[0]), &C_radius, &C_thickness, &C_color,
              &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
              medium_E_dir.dataPtr(), &watch_draw_disk, &return_status);
          if (¬return_status) {
            pout() << "\t\tERROR(f_draw_disk_): Fortran problem, dir = " << dir << endl;
            goto exit;
          }
          BaseFab<int> &medium_H_dir = current_level_ptr->medium_H[data_index][dir];
          Box H_box_dir = medium_H_dir.box();
          IntVect H_lo = H_box_dir.smallEnd();
          IntVect H_hi = H_box_dir.bigEnd();
          f_draw_disk_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_center[0]), &(C_normal[0]), &C_radius, &C_thickness, &C_color,
              &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
              medium_H_dir.dataPtr(), &watch_draw_disk, &return_status);
          if (¬return_status) {
            pout() << "\t\tERROR(f_draw_disk_): Fortran problem, dir = " << dir << endl;
            goto exit;
          }
        }
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 165.

8.6.2 Drawing Disk on Tags

170 ⟨Function *draw_tags_disk* 170⟩ ≡

```
  int draw_tags_disk(Vector<Real> &C_center, Vector<Real> &C_normal, Real &C_radius, Real
          &C_thickness)
  {
    pout() << "ERROR(draw_disk): Drawing on tags not implemented yet." << endl;
    return false;
  }
```

This code is used in chunk 165.

8.7 Draw Arbitrary Ellipsoid

171 ⟨ Function *draw_ary_ellipsoid* 171 ⟩ ≡

```
SCM draw_ary_ellipsoid(SCM center, SCM e_1, SCM r_1, SCM e_2, SCM r_2, SCM e_3, SCM
        r_3, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_center(SpaceDim, 0.0), C_e_1(SpaceDim, 0.0), C_e_2(SpaceDim, 0.0),
      C_e_3(SpaceDim, 0.0);
  Real C_r_1, C_r_2, C_r_3;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_ary_ellipsoid = 0;
  ⟨ draw_ary_ellipsoid: introduce yourself 172 ⟩
  ⟨ draw_ary_ellipsoid: read, check, and translate arguments 173 ⟩
  ⟨ draw_ary_ellipsoid: call appropriate utility 174 ⟩
exit:
  if (watch_draw_ary_ellipsoid) pout() << "draw_ary_ellipsoid: returning to caller" << endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_ary_ellipsoid 175 ⟩
⟨ Function draw_tags_ary_ellipsoid 176 ⟩
```

This code is used in chunk 134.

172 ¶⟨ *draw_ary_ellipsoid*: introduce yourself 172 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ary_ellipsoid", watch_draw_ary_ellipsoid);
if (watch_draw_ary_ellipsoid) pout() << "draw_ary_ellipsoid:" << endl;
```

This code is used in chunk 171.

173 ¶⟨ *draw_ary_ellipsoid*: read, check, and translate arguments 173 ⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(center)) ∧ (scm_to_int(scm_f64vector_length(center)) ≡
      SpaceDim) ∧ scm_is_true(scm_f64vector_p(e_1)) ∧ (scm_to_int(scm_f64vector_length(e_1)) ≡
      SpaceDim) ∧ scm_is_true(scm_f64vector_p(e_2)) ∧ (scm_to_int(scm_f64vector_length(e_2)) ≡
      SpaceDim) ∧ scm_is_true(scm_f64vector_p(e_3)) ∧ (scm_to_int(scm_f64vector_length(e_3)) ≡ SpaceDim))
  {
  for (int i = 0; i < SpaceDim; i++) {
    C_center[i] = scm_to_double(scm_f64vector_ref(center, scm_from_int(i)));
    C_e_1[i] = scm_to_double(scm_f64vector_ref(e_1, scm_from_int(i)));
    C_e_2[i] = scm_to_double(scm_f64vector_ref(e_2, scm_from_int(i)));
    C_e_3[i] = scm_to_double(scm_f64vector_ref(e_3, scm_from_int(i)));
  }
  if (watch_draw_ary_ellipsoid) {
    pout() << "\tellipsoid center: #f64(" << C_center << ")" << endl;
    pout() << "\t          e_1:    #f64(" << C_e_1 << ")" << endl;
    pout() << "\t          e_2:    #f64(" << C_e_2 << ")" << endl;
    pout() << "\t          e_3:    #f64(" << C_e_3 << ")" << endl;
  }
}
else {
  pout() << "\tERROR(draw_ary_ellipsoid): " << "args 1, 2, 4, and 6 must be #f64(a b c)\
      vectors" << endl;
```

```
    return_status = false;
    goto exit;
  }
  if (scm_is_real(r_1) ∧ scm_is_real(r_2) ∧ scm_is_real(r_3)) {
    C_r_1 = scm_to_double(r_1);
    C_r_2 = scm_to_double(r_2);
    C_r_3 = scm_to_double(r_3);
    if (watch_draw_ary_ellipsoid) {
      pout() ≪ "\t␣␣␣␣␣␣␣␣␣␣r_1:␣␣␣␣" ≪ C_r_1 ≪ endl;
      pout() ≪ "\t␣␣␣␣␣␣␣␣␣␣r_2:␣␣␣␣" ≪ C_r_2 ≪ endl;
      pout() ≪ "\t␣␣␣␣␣␣␣␣␣␣r_3:␣␣␣␣" ≪ C_r_3 ≪ endl;
    }
  }
  else {
    pout() ≪ "\tERROR(draw_ary_ellipsoid):␣" ≪ "args␣3,␣5␣and␣7␣must␣be␣real␣(radii)" ≪ endl;
    return_status = false;
    goto exit;
  }
        /* Is the fifth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() ≪ "\tERROR(draw_ary_ellipsoid):␣negative␣colors␣are␣reserved" ≪ endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() ≪ "\tERROR(draw_ary_ellipsoid):␣drawing␣color␣must␣be␣an␣integer" ≪ endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 171.

174 ¶⟨ *draw_ary_ellipsoid*: call appropriate utility 174 ⟩ ≡

```
  if (draw_on_media)
    return_status = draw_media_ary_ellipsoid(C_center, C_e_1, C_r_1, C_e_2, C_r_2, C_e_3, C_r_3, C_color);
  else return_status = draw_tags_ary_ellipsoid(C_center, C_e_1, C_r_1, C_e_2, C_r_2, C_e_3, C_r_3);
```

This code is used in chunk 171.

8.7.1 Drawing Arbitrary Ellipsoid on Media

175 ⟨ Function *draw_media_ary_ellipsoid* 175 ⟩ ≡

```
  int draw_media_ary_ellipsoid(Vector⟨Real⟩ &C_center, Vector⟨Real⟩ &C_e_1, Real &C_r_1, Vector⟨Real⟩
          &C_e_2, Real &C_r_2, Vector⟨Real⟩ &C_e_3, Real &C_r_3, int &C_color)
  {
    extern level *current_level_ptr;
```

```
Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
Vector⟨Real⟩ &origin = current_level_ptr→origin;
Real &delta = current_level_ptr→delta;
DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
DataIterator &data_iterator = current_level_ptr→data_iterator;
IntVect bounding_box_lo = IntVect::Zero;
IntVect bounding_box_hi = IntVect::Zero;
IntVect total_field_lo = IntVect::Zero;
IntVect total_field_hi = IntVect::Zero;
int buffer_size = REFINE_BUFFER_SIZE;
int ghost_margin = N_GHOST_CELLS;
int watch_draw_ary_ellipsoid = false;
int return_status = true;      /* this is for Scheme handling */
Vector⟨Real⟩ C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
Real r_max;
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ary_ellipsoid", watch_draw_ary_ellipsoid);
if (watch_draw_ary_ellipsoid) pout() << "\tdraw_media_ary_ellipsoid:" << endl;
if ((C_center.size() ≠ SpaceDim) ∨ (C_e_1.size() ≠ SpaceDim) ∨ (C_e_2.size() ≠
      SpaceDim) ∨ (C_e_3.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_ary_ellipsoid): badly sized vectors" << endl;
  return_status = false;
  goto exit;
}
if (((C_e_1[0] ≡ 0.0) ∧ (C_e_1[1] ≡ 0.0) ∧ (C_e_1[2] ≡ 0.0)) ∨ ((C_e_2[0] ≡ 0.0) ∧ (C_e_2[1] ≡
      0.0) ∧ (C_e_2[2] ≡ 0.0)) ∨ ((C_e_3[0] ≡ 0.0) ∧ (C_e_3[1] ≡ 0.0) ∧ (C_e_3[2] ≡ 0.0))) {
  pout() << "\t\tERROR(draw_media_ary_ellipsoid): zero direction vectors" << endl;
  return_status = false;
  goto exit;
}
if ((C_r_1 ≤ 0.0) ∨ (C_r_2 ≤ 0.0) ∨ (C_r_3 ≤ 0.0)) {
  pout() << "\t\tERROR(draw_media_ary_ellipsoid): zero or negative radii" << endl;
  return_status = false;
  goto exit;
}
r_max = max(C_r_1, C_r_2);
r_max = max(r_max, C_r_3);
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = C_center[i] − r_max;
  C_hi[i] = C_center[i] + r_max;
}
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_ary_ellipsoid): bounding box empty" << endl;
  return_status = false;
```

```
      goto exit;
    }
    if (¬total_field_box.contains(bounding_box)) {
      pout() << "\tERROR(draw_ary_ellipsoid):␣bounding␣box␣not␣in␣total␣field␣region" << endl;
      return_status = false;
      goto exit;
    }
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      Box layout_box = box_layout.get(data_index);
      if (bounding_box.intersects(layout_box)) {
          /* This box may need work. This will be ultimately decided by the Fortran routine. */
        for (int dir = 0; dir < SpaceDim; dir++) {
          BaseFab<int> &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
          Box E_box_dir = medium_E_dir.box();
          IntVect E_lo = E_box_dir.smallEnd();
          IntVect E_hi = E_box_dir.bigEnd();
          f_draw_ary_ellipsoid_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_center[0]), &(C_e_1[0]), &C_r_1, &(C_e_2[0]), &C_r_2, &(C_e_3[0]),
              &C_r_3, &C_color, &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]),
              &(E_hi[1]), &(E_hi[2]), medium_E_dir.dataPtr(), &watch_draw_ary_ellipsoid, &return_status);
          if (¬return_status) {
            pout() << "\t\tERROR(f_draw_ary_ellipsoid_):␣Fortran␣problem,␣dir␣=␣" << dir << endl;
            goto exit;
          }
          BaseFab<int> &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
          Box H_box_dir = medium_H_dir.box();
          IntVect H_lo = H_box_dir.smallEnd();
          IntVect H_hi = H_box_dir.bigEnd();
          f_draw_ary_ellipsoid_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_center[0]), &(C_e_1[0]), &C_r_1, &(C_e_2[0]), &C_r_2, &(C_e_3[0]),
              &C_r_3, &C_color, &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]),
              &(H_hi[1]), &(H_hi[2]), medium_H_dir.dataPtr(), &watch_draw_ary_ellipsoid, &return_status);
          if (¬return_status) {
            pout() << "\t\tERROR(f_draw_ary_ellipsoid_):␣Fortran␣problem,␣dir␣=␣" << dir << endl;
            goto exit;
          }
        }
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 171.

8.7.2 Drawing Arbitrary Ellipsoid on Tags

176 ⟨Function *draw_tags_ary_ellipsoid* 176⟩ ≡

```
  int draw_tags_ary_ellipsoid(Vector<Real> &C_center, Vector<Real> &C_e_1, Real &C_r_1, Vector<Real>
          &C_e_2, Real &C_r_2, Vector<Real> &C_e_3, Real &C_r_3)
  {
    pout() << "ERROR(draw_ary_ellipsoid):␣Drawing␣on␣tags␣not␣implemented␣yet." << endl;
    return false;
  }
```

This code is used in chunk 171.

8.8 Draw Parallelepiped

177 ⟨Function *draw_parallelepiped* 177⟩ ≡

```
SCM draw_parallelepiped(SCM corner, SCM e_1, SCM e_2, SCM e_3, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_corner(SpaceDim, 0.0), C_e_1(SpaceDim, 0.0), C_e_2(SpaceDim, 0.0),
      C_e_3(SpaceDim, 0.0);
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_parallelepiped = 0;
  ⟨ draw_parallelepiped: introduce yourself 178 ⟩
  ⟨ draw_parallelepiped: read, check, and translate arguments 179 ⟩
  ⟨ draw_parallelepiped: call appropriate utility 180 ⟩
exit:
  if (watch_draw_parallelepiped) pout() << "draw_parallelepiped:␣returning␣to␣caller" << endl;
  return scm_from_bool(return_status);
}
⟨Function draw_media_parallelepiped 181⟩
⟨Function draw_tags_parallelepiped 182⟩
```

This code is used in chunk 134.

178 ¶⟨ *draw_parallelepiped*: introduce yourself 178⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_parallelepiped", watch_draw_parallelepiped);
if (watch_draw_parallelepiped) pout() << "draw_parallelepiped:" << endl;
```

This code is used in chunk 177.

179 ¶⟨ *draw_parallelepiped*: read, check, and translate arguments 179⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(corner)) ∧ (scm_to_int(scm_f64vector_length(corner)) ≡
      SpaceDim) ∧ scm_is_true(scm_f64vector_p(e_1)) ∧ (scm_to_int(scm_f64vector_length(e_1)) ≡
      SpaceDim) ∧ scm_is_true(scm_f64vector_p(e_2)) ∧ (scm_to_int(scm_f64vector_length(e_2)) ≡
      SpaceDim) ∧ scm_is_true(scm_f64vector_p(e_3)) ∧ (scm_to_int(scm_f64vector_length(e_3)) ≡ SpaceDim))
  {
  for (int i = 0; i < SpaceDim; i++) {
    C_corner[i] = scm_to_double(scm_f64vector_ref(corner, scm_from_int(i)));
    C_e_1[i] = scm_to_double(scm_f64vector_ref(e_1, scm_from_int(i)));
    C_e_2[i] = scm_to_double(scm_f64vector_ref(e_2, scm_from_int(i)));
    C_e_3[i] = scm_to_double(scm_f64vector_ref(e_3, scm_from_int(i)));
  }
  if (watch_draw_parallelepiped) {
    pout() << "\tparallelepiped␣corner:␣#f64(" << C_corner << ")" << endl;
    pout() << "\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣e_1:␣␣␣␣#f64(" << C_e_1 << ")" << endl;
    pout() << "\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣e_2:␣␣␣␣#f64(" << C_e_2 << ")" << endl;
    pout() << "\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣e_3:␣␣␣␣#f64(" << C_e_3 << ")" << endl;
  }
}
else {
  pout() << "\tERROR(draw_parallelepiped):␣" << "args␣1␣through␣4␣must␣be␣#f64(a␣b␣c)␣ve\
      ctors" << endl;
  return_status = false;
```

```
    goto exit;
  }
        /* Is the fifth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_parallelepiped): negative colors are reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_parallelepiped): drawing color must be an integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 177.

180 ¶⟨ *draw_parallelepiped*: call appropriate utility 180 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_parallelepiped(C_corner, C_e_1, C_e_2, C_e_3, C_color);
  else return_status = draw_tags_parallelepiped(C_corner, C_e_1, C_e_2, C_e_3);
```

This code is used in chunk 177.

8.8.1 Drawing Parallelepiped on Media

181 ⟨ Function *draw_media_parallelepiped* 181 ⟩ ≡

```
  int draw_media_parallelepiped(Vector<Real> &C_corner, Vector<Real> &C_e_1, Vector<Real>
          &C_e_2, Vector<Real> &C_e_3, int &C_color)
  {
    extern level *current_level_ptr;
    Vector<Real> &signal_xyz_min = current_level_ptr->signal_xyz_min;
    Vector<Real> &signal_xyz_max = current_level_ptr->signal_xyz_max;
    Vector<Real> &origin = current_level_ptr->origin;
    Real &delta = current_level_ptr->delta;
    DisjointBoxLayout &box_layout = current_level_ptr->box_layout;
    DataIterator &data_iterator = current_level_ptr->data_iterator;
    IntVect bounding_box_lo = IntVect::Zero;
    IntVect bounding_box_hi = IntVect::Zero;
    IntVect total_field_lo = IntVect::Zero;
    IntVect total_field_hi = IntVect::Zero;
    int buffer_size = REFINE_BUFFER_SIZE;
    int ghost_margin = N_GHOST_CELLS;
    int watch_draw_parallelepiped = false;
    int return_status = true;     /* this is for Scheme handling */
    Vector<Real> C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
    Vector<Vector<Real>> corners(8);
    int count;
```

```
Real determinant;
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_parallelepiped", watch_draw_parallelepiped);
if (watch_draw_parallelepiped) pout() << "\tdraw_media_parallelepiped:" << endl;
if ((C_corner.size() ≠ SpaceDim) ∨ (C_e_1.size() ≠ SpaceDim) ∨ (C_e_2.size() ≠
        SpaceDim) ∨ (C_e_3.size() ≠ SpaceDim)) {
   pout() << "\t\tERROR(draw_media_parallelepiped): badly sized vectors" << endl;
   return_status = false;
   goto exit;
}
if (((C_e_1[0] ≡ 0.0) ∧ (C_e_1[1] ≡ 0.0) ∧ (C_e_1[2] ≡ 0.0)) ∨ ((C_e_2[0] ≡ 0.0) ∧ (C_e_2[1] ≡
        0.0) ∧ (C_e_2[2] ≡ 0.0)) ∨ ((C_e_3[0] ≡ 0.0) ∧ (C_e_3[1] ≡ 0.0) ∧ (C_e_3[2] ≡ 0.0))) {
   pout() << "\t\tERROR(draw_media_parallelepiped): zero direction vectors" << endl;
   return_status = false;
   goto exit;
}
determinant = C_e_1[0] * C_e_2[1] * C_e_3[2] + C_e_2[0] * C_e_3[1] * C_e_1[2] + C_e_3[0] * C_e_1[1] *
      C_e_2[2] − C_e_3[0] * C_e_2[1] * C_e_1[2] − C_e_2[0] * C_e_1[1] * C_e_3[2] − C_e_1[0] * C_e_3[1] * C_e_2[2];
if (determinant ≡ 0.0) {
   pout() << "\t\tERROR(draw_media_parallelepiped): zero volume" << endl;
   return_status = false;
   goto exit;
}
for (int count = 0; count < 8; count++) {
   (corners[count]).resize(SpaceDim, 0.0);
}
for (int i = 0; i < SpaceDim; i++) {
   (corners[0])[i] = C_corner[i];
   (corners[1])[i] = C_corner[i] + C_e_1[i];
   (corners[2])[i] = C_corner[i] + C_e_2[i];
   (corners[3])[i] = C_corner[i] + C_e_3[i];
   (corners[4])[i] = C_corner[i] + C_e_1[i] + C_e_2[i];
   (corners[5])[i] = C_corner[i] + C_e_1[i] + C_e_3[i];
   (corners[6])[i] = C_corner[i] + C_e_2[i] + C_e_3[i];
   (corners[7])[i] = C_corner[i] + C_e_1[i] + C_e_2[i] + C_e_3[i];
}
for (int i = 0; i < SpaceDim; i++) {
   C_lo[i] = min((corners[0])[i], (corners[1])[i]);
   for (count = 1; count < 8; count++) C_lo[i] = min(C_lo[i], (corners[count])[i]);
   C_hi[i] = max((corners[0])[i], (corners[1])[i]);
   for (count = 1; count < 8; count++) C_hi[i] = max(C_hi[i], (corners[count])[i]);
}
for (int i = 0; i < SpaceDim; i++) {
   bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
   bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
   total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
   total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
   pout() << "\tERROR(draw_parallelepiped): bounding box empty" << endl;
   return_status = false;
```

```
      goto exit;
    }
    if (¬total_field_box.contains(bounding_box)) {
      pout() ≪ "\tERROR(draw_parallelepiped):␣bounding␣box␣not␣in␣total␣field␣region" ≪ endl;
      return_status = false;
      goto exit;
    }
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      Box layout_box = box_layout.get(data_index);
      if (bounding_box.intersects(layout_box)) {
          /* This box may need work. This will be ultimately decided by the Fortran routine. */
        for (int dir = 0; dir < SpaceDim; dir++) {
          BaseFab⟨int⟩ &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
          Box E_box_dir = medium_E_dir.box();
          IntVect E_lo = E_box_dir.smallEnd();
          IntVect E_hi = E_box_dir.bigEnd();
          f_draw_parallelepiped_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_corner[0]), &(C_e_1[0]), &(C_e_2[0]), &(C_e_3[0]), &C_color,
              &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
              medium_E_dir.dataPtr(), &watch_draw_parallelepiped, &return_status);
          if (¬return_status) {
            pout() ≪ "\t\tERROR(f_draw_parallelepiped_):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
            goto exit;
          }
          BaseFab⟨int⟩ &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
          Box H_box_dir = medium_H_dir.box();
          IntVect H_lo = H_box_dir.smallEnd();
          IntVect H_hi = H_box_dir.bigEnd();
          f_draw_parallelepiped_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_corner[0]), &(C_e_1[0]), &(C_e_2[0]), &(C_e_3[0]), &C_color,
              &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
              medium_H_dir.dataPtr(), &watch_draw_parallelepiped, &return_status);
          if (¬return_status) {
            pout() ≪ "\t\tERROR(f_draw_parallelepiped_):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
            goto exit;
          }
        }
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 177.

8.8.2 Drawing Parallelepiped on Tags

182 ⟨Function *draw_tags_parallelepiped* 182⟩ ≡

```
  int draw_tags_parallelepiped(Vector⟨Real⟩ &C_corner, Vector⟨Real⟩ &C_v_1, Vector⟨Real⟩
          &C_v_2, Vector⟨Real⟩ &C_v_3)
  {
    pout() ≪ "ERROR(draw_parallelepiped):␣Drawing␣on␣tags␣not␣implemented␣yet." ≪ endl;
    return false;
  }
```

This code is used in chunk 177.

8.9 Draw Torus

183 ⟨ Function *draw_torus* 183 ⟩ ≡

```
SCM draw_torus(SCM center, SCM normal, SCM radius_1, SCM radius_2, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_center(SpaceDim, 0.0), C_normal(SpaceDim, 0.0);
  Real C_radius_1, C_radius_2;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_torus = 0;
  ⟨ draw_torus: introduce yourself 184 ⟩
  ⟨ draw_torus: read, check, and translate arguments 185 ⟩
  ⟨ draw_torus: call appropriate utility 186 ⟩
exit:
  if (watch_draw_torus) pout() ≪ "draw_torus:␣returning␣to␣caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_torus 187 ⟩
⟨ Function draw_tags_torus 188 ⟩
```

This code is used in chunk 134.

184 ¶⟨ *draw_torus*: introduce yourself 184 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_torus", watch_draw_torus);
if (watch_draw_torus) pout() ≪ "draw_torus:" ≪ endl;
```

This code is used in chunk 183.

185 ¶⟨ *draw_torus*: read, check, and translate arguments 185 ⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(center)) ∧ (scm_to_int(scm_f64vector_length(center)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(normal)) ∧ (scm_to_int(scm_f64vector_length(normal)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_center[i] = scm_to_double(scm_f64vector_ref(center, scm_from_int(i)));
    C_normal[i] = scm_to_double(scm_f64vector_ref(normal, scm_from_int(i)));
  }
  if (watch_draw_torus) {
    pout() ≪ "\ttorus␣center:␣#f64(" ≪ C_center ≪ ")" ≪ endl;
    pout() ≪ "\t␣␣␣␣␣␣normal:␣#f64(" ≪ C_normal ≪ ")" ≪ endl;
  }
}
else {
  pout() ≪ "\tERROR(draw_torus):␣" ≪ "the␣first␣two␣arguments␣must␣be␣#f64(a␣b␣c)␣vectors" ≪
      endl;
  return_status = false;
  goto exit;
}
if (scm_is_real(radius_1) ∧ scm_is_real(radius_2)) {
  C_radius_1 = scm_to_double(radius_1);
  C_radius_2 = scm_to_double(radius_2);
  if (watch_draw_torus) {
    pout() ≪ "\t␣␣␣␣␣␣radius_1:␣" ≪ C_radius_1 ≪ endl;
```

```
      pout() << "\t␣␣␣␣␣␣radius_2:␣" << C_radius_2 << endl;
    }
  }
  else {
    pout() << "\tERROR(draw_torus):␣" << "arguments␣3␣and␣4␣must␣be␣reals␣(radii)" << endl;
    return_status = false;
    goto exit;
  }
        /* Is the fifth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_torus):␣negative␣colors␣are␣reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_torus):␣drawing␣color␣must␣be␣an␣integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 183.

186 ¶⟨ *draw_torus*: call appropriate utility 186 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_torus(C_center, C_normal, C_radius_1, C_radius_2, C_color);
  else return_status = draw_tags_torus(C_center, C_normal, C_radius_1, C_radius_2);
```

This code is used in chunk 183.

8.9.1 Drawing Torus on Media

187 ⟨ Function *draw_media_torus* 187 ⟩ ≡

```
  int draw_media_torus(Vector<Real> &C_center, Vector<Real> &C_normal, Real &C_radius_1, Real
          &C_radius_2, int &C_color)
  {
    extern level *current_level_ptr;
    Vector<Real> &signal_xyz_min = current_level_ptr→signal_xyz_min;
    Vector<Real> &signal_xyz_max = current_level_ptr→signal_xyz_max;
    Vector<Real> &origin = current_level_ptr→origin;
    Real &delta = current_level_ptr→delta;
    DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
    DataIterator &data_iterator = current_level_ptr→data_iterator;
    IntVect bounding_box_lo = IntVect::Zero;
    IntVect bounding_box_hi = IntVect::Zero;
    IntVect total_field_lo = IntVect::Zero;
    IntVect total_field_hi = IntVect::Zero;
    int buffer_size = REFINE_BUFFER_SIZE;
```

```
int ghost_margin = N_GHOST_CELLS;
int watch_draw_torus = false;
int return_status = true;      /* this is for Scheme handling */
Vector<Real> C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
Vector<Real> C_end_1(SpaceDim, 0.0), C_end_2(SpaceDim, 0.0);
Vector<Real> n(SpaceDim, 0.0);
Real r_max, r_min, r_tube, C_normal_norm;
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_torus", watch_draw_torus);
if (watch_draw_torus) pout() << "\tdraw_media_torus:" << endl;
if ((C_center.size() ≠ SpaceDim) ∨ (C_normal.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_torus): badly sized vectors" << endl;
  return_status = false;
  goto exit;
}
if ((C_normal[0] ≡ 0.0) ∧ (C_normal[1] ≡ 0.0) ∧ (C_normal[2] ≡ 0.0)) {
  pout() << "\t\tERROR(draw_media_torus): normal vector zero" << endl;
  return_status = false;
  goto exit;
}
C_normal_norm = sqrt(C_normal[0]*C_normal[0]+C_normal[1]*C_normal[1]+C_normal[2]*C_normal[2]);
n[0] = C_normal[0]/C_normal_norm;
n[1] = C_normal[1]/C_normal_norm;
n[2] = C_normal[2]/C_normal_norm;
     /* The logic of making a bounding box is similar to that used in the draw_disk code. */
r_max = max(C_radius_1, C_radius_2);
r_min = min(C_radius_1, C_radius_2);
if (r_max ≡ r_min) {
  pout() << "\t\tERROR(draw_media_torus): equal radii" << endl;
  return_status = false;
  goto exit;
}
r_tube = (r_max − r_min)/2.0;
for (int i = 0; i < SpaceDim; i++) {
  C_end_1[i] = C_center[i] − n[i] * r_tube;
  C_end_2[i] = C_center[i] + n[i] * r_tube;
}
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = min(C_end_1[i], C_end_2[i]) − r_max;
  C_hi[i] = max(C_end_1[i], C_end_2[i]) + r_max;
}
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_torus): bounding box empty" << endl;
  return_status = false;
  goto exit;
```

```
    }
    if (¬total_field_box.contains(bounding_box)) {
      pout() ≪ "\tERROR(draw_torus):␣bounding␣box␣not␣in␣total␣field␣region" ≪ endl;
      return_status = false;
      goto exit;
    }
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      Box layout_box = box_layout.get(data_index);
      if (bounding_box.intersects(layout_box)) {
          /* This box may need work. This will be ultimately decided by the Fortran routine. */
        for (int dir = 0; dir < SpaceDim; dir++) {
          BaseFab⟨int⟩ &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
          Box E_box_dir = medium_E_dir.box();
          IntVect E_lo = E_box_dir.smallEnd();
          IntVect E_hi = E_box_dir.bigEnd();
          f_draw_torus_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_center[0]), &(C_normal[0]), &C_radius_1, &C_radius_2, &C_color,
              &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
              medium_E_dir.dataPtr(), &watch_draw_torus, &return_status);
          if (¬return_status) {
            pout() ≪ "\t\tERROR(f_draw_torus_):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
            goto exit;
          }
          BaseFab⟨int⟩ &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
          Box H_box_dir = medium_H_dir.box();
          IntVect H_lo = H_box_dir.smallEnd();
          IntVect H_hi = H_box_dir.bigEnd();
          f_draw_torus_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
              &(bounding_box_hi[0]), &(C_center[0]), &(C_normal[0]), &C_radius_1, &C_radius_2, &C_color,
              &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
              medium_H_dir.dataPtr(), &watch_draw_torus, &return_status);
          if (¬return_status) {
            pout() ≪ "\t\tERROR(f_draw_torus_):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
            goto exit;
          }
        }
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 183.

8.9.2 Drawing Torus on Tags

188 ⟨Function *draw_tags_torus* 188⟩ ≡

```
  int draw_tags_torus(Vector⟨Real⟩ &C_center, Vector⟨Real⟩ &C_normal, Real &C_r_1, Real &C_r_2)
  {
    pout() ≪ "ERROR(draw_torus):␣Drawing␣on␣tags␣not␣implemented␣yet." ≪ endl;
    return false;
  }
```

This code is used in chunk 183.

8.10 Draw Lens

189 ⟨ Function *draw_lens* 189 ⟩ ≡

```
SCM draw_lens(SCM center_1, SCM radius_1, SCM center_2, SCM radius_2, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_center_1(SpaceDim, 0.0), C_center_2(SpaceDim, 0.0);
  Real C_radius_1, C_radius_2;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_lens = 0;
  ⟨ draw_lens: introduce yourself 190 ⟩
  ⟨ draw_lens: read, check, and translate arguments 191 ⟩
  ⟨ draw_lens: call appropriate utility 192 ⟩
exit:
  if (watch_draw_lens) pout() ≪ "draw_lens:␣returning␣to␣caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_lens 193 ⟩
⟨ Function draw_tags_lens 194 ⟩
```

This code is used in chunk 134.

190 ¶⟨ *draw_lens*: introduce yourself 190 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_lens", watch_draw_lens);
if (watch_draw_lens) pout() ≪ "draw_lens:" ≪ endl;
```

This code is used in chunk 189.

191 ¶⟨ *draw_lens*: read, check, and translate arguments 191 ⟩ ≡

```
    /* Are the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(center_1)) ∧ (scm_to_int(scm_f64vector_length(center_1)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(center_2)) ∧ (scm_to_int(scm_f64vector_length(center_2)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_center_1[i] = scm_to_double(scm_f64vector_ref(center_1, scm_from_int(i)));
    C_center_2[i] = scm_to_double(scm_f64vector_ref(center_2, scm_from_int(i)));
  }
  if (watch_draw_lens) {
    pout() ≪ "\tlens␣center_1:␣#f64(" ≪ C_center_1 ≪ ")" ≪ endl;
    pout() ≪ "\t␣␣␣␣␣center_2:␣#f64(" ≪ C_center_2 ≪ ")" ≪ endl;
  }
}
else {
  pout() ≪ "\tERROR(draw_lens):␣" ≪ "args␣1␣and␣3␣must␣be␣#f64(a␣b␣c)␣vectors" ≪ endl;
  return_status = false;
  goto exit;
}
if (scm_is_real(radius_1) ∧ scm_is_real(radius_2)) {
  C_radius_1 = scm_to_double(radius_1);
  C_radius_2 = scm_to_double(radius_2);
  if (watch_draw_lens) {
    pout() ≪ "\t␣␣␣␣␣␣radius_1:␣" ≪ C_radius_1 ≪ endl;
    pout() ≪ "\t␣␣␣␣␣␣radius_2:␣" ≪ C_radius_2 ≪ endl;
```

```
    }
  }
  else {
    pout() << "\tERROR(draw_lens): " << "arguments 2 and 4 must be reals (radii)" << endl;
    return_status = false;
    goto exit;
  }
        /* Is the fifth, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() << "\tERROR(draw_lens): negative colors are reserved" << endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() << "\tERROR(draw_lens): drawing color must be an integer" << endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 189.

192 ¶⟨ *draw_lens*: call appropriate utility 192 ⟩ ≡

```
  if (draw_on_media)
    return_status = draw_media_lens(C_center_1, C_radius_1, C_center_2, C_radius_2, C_color);
  else return_status = draw_tags_lens(C_center_1, C_radius_1, C_center_2, C_radius_2);
```

This code is used in chunk 189.

8.10.1 Drawing Lens on Media

193 ⟨ Function *draw_media_lens* 193 ⟩ ≡

```
  int draw_media_lens(Vector<Real> &C_center_1, Real &C_radius_1, Vector<Real> &C_center_2, Real
          &C_radius_2, int &C_color)
  {
    extern level *current_level_ptr;
    Vector<Real> &signal_xyz_min = current_level_ptr->signal_xyz_min;
    Vector<Real> &signal_xyz_max = current_level_ptr->signal_xyz_max;
    Vector<Real> &origin = current_level_ptr->origin;
    Real &delta = current_level_ptr->delta;
    DisjointBoxLayout &box_layout = current_level_ptr->box_layout;
    DataIterator &data_iterator = current_level_ptr->data_iterator;
    IntVect bounding_box_lo = IntVect::Zero;
    IntVect bounding_box_hi = IntVect::Zero;
    IntVect total_field_lo = IntVect::Zero;
    IntVect total_field_hi = IntVect::Zero;
    int buffer_size = REFINE_BUFFER_SIZE;
```

```
int ghost_margin = N_GHOST_CELLS;
int watch_draw_lens = false;
int return_status = true;      /* this is for Scheme handling */
Vector<Real> C_lo(SpaceDim, 0.0), C_hi(SpaceDim, 0.0);
Box bounding_box, total_field_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_lens", watch_draw_lens);
if (watch_draw_lens) pout() << "\tdraw_media_lens:" << endl;
if ((C_center_1.size() ≠ SpaceDim) ∨ (C_center_2.size() ≠ SpaceDim)) {
  pout() << "\t\tERROR(draw_media_lens): badly sized vectors" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  C_lo[i] = min(C_center_1[i], C_center_2[i]) − max(C_radius_1, C_radius_2);
  C_hi[i] = max(C_center_1[i], C_center_2[i]) + max(C_radius_1, C_radius_2);
}
for (int i = 0; i < SpaceDim; i++) {
  bounding_box_lo[i] = (int) floor((C_lo[i] − origin[i])/delta) − buffer_size;
  bounding_box_hi[i] = (int) ceil((C_hi[i] − origin[i])/delta) + buffer_size;
  total_field_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta);
  total_field_hi[i] = (int) ceil((signal_xyz_max[i] − origin[i])/delta);
}
bounding_box.define(bounding_box_lo, bounding_box_hi);
total_field_box.define(total_field_lo, total_field_hi);
if (bounding_box.isEmpty()) {
  pout() << "\tERROR(draw_lens): bounding box empty" << endl;
  return_status = false;
  goto exit;
}
if (¬total_field_box.contains(bounding_box)) {
  pout() << "\tERROR(draw_lens): bounding box not in total field region" << endl;
  return_status = false;
  goto exit;
}
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (bounding_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr→medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_lens_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
          &(bounding_box_hi[0]), &(C_center_1[0]), &C_radius_1, &(C_center_2[0]), &C_radius_2,
          &C_color, &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]),
          &(E_hi[2]), medium_E_dir.dataPtr(), &watch_draw_lens, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_lens_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr→medium_H[data_index][dir];
```

```
            Box H_box_dir = medium_H_dir.box();
            IntVect H_lo = H_box_dir.smallEnd();
            IntVect H_hi = H_box_dir.bigEnd();
            f_draw_lens_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(bounding_box_lo[0]),
                &(bounding_box_hi[0]), &(C_center_1[0]), &C_radius_1, &(C_center_2[0]), &C_radius_2,
                &C_color, &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]),
                &(H_hi[2]), medium_H_dir.dataPtr(), &watch_draw_lens, &return_status);
            if (¬return_status) {
              pout() ≪ "\t\tERROR(f_draw_lens_): Fortran problem, dir = " ≪ dir ≪ endl;
              goto exit;
            }
          }
        }
      }
    exit: return return_status;
    }
```

This code is used in chunk 189.

8.10.2 Drawing Lens on Tags

194 ⟨Function *draw_tags_lens* 194⟩ ≡

```
  int draw_tags_lens(Vector⟨Real⟩ &C_center_1, Real &C_radius_1, Vector⟨Real⟩ &C_center_2, Real
          &C_radius_2)
  {
    pout() ≪ "ERROR(draw_lens): Drawing on tags not implemented yet." ≪ endl;
    return false;
  }
```

This code is used in chunk 189.

8.11 Auxiliary Scheme Procedures

8.11.1 Is a Symbol Defined

Function *does_scm_symbol_exist*, borrowed from [9] with one small modification that fixes the guile-1.8.1 note about deprecated features, does a job that is similar to *scm_c_lookup*( )[13] with one difference: it does not bomb out if the symbol is not found, just returns *false*.

Implementation notes:

scm_str2symbol This function is defined on `"libguile/discouraged.h"`. I have replaced it with *scm_string_to_symbol*.

scm_sym2var This is a Guile internal, defined on `"libguile/modules.h"`.

scm_current_module_lookup_closure( ) Ditto.

This function is also declared and defined in **SCMParmParse** as a protected member. See Section 12.1, page 350.

196 ⟨ Auxiliary Scheme Procedures 196 ⟩ ≡

```
bool does_scm_symbol_exist(const char *name)
{
  SCM symbol;
  SCM variable;
  assert(name ≠ (const char *) Λ);
  symbol = scm_string_to_symbol(scm_from_locale_string(name));
      /* Lookup this symbol in the current module lookup closure, but do not create it automatically if it
      does not exist. This is what the third argument to this function is about. Return SCM_BOOL_F if this is
      the case. */
  variable = scm_sym2var(symbol, scm_current_module_lookup_closure( ), SCM_BOOL_F);
  return (variable ≠ SCM_BOOL_F);
}
```

This code is used in chunk 134.

[13] Defined in "Accessing Modules from C" in "The Guile Module System" section of the "Modules" chapter.

8.12 Functions for MPI Connectivity

These are wrappers around Chombo functions that themselves are wrappers around MPI functions. Nothing special. The purpose is to extend the most basic MPI functionality to the Scheme layer of the program. Expected use is in user defined flux functions and similar. See Section 3.5.3, page 181 for an example of their use.

197 ⟨MPI Functions 197⟩ ≡

```
SCM SCMnumProc()
{   /* Return number of processes in the MPI pool. */
  int pool_size = 0;
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  pool_size = numProc();
  if (watch_scheme) pout() << "\tSCMnumProc: procs in pool = " << pool_size << endl;
  return (scm_from_int(pool_size));
}
SCM SCMprocID()
{   /* Return this process' rank. */
  int rank = 0;
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  rank = procID();
  if (watch_scheme) pout() << "\tSCMprocID: my rank = " << rank << endl;
  return (scm_from_int(rank));
}
SCM SCMbarrier()
{   /* Synchronize all processes. */
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  if (watch_scheme) pout() << "\tSCMBarrier: synchronizing... ";
  barrier();
  if (watch_scheme) pout() << "done." << endl;
  return (SCM_BOOL_T);
}
SCM SCMuniqueProc()
{   /* Return a unique process ID for this task. */   /* The only task defined is compute. */
  int watch_scheme = 0;
  int unique_process = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  unique_process = uniqueProc(SerialTask::compute);
  if (watch_scheme) pout() << "\tSCMuniqueProc: unique proc rank = " << unique_process << endl;
  return (scm_from_int(unique_process));
}
SCM SCMgather(SCM collection_vector, SCM item, SCM destination)
{   /* Gather items from all processes on the collection_vector. */   /* Only the destination process
      does this. */   /* Defined for item being Scheme integer or real only. */
  int watch_scheme = 0;
  int root_process_rank = scm_to_int(destination);
  int number_of_processes = numProc();
```

```
    int my_rank = procID();
    SCMParmParse watch_parameter_parser("forms.watch");
    watch_parameter_parser.query("scheme", watch_scheme);
    if (watch_scheme) pout() << "SCMgather:" << endl;      /* Is collection_vector appropriately sized? */
    if (scm_c_vector_length(collection_vector) < number_of_processes) {
      pout() << "\tSCMgather: ERROR: data vector size too small." << endl;
      return (SCM_BOOL_F);
    }
    else if (scm_is_real(item)) {
        /* Scheme reals will answer positively to integer? if they are also integers, so we cannot capture
          this. We treat them all like doubles. */
      Vector<Real> vector_of_items(number_of_processes, (Real) 0.0);
      Real datum = (Real) scm_to_double(item);
      if (watch_scheme) pout() << "\tSCMgather: sending " << datum << endl;
      gather(vector_of_items, datum, root_process_rank);
      if (my_rank ≡ root_process_rank) {
        for (int count = 0; count < number_of_processes; count++)
          scm_c_vector_set_x(collection_vector, count, scm_from_double((double) vector_of_items[count]));
        if (watch_scheme) pout() << "\tSCMgather: received " << vector_of_items << endl;
      }
      return (SCM_BOOL_T);
    }
    else {
      pout() << "\tSCMgather: ERROR: cannot gather this type." << endl;
      return (SCM_BOOL_F);
    }
  }
  SCM SCMbroadcast(SCM item, SCM source)
  {     /* The source process broadcasts item to all processes. */
        /* The item may be Scheme integer or real. */
    int watch_scheme = 0;
    int source_process_rank = scm_to_int(source);
    int my_rank = procID();
    SCMParmParse watch_parameter_parser("forms.watch");
    watch_parameter_parser.query("scheme", watch_scheme);
    if (watch_scheme) pout() << "SCMbroadcast:" << endl;
    if (scm_is_real(item)) {
        /* Scheme reals will answer positively to integer? if they are also integers, so we can't capture this
          condition. We treat all numbers like doubles. */
      Real receive = (Real) scm_to_double(item);
      if (watch_scheme ∧ (my_rank ≡ source_process_rank))
        pout() << "\tSCMbroadcast: sending: " << receive << endl;
      broadcast(receive, source_process_rank);
      if (watch_scheme) pout() << "\tSCMbroadcast: received: " << receive << endl;
      return (scm_from_double(receive));
    }
    else {
      pout() << "\tSCMbroadcast: ERROR: cannot broadcast this type." << endl;
      return (SCM_BOOL_F);
    }
  }
```

This code is cited in chunks 13 and 66.

This code is used in chunk 134.

8.13 Interpolation

Here we implement functions discussed in Section 3.5.4, page 183.

198 ⟨Interpolation 198⟩ ≡

```
SCM SCMinterpolate(SCM scm_name, SCM scm_x, SCM scm_y, SCM scm_z)
{
    /* Return the interpolated value of the scm_name field, which must be a two character string such as
       "Ex", "Hy", or "Bz", at (x, y, z), which are expected to be passed as three Scheme reals—these map
       directly onto C's double s. The interpolation is restricted to level 0 at present. */
  extern Vector<level *> *levels_ptr;
  level *level_0_ptr = (*levels_ptr)[0];
  Vector<Real> &origin = level_0_ptr→origin;
  Real &delta = level_0_ptr→delta;
  DisjointBoxLayout &box_layout = level_0_ptr→box_layout;
  DataIterator &data_iterator = level_0_ptr→data_iterator;
  Vector<Real> position(SpaceDim);
  string name = "AA";
  const int name_length = 2;
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
      /* Here we check if the input paramers, scm_name, scm_x, scm_y, and scm_z are correct. We abort
      returning SCM_BOOL_F if not. */
  if (scm_is_real(scm_x)) {
    position[0] = (Real) scm_to_double(scm_x);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: x = " << position[0] << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: x must be real" << endl;
    return (SCM_BOOL_F);
  }
  if (scm_is_real(scm_y)) {
    position[1] = (Real) scm_to_double(scm_y);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: y = " << position[1] << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: y must be real" << endl;
    return (SCM_BOOL_F);
  }
  if (scm_is_real(scm_z)) {
    position[2] = (Real) scm_to_double(scm_z);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: z = " << position[2] << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: z must be real" << endl;
    return (SCM_BOOL_F);
  }
  if (scm_is_string(scm_name)) {
    char buffer[name_length + 1];
    buffer[name_length] = (char) 0;
    int count = scm_to_locale_stringbuf(scm_name, buffer, name_length);
    if (count ≠ name_length) {
      pout() << "\tSCMinterpolate: ERROR: bad field name length (must be 2)" << endl;
      return (SCM_BOOL_F);
```

```
    }
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: buffer = " << buffer << endl;
    name.replace(0, 2, buffer);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: name = " << name << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: field name must be string" << endl;
    return (SCM_BOOL_F);
  }
        /* Loop over all boxes under this process control. */
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    FArrayBox *field;
    int current_idx, old_idx;
    bool need_old_field = false;
    DataIndex data_index = data_iterator();
        /* We match the name against all fields that we know about. If we find a match, we point field to
        the field and set current_idx to the current index of the field. For magnetic fields, we also set the
        old_idx, because it will be needed for time interpolation. */
    if (name == "Ex") {
      field = &(level_0_ptr->E[data_index][0]);
      current_idx = level_0_ptr->E_idx[0];
    }
    else if (name == "Ey") {
      field = &(level_0_ptr->E[data_index][1]);
      current_idx = level_0_ptr->E_idx[0];
    }
    else if (name == "Ez") {
      field = &(level_0_ptr->E[data_index][2]);
      current_idx = level_0_ptr->E_idx[0];
    }
    else if (name == "Dx") {
      field = &(level_0_ptr->D[data_index][0]);
      current_idx = level_0_ptr->D_idx[0];
    }
    else if (name == "Dy") {
      field = &(level_0_ptr->D[data_index][1]);
      current_idx = level_0_ptr->D_idx[0];
    }
    else if (name == "Dz") {
      field = &(level_0_ptr->D[data_index][2]);
      current_idx = level_0_ptr->D_idx[0];
    }
    else if (name == "Hx") {
      field = &(level_0_ptr->H[data_index][0]);
      current_idx = level_0_ptr->H_idx[0];
      old_idx = level_0_ptr->H_idx[1];
      need_old_field = true;
    }
    else if (name == "Hy") {
      field = &(level_0_ptr->H[data_index][1]);
      current_idx = level_0_ptr->H_idx[0];
      old_idx = level_0_ptr->H_idx[1];
      need_old_field = true;
    }
```

```
else if (name ≡ "Hz") {
  field = &(level_0_ptr→H[data_index][2]);
  current_idx = level_0_ptr→H_idx[0];
  old_idx = level_0_ptr→H_idx[1];
  need_old_field = true;
}
else if (name ≡ "Bx") {
  field = &(level_0_ptr→B[data_index][0]);
  current_idx = level_0_ptr→B_idx[0];
  old_idx = level_0_ptr→B_idx[1];
  need_old_field = true;
}
else if (name ≡ "By") {
  field = &(level_0_ptr→B[data_index][1]);
  current_idx = level_0_ptr→B_idx[0];
  old_idx = level_0_ptr→B_idx[1];
  need_old_field = true;
}
else if (name ≡ "Bz") {
  field = &(level_0_ptr→B[data_index][2]);
  current_idx = level_0_ptr→B_idx[0];
  old_idx = level_0_ptr→B_idx[1];
  need_old_field = true;
}
else {
  pout() ≪ "\tSCMinterpolate:␣ERROR:␣field␣name␣not␣recognized" ≪ endl;
  return (SCM_BOOL_F);
}
```

/* Find the low corner of the cell from which to obtain data. Shrink the field box, so as to drop the ghost regions, and check if the cell's low corner fits in the box. Proceed only, if it does, otherwise go to the next *data_index*. Why do we do all of this here only, and not up-front? This is because we need to know about the type of the box—even to evaluate its low corner—and this we can extract from the specific **FArrayBox** only. */

```
Box full_box = field→box();
IntVect low_corner, box_type = full_box.type();
for (int i = 0; i < SpaceDim; i++)
  low_corner[i] = (int) floor((position[i] − (origin[i] − 0.5 ∗ box_type[i] ∗ delta))/delta);
Box proper_box = grow(full_box, (−1) ∗ N_GHOST_CELLS);
for (int i = 0; i < SpaceDim; i++)
  if (box_type[i]) proper_box.enclosedCells(i);
if (proper_box.contains(low_corner)) {
  Vector⟨IntVect⟩ corner(8);
  Vector⟨Real⟩ f(8);
  Real one_by_delta = 1.0/delta;
  Real one_by_delta_square = one_by_delta ∗ one_by_delta;
  Real one_by_delta_cube = one_by_delta_square ∗ one_by_delta;
  if (watch_scheme > 1)
    pout() ≪ "\tSCMinterpolate:␣found␣my␣cell,␣interpolating..." ≪ endl;
  corner[0] = low_corner;      /* (0,0,0) */
  corner[1] = low_corner + BASISV(2);      /* (0,0,1) */
  corner[2] = low_corner + BASISV(1);      /* (0,1,0) */
  corner[3] = low_corner + BASISV(1) + BASISV(2);      /* (0,1,1) */
  corner[4] = low_corner + BASISV(0);      /* (1,0,0) */
  corner[5] = low_corner + BASISV(0) + BASISV(2);      /* (1,0,1) */
```

```
      corner[6] = low_corner + BASISV(0) + BASISV(1);      /* (1,1,0) */
      corner[7] = low_corner + BASISV(0) + BASISV(1) + BASISV(2);      /* (1,1,1) */
          /* Compare (above and below) with equations (407–414) in Section 3.5.4, page 183. */
      for (int i = 0; i < 8; i++) f[i] = field→get(corner[i], current_idx);
            /* For B and H interpolate in time as well. */
      if (need_old_field)
        for (int i = 0; i < 8; i++) f[i] = 0.5 * (f[i] + field→get(corner[i], old_idx));
              /* Compare with equations (419–426) in Section 3.5.4, page 183. */
      Real c0 = f[0];
      Real cz = (f[1] − f[0]) * one_by_delta;
      Real cy = (f[2] − f[0]) * one_by_delta;
      Real cx = (f[4] − f[0]) * one_by_delta;
      Real cyz = (f[3] − f[2] − f[1] + f[0]) * one_by_delta_square;
      Real czx = (f[5] − f[4] − f[1] + f[0]) * one_by_delta_square;
      Real cxy = (f[6] − f[4] − f[2] + f[0]) * one_by_delta_square;
      Real cxyz = (f[7] + f[1] + f[2] − f[3] + f[4] − f[5] − f[6] − f[0]) * one_by_delta_cube;
      Real x0 = origin[0] + (low_corner[0] − 0.5 * box_type[0]) * delta;
      Real y0 = origin[1] + (low_corner[1] − 0.5 * box_type[1]) * delta;
      Real z0 = origin[2] + (low_corner[2] − 0.5 * box_type[2]) * delta;
      Real dx = position[0] − x0;
      Real dy = position[1] − y0;
      Real dz = position[2] − z0;
          /* Compare with equation 406 in Section 3.5.4, page 183. */
      Real f_int = cxyz * dx * dy * dz + cyz * dy * dz + czx * dz * dx + cxy * dx * dy + cx * dx + cy * dy + cz * dz + c0;
      if (watch_scheme > 1) pout() << "\tSCMinterpolate:␣returning␣" << f_int << endl;
      return (scm_from_double((double) f_int));
    }     /* if (proper_box.contains(low_corner)) */
  }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  if (watch_scheme > 1) pout() << "\tSCMinterpolate:␣cell␣not␣found,␣returning␣zero" << endl;
          /* Perhaps we should return SCM_BOOL_F instead and leave it to the caller to sort things out? But
          then, how will the caller know why it's SCM_BOOL_F? Some other marker is needed here. */
  return (scm_from_double((double) 0.0));
}
```

This code is cited in chunks 13 and 41.

This code is used in chunk 134.

8.14 Initialize Scheme

Here we condition the Scheme process on its start up. This is done *after* the input file is read.

We begin by registering the C++ implemented Scheme procedures with Scheme. This, as was pointed out in Section 5, page 193, module ⟨ Inner Main 73 ⟩, must be done at the beginning of the program, so that the procedures become available to Scheme before any user code is processed.

Also, on this occasion we create certain bindings and initialize certain variables—especially vectors. These can be used in lambda expressions that will be provided by users.

We *define* primitive subroutines, written in C++, by calling *scm_c_define_gsubr*. The arguments to *scm_c_define_gsubr* are as follows:

`"draw-box"` This is the Scheme name of the function.

$2, 2, 0$ Here we tell Scheme that the function has two required arguments, two optional arguments, and zero *rest* arguments.

$(\mathbf{SCM}(*)(\,))$*draw_box* This is the C++ name of the function, and it is also *cast* here on what *scm_c_define_gsubr* expects in this slot.

In the "Tutorial Introduction to Guile" by David Drysdale [6] *gh_new_procedure* from the currently deprecated GH toolkit was used to register C-functions with Scheme. The Guile-1.8.1 documentation recommends using *scm_c_define_gsubr* instead, but the example that is provided does not work. Luckily, Michael Gran shows how to do this in [9]. It turns out that an explicit cast to $(\mathbf{SCM}(*)(\,))$ is needed in front of the function C-name. The compiler complains otherwise.

199 ⟨ Initialize Scheme 199 ⟩ ≡

```
    /* global variables */
Real nx, ny, nz;
int normalized;
bool have_post_iteration_lambda, have_post_all_lambda;
SCM ex_lambda, ey_lambda, ez_lambda, e_lambda, post_iteration_lambda, post_all_lambda;
SCM E_var, E_old_var, D_var, D_old_var, S_var, S_var_ref;
SCM medium_var, direction_var, t_e_var, dt_var;
SCM iterate_count_var, time_to_dump_var;
int initialize_scheme()
{
    /* To be run before the first instantiation of SCMParmParse class, which reads the input file. */
  int return_status = EXIT_SUCCESS;
      /* Register new Scheme functions */
      /* ... Functions for drawing on media and tag arrays */
  scm_c_define_gsubr("draw-box", 2, 1, 0, (SCM(*)())draw_box);
  scm_c_define_gsubr("draw-ball", 2, 1, 0, (SCM(*)())draw_ball);
  scm_c_define_gsubr("draw-ellipsoid", 3, 1, 0, (SCM(*)())draw_ellipsoid);
  scm_c_define_gsubr("draw-disk", 4, 1, 0, (SCM(*)())draw_disk);
  scm_c_define_gsubr("draw-ary-ellipsoid", 7, 1, 0, (SCM(*)())draw_ary_ellipsoid);
  scm_c_define_gsubr("draw-cylinder", 3, 1, 0, (SCM(*)())draw_cylinder);
  scm_c_define_gsubr("draw-cone", 3, 1, 0, (SCM(*)())draw_cone);
  scm_c_define_gsubr("draw-parallelepiped", 4, 1, 0, (SCM(*)())draw_parallelepiped);
  scm_c_define_gsubr("draw-torus", 4, 1, 0, (SCM(*)())draw_torus);
  scm_c_define_gsubr("draw-lens", 4, 1, 0, (SCM(*)())draw_lens);
      /* ... Functions for MPI operations */
  scm_c_define_gsubr("num-proc", 0, 0, 0, (SCM(*)())SCMnumProc);
  scm_c_define_gsubr("proc-id", 0, 0, 0, (SCM(*)())SCMprocID);
  scm_c_define_gsubr("barrier", 0, 0, 0, (SCM(*)())SCMbarrier);
  scm_c_define_gsubr("unique-proc", 0, 0, 0, (SCM(*)())SCMuniqueProc);
  scm_c_define_gsubr("gather", 3, 0, 0, (SCM(*)())SCMgather);
```

```
  scm_c_define_gsubr("broadcast", 2, 0, 0, (SCM(*)( ))SCMbroadcast);
      /* ... Functions to access field data */
  scm_c_define_gsubr("interpolate", 4, 0, 0, (SCM(*)( ))SCMinterpolate);
      /* End of function registration section */
      /* Define some Scheme symbols to be used by lambdas. This part used to live inside what is
      now called post_initialize_scheme( ), but we want it up-front, so that the symbols are available to
      user defined pre-iteration operations. Pointers to the associated variables are extracted in the
      post_initialize_scheme( ) below, because SCMParmParse::variable function is needed to do this. */
  scm_c_define("forms.spacedim", scm_from_int(SpaceDim));
  scm_c_eval_string("(define␣forms.E␣0.0)\n␣␣␣␣␣␣␣(define␣forms.E_old␣0.0)\n␣␣␣␣\
      ␣␣␣(define␣forms.D␣0.0)\n␣␣␣␣␣␣␣(define␣forms.D_old␣0.0)\n␣␣␣␣␣␣␣(define␣forms.m\
      edium␣0)\n␣␣␣␣␣␣␣(define␣forms.direction␣0)\n␣␣␣␣␣␣␣(define␣forms.time_e␣0.0)\n␣\
      ␣␣␣␣␣␣(define␣forms.iterate.count␣0)\n␣␣␣␣␣␣␣(define␣forms.iterate.time_to_dump␣\
      0)\n␣␣␣␣␣␣␣(define␣forms.dt␣0.1)\n");
  return return_status;
}
int post_initialize_scheme( )
{
  SCMParmParse forms_parser("forms");
  SCMParmParse watch_parser("forms.watch");
  int watch_initialize_scheme = (int) false;
  int return_status = EXIT_SUCCESS;
  watch_parser.query("initialize_scheme", watch_initialize_scheme);
  if (watch_initialize_scheme) {
    pout( ) << "initialize_scheme:" << endl;
    pout( ) << "\tloading␣C++␣functions" << endl;
  }
  if (watch_initialize_scheme) pout( ) << "\tdefining␣parameters" << endl;
  if (SpaceDim ≠ 3) {
    pout( ) << "\tERROR(initialize_scheme):␣SpaceDim␣=/=␣3" << endl;
    return_status = ERR_INITIALIZE_SCHEME;
    goto exit;
  }
  else {
      /* Define Scheme variables to be used by C-lambdas. This part used to live inside one of the if
        statements below, but it's needed globally. The symbols themselves are now defined up-front before
        the first SCMParmParse is invoked to read the input file. */
    E_var = forms_parser.variable("E");
    E_old_var = forms_parser.variable("E_old");
    D_var = forms_parser.variable("D");
    D_old_var = forms_parser.variable("D_old");
    medium_var = forms_parser.variable("medium");
    direction_var = forms_parser.variable("direction");
    t_e_var = forms_parser.variable("time_e");
    dt_var = forms_parser.variable("dt");
    iterate_count_var = forms_parser.variable("iterate.count");
    time_to_dump_var = forms_parser.variable("iterate.time_to_dump");
        /* Read the signal group */
    if (watch_initialize_scheme) pout( ) << "\treading␣the␣signal␣group␣..." << flush;
          /* Read n and transfer the data to a global variable. */
    SCMParmParse signal_parameter_parser("forms.signal");
    Vector<Real> n(SpaceDim);
    if (¬signal_parameter_parser.queryarr("direction", n, 0, SpaceDim)) {
      pout( ) << "\n\tERROR(initialize_scheme):␣no␣signal␣propagation␣direction" << endl;
```

```
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
    /* have acquired n */
  Real length = sqrt(n[0] * n[0] + n[1] * n[1] + n[2] * n[2]);
  if (length > 0.0) {
    extern Real nx, ny, nz;
    nx = n[0]/length;
    ny = n[1]/length;
    nz = n[2]/length;
    if (watch_initialize_scheme) pout() << " direction," << flush;
  }
  else {
    pout() << "\n\tERROR(initialize_scheme): zero direction" << endl;
    return_status = ERR_INITIALIZE_SCHEME;
    goto exit;
  }
}
if (¬signal_parameter_parser.query("normalized", normalized)) {
  pout() << "\n\tWARNING(initialize_scheme): signal not normalized" << endl;
  normalized = (int) false;
}
else if (¬normalized) {
  pout() << "\n\tWARNING(initialize_scheme): signal not normalized" << endl;
}
else {
  if (watch_initialize_scheme) pout() << " normalization," << flush;
}
if (¬signal_parameter_parser.isProcedure("ex_lambda")) {
  pout() << "\n\tERROR(initialize_scheme): ex_lambda not provided" << endl;
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
  ex_lambda = scm_variable_ref(signal_parameter_parser.variable("ex_lambda"));
  if (watch_initialize_scheme) pout() << " ex_lambda," << flush;
}
if (¬signal_parameter_parser.isProcedure("ey_lambda")) {
  pout() << "\n\tERROR(initialize_scheme): ey_lambda not provided" << endl;
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
  ey_lambda = scm_variable_ref(signal_parameter_parser.variable("ey_lambda"));
  if (watch_initialize_scheme) pout() << " ey_lambda," << flush;
}
if (¬signal_parameter_parser.isProcedure("ez_lambda")) {
  pout() << "\n\tERROR(initialize_scheme): ez_lambda not provided" << endl;
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
  ez_lambda = scm_variable_ref(signal_parameter_parser.variable("ez_lambda"));
```

```
  if (watch_initialize_scheme) pout() << " ez_lambda" << endl << flush;
}
        /* Read the media group */
if (watch_initialize_scheme) pout() << "\treading the media group ..." << flush;
SCMParmParse media_parameter_parser("forms.media");
int number_of_auxiliary_fields = 0;
if (¬media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields)) {
  pout() << "\n\tWARNING(initialize_scheme): number_of_auxiliary_fields not defined" <<
      endl;
}
else {
  if (watch_initialize_scheme) pout() << " number_of_auxiliary_fields" << flush;
}
if (¬media_parameter_parser.isProcedure("distribution.lambda")) {
  pout() << "\n\tWARNING(initialize_scheme): distribution lambda not provided" << endl;
}
else {
  if (watch_initialize_scheme) pout() << " distribution.lambda" << flush;
  if (¬media_parameter_parser.isProcedure("model.e_lambda")) {
    pout() << "\n\tERROR(initialize_scheme): model.e_lambda not provided" << endl;
    return_status = ERR_INITIALIZE_SCHEME;
    goto exit;
  }
  else {
    e_lambda = scm_variable_ref(media_parameter_parser.variable("model.e_lambda"));
    if (watch_initialize_scheme) pout() << " model.e_lambda" << flush;
    if (number_of_auxiliary_fields) {
      char command[BUFSIZ];
      sprintf(command, "(define forms.S (make-f64vector %d 0.0))\n",
          number_of_auxiliary_fields);
      scm_c_eval_string(command);
      S_var = forms_parser.variable("S");
      S_var_ref = scm_variable_ref(S_var);
    }
  }     /* if (¬media_parameter_parser.isProcedure("model.e_lambda")) else clause */
}     /* if (¬media_parameter_parser.isProcedure("distribution.lambda")) else clause */
        /* Read the post group */
if (watch_initialize_scheme) pout() << "\treading the post group ..." << endl;
SCMParmParse post_parser("forms.post");
if (¬post_parser.isProcedure("iteration.lambda")) {
  pout() << "\n\tWARNING(initialize_scheme): post-iteration lambda not provided" << endl;
  have_post_iteration_lambda = false;
}
else {
  post_iteration_lambda = scm_variable_ref(post_parser.variable("iteration.lambda"));
  have_post_iteration_lambda = true;
}     /* if (¬post_parser.isProcedure("iteration.lambda")) else clause */
if (¬post_parser.isProcedure("all.lambda")) {
  pout() << "\n\tWARNING(initialize_scheme): post-all lambda not provided" << endl;
  have_post_all_lambda = false;
}
else {
  post_all_lambda = scm_variable_ref(post_parser.variable("all.lambda"));
  have_post_all_lambda = true;
```

```
    }    /* if (¬post_parser.isProcedure("all.lambda")) else clause */
  }    /* if (SpaceDim ≠ 3) { } else */
exit:
  if (watch_initialize_scheme) pout() << "initialize_scheme:␣returning␣to␣caller" << endl;
  return return_status;
}
```

This code is cited in chunks 22 and 66.

This code is used in chunk 134.

9 IO Functions

This section describes dumping of HDF5 images for processing with `Chombovis` or `VisIt`, see section 2.3, page 35, for comments about data postprocessing and use of these tools.

The primary output tool here is a Chombo function *WriteAMRHierarchyHDF5*. The function, defined on `NodeAMRIO.H`, takes the following twelve arguments:

file_name The full name of the output file. In our case, the name is made of a root string bound by the user to forms.output.filename.root, and a dump number, which is padded with zeros up to a number of digits that's bound by the user to forms.output.filename.number_of_digits.

vector_of_box_layouts This is a **Vector** of **DisjointBoxLayout**s one for each level in the AMR (or just multigrid) hierarchy.

vector_of_ptrs_to_level_data This is the data that is being dumped. It is a **Vector**, each element of which is a *pointer* to **LevelData** associated with the corresponding **DisjointBoxLayout** in *vector_of_box_layouts*. **LevelData** can normally be a collection of **BaseFab**⟨**int**⟩s, as *tags* are in this program, or a collection of **FluxBox**es, as $\boldsymbol{H}$ and $\boldsymbol{B}$ are in this program. But *WriteAMRHierarchyHDF5* expects **LevelData**⟨**FArrayBox**⟩. So, we cannot dump $\boldsymbol{B}$ or $\boldsymbol{E}$ (which is **LevelData**⟨**CurlBox**⟩) directly. **FArrayBox** attaches an array of **Real**s at the center of each cell, so we need to interpolate our edge and face centered fields onto cell centers for every box, and then *assemble* them into arrays of output fields values at every cell center in such order as the user requires.

More about this below.

vector_of_outputs This is a vector of strings. For each field that is going to be accumulated in the **LevelData**⟨**FArrayBox**⟩ for a given level, pointed to by an entry in *vector_of_ptrs_to_level_data*, we need to provide a string that describes it. For example, if the **Real** slots in the **FArrayBox** associated with each box of the **DisjointBoxLayout** are E_x, E_y and E_z, in this order, then the entries in the *vector_of_outputs* should be `"Ex"`, `"Ey"`, and `"Ez"`. When the data is written on the HDF5 files, it will be labeled so.

domain This is the zero level computational domain of the grid structure, i.e., the level zero box that encapsulates everything.

delta This is the grid constant of level zero. It is assumed to be the same in all directions.

dt This is the time step of level zero.

t This is the time that the system attains at the point of dump at level zero.

vector_of_refinements This is a vector of refinement ratios between the levels of the multigrid system.

number_of_levels Number of levels in the multigrid.

Let us return now to the discussion of how the dump data is produced. The user defines two arrays of field names on the input file, forms.output.precompute and forms.output.**write**. The latter specifies fields to be written out. Some of the fields to be written out may require precomputation of other fields, which we may not wish to dump. So, we need to specify all fields that have to be precomputed, in the correct order, on forms.output.precompute, and then select the ones for output on forms.output.**write**.

For example, let us suppose that we want to output `"Pz"`, that is $P_z = E_x H_y - E_y H_x$. It requires precomputation of E_x, E_y, H_x and H_y, the precomputation in this case is just interpolation onto cell centers, and then the precomputation of P_z itself. So we define

(**define** forms.output.precompute '("Ex" "Ey" "Hx" "Hy" "Pz"))

Then, to actually write P_z out, we define

(**define** forms.output.**write** '("Pz"))

The code knows about the following hardwired field names: "Ex", "Ey", "Ez", "Dx", "Dy", "Dz", "Bx", "By", "Bz", "Hx", "Hy", "Hz", "En", "Px", "Py", and "Pz". "En" is energy density. Eventually we want to let users specify what to output by providing an arbitrary lambda. For the time being though energy density and Poynting flux are predefined.

The **LevelData**⟨**FArrayBox**⟩ fields that are used to hold precomputed data and then data for writing are a part of the **level** structure, defined on `Forms.H`, and are called *Out* and *OutWrite*. Like all other field structures, they're allocated in `Initialize.cpp` by function *fill_levels*, see Section 6.2, page 213, module ⟨ Function *fill_levels* 94 ⟩:

```
levels[n]->Out.define(level_box_layout, number_of_precompute_fields,
IntVect::Zero);
levels[n]->OutWrite.define(level_box_layout, number_of_write_fields,
IntVect::Zero);
```

Here *number_of_precomputed_fields* and *number_of_write_fields* is obtained by interrogating the Guile namespace with **SCMParmParse**, as follows:

```
number_of_write_fields = output_parameter_parser.countval("write");
number_of_precompute_fields = output_parameter_parser.countval("precompute");
```

Before function *dump_data* gets to call *WriteAMRHierarchyHDF5*, we

1. loop over the levels of the grid hierarchy. For each level, we
   (a) loop over the boxes of the level. And for each box we
      i. loop over the entries in the list of precomputes in the order provided. If the field to be precomputed is E_i or D_i we call Fortran subroutine *from_edge_to_center_* to do the job. For B_i or H_i we call Fortran subroutine *from_face_to_center_* to carry out the interpolation. These are defined on `output.f` and listed in Section 13.8, page 496. The order in which the fields appear is remembered on *Ex_idx* through *Pz_idx*, all of which are initialized to -1, which remains so if the fields have not been precomputed. The code that computes `"En"`, `"Px"`, `"Py"` and `"Pz"` uses these indexes first to ensure that all required fields have been precomputed, and then to select them for its own computation. Having precomputed the fields we
      ii. loop over the entries in the list of outputs. For each entry we find its index in the precomputed array and call Fortran subroutine *output_field_copy_*, to copy its **FArrayBox**es from precomputes to the appropriate slot in outputs.

Thus having dealt with all precompute and output fields for each box, and with all boxes for each level, and with all levels for the hierarchy, we will have fully constructed the data to which we then point entries in *vector_of_ptrs_to_level_data*.

The rest is reporting, when requested, and... *writeAMRHierarchyHDF5*.

200 ⟨ `IO.c` 200 ⟩ ≡

```
#include "Forms.H"
  ⟨ Function dump_data 201 ⟩
```

9.1 Dump Data

201 ⟨Function *dump_data* 201⟩ ≡

```
int dump_data(Vector⟨level *⟩ &levels, int label)
{
  char file_name[BUFSIZ];
  char format[BUFSIZ];
  string root;
  int number_of_precomputes = 0;
  int number_of_outputs = 0;
  int number_of_digits = 0;
  int number_of_levels = levels.size();
  int return_status = EXIT_SUCCESS;
  int watch_dump_data = (int) false;
  int report_outliers = (int) false;
  int Ex_idx = −1, Ey_idx = −1, Ez_idx = −1, Dx_idx = −1, Dy_idx = −1, Dz_idx = −1, Hx_idx = −1,
      Hy_idx = −1, Hz_idx = −1, Bx_idx = −1, By_idx = −1, Bz_idx = −1, En_idx = −1, Px_idx = −1,
      Py_idx = −1, Pz_idx = −1, eMdx_idx = −1, eMdy_idx = −1, eMdz_idx = −1, hMdx_idx = −1,
      hMdy_idx = −1, hMdz_idx = −1;
  Vector⟨string⟩ vector_of_precomputes, vector_of_outputs;
  Vector⟨DisjointBoxLayout⟩ vector_of_box_layouts;
  Vector⟨LevelData⟨FArrayBox⟩ *⟩ vector_of_ptrs_to_level_data;
  Vector⟨int⟩ vector_of_refinements;
  vector_of_precomputes.clear();
  vector_of_outputs.clear();
  vector_of_box_layouts.clear();
  vector_of_ptrs_to_level_data.clear();
  vector_of_refinements.clear();
  SCMParmParse output_parameter_parser("forms.output");
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("dump_data", watch_dump_data);
  if (watch_dump_data) pout() ≪ "\t\tdump_data:" ≪ endl;
  output_parameter_parser.query("report_outliers", report_outliers);
  if ((number_of_precomputes = output_parameter_parser.countval("precompute")) ≤ 0) {
    if (watch_dump_data) pout() ≪ "\t\t\tnothing␣to␣precompute,␣exiting..." ≪ endl;
    goto exit;
  }
  if ((number_of_outputs = output_parameter_parser.countval("write")) ≤ 0) {
    if (watch_dump_data) pout() ≪ "\t\t\tnothing␣to␣write,␣exiting..." ≪ endl;
    goto exit;
  }
        /* If we're here, it means that we have something to output and that we have something to
        precompute. */
  output_parameter_parser.getarr("precompute", vector_of_precomputes, 0, number_of_precomputes);
  output_parameter_parser.getarr("write", vector_of_outputs, 0, number_of_outputs);
  if (¬output_parameter_parser.query("filename.root", root)) root = FORMS_OUTPUT;
  if (¬output_parameter_parser.query("filename.number_of_digits", number_of_digits))
    number_of_digits = OUTPUT_NUMBER_OF_DIGITS;
  sprintf(format, "%%s_%%%d.%dd.hdf5", number_of_digits, number_of_digits);
  sprintf(file_name, format, root.c_str(), label);
  if (watch_dump_data) pout() ≪ "\t\t\tprecomputing␣" ≪ vector_of_precomputes ≪ endl;
        /* Now we precompute the fields specified */
  for (int n = 0; n < number_of_levels; n++) {
    DataIterator &data_iterator = levels[n]→data_iterator;
    int &E_idx = levels[n]→E_idx[0];
```

```
    int &D_idx = levels[n]→D_idx[0];
    int &B_idx = levels[n]→B_idx[0];
    int &H_idx = levels[n]→H_idx[0];
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      CurlBox &E = levels[n]→E[data_index];
      CurlBox &D = levels[n]→D[data_index];
      FluxBox &B = levels[n]→B[data_index];
      FluxBox &H = levels[n]→H[data_index];
      EdgeFab⟨int⟩ &medium_E = levels[n]→medium_E[data_index];
      FaceFab⟨int⟩ &medium_H = levels[n]→medium_H[data_index];
      FArrayBox &Out = levels[n]→Out[data_index];
      FArrayBox &OutWrite = levels[n]→OutWrite[data_index];
      if (watch_dump_data) {
        pout() << "\t\t\tbox:␣" << Out.box() << endl;
        pout() << "\t\t\t\t" << flush;
      }
      for (int count = 0; count < number_of_precomputes; count++) {
        string field = vector_of_precomputes[count];
        if (watch_dump_data) pout() << field << "␣" << flush;
        ⟨ dump_data: precompute field 203 ⟩
      }
      if (watch_dump_data) pout() << endl;
            /* Alas, this is not the end of the story. Having precomputed the fields, we now must select the
            ones for writing and assemble them in OutWrite. It should not be necessary to do this with
            copy, but right now this is how I do it, because WriteAMRHierarchyHDF5 can only output the
            whole LevelData and not specified components of it. I will have to write my own version of it
            to optimize this part of the code. */
      if (watch_dump_data) {
        pout() << "\t\t\t\tAssembling␣fields␣for␣writing␣out" << endl;
        pout() << "\t\t\t\t" << flush;
      }
      for (int count = 0; count < number_of_outputs; count++) {
        string field = vector_of_outputs[count];
        if (watch_dump_data) pout() << field << "␣" << flush;
        ⟨ dump_data: extract field from precomputes 210 ⟩
      }
      if (watch_dump_data) pout() << endl;
    }
    vector_of_box_layouts.push_back(levels[n]→box_layout);
    vector_of_ptrs_to_level_data.push_back(&(levels[n]→OutWrite));
    vector_of_refinements.push_back(REFINE_RATIO);
  }
  if (watch_dump_data) {
    pout() << "\t\t\tWriting␣" << vector_of_outputs << "on␣" << file_name << endl;
    pout() << "\t\t\t\trefinements␣␣" << vector_of_refinements << endl;
    pout() << "\t\t\t\tdomain␣␣␣␣␣␣␣" << levels[0]→domain << endl;
    pout() << "\t\t\t\tdelta␣␣␣␣␣␣␣␣" << levels[0]→delta << endl;
    pout() << "\t\t\t\tdt␣␣␣␣␣␣␣␣␣␣" << levels[0]→dt << endl;
    pout() << "\t\t\t\ttime_e␣␣␣␣␣␣" << levels[0]→time_e << endl;
    pout() << "\t\t\t\tlevels␣␣␣␣␣␣␣" << number_of_levels << endl;
    pout() << "\t\t\t\t\tlayouts␣␣␣␣" << vector_of_box_layouts.size() << endl;
    pout() << "\t\t\t\t\tdatasets␣␣␣" << vector_of_ptrs_to_level_data.size() << endl;
    pout() << "\t\t\t\t\tcomponents␣" << vector_of_ptrs_to_level_data[0]→nComp() << endl;
```

```
    }
    if (report_outliers) {
      pout() << "\t\t\t\toutliers:" << endl;
      for (int level_number = 0; level_number < number_of_levels; level_number++) {
        LevelData<FArrayBox> &OutWrite = *(vector_of_ptrs_to_level_data[level_number]);
        DataIterator &data_iterator = levels[level_number]->data_iterator;
        pout() << "\t\t\t\t\tlevel␣" << level_number << endl;
        for (int count = 0; count < number_of_outputs; count++) {
          pout() << "\t\t\t\t\t\t" << vector_of_outputs[count] << "␣" << flush;
          Real minimum = 0.0, maximum = 0.0;
          IntVect minimum_location = IntVect::Zero, maximum_location = IntVect::Zero;
          for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
            FArrayBox &field = OutWrite[data_iterator()];
            Real local_minimum, local_maximum;
            IntVect local_minimum_location, local_maximum_location;
            local_minimum = field.min(count);
            local_minimum_location = field.minIndex(count);
            local_maximum = field.max(count);
            local_maximum_location = field.maxIndex(count);
            if (local_minimum < minimum) {
              minimum = local_minimum;
              minimum_location = local_minimum_location;
            }
            if (local_maximum > maximum) {
              maximum = local_maximum;
              maximum_location = local_maximum_location;
            }
          }
          pout() << minimum << "␣at␣" << minimum_location << ",␣" << maximum << "␣at␣" <<
              maximum_location << endl;
        }
      }
    }
    WriteAMRHierarchyHDF5(file_name, vector_of_box_layouts, vector_of_ptrs_to_level_data, vector_of_outputs,
        levels[0]->domain, levels[0]->delta, levels[0]->dt, levels[0]->time_e, vector_of_refinements, number_of_levels);
    if (watch_dump_data) pout() << "\t\t\t...␣done." << endl;
  exit: return return_status;
  }
```

This code is cited in chunk 73.

This code is used in chunk 200.

9.2 Precompute Fields for Output

9.2.1 Space Interpolate Electric Fields

203 ⟨ *dump_data*: precompute field 203 ⟩ ≡

```
if ((field ≡ "Ex") ∨ (field ≡ "Ey") ∨ (field ≡ "Ez")) {
  int dir;
  if (field ≡ "Ex") {
    dir = 0;
    Ex_idx = count;
  }
  else if (field ≡ "Ey") {
    dir = 1;
    Ey_idx = count;
  }
  else {
    dir = 2;
    Ez_idx = count;
  }
  from_edge_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &E_idx, &(Out.loVect()[0]),
      &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]), &(Out.hiVect()[2]),
      Out.dataPtr(), &(E[dir].loVect()[0]), &(E[dir].loVect()[1]), &(E[dir].loVect()[2]),
      &(E[dir].hiVect()[0]), &(E[dir].hiVect()[1]), &(E[dir].hiVect()[2]), E[dir].dataPtr(),
      &watch_dump_data, &return_status);
  if (return_status ≠ EXIT_SUCCESS) {
    pout() ≪ "\n\t\t\tERROR(dump_data):␣from_edge_to_center_␣returned␣" ≪ return_status ≪ endl;
    goto exit;
  }
}
else if ((field ≡ "Dx") ∨ (field ≡ "Dy") ∨ (field ≡ "Dz")) {
  int dir;
  if (field ≡ "Dx") {
    dir = 0;
    Dx_idx = count;
  }
  else if (field ≡ "Dy") {
    dir = 1;
    Dy_idx = count;
  }
  else {
    dir = 2;
    Dz_idx = count;
  }
  from_edge_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &D_idx, &(Out.loVect()[0]),
      &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]), &(Out.hiVect()[2]),
      Out.dataPtr(), &(D[dir].loVect()[0]), &(D[dir].loVect()[1]), &(D[dir].loVect()[2]),
      &(D[dir].hiVect()[0]), &(D[dir].hiVect()[1]), &(D[dir].hiVect()[2]), D[dir].dataPtr(),
      &watch_dump_data, &return_status);
  if (return_status ≠ EXIT_SUCCESS) {
    pout() ≪ "\n\t\t\tERROR(dump_data):␣from_edge_to_center_␣returned␣" ≪ return_status ≪ endl;
    goto exit;
  }
}
```

See also chunks 204, 205, 206, 207, 208, and 209.

This code is used in chunk 201.

9.2.2 Space and Time Interpolate Magnetic Fields

```
204 ⟨ dump_data: precompute field 203 ⟩ +≡
  else
    if ((field ≡ "Bx") ∨ (field ≡ "By") ∨ (field ≡ "Bz")) {
      int dir;
      if (field ≡ "Bx") {
        dir = 0;
        Bx_idx = count;
      }
      else if (field ≡ "By") {
        dir = 1;
        By_idx = count;
      }
      else {
        dir = 2;
        Bz_idx = count;
      }
      from_face_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &B_idx, &(Out.loVect()[0]),
          &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
          &(Out.hiVect()[2]), Out.dataPtr(), &(B[dir].loVect()[0]), &(B[dir].loVect()[1]),
          &(B[dir].loVect()[2]), &(B[dir].hiVect()[0]), &(B[dir].hiVect()[1]), &(B[dir].hiVect()[2]),
          B[dir].dataPtr(), &watch_dump_data, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() ≪ "\n\t\t\tERROR(dump_data):␣from_face_to_center_␣returned␣" ≪ return_status ≪
            endl;
        goto exit;
      }
    }
    else if ((field ≡ "Hx") ∨ (field ≡ "Hy") ∨ (field ≡ "Hz")) {
      int dir;
      if (field ≡ "Hx") {
        dir = 0;
        Hx_idx = count;
      }
      else if (field ≡ "Hy") {
        dir = 1;
        Hy_idx = count;
      }
      else {
        dir = 2;
        Hz_idx = count;
      }
      from_face_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &H_idx, &(Out.loVect()[0]),
          &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
          &(Out.hiVect()[2]), Out.dataPtr(), &(H[dir].loVect()[0]), &(H[dir].loVect()[1]),
          &(H[dir].loVect()[2]), &(H[dir].hiVect()[0]), &(H[dir].hiVect()[1]), &(H[dir].hiVect()[2]),
          H[dir].dataPtr(), &watch_dump_data, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() ≪ "\n\t\t\tERROR(dump_data):␣from_face_to_center_␣returned␣" ≪ return_status ≪
            endl;
        goto exit;
      }
    }
```

9.2.3 Evaluate Energy

205 ⟨ *dump_data*: precompute field 203 ⟩ +≡

```
else
  if (field ≡ "En") {
    En_idx = count;
        /* For this field we need to have E and B available. */
    if ((Ex_idx ≡ −1) ∨ (Ey_idx ≡ −1) ∨ (Ez_idx ≡ −1) ∨ (Bx_idx ≡ −1) ∨ (By_idx ≡ −1) ∨ (Bz_idx ≡ −1)) {
      pout( ) ≪ "\n\t\t\tERROR(dump_data): precompute E and B before Energy" ≪ endl;
      return_status = ERR_DUMP_DATA;
      goto exit;
    }
    evaluate_energy_(&SpaceDim, &number_of_precomputes, &count, &Ex_idx, &Ey_idx, &Ez_idx, &Bx_idx,
        &By_idx, &Bz_idx, &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
        &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
    if (return_status ≠ EXIT_SUCCESS) {
      pout( ) ≪ "\n\t\t\tERROR(dump_data): evaluate_energy_ returned " ≪ return_status ≪ endl;
      goto exit;
    }
  }
```

9.2.4 Evaluate Poynting Vector

206 ⟨ *dump_data*: precompute field 203 ⟩ +≡

```
else
  if ((field ≡ "Px") ∨ (field ≡ "Py") ∨ (field ≡ "Pz")) {
    int dir;
    if (field ≡ "Px") {
      dir = 0;
      Px_idx = count;
      if ((Ey_idx ≡ −1) ∨ (Hz_idx ≡ −1) ∨ (Ez_idx ≡ −1) ∨ (Hy_idx ≡ −1)) {
        pout( ) ≪ "\n\t\t\tERROR(dump_data): precompute Ey, Hz, Ez and Hy" ≪ " before Px" ≪
            endl;
        return_status = ERR_DUMP_DATA;
        goto exit;
      }
      evaluate_flow_(&SpaceDim, &number_of_precomputes, &count, &Ey_idx, &Hz_idx, &Ez_idx, &Hy_idx,
          &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
          &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
    }
    else if (field ≡ "Py") {
      dir = 1;
      Py_idx = count;
      if ((Ez_idx ≡ −1) ∨ (Hx_idx ≡ −1) ∨ (Ex_idx ≡ −1) ∨ (Hz_idx ≡ −1)) {
        pout( ) ≪ "\n\t\t\tERROR(dump_data): precompute Ez, Hx, Ex and Hz" ≪ " before Py" ≪
            endl;
        return_status = ERR_DUMP_DATA;
        goto exit;
      }
      evaluate_flow_(&SpaceDim, &number_of_precomputes, &count, &Ez_idx, &Hx_idx, &Ex_idx, &Hz_idx,
          &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
          &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
    }
    else {
```

```
      dir = 2;
      Pz_idx = count;
      if ((Ex_idx ≡ −1) ∨ (Hy_idx ≡ −1) ∨ (Ey_idx ≡ −1) ∨ (Hx_idx ≡ −1)) {
        pout() ≪ "\n\t\t\tERROR(dump_data): precompute Ex, Hy, Ey and Hx" ≪ " before Pz" ≪
            endl;
        return_status = ERR_DUMP_DATA;
        goto exit;
      }
      evaluate_flow_(&SpaceDim, &number_of_precomputes, &count, &Ex_idx, &Hy_idx, &Ey_idx, &Hx_idx,
          &(Out.loVect()[0]), &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]),
          &(Out.hiVect()[1]), &(Out.hiVect()[2]), Out.dataPtr(), &watch_dump_data, &return_status);
    }
    if (return_status ≠ EXIT_SUCCESS) {
      pout() ≪ "\n\t\t\tERROR(dump_data): evaluate_flow_ returned " ≪ return_status ≪ endl;
      goto exit;
    }
  }
```

9.2.5 Space Interpolate Electric Media Distribution

207 ⟨ *dump_data*: precompute field 203 ⟩ +≡

```
else
  if ((field ≡ "eMdx") ∨ (field ≡ "eMdy") ∨ (field ≡ "eMdz")) {
    int dir;
    if (field ≡ "eMdx") {
      dir = 0;
      eMdx_idx = count;
    }
    else if (field ≡ "eMdy") {
      dir = 1;
      eMdy_idx = count;
    }
    else {
      dir = 2;
      eMdz_idx = count;
    }
    from_int_edge_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count,
        /* There is no E_idx here, because we do not have an "old" and a "new" medium field. There's just
        one that's the same all the time. */
        /* This is what we are going to write on. */
    &(Out.loVect()[0]), &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
        &(Out.hiVect()[2]), Out.dataPtr(),
        /* This is what we are going to read from. */
    &(medium_E[dir].loVect()[0]), &(medium_E[dir].loVect()[1]), &(medium_E[dir].loVect()[2]),
        &(medium_E[dir].hiVect()[0]), &(medium_E[dir].hiVect()[1]), &(medium_E[dir].hiVect()[2]),
        medium_E[dir].dataPtr(),
        /* Verbosity and return. */
    &watch_dump_data, &return_status);
  }    /* else if ( (field ≡ "eMdx") ... */
```

9.2.6 Space Interpolate Magnetic Media Distribution

208 ⟨ *dump_data*: precompute field 203 ⟩ +≡

```
else
  if ((field ≡ "hMdx") ∨ (field ≡ "hMdy") ∨ (field ≡ "hMdz")) {
    int dir;
    if (field ≡ "hMdx") {
      dir = 0;
      hMdx_idx = count;
    }
    else if (field ≡ "hMdy") {
      dir = 1;
      hMdy_idx = count;
    }
    else {
      dir = 2;
      hMdz_idx = count;
    }
    from_int_face_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count,
        /* There is no B_idx here, because we do not have an "old" and a "new" medium field. There's just
        one that's the same all the time. */
        /* This is what we are going to write on. */
    &(Out.loVect()[0]), &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
        &(Out.hiVect()[2]), Out.dataPtr(),
        /* This is what we are going to read from. */
    &(medium_H[dir].loVect()[0]), &(medium_H[dir].loVect()[1]), &(medium_H[dir].loVect()[2]),
        &(medium_H[dir].hiVect()[0]), &(medium_H[dir].hiVect()[1]), &(medium_H[dir].hiVect()[2]),
        medium_H[dir].dataPtr(),
        /* Verbosity and return. */
    &watch_dump_data, &return_status);
  }    /* else if ( (field ≡ "hMdx") ... */
```

9.2.7 Evaluate User Defined Fields

209 ⟨ *dump_data*: precompute field 203 ⟩ +≡

```
else { }
```

9.3 Transfer Fields to the Output Object

210 ⟨ *dump_data*: extract field from precomputes 210 ⟩ ≡

```
/* Find the field's index in the precomputes arrays */
int index = -1;
for (int i = 0; i < number_of_precomputes; i++)
  if (vector_of_precomputes[i] ≡ field) index = i;
if (index < 0) {
  pout() << "\n\t\t\tERROR(dump_data): field " << field << " not precomputed" << endl;
  return_status = ERR_DUMP_DATA;
  goto exit;
}
/* Now copy from Out[index] to OutWrite[count]. I use my own Fortran copy function here, because
   I don't quite trust Chombo's FArrayBox::copy. */
output_field_copy_(&SpaceDim, &number_of_precomputes, &index, &number_of_outputs, &count,
    &(Out.loVect()[0]), &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
    &(Out.hiVect()[2]), Out.dataPtr(), OutWrite.dataPtr(), &watch_dump_data, &return_status);
if (return_status ≠ EXIT_SUCCESS) {
  pout() << "\n\t\t\tERROR(dump_data): output_field_copy_ returned " << return_status << endl;
  goto exit;
}
```

This code is used in chunk 201.

10 Auxiliary Functions

211 ⟨ `Auxiliary.c` 211 ⟩ ≡

```
#include "Forms.H"
  ⟨ Function is_a_pwr_of_two 212 ⟩
  ⟨ Function is_file_readable 213 ⟩
  ⟨ Function effective_grid_bounds 214 ⟩
  ⟨ Function make_tag_set 222 ⟩
  ⟨ Function upml_sigma 230 ⟩
  ⟨ Function mark_regions 231 ⟩
  ⟨ Function complement 232 ⟩
  ⟨ Function mark_TFR_faces 233 ⟩
  ⟨ Function fill_PML_arrays 235 ⟩
  ⟨ Function swap_idx 236 ⟩
```

10.1 Is It a Power of Two

This is a little charming function that checks if a given integer, call it n is a power of 2. If it is then $n = 2^m$ where m is some other integer, and thus $m = \log_2 n$.

A logarithm of base 2 can be expressed in terms of natural logarithms as follows. Let x be such a number that $\mathrm{e}^x = 2$. Then $n = 2^m = (\mathrm{e}^x)^m = \mathrm{e}^{xm}$. Therefore $xm = \ln n$. But since $x = \ln 2$, we find that

$$m = \frac{\ln n}{\ln 2} = \log_2 n \tag{462}$$

So, now if m is an integer then

$$\mathrm{floor}\left(\frac{\ln n}{\ln 2}\right) = \left(\frac{\ln n}{\ln 2}\right) = m \tag{463}$$

and we should get that

$$n = \mathrm{floor}\left(2^{\mathrm{floor}(\ln n/\ln 2)}\right) = 2^m \tag{464}$$

But if m is not an integer, the "floors" will chop the non-integer part and we won't get the equality. Hence, testing for the equality is equivalent to testing if n is a power of 2.

212 ⟨Function *is_a_pwr_of_two* 212⟩ ≡

```
int is_a_pwr_of_two(int n)
{
  const double eps = 0.001;
  int n_compare;    /* A slight problem here with floating point arithmetic, hence the dirty trick with the
      epsilon. This needs to be watched... */
  n_compare = (int)(floor(pow(2.0, (log((double) n + eps)/log((double) 2)))));
  pout() << "\tn␣=␣" << n << ",␣n_compare␣=␣" << n_compare << endl;    /* Intel's icpc objected to
      comparing integers and floats, so we try a clean int-to-int comparison here. */
  return (n ≡ n_compare);
}
```

This code is used in chunk 211.

10.2 Is File Readable

This is a simple function that returns the file size in bytes if the file named in its argument is readable and NO (zero) otherwise. If it so happens that the file is readable, but its size is zero, NO is returned as well.

We first call the UNIX *stat* function on the file, and if this fails, we return NO.

Otherwise we check if the file is regular, i.e., not a device file and not a directory either.[14] If the file is not regular, we return NO.

Otherwise we attempt to open the file with the C++ **ifstream** :: *open*( ). If it fails we return NO.

Otherwise, we close the file and return its size in bytes.

Why not try **ifstream** :: *open*( ) right away? This is because the *open*( ) would work on a directory or a device file, and we don't want this.

213 ⟨ Function *is_file_readable* 213 ⟩ ≡

```
off_t is_file_readable(const char *file_name)
{
  struct stat file_status;
  off_t exit_status;
  ifstream file_stream;
  if (stat(file_name, &file_status) < 0) {
    pout() << "\tERROR(is_file_readable):␣cannot␣stat␣" << file_name << endl;
    pout() << "\t\tError␣number␣" << errno << endl;
    exit_status = (off_t) NO;
    goto exit;
  }
  if (¬(S_ISREG(file_status.st_mode))) {
    pout() << "\tERROR(is_file_readable):␣file␣" << file_name << "␣is␣not␣regular" << endl;
    exit_status = (off_t) NO;
    goto exit;
  }
  file_stream.open(file_name);
  if (file_stream.fail()) {
    pout() << "\tERROR(is_file_readable):␣file␣" << file_name <<
        "␣cannot␣be␣opened␣for␣reading" << endl;
    exit_status = (off_t) NO;
    goto exit;
  }
  file_stream.close();
  exit_status = file_status.st_size;
exit: return exit_status;
}
```

This code is used in chunk 211.

[14] (Will a soft link show as a regular file? Hopefully not.

10.3 Effective Grid Bounds

We just scan through all the boxes of the layout, find min and max, and this is it. Because we use **LayoutIterator** every process scans through all the boxes of the box layout, including the ones that live on other processes.

Why do I need this silly function at all? This is because when *domainSplit* is called on the top level grid, it may sometimes generate a grid that does not cover the original domain exactly. Calling this function lets me find out what the actual computational domain is.

I used to mess about with parallel version of this function, not quite understanding that it wasn't at all necessary, because every process has full knowledge of the *whole* box layout, which can be accessed by extracting **LayoutIterator** from it, and then iterating over the layout.

214 ⟨Function *effective_grid_bounds* 214⟩ ≡

```
int effective_grid_bounds(level *level_ptr) {
    int return_status = EXIT_SUCCESS;
    int watch_effective_grid_bounds = false;
    SCMParmParse watch_parameter_parser("forms.watch");
```

See also chunks 215, 216, 217, 218, 219, 220, and 221.

This code is cited in chunks 90 and 222.

This code is used in chunk 211.

¶ These two **IntVect**s correspond to the lower and upper corners of a box within which the given subgrid is embedded tightly.

The *min*s are all initialized to the largest integer possible and the *max*es are all initialized to the lowest (most negative) integer possible.

215 ⟨Function *effective_grid_bounds* 214⟩ +≡

```
IntVect
ijk_min(D_DECL(numeric_limits⟨int⟩::max(), numeric_limits⟨int⟩::max(), numeric_limits⟨int⟩::max())),
ijk_max(D_DECL(numeric_limits⟨int⟩::min(), numeric_limits⟨int⟩::min(), numeric_limits⟨int⟩::min()));
```

¶ Vectors *xyz_min* and *xyz_max* are for the physical (as opposed to **IntVect**) coordinates of *ijk_min* and *ijk_max*.

216 ⟨Function *effective_grid_bounds* 214⟩ +≡

```
Vector⟨Real⟩ xyz_min(SpaceDim, 0.0), xyz_max(SpaceDim, 0.0);
```

¶ These two variables are the box layout of the current level and the layout iterator associated with it.

217 ⟨Function *effective_grid_bounds* 214⟩ +≡

```
DisjointBoxLayout &box_layout = level_ptr→box_layout;
LayoutIterator layout_iterator = box_layout.layoutIterator();
```

¶ Action starts in this chunk. The first thing, as usual, is to check how much we should be telling the user about our actions.

218 ⟨Function *effective_grid_bounds* 214⟩ +≡

```
    /* Action */
watch_parameter_parser.query("effective_grid_bounds", watch_effective_grid_bounds);
if (watch_effective_grid_bounds) pout() ≪ "effective_grid_bounds:" ≪ endl;
```

¶ Now, we loop over all the boxes of the layout and find the *smallEnd*( ) and the *bigEnd*( ) of each. Then compare with what we have in *ijk_min* and *ijk_max* and adjust the latter two vectors to hold the min and max values always.

219 ⟨ Function *effective_grid_bounds* 214 ⟩ +≡

```
    /* Find the extremal values for i, j, and k */
  for (layout_iterator.reset( ); layout_iterator.ok( ); ++layout_iterator) {
    Box box = box_layout.get(layout_iterator( ));
    IntVect lo_vector = box.smallEnd( );
    IntVect hi_vector = box.bigEnd( );
    for (int i = 0; i < SpaceDim; i++) {
      if (lo_vector[i] < ijk_min[i]) ijk_min[i] = lo_vector[i];
      if (hi_vector[i] > ijk_max[i]) ijk_max[i] = hi_vector[i];
    }
  }
  if (watch_effective_grid_bounds)
    pout( ) << "\tijk_min␣=␣" << ijk_min << ",␣ijk_max␣=␣" << ijk_max << endl;
```

¶ Each process, having found *ijk_min* and *ijk_max*, can use its own knowledge of *origin* and *delta* to evaluate physical coordinates that correspond to the min and max **IntVect**s of this level.

220 ⟨ Function *effective_grid_bounds* 214 ⟩ +≡

```
  for (int i = 0; i < SpaceDim; i++) {
    xyz_min[i] = level_ptr→origin[i] + ijk_min[i] * level_ptr→delta;
    xyz_max[i] = level_ptr→origin[i] + ijk_max[i] * level_ptr→delta;
  }
  level_ptr→ijk_min = ijk_min;
  level_ptr→ijk_max = ijk_max;
  level_ptr→xyz_min.resize(SpaceDim);
  level_ptr→xyz_min = xyz_min;
  level_ptr→xyz_max.resize(SpaceDim);
  level_ptr→xyz_max = xyz_max;
  if (watch_effective_grid_bounds)
    pout( ) << "\tlevel_ptr->xyz_min␣=␣(" << level_ptr→xyz_min << "),␣level_ptr->xyz_max␣=␣(" <<
        level_ptr→xyz_max << ")" << endl;
```

¶ And this is it. We return to the caller

221 ⟨ Function *effective_grid_bounds* 214 ⟩ +≡ /* exit: */

```
  if (watch_effective_grid_bounds) pout( ) << "effective_grid_bounds:␣returning␣to␣caller" << endl;
  return return_status;
  }
```

10.4 Make Tag Set

The effect of calling *scm_c_eval_string*(*lambda*) is some pattern drawn on the **level** . *tags* field. But this is not what **BRMeshRefine** wants. The latter wants to see a *set* of cells, an **IntVectSet**. Such a set is easy to construct. All we need to do is to run through every **BaseFab**⟨**int**⟩ of the **level** . *tags* field, then run through its every cell, and if the value of the field in the cell is 1, then we add the cell to the set, otherwise, we don't.

But this will produce a set of cells for a given MPI process that is limited to what the process can see of the whole grid. So we have to carry out a procedure similar to the one outlined in *effective_grid_bounds*, module ⟨ Function *effective_grid_bounds* 214 ⟩, section 10.3, page 318. We have to collect the sets from all MPI processes, combined them into a single set that contains all tagged cells, and then return the set to the processes.

We begin by presenting the interface of the function. It takes a pointer to a **level** as its argument and returns an integer: `EXIT_SUCCESS` if all has worked fine. The first three internal variables of the functions are as we have seen before in other functions.

222 ⟨ Function *make_tag_set* 222 ⟩ ≡

```
int make_tag_set(level *level_ptr) {
    int return_status = EXIT_SUCCESS;
    int watch_make_tag_set = false;
    SCMParmParse watch_parameter_parser("forms.watch");
```

See also chunks 223, 224, 225, 226, 227, 228, and 229.

This code is cited in chunk 93.

This code is used in chunk 211.

¶

tags This is the tagged **BaseFab**⟨**int**⟩ field of the level.

local_tag_set This is the set which every MPI process is going to construct on its own by scanning its portion of the *tags* field.

tag_set This is the global set of tagged cells that is the sum of all *local_tag_set*s.

223 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
LevelData⟨BaseFab⟨int⟩⟩ &tags = level_ptr→tags;
IntVectSet local_tag_set;
IntVectSet &tag_set = level_ptr→tag_set;
```

¶

box_layout This is the disjoint box layout of the current level.

data_iterator This is the level data iterator associated with the current level's box layout.

224 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
DisjointBoxLayout &box_layout = level_ptr→box_layout;
DataIterator data_iterator(box_layout);
```

¶

root_process This is a process that will collect sets from all other MPI processes, and merge them.

number_of_processes The number of processes in the current MPI pool.

vector_of_tag_sets The *root_process* will collect sets sent to it from other processes on this vector.

225 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
int root_process = uniqueProc(SerialTask::compute);
int number_of_processes = numProc();
Vector⟨IntVectSet⟩ vector_of_tag_sets(number_of_processes);
```

¶ Action starts here. We begin by checking the verbosity level.

226 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
watch_parameter_parser.query("make_tag_set", watch_make_tag_set);
if (watch_make_tag_set) pout() << "make_tag_set:" << endl;
```

¶ Here each MPI process collects tagged cells and puts them in its *local_tag_set*. The iteration is over the **BaseFab**s that the process looks after. For each **BaseFab**⟨**int**⟩, here referred to as *local_tags*, we extract its **Box**, then loop over its cells and add the cell to the set if *local_tags*(*cell*) is other than zero.

227 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
local_tag_set.makeEmpty();
if (watch_make_tag_set) pout() << "\tProcess␣" << procID() << ":␣collecting␣tags␣locally" << endl;
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  BaseFab<int> &local_tags = tags[data_iterator()];
  Box tag_box = local_tags.box();
  BoxIterator box_iterator(tag_box);
  for (box_iterator.reset(); box_iterator.ok(); ++box_iterator) {
    IntVect cell = box_iterator();
    if (local_tags(cell)) local_tag_set |= cell;
  }
}
```

¶ Here the *root_process* gathers *local_tag_set*s from all MPI processes on its *vector_of_tag_sets*. Then it adds them all to *tag_set*, which has been emptied just before this operation, and sends the latter back to the processes.

228 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
tag_set.makeEmpty();
if (watch_make_tag_set) pout() << "\tExchanging␣data..." << endl;
gather(vector_of_tag_sets, local_tag_set, root_process);
if (procID() ≡ root_process)
  for (int count = 0; count < number_of_processes; count++) tag_set |= vector_of_tag_sets[count];
broadcast(tag_set, root_process);
if (watch_make_tag_set) pout() << "\tProcess␣" << procID() << ":␣" << level_ptr→tag_set.numPts() <<
      "␣points␣in␣tag_set" << endl;
```

¶ And this it. We return to the caller.

229 ⟨ Function *make_tag_set* 222 ⟩ +≡

```
/* exit: */
if (watch_make_tag_set) pout() << "make_tag_set:␣returning␣to␣caller" << endl;
return return_status;
}
```

10.5 The UPML sigma

Here we evaluate [8]

$$\sigma(x) = \frac{\sigma_{\max} \left|x - x_0\right|^m}{d^m}, \tag{465}$$

where x_0 is the position of the PML boundary, m is an experimental parameter, which seems to return best results for $m = 4$, and d is the width of the UPML layer.

230 ⟨Function *upml_sigma* 230⟩ ≡

```
Real upml_sigma(int count, int ghost_margin, Real origin, Real delta, Real x_min, Real x_max, Real
        pml_min, Real pml_max, Real sigma_max)
{
  const int m = PML_M;
  int index = count − ghost_margin;
  Real position = index ∗ delta + origin;
  if (position < pml_min) {
    Real width = pml_min − x_min;
    Real depth = pml_min − position;
    if (depth ≥ width) return sigma_max;
    else return (sigma_max ∗ pow(depth/width, m));
  }
  else if (position > pml_max) {
    Real width = x_max − pml_max;
    Real depth = position − pml_max;
    if (depth ≥ width) return sigma_max;
    else return (sigma_max ∗ pow(depth/width, m));
  }
  else return 0.0;
}
```

This code is used in chunk 211.

10.6 Marking Regions

This function is slow, because drawing data on **BaseFab⟨int⟩** with *setVal* is slow, and because we calculate and compare coordinates for *every* edge and for every face of the 3D grid (of which there are six per cell altogether). Luckily, we only have to do this once.

The way to speed up this and similar functions would be to redo the computations within Fortran, thus avoiding *setVal*s.

231 ⟨Function *mark_regions* 231⟩ ≡

```
int mark_regions(level *level_ptr)
{    /* Mark UPML and scattered field regions on the medium_E and medium_H fields. The UPML
        region gets medium value of PML_MARKER (-2) and the scattered field region gets medium value of
        SCATTERED_FIELD_MARKER (-1). */
  SCMParmParse watch_parameter_parser("forms.watch");
  int watch_mark_regions = (int) false;
  int return_value = OK;
  const int pml_marker = PML_MARKER;
  const int scattered_field_marker = SCATTERED_FIELD_MARKER;
  Vector⟨Real⟩ &pml_xyz_min = level_ptr→pml_xyz_min;
  Vector⟨Real⟩ &pml_xyz_max = level_ptr→pml_xyz_max;
  Vector⟨Real⟩ &signal_xyz_min = level_ptr→signal_xyz_min;
  Vector⟨Real⟩ &signal_xyz_max = level_ptr→signal_xyz_max;
  Vector⟨Real⟩ &origin = level_ptr→origin;
  Real &delta = level_ptr→delta;
  LevelData⟨EdgeFab⟨int⟩⟩ &medium_E = level_ptr→medium_E;
  LevelData⟨FaceFab⟨int⟩⟩ &medium_H = level_ptr→medium_H;
  DataIterator &data_iterator = level_ptr→data_iterator;
  watch_parameter_parser.query("mark_regions", watch_mark_regions);
  if (watch_mark_regions) pout() ≪ "\tmark_regions:␣" ≪ endl;
  if (level_ptr→number ≠ 0) {
    pout() ≪ "\t\tERROR(mark_regions):␣cannot␣mark␣on␣level␣" ≪ level_ptr→number ≪ endl;
    return_value = ERR_MARK_REGIONS;
    goto exit;
  }
        /* Mark medium_E */
  if (watch_mark_regions) pout() ≪ "\t\tmarking␣edges" ≪ endl;
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab⟨int⟩ &medium_E_dir = medium_E[data_iterator()][dir];
      Box medium_E_dir_box = medium_E_dir.box();
      BoxIterator box_iterator(medium_E_dir_box);
      IntVect box_type = medium_E_dir_box.type();
      if (watch_mark_regions > 1) {
        pout() ≪ "\t\tdir␣=␣" ≪ dir ≪ "␣box␣=␣" ≪ medium_E_dir_box ≪ endl;
      }
      for (box_iterator.reset(); box_iterator.ok(); ++box_iterator) {
        IntVect cell = box_iterator();
        Box cell_box(cell, cell, box_type);
        Vector⟨Real⟩ position(SpaceDim);
        for (int i = 0; i < SpaceDim; i++) {
          position[i] = origin[i] + (cell[i] − 0.5 * box_type[i]) * delta;
        }
        if ((position[0] ≤ pml_xyz_min[0]) ∨ (pml_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤ pml_xyz_min[1]) ∨
              (pml_xyz_max[1] ≤ position[1])∨(position[2] ≤ pml_xyz_min[2])∨(pml_xyz_max[2] ≤ position[2]))
          {
```

```
            medium_E_dir.setVal(pml_marker, cell_box, 0);
          }
          else if ((position[0] ≤ signal_xyz_min[0]) ∨ (signal_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤
                signal_xyz_min[1]) ∨ (signal_xyz_max[1] ≤ position[1]) ∨ (position[2] ≤
                signal_xyz_min[2]) ∨ (signal_xyz_max[2] ≤ position[2])) {
            medium_E_dir.setVal(scattered_field_marker, cell_box, 0);
          }
        }
      }
    }
          /* Mark medium_H */
    if (watch_mark_regions) pout() << "\t\tmarking faces" << endl;
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      for (int dir = 0; dir < SpaceDim; dir++) {
        BaseFab<int> &medium_H_dir = medium_H[data_iterator()][dir];
        Box medium_H_dir_box = medium_H_dir.box();
        BoxIterator box_iterator(medium_H_dir_box);
        IntVect box_type = medium_H_dir_box.type();
        if (watch_mark_regions > 1) {
          pout() << "\t\tdir = " << dir << " box = " << medium_H_dir_box << endl;
        }
        for (box_iterator.reset(); box_iterator.ok(); ++box_iterator) {
          IntVect cell = box_iterator();
          Box cell_box(cell, cell, box_type);
          Vector<Real> position(SpaceDim);
          for (int i = 0; i < SpaceDim; i++) {
            position[i] = origin[i] + (cell[i] - 0.5 * box_type[i]) * delta;
          }
          if ((position[0] ≤ pml_xyz_min[0]) ∨ (pml_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤ pml_xyz_min[1]) ∨
                (pml_xyz_max[1] ≤ position[1])∨(position[2] ≤ pml_xyz_min[2])∨(pml_xyz_max[2] ≤ position[2]))
            {
            medium_H_dir.setVal(pml_marker, cell_box, 0);
          }
          else if ((position[0] ≤ signal_xyz_min[0]) ∨ (signal_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤
                signal_xyz_min[1]) ∨ (signal_xyz_max[1] ≤ position[1]) ∨ (position[2] ≤
                signal_xyz_min[2]) ∨ (signal_xyz_max[2] ≤ position[2])) {
            medium_H_dir.setVal(scattered_field_marker, cell_box, 0);
          }
        }
      }
    }
  exit:
    if (watch_mark_regions) pout() << "\tmark_regions: returning to caller" << endl;
    return return_value;
  }
```

This code is used in chunk 211.

10.7 Return Third Direction

Return a complementary third direction if two different directions are given. Flag (negative) error otherwise.

232 ⟨Function *complement* 232⟩ ≡

```
int complement(int i, int j)
{
  int return_value;
      /* Works for SpaceDim ≡ 3 only */
  if (((i ≡ 0) ∧ (j ≡ 1)) ∨ ((i ≡ 1) ∧ (j ≡ 0))) return_value = 2;
  else if (((i ≡ 0) ∧ (j ≡ 2)) ∨ ((i ≡ 2) ∧ (j ≡ 0))) return_value = 1;
  else if (((i ≡ 1) ∧ (j ≡ 2)) ∨ ((i ≡ 2) ∧ (j ≡ 1))) return_value = 0;
  else {
    pout() ≪ "\tERROR(complement): bad arguments" ≪ endl;
    return_value = ERR_COMPLEMENT;
  }
  return return_value;
}
```

This code is used in chunk 211.

10.8 Mark Total Field Region Faces

Fill the **Vector**⟨**Vector**⟨**Vector**⟨**DataIndex**⟩⟩⟩ *total_field_side_boxes* slot of *levels*[0].

This form of the function is merely a wrapper. The function just invokes itself six times with direction and side specified.

233 ⟨ Function *mark_TFR_faces* 233 ⟩ ≡

```
int mark_TFR_faces(level *level_ptr)
{
  int return_status = EXIT_SUCCESS;
  int watch_mark_TFR_faces = (int) false;
  SideIterator side_iterator;
  IntVect &signal_ijk_lo = level_ptr→signal_ijk_lo;
  IntVect &signal_ijk_hi = level_ptr→signal_ijk_hi;
  Vector⟨Vector⟨Vector⟨DataIndex⟩⟩⟩ &TFR_face_boxes = level_ptr→TFR_face_boxes;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("mark_TFR_faces", watch_mark_TFR_faces);
  if (watch_mark_TFR_faces) {
    pout() ≪ "\tmark_TFR_faces:" ≪ endl;
    if (watch_mark_TFR_faces > 1) {
      pout() ≪ "\t\tsignal_ijk_lo␣=␣" ≪ signal_ijk_lo ≪ endl;
      pout() ≪ "\t\tsignal_ijk_hi␣=␣" ≪ signal_ijk_hi ≪ endl;
    }
  }
  if (watch_mark_TFR_faces) pout() ≪ "\t\tresizing␣TFR_face_boxes" ≪ endl;
  TFR_face_boxes.resize(SpaceDim);
  for (int dir = 0; dir < SpaceDim; dir++) TFR_face_boxes[dir].resize(2);
  for (int dir = 0; dir < SpaceDim; dir++) {
    for (side_iterator.reset(); side_iterator.ok(); ++side_iterator) {
      Side::LoHiSide side = side_iterator();
      if (watch_mark_TFR_faces > 1)
        pout() ≪ "\t\tcalling␣mark_TFR_faces:␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪ endl;
      if (return_status = mark_TFR_faces(level_ptr, dir, side)) {
        pout() ≪ "\t\tERROR(mart_TFR_faces):␣failed␣on␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪
            endl;
        goto exit;
      }
      else {
        if (watch_mark_TFR_faces > 1) {
          Vector⟨DataIndex⟩ &face_boxes = level_ptr→TFR_face_boxes[dir][side];
          DisjointBoxLayout &box_layout = level_ptr→box_layout;
          pout() ≪ "\t\tSelected␣" ≪ face_boxes.size() ≪ "␣boxes␣out␣of␣" ≪
              box_layout.numBoxes(procID()) ≪ endl;
        }
        if (watch_mark_TFR_faces > 2) {
          Vector⟨DataIndex⟩ &face_boxes = level_ptr→TFR_face_boxes[dir][side];
          DisjointBoxLayout &box_layout = level_ptr→box_layout;
          pout() ≪ "\t\tTFR␣boxes␣for␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪ ":" ≪ endl;
          for (int count = 0; count < face_boxes.size(); count++) {
            DataIndex &data_index = face_boxes[count];
            Box the_box = box_layout.get(data_index);
            pout() ≪ "\t\t\t" ≪ the_box ≪ endl;
          }
        }
      }
```

```
      }
    }
  exit:
    if (watch_mark_TFR_faces) pout() << "\tmark_TFR_faces:␣returning␣to␣caller" << endl;
    return return_status;
  }
```

See also chunk 234.

This code is used in chunk 211.

¶ Here we perform the task for a specific direction and side.

First we select the index that corresponds to the plane. It's called *ns*. Then we enter a loop over all boxes of the domain and for each check if it intersects with *ns* in direction *dir*.

If it does, we check if it fits in the total field region in the other two directions. If it does then only we push it onto the *TFR_face_boxes*[*dir*][*side*] vector.

234 ⟨Function *mark_TFR_faces* 233⟩ +≡

```
  int mark_TFR_faces(level *level_ptr, int dir, Side::LoHiSide side)
  {
    int return_status = EXIT_SUCCESS;
    int watch_mark_TFR_faces = (int) false;
    SCMParmParse watch_parameter_parser("forms.watch");
    IntVect &signal_ijk_lo = level_ptr→signal_ijk_lo;
    IntVect &signal_ijk_hi = level_ptr→signal_ijk_hi;
    int ns = (side ≡ Side::Lo) ? signal_ijk_lo[dir] : signal_ijk_hi[dir];
    watch_parameter_parser.query("mark_TFR_faces", watch_mark_TFR_faces);
    if (watch_mark_TFR_faces) {
      pout() << "\t\tmark_TFR_faces,␣dir␣=␣" << dir << ",␣side␣=␣" << side << endl;
      if (watch_mark_TFR_faces > 1) {
        for (int i = 0; i < SpaceDim; i++) {
          if (i ≡ dir) pout() << "\t\t\tns␣=␣" << ns << endl;
          else pout() << "\t\t\tlo␣=␣" << signal_ijk_lo[i] << ",␣hi␣=␣" << signal_ijk_hi[i] << endl;
        }
      }
    }
    DataIterator &data_iterator = level_ptr→data_iterator;
    DisjointBoxLayout &box_layout = level_ptr→box_layout;
    Vector⟨DataIndex⟩ &face_boxes = level_ptr→TFR_face_boxes[dir][side];
    face_boxes.clear();
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      Box the_box = box_layout.get(data_index);
          /* This box has to be enlarged to cover all subboxes of the EdgeFab or FaceFab for this dir. */
      Box enlarged_box(surroundingNodes(the_box));
      IntVect small_end = enlarged_box.smallEnd();
      IntVect big_end = enlarged_box.bigEnd();
      if ((small_end[dir] ≤ ns) ∧ (ns ≤ big_end[dir]))      /* is on the plane */
      {
        vector⟨bool⟩ is_in(2, false);
        int count = 0;
        for (int i = 0; i < SpaceDim; i++) {
          if (i ≠ dir)
            if ((big_end[i] ≥ signal_ijk_lo[i]) ∧ (small_end[i] ≤ signal_ijk_hi[i])) is_in[count++] = true;
        }
```

```
        if (is_in[0] ∧ is_in[1]) face_boxes.push_back(data_index);
      }
    }
    return return_status;
}
```

10.9 Fill PML Arrays

235 ⟨Function *fill_PML_arrays* 235⟩ ≡

```
int fill_PML_arrays(level *level_ptr)
{
  int return_status = EXIT_SUCCESS;
  int watch_fill_PML_arrays = (int) false;
  int level_number = level_ptr→number;
  int const ghost_margin = N_GHOST_CELLS;
  Real sigma_max = PML_SIGMA_MAX;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse pml_parameter_parser("forms.pml");
  IntVect &ijk_min = level_ptr→ijk_min;
  IntVect &ijk_max = level_ptr→ijk_max;
  Vector⟨Real⟩ &xyz_min = level_ptr→xyz_min;
  Vector⟨Real⟩ &xyz_max = level_ptr→xyz_max;
  Vector⟨Real⟩ &pml_xyz_min = level_ptr→pml_xyz_min;
  Vector⟨Real⟩ &pml_xyz_max = level_ptr→pml_xyz_max;
  Vector⟨Real⟩ &origin = level_ptr→origin;
  const Real &delta = level_ptr→delta;
  const Real &dt = level_ptr→dt;
  Vector⟨Real⟩ &C0x = level_ptr→C0x;
  Vector⟨Real⟩ &C1x = level_ptr→C1x;
  Vector⟨Real⟩ &C2x = level_ptr→C2x;
  Vector⟨Real⟩ &C3x = level_ptr→C3x;
  Vector⟨Real⟩ &C4x = level_ptr→C4x;
  Vector⟨Real⟩ &C0y = level_ptr→C0y;
  Vector⟨Real⟩ &C1y = level_ptr→C1y;
  Vector⟨Real⟩ &C2y = level_ptr→C2y;
  Vector⟨Real⟩ &C3y = level_ptr→C3y;
  Vector⟨Real⟩ &C4y = level_ptr→C4y;
  Vector⟨Real⟩ &C0z = level_ptr→C0z;
  Vector⟨Real⟩ &C1z = level_ptr→C1z;
  Vector⟨Real⟩ &C2z = level_ptr→C2z;
  Vector⟨Real⟩ &C3z = level_ptr→C3z;
  Vector⟨Real⟩ &C4z = level_ptr→C4z;
  int Cx_length = ijk_max[0] − ijk_min[0] + 1 + 2 ∗ ghost_margin;
  int Cy_length = ijk_max[1] − ijk_min[1] + 1 + 2 ∗ ghost_margin;
  int Cz_length = ijk_max[2] − ijk_min[2] + 1 + 2 ∗ ghost_margin;
  watch_parameter_parser.query("fill_PML_arrays", watch_fill_PML_arrays);
  pml_parameter_parser.query("sigma_max", sigma_max);
  if (watch_fill_PML_arrays) {
    pout() ≪ "\tfill_PML_arrays:" ≪ endl;
  }
  if (level_number ≠ 0) {
    pout() ≪ "\t\tERROR(fill_PML_arrays):␣level␣number␣not␣equal␣zero" ≪ endl;
    return_status = ERR_PML_ARRAYS;
    goto exit;
  }
  C0x.resize(Cx_length);
  C1x.resize(Cx_length);
  C2x.resize(Cx_length);
  C3x.resize(Cx_length);
  C4x.resize(Cx_length);
  C0y.resize(Cy_length);
```

```
    C1y.resize(Cy_length);
    C2y.resize(Cy_length);
    C3y.resize(Cy_length);
    C4y.resize(Cy_length);
    C0z.resize(Cz_length);
    C1z.resize(Cz_length);
    C2z.resize(Cz_length);
    C3z.resize(Cz_length);
    C4z.resize(Cz_length);
    if (watch_fill_PML_arrays) pout() << "\t\t\tcalling upml_sigma(count, " << ghost_margin << ", "
          << origin[0] << ", " << delta << ", " << xyz_min[0] << ", " << xyz_max[0] << ", " << pml_xyz_min[0] <<
          ", " << pml_xyz_max[0] << ", " << sigma_max << ")" << endl;
    for (int count = 0; count < Cx_length; count++) {
      Real sigma = upml_sigma(count, ghost_margin, origin[0], delta, xyz_min[0], xyz_max[0], pml_xyz_min[0],
          pml_xyz_max[0], sigma_max);
      C3x[count] = 2.0 + sigma * dt;
      C4x[count] = 2.0 - sigma * dt;
      C0x[count] = 1.0/C3x[count];
      C1x[count] = C4x[count] * C0x[count];
      C2x[count] = 2 * dt/delta * C0x[count];
    }
    if (watch_fill_PML_arrays) pout() << "\t\t\tCx: " << C0x.size() << " entries initialized" << endl;
    if (watch_fill_PML_arrays) pout() << "\t\t\tcalling upml_sigma(count, " << ghost_margin << ", " <<
          origin[1] << ", " << delta << ", " << xyz_min[1] << ", " << xyz_max[1] << ", " << pml_xyz_min[1] <<
          ", " << pml_xyz_max[1] << ", " << sigma_max << ")" << endl;
    for (int count = 0; count < Cy_length; count++) {
      Real sigma = upml_sigma(count, ghost_margin, origin[1], delta, xyz_min[1], xyz_max[1], pml_xyz_min[1],
          pml_xyz_max[1], sigma_max);
      C3y[count] = 2.0 + sigma * dt;
      C4y[count] = 2.0 - sigma * dt;
      C0y[count] = 1.0/C3y[count];
      C1y[count] = C4y[count] * C0y[count];
      C2y[count] = 2 * dt/delta * C0y[count];
    }
    if (watch_fill_PML_arrays) pout() << "\t\t\tCy: " << C0y.size() << " entries initialized" << endl;
    if (watch_fill_PML_arrays) pout() << "\t\t\tcalling upml_sigma(count, " << ghost_margin << ", " <<
          origin[2] << ", " << delta << ", " << xyz_min[2] << ", " << xyz_max[2] << ", " << pml_xyz_min[2] <<
          ", " << pml_xyz_max[2] << ", " << sigma_max << ")" << endl;
    for (int count = 0; count < Cz_length; count++) {
      Real sigma = upml_sigma(count, ghost_margin, origin[2], delta, xyz_min[2], xyz_max[2], pml_xyz_min[2],
          pml_xyz_max[2], sigma_max);
      C3z[count] = 2.0 + sigma * dt;
      C4z[count] = 2.0 - sigma * dt;
      C0z[count] = 1.0/C3z[count];
      C1z[count] = C4z[count] * C0z[count];
      C2z[count] = 2 * dt/delta * C0z[count];
    }
    if (watch_fill_PML_arrays) pout() << "\t\t\tCz: " << C0z.size() << " entries initialized" << endl;
    if (watch_fill_PML_arrays) pout() << "\t\tdone." << endl;
  exit: return return_status;
  }
```

This code is used in chunk 211.

10.10 Swap New-Old Indexes

236 ⟨Function *swap_idx* 236⟩ ≡

```
void swap_idx(int idx_array[2])
{
  int hold = idx_array[0];
  idx_array[0] = idx_array[1];
  idx_array[1] = hold;
}
```

This code is used in chunk 211.

11 Header File

This is a listing of the header file. The actual file that is dumped by `tangle` is called `Forms.hw`. This file is then additionally massaged by `sed` to remove CWEB line numbers, comments, and empty lines. Finally, `astyle` is called to prettify it and the output is written on `Forms.H` eventually.

Although CWEB-related line numbers are lost in the process, the resulting code is cleaner and can be exported on its own. If references to the CWEB source are needed, these steps should be removed from the `Makefile` and `Forms.hw` simply renamed to `Forms.H`.

237 ⟨ `Forms.hw` 237 ⟩ ≡

```
#ifndef _FORMS_H_
# define _FORMS_H_
# defineprogram_author "Zdzislaw (Gustav) Meglicki, Indiana University"
# defineprogram_version "$Id: Includes.w,v 1.69 2009/08/20 18:09:09 gustav Exp $"
    /* C includes */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#ifndef CH_Linux
  extern int errno;
#endif
    /* Guile includes */
#include <libguile.h>
    /* C++ includes */
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>
#include <fstream>
#include <ios>
#include <vector>
#include <cmath>
#include <cstdlib>
  using std::cout;
  using std::cin;
  using std::cerr;
  using std::endl;
  using std::flush;
  using std::setw;
  using std::numeric_limits;
  using std::ifstream;
  using std::string;
```

See also chunks 238, 239, 240, 241, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 254, 256, 257, 259, 260, 261, 263, 264, and 265.

11.1 Chombo Includes

238 ⟨Forms.hw 237⟩ +≡

```
/* Standard Chombo includes */
#include <parstream.H>      /* pout() lives here */
#include <SPACE.H>      /* SpaceDim lives here */
#include <SPMD.H>      /* procID() and numProc() live here */
#include <IntVectSet.H>
#include <REAL.H>
#include <Box.H>
#include <DisjointBoxLayout.H>
#include <FArrayBox.H>
#include <FluxBox.H>
#include <LevelData.H>
#include <BRMeshRefine.H>
#include <LoadBalance.H>
#include <BoxIterator.H>
#include <LoHiSide.H>
#include <AMRIO.H>
/* My Chombo extensions: includes */
#include <SCMParmParse.H>
#include <EdgeFab.H>
#include <FaceFab.H>
#include <CurlBox.H>
```

11.2 Program Constants

239 ⟨Forms.hw 237⟩ +≡

```
/* Exit statuses */
#ifdef EXIT_SUCCESS
#undef EXIT_SUCCESS
#endif
# define EXIT_SUCCESS 0
# define ERR_COMPLEMENT -1
# define ERR_BAD_DIMENSION 1
# define ERR_NO_INPUT_FILE 2
# define ERR_BUILD_LEVELS 3
# define ERR_FILL_LEVELS 4
# define ERR_ADVANCE_D 5
# define ERR_ADVANCE_B 6
# define ERR_DUMP_DATA 7
# define ERR_INITIALIZE_SCHEME 8
# define ERR_D_TO_E 9
# define ERR_INJECT_D 10
# define ERR_B_TO_H 11
# define ERR_INJECT_B 12
# define ERR_MARK_REGIONS 30
# define ERR_PML_ARRAYS 31
/* Other program specific constants */
#ifdef NO
#undef NO
#endif
# define NO 0
#ifdef YES
#undef YES
#endif
# define YES 1
#ifdef OK
#undef OK
#endif
# define OK 0
#ifdef OUCH
#undef OUCH
#endif
# define OUCH 1
#ifdef ZERO
#undef ZERO
#endif
# define ZERO 0
#ifdef ONE
#undef ONE
#endif
# define ONE 1
#ifdef EPSILON
#undef EPSILON
#endif
# define EPSILON 0.0001
/* Default iteration specifications */
# define NUMBER_OF_STEPS 100
# define IMAGE_FREQUENCY 10
```

```
# define STRIDE 2
# define T0 0.0
# define INITIAL_LABEL 1
    /* Default grid specifications */
# define N_CELLS 128
# define N_CELLS_MIN 8
# define N_CELLS_MAX 1024
# define DELTA 1.0
# define X0 0.0
# define PML_XYZ_MIN 10
# define PML_XYZ_MAX 118
# define PML_MARGIN 3.0
# define PML_M 4
# define PML_SIGMA_MAX 10.0
# define SIGNAL_XYZ_MIN 20
# define SIGNAL_XYZ_MAX 108
# define SIGNAL_MARGIN 3.0    /* PML_MARGIN and SIGNAL_MARGIN should really be 7.0, but we make them
    less temporarily for test purposes. */
# define N_LEVELS 1
# define N_LEVELS_MIN 1
# define N_LEVELS_MAX 10
# define N_GHOST_CELLS 2
# define N_AUXILIARIES 0
# define N_AUXILIARIES_MIN 0
# define N_AUXILIARIES_MAX 10
# define MAX_BOX_SIZE 64
# define MAX_BOX_SIZE_MIN 4
# define MAX_BOX_SIZE_MAX 1024
# define LAMBDA_MIN_SIZE 5
# define REFINE_RATIO 2
# define REFINE_BLOCK_FACTOR 2
# define REFINE_BUFFER_SIZE 4
# define REFINE_MAX_SIZE 50
# define REFINE_FILL_RATIO 1.0    /* REFINE_BUFFER_SIZE should really be 8, but here we make it smaller
    for test purposes. */    /* Region markers */
# define PML_MARKER -2
# define SCATTERED_FIELD_MARKER -1
# define FORMS_OUTPUT "forms_output"
# define OUTPUT_NUMBER_OF_DIGITS 3
```

11.3 C++ Functions

240 ⟨ Forms.hw 237 ⟩ +≡
⟨ Structure level 266 ⟩

```
int build_levels(Vector⟨level *⟩ &levels);
int fill_levels(Vector⟨level *⟩ &levels);
int mark_regions(level *level_ptr);
int advance_d(level *level_ptr);
int advance_d_0(level *level_ptr);
int advance_d_n(level *level_ptr);
int inject_d(level *level_ptr);
int inject_d(level *level_ptr, int dir, Side::LoHiSide side);
int d_to_e(level *level_ptr);
int d_to_e_0(level *level_ptr);
int d_to_e_0_0(level *level_ptr);
int d_to_e_0_n(level *level_ptr);
int d_to_e_n(level *level_ptr);
int d_to_e_n_0(level *level_ptr);
int d_to_e_n_n(level *level_ptr);
int advance_b(level *level_ptr);
int advance_b_0(level *level_ptr);
int advance_b_n(level *level_ptr);
int inject_b(level *level_ptr);
int inject_b(level *level_ptr, int dir, Side::LoHiSide side);
int b_to_h(level *level_ptr);
int b_to_h_0(level *level_ptr);
int b_to_h_n(level *level_ptr);
int dump_data(Vector⟨level *⟩ &levels, int label);
int is_a_pwr_of_two(int n);
off_t is_file_readable(const char *file_name);
int effective_grid_bounds(level *level_ptr);
int make_tag_set(level *level_ptr);
Real upml_sigma(int count, int ghost_margin, Real origin, Real delta, Real x_min, Real x_max, Real
    pml_min, Real pml_max, Real sigma_max);
int mark_TFR_faces(level *level_ptr);
int mark_TFR_faces(level *level_ptr, int dir, Side::LoHiSide side);
int fill_PML_arrays(level *level_ptr);
void swap_idx(int idx_array[2]);
```

11.4 Scheme Functions

241 ⟨Forms.hw 237⟩ +≡

```
/* Scheme stuff */
bool does_scm_symbol_exist(const char *name);
int initialize_scheme();
int post_initialize_scheme();
```

11.4.1 Drawing Figures

A box

243 ⟨Forms.hw 237⟩ +≡

```
SCM draw_box(SCM low_corner, SCM high_corner, SCM color);
int draw_media_box(Vector⟨Real⟩ &low_corner, Vector⟨Real⟩ &high_corner, int &color);
int draw_tags_box(Vector⟨Real⟩ &low_corner, Vector⟨Real⟩ &high_corner);
extern "C"
{
  void f_draw_box_(
  const int *const SpaceDim, const int *const ghost_margin, const Real *const origin, const
      Real *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const
      Real *const low_corner, const Real *const high_corner, const int *const color, const int
      *const box_type, const int *const i_lo, const int *const j_lo, const int *const k_lo, const int
      *const i_hi, const int *const j_hi, const int *const k_hi, const int *const medium, const int
      *const verbosity, const int *const return_status);
}
```

A ball

244 ⟨Forms.hw 237⟩ +≡

```
SCM draw_ball(SCM center, SCM radius, SCM color);
int draw_media_ball(Vector⟨Real⟩ &center, Real &radius, int &color);
int draw_tags_ball(Vector⟨Real⟩ &center, Real &radius);
extern "C"
{
  void f_draw_ball_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
      *const center, const Real *const radius, const int *const color, const int *const box_type, const
      int *const i_lo, const int *const j_lo, const int *const k_lo, const int *const i_hi, const int
      *const j_hi, const int *const k_hi, const int *const medium, const int *const verbosity, const
      int *const return_status);
}
```

An ellipsoid

245 ⟨Forms.hw 237⟩ +≡

```
SCM draw_ellipsoid(SCM focus_1, SCM focus_2, SCM sum, SCM color);
int draw_media_ellipsoid(Vector⟨Real⟩ &focus_1, Vector⟨Real⟩ &focus_2, Real &sum, int &color);
int draw_tags_ellipsoid(Vector⟨Real⟩ &focus_1, Vector⟨Real⟩ &focus_2, Real &sum);
```

```
extern "C"
{
  void f_draw_ellipsoid_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
      *const focus_1, const Real *const focus_2, const Real *const sum, const int *const color, const
      int *const box_type, const int *const i_lo, const int *const j_lo, const int *const k_lo, const int
      *const i_hi, const int *const j_hi, const int *const k_hi, const int *const medium, const int
      *const verbosity, const int *const return_status);
}
```

A cylinder

246 ⟨Forms.hw 237⟩ +≡

```
SCM draw_cylinder(SCM end_1, SCM end_2, SCM radius, SCM color);
int draw_media_cylinder(Vector⟨Real⟩ &end_1, Vector⟨Real⟩ &end_2, Real &radius, int &color);
int draw_tags_cylinder(Vector⟨Real⟩ &end_1, Vector⟨Real⟩ &end_2, Real &radius);
extern "C"
{
  void f_draw_cylinder_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
      *const end_1, const Real *const end_2, const Real *const radius, const int *const color, const
      int *const box_type, const int *const i_lo, const int *const j_lo, const int *const k_lo, const int
      *const i_hi, const int *const j_hi, const int *const k_hi, const int *const medium, const int
      *const verbosity, const int *const return_status);
}
```

A cone

247 ⟨Forms.hw 237⟩ +≡

```
SCM draw_cone(SCM end_1, SCM end_2, SCM radius, SCM color);
int draw_media_cone(Vector⟨Real⟩ &end_1, Vector⟨Real⟩ &end_2, Real &radius, int &color);
int draw_tags_cone(Vector⟨Real⟩ &end_1, Vector⟨Real⟩ &end_2, Real &radius);
extern "C"
{
  void f_draw_cone_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
      *const end_1, const Real *const end_2, const Real *const radius, const int *const color, const
      int *const box_type, const int *const i_lo, const int *const j_lo, const int *const k_lo, const int
      *const i_hi, const int *const j_hi, const int *const k_hi, const int *const medium, const int
      *const verbosity, const int *const return_status);
}
```

A disk

248 ⟨Forms.hw 237⟩ +≡

```
SCM draw_disk(SCM center, SCM normal, SCM radius, SCM thickness, SCM color);
int draw_media_disk(Vector⟨Real⟩ &center, Vector⟨Real⟩ &normal, Real &radius, Real &thickness, int
    &color);
int draw_tags_disk(Vector⟨Real⟩ &center, Vector⟨Real⟩ &normal, Real &radius, Real &thickness);
```

```
extern "C"
{
  void f_draw_disk_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const
      Real *const center, const Real *const normal, const Real *const radius, const Real
      *const thickness, const int *const color, const int *const box_type, const int *const i_lo, const
      int *const j_lo, const int *const k_lo, const int *const i_hi, const int *const j_hi, const int
      *const k_hi, const int *const medium, const int *const verbosity, const int *const return_status);
}
```

An arbitrary ellipsoid

249 ⟨Forms.hw 237⟩ +≡

```
SCM draw_ary_ellipsoid(SCM center, SCM e_1, SCM r_1, SCM e_2, SCM r_2, SCM e_3, SCM
    r_3, SCM color);
int draw_media_ary_ellipsoid(Vector⟨Real⟩ &center, Vector⟨Real⟩ &e_1, Real &r_1, Vector⟨Real⟩
    &e_2, Real &r_2, Vector⟨Real⟩ &e_3, Real &r_3, int &color);
int draw_tags_ary_ellipsoid(Vector⟨Real⟩ &center, Vector⟨Real⟩ &e_1, Real &r_1, Vector⟨Real⟩
    &e_2, Real &r_2, Vector⟨Real⟩ &e_3, Real &r_3);
extern "C"
{
  void f_draw_ary_ellipsoid_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
      *const center, const Real *const e_1, const Real *const r_1, const Real *const e_2, const
      Real *const r_2, const Real *const e_3, const Real *const r_3, const int *const color, const
      int *const box_type, const int *const i_lo, const int *const j_lo, const int *const k_lo, const int
      *const i_hi, const int *const j_hi, const int *const k_hi, const int *const medium, const int
      *const verbosity, const int *const return_status);
}
```

A parallelepiped

250 ⟨Forms.hw 237⟩ +≡

```
SCM draw_parallelepiped(SCM corner, SCM e_1, SCM e_2, SCM e_3, SCM color);
int draw_media_parallelepiped(Vector⟨Real⟩ &corner, Vector⟨Real⟩ &e_1, Vector⟨Real⟩
    &e_2, Vector⟨Real⟩ &e_3, int &color);
int draw_tags_parallelepiped(Vector⟨Real⟩ &corner, Vector⟨Real⟩ &e_1, Vector⟨Real⟩
    &e_2, Vector⟨Real⟩ &e_3);
extern "C"
{
  void f_draw_parallelepiped_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
      *const corner, const Real *const e_1, const Real *const e_2, const Real *const e_3, const
      int *const color, const int *const box_type, const int *const i_lo, const int *const j_lo, const
      int *const k_lo, const int *const i_hi, const int *const j_hi, const int *const k_hi, const int
      *const medium, const int *const verbosity, const int *const return_status);
}
```

A torus

251 ⟨Forms.hw 237⟩ +≡

```
SCM draw_torus(SCM center, SCM normal, SCM r_1, SCM r_2, SCM color);
int draw_media_torus(Vector⟨Real⟩ &center, Vector⟨Real⟩ &normal, Real &radius_1, Real &radius_2, int
    &color);
int draw_tags_torus(Vector⟨Real⟩ &center, Vector⟨Real⟩ &normal, Real &radius_1, Real &radius_2);
extern "C"
{
  void f_draw_torus_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const
      Real *const center, const Real *const normal, const Real *const radius_1, const Real
      *const radius_2, const int *const color, const int *const box_type, const int *const i_lo, const
      int *const j_lo, const int *const k_lo, const int *const i_hi, const int *const j_hi, const int
      *const k_hi, const int *const medium, const int *const verbosity, const int *const return_status);
}
```

A lens

252 ⟨Forms.hw 237⟩ +≡

```
SCM draw_lens(SCM center_1, SCM r_1, SCM center_2, SCM r_2, SCM color);
int draw_media_lens(Vector⟨Real⟩ &center_1, Real &radius_1, Vector⟨Real⟩ &center_2, Real
    &radius_2, int &color);
int draw_tags_lens(Vector⟨Real⟩ &center_1, Real &radius_1, Vector⟨Real⟩ &center_2, Real &radius_2);
extern "C"
{
  void f_draw_lens_(
  const int *const SpaceDim, const int *const margin, const Real *const origin, const Real
      *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const
      Real *const center_1, const Real *const radius_1, const Real *const center_2, const Real
      *const radius_2, const int *const color, const int *const box_type, const int *const i_lo, const
      int *const j_lo, const int *const k_lo, const int *const i_hi, const int *const j_hi, const int
      *const k_hi, const int *const medium, const int *const verbosity, const int *const return_status);
}
```

11.5 Fortran Functions

11.5.1 Update Functions

254 ⟨Forms.hw 237⟩ +≡

```
/* Fortran stuff */
extern "C"
{
  void update_d_(
  const int *const SpaceDim, const int *const ghost_margin, const int *const D_idx, const int
      *const H_idx, const Real *const dt_by_dg,
  const int *const imin_Dx, const int *const jmin_Dx, const int *const kmin_Dx, const int
      *const imax_Dx, const int *const jmax_Dx, const int *const kmax_Dx, const Real *const Dx,
  const int *const imin_Dy, const int *const jmin_Dy, const int *const kmin_Dy, const int
      *const imax_Dy, const int *const jmax_Dy, const int *const kmax_Dy, const Real *const Dy,
  const int *const imin_Dz, const int *const jmin_Dz, const int *const kmin_Dz, const int
      *const imax_Dz, const int *const jmax_Dz, const int *const kmax_Dz, const Real *const Dz,
  const int *const imin_Hx, const int *const jmin_Hx, const int *const kmin_Hx, const int
      *const imax_Hx, const int *const jmax_Hx, const int *const kmax_Hx, const Real *const Hx,
  const int *const imin_Hy, const int *const jmin_Hy, const int *const kmin_Hy, const int
      *const imax_Hy, const int *const jmax_Hy, const int *const kmax_Hy, const Real *const Hy,
  const int *const imin_Hz, const int *const jmin_Hz, const int *const kmin_Hz, const int
      *const imax_Hz, const int *const jmax_Hz, const int *const kmax_Hz, const Real *const Hz,
  const int *const fortran_verbose, const int *const return_status
  );
}
extern "C"
{
  void update_d_upml_(
  const int *const SpaceDim, const int *const ghost_margin, const int *const D_idx, const int
      *const H_idx,
  const int *const imin_Cx, const int *const imax_Cx, const Real *const C1x, const Real
      *const C2x,
  const int *const jmin_Cy, const int *const jmax_Cy, const Real *const C1y, const Real
      *const C2y,
  const int *const kmin_Cz, const int *const kmax_Cz, const Real *const C1z, const Real
      *const C2z,
  const int *const imin_Dx, const int *const jmin_Dx, const int *const kmin_Dx, const int
      *const imax_Dx, const int *const jmax_Dx, const int *const kmax_Dx, const Real *const Dx,
  const int *const imin_Dy, const int *const jmin_Dy, const int *const kmin_Dy, const int
      *const imax_Dy, const int *const jmax_Dy, const int *const kmax_Dy, const Real *const Dy,
  const int *const imin_Dz, const int *const jmin_Dz, const int *const kmin_Dz, const int
      *const imax_Dz, const int *const jmax_Dz, const int *const kmax_Dz, const Real *const Dz,
  const int *const imin_Hx, const int *const jmin_Hx, const int *const kmin_Hx, const int
      *const imax_Hx, const int *const jmax_Hx, const int *const kmax_Hx, const Real *const Hx,
  const int *const imin_Hy, const int *const jmin_Hy, const int *const kmin_Hy, const int
      *const imax_Hy, const int *const jmax_Hy, const int *const kmax_Hy, const Real *const Hy,
  const int *const imin_Hz, const int *const jmin_Hz, const int *const kmin_Hz, const int
      *const imax_Hz, const int *const jmax_Hz, const int *const kmax_Hz, const Real *const Hz,
  const int *const fortran_verbose, const int *const return_status
  );
}
extern "C"
{
  void update_b_(
```

```
const int *const SpaceDim, const int *const ghost_margin, const int *const B_idx, const int
    *const E_idx, const Real *const dt_by_dg,
const int *const imin_Bx, const int *const jmin_Bx, const int *const kmin_Bx, const int
    *const imax_Bx, const int *const jmax_Bx, const int *const kmax_Bx, const Real *const Bx,
const int *const imin_By, const int *const jmin_By, const int *const kmin_By, const int
    *const imax_By, const int *const jmax_By, const int *const kmax_By, const Real *const By,
const int *const imin_Bz, const int *const jmin_Bz, const int *const kmin_Bz, const int
    *const imax_Bz, const int *const jmax_Bz, const int *const kmax_Bz, const Real *const Bz,
const int *const imin_Ex, const int *const jmin_Ex, const int *const kmin_Ex, const int
    *const imax_Ex, const int *const jmax_Ex, const int *const kmax_Ex, const Real *const Ex,
const int *const imin_Ey, const int *const jmin_Ey, const int *const kmin_Ey, const int
    *const imax_Ey, const int *const jmax_Ey, const int *const kmax_Ey, const Real *const Ey,
const int *const imin_Ez, const int *const jmin_Ez, const int *const kmin_Ez, const int
    *const imax_Ez, const int *const jmax_Ez, const int *const kmax_Ez, const Real *const Ez,
const int *const fortran_verbose, const int *const return_status
);
}
extern "C"
{
void update_b_upml_(
const int *const SpaceDim, const int *const ghost_margin,
const int *const B_idx, const int *const E_idx,
const int *const imin_Cx, const int *const imax_Cx, const Real *const C1x, const Real
    *const C2x,
const int *const jmin_Cy, const int *const jmax_Cy, const Real *const C1y, const Real
    *const C2y,
const int *const kmin_Cz, const int *const kmax_Cz, const Real *const C1z, const Real
    *const C2z,
const int *const imin_Bx, const int *const jmin_Bx, const int *const kmin_Bx, const int
    *const imax_Bx, const int *const jmax_Bx, const int *const kmax_Bx, const Real *const Bx,
const int *const imin_By, const int *const jmin_By, const int *const kmin_By, const int
    *const imax_By, const int *const jmax_By, const int *const kmax_By, const Real *const By,
const int *const imin_Bz, const int *const jmin_Bz, const int *const kmin_Bz, const int
    *const imax_Bz, const int *const jmax_Bz, const int *const kmax_Bz, const Real *const Bz,
const int *const imin_Ex, const int *const jmin_Ex, const int *const kmin_Ex, const int
    *const imax_Ex, const int *const jmax_Ex, const int *const kmax_Ex, const Real *const Ex,
const int *const imin_Ey, const int *const jmin_Ey, const int *const kmin_Ey, const int
    *const imax_Ey, const int *const jmax_Ey, const int *const kmax_Ey, const Real *const Ey,
const int *const imin_Ez, const int *const jmin_Ez, const int *const kmin_Ez, const int
    *const imax_Ez, const int *const jmax_Ez, const int *const kmax_Ez, const Real *const Ez,
const int *const fortran_verbose, const int *const return_status
);
}
```

11.5.2 *D*-to-*E* Conversion Functions

Level Zero Functions

256 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
void d_to_e_0_0_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
    int *const E_idx, const int *const D_idx, const Real *const time_e, const Real *const dt, const
    int *const imin_Cx, const int *const imax_Cx, const Real *const C0x, const Real
```

```
*const C1x, const Real *const C3x, const Real *const C4x, const int *const imin_Cy, const int
*const imax_Cy, const Real *const C0y, const Real *const C1y, const Real *const C3y, const
Real *const C4y, const int *const imin_Cz, const int *const imax_Cz, const Real
*const C0z, const Real *const C1z, const Real *const C3z, const Real *const C4z, const
int *const imin_D, const int *const jmin_D, const int *const kmin_D, const int
*const imax_D, const int *const jmax_D, const int *const kmax_D, const Real *const D, const
Real *const E, const int *const medium_E, const int *const watch_d_to_e, const int
*const return_status);
}
extern "C"
{
  void d_to_e_0_n_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
    int *const number_of_auxiliary_fields, const int *const E_idx, const int *const D_idx, const Real
    *const time_e, const Real *const dt, const int *const imin_Cx, const int *const imax_Cx, const
    Real *const C0x, const Real *const C1x, const Real *const C3x, const Real *const C4x, const
    int *const imin_Cy, const int *const imax_Cy, const Real *const C0y, const Real
    *const C1y, const Real *const C3y, const Real *const C4y, const int *const imin_Cz, const
    int *const imax_Cz, const Real *const C0z, const Real *const C1z, const Real
    *const C3z, const Real *const C4z, const int *const imin_D, const int *const jmin_D, const
    int *const kmin_D, const int *const imax_D, const int *const jmax_D, const int
    *const kmax_D, const Real *const D, const Real *const E, const Real *const S, const int
    *const medium_E, const int *const watch_d_to_e, const int *const return_status);
}
```

Higher Level Functions

257 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
  void d_to_e_n_0_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
    int *const E_idx, const int *const D_idx, const Real *const time_e, const Real *const dt, const
    int *const imin_D, const int *const jmin_D, const int *const kmin_D, const int
    *const imax_D, const int *const jmax_D, const int *const kmax_D, const Real *const D, const
    Real *const E, const int *const medium_E, const int *const watch_d_to_e, const int
    *const return_status);
}
extern "C"
{
  void d_to_e_n_n_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
    int *const number_of_auxiliary_fields, const int *const E_idx, const int *const D_idx, const Real
    *const time_e, const Real *const dt, const int *const imin_D, const int *const jmin_D, const
    int *const kmin_D, const int *const imax_D, const int *const jmax_D, const int
    *const kmax_D, const Real *const D, const Real *const E, const Real *const S, const int
    *const medium_E, const int *const watch_d_to_e, const int *const return_status);
}
```

11.5.3 *B*-to-*H* Conversion Functions

Level Zero Functions

259 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
```

```
void b_to_h_0_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
    int *const H_idx, const int *const B_idx, const Real *const time_h, const Real *const dt, const
    int *const imin_Cx, const int *const imax_Cx, const Real *const C0x, const Real
    *const C1x, const Real *const C3x, const Real *const C4x, const int *const imin_Cy, const int
    *const imax_Cy, const Real *const C0y, const Real *const C1y, const Real *const C3y, const
    Real *const C4y, const int *const imin_Cz, const int *const imax_Cz, const Real
    *const C0z, const Real *const C1z, const Real *const C3z, const Real *const C4z, const int
    *const imin_B, const int *const jmin_B, const int *const kmin_B, const int *const imax_B, const
    int *const jmax_B, const int *const kmax_B, const Real *const B, const Real *const H, const
    int *const medium_H, const int *const watch_b_to_h, const int *const return_status);
}
```

Higher Level Functions

260 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
  void b_to_h_n_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
      int *const H_idx, const int *const B_idx, const Real *const time_h, const Real
      *const dt, const int *const imin_B, const int *const jmin_B, const int *const kmin_B, const int
      *const imax_B, const int *const jmax_B, const int *const kmax_B, const Real *const B, const
      Real *const H, const int *const medium_H, const int *const watch_b_to_h, const int
      *const return_status);
}
```

11.5.4 Signal Injection Functions

261 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
  void inject_d_(const int *const dir, const int *const side_number, const int *const margin, const
      int *const new_component, const Real *const x_0, const Real *const y_0, const
      Real *const z_0, const Real *const delta, const Real *const dt_by_delta, const Real
      *const time_h, const int *const i_pml_lo, const int *const j_pml_lo, const int *const k_pml_lo,
      const int *const i_pml_hi, const int *const j_pml_hi, const int *const k_pml_hi, const
      int *const i_Dx_lo, const int *const j_Dx_lo, const int *const k_Dx_lo, const int
      *const i_Dx_hi, const int *const j_Dx_hi, const int *const k_Dx_hi, const Real *const Dx, const
      int *const i_Dy_lo, const int *const j_Dy_lo, const int *const k_Dy_lo, const int
      *const i_Dy_hi, const int *const j_Dy_hi, const int *const k_Dy_hi, const Real *const Dy, const
      int *const watch_inject_d, const int *const return_status);
}
extern "C"
{
  void inject_b_(const int *const dir, const int *const side_number, const int *const margin, const
      int *const new_component, const Real *const x_0, const Real *const y_0, const
      Real *const z_0, const Real *const delta, const Real *const dt_by_delta, const Real
      *const time_e, const int *const i_pml_lo, const int *const j_pml_lo, const int *const k_pml_lo,
      const int *const i_pml_hi, const int *const j_pml_hi, const int *const k_pml_hi, const
      int *const i_Bx_lo, const int *const j_Bx_lo, const int *const k_Bx_lo, const int
      *const i_Bx_hi, const int *const j_Bx_hi, const int *const k_Bx_hi, const Real *const Bx, const
      int *const i_By_lo, const int *const j_By_lo, const int *const k_By_lo, const int
      *const i_By_hi, const int *const j_By_hi, const int *const k_By_hi, const Real *const By, const
      int *const watch_inject_b, const int *const return_status);
}
```

11.5.5 Output Functions

Interpolation Functions

263 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
  void from_edge_to_center_(const int *const spacedim, const int *const dir, const int
      *const n_out, const int *const count, const int *const idx, const int *const i_out_lo, const
      int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const out, const int *const i_fld_lo, const
      int *const j_fld_lo, const int *const k_fld_lo, const int *const i_fld_hi, const int
      *const j_fld_hi, const int *const k_fld_hi, const Real *const fld, const int *const watch, const
      int *const status);
}
extern "C"
{
  void from_int_edge_to_center_(const int *const spacedim, const int *const dir, const int
      *const n_out, const int *const count, const int *const i_out_lo, const int *const j_out_lo, const
      int *const k_out_lo, const int *const i_out_hi, const int *const j_out_hi, const int
      *const k_out_hi, const Real *const out, const int *const i_fld_lo, const int *const j_fld_lo, const
      int *const k_fld_lo, const int *const i_fld_hi, const int *const j_fld_hi, const int
      *const k_fld_hi, const int *const fld, const int *const watch, const int *const status);
}
extern "C"
{
  void from_face_to_center_(const int *const spacedim, const int *const dir, const int
      *const n_out, const int *const count, const int *const idx, const int *const i_out_lo, const
      int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const out, const int *const i_fld_lo, const
      int *const j_fld_lo, const int *const k_fld_lo, const int *const i_fld_hi, const int
      *const j_fld_hi, const int *const k_fld_hi, const Real *const fld, const int *const watch, const
      int *const status);
}
extern "C"
{
  void from_int_face_to_center_(const int *const spacedim, const int *const dir, const int
      *const n_out, const int *const count, const int *const i_out_lo, const int *const j_out_lo, const
      int *const k_out_lo, const int *const i_out_hi, const int *const j_out_hi, const int
      *const k_out_hi, const Real *const out, const int *const i_fld_lo, const int *const j_fld_lo, const
      int *const k_fld_lo, const int *const i_fld_hi, const int *const j_fld_hi, const int
      *const k_fld_hi, const int *const fld, const int *const watch, const int *const status);
}
```

Energy and Flux Functions

264 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
  void evaluate_energy_(const int *const spacedim, const int *const n_out, const int *const count, const
      int *const Ex_idx, const int *const Ey_idx, const int *const Ez_idx, const int
      *const Bx_idx, const int *const By_idx, const int *const Bz_idx, const int *const i_out_lo, const
      int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const out, const int *const watch, const
      int *const status);
```

```
}
extern "C"
{
  void evaluate_flow_(const int *const spacedim, const int *const n_out, const int *const count, const int
      *const Ey_idx, const int *const Bz_idx, const int *const Ez_idx, const int *const By_idx, const
      int *const i_out_lo, const int *const j_out_lo, const int *const k_out_lo, const int
      *const i_out_hi, const int *const j_out_hi, const int *const k_out_hi, const Real *const out, const
      int *const watch, const int *const status);
}
```

Copy Functions

265 ⟨Forms.hw 237⟩ +≡

```
extern "C"
{
  void output_field_copy_(const int *const spacedim, const int *const n_pcmp, const int
      *const idx_pcmp, const int *const n_outp, const int *const idx_outp, const int *const i_out_lo,
      const int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const pcmp, const Real *const outp, const
      int *const watch, const int *const status);
}
#endif
```

11.6 Structure level

This is the central structure of the program. We'll have to work on this, because as it is, it is too clumsy. This has been copied from SHAPES.

266 ⟨ Structure level 266 ⟩ ≡

```
struct level {
    /* level number */
  int number;
      /* electric and magnetic time and time step */
  Real time_e, time_h, dt;
      /* physics regions: UPML boundaries and signal injection */
  Vector⟨Real⟩ pml_xyz_min;
  Vector⟨Real⟩ pml_xyz_max;
  Vector⟨Real⟩ signal_xyz_min;
  Vector⟨Real⟩ signal_xyz_max;
  IntVect signal_ijk_lo;
  IntVect signal_ijk_hi;
      /* general grid parameters */
  Box domain;
  Vector⟨Real⟩ origin;
  Real delta;    /* identical grid spacing in every direction */
  IntVect ijk_min;
  IntVect ijk_max;
  Vector⟨Real⟩ xyz_min;
  Vector⟨Real⟩ xyz_max;
      /* level distribution and structure */
  Vector⟨Box⟩ vector_of_boxes;
  Vector⟨int⟩ vector_of_processes;
  DisjointBoxLayout box_layout;
  LayoutIterator layout_iterator;
  DataIterator data_iterator;
  LevelData⟨BaseFab⟨int⟩⟩ tags;
  IntVectSet tag_set;
  Vector⟨Vector⟨Vector⟨DataIndex⟩⟩⟩ TFR_face_boxes;
      /* fields */
      /* magnetic */
  LevelData⟨FluxBox⟩ H, B;
      /* electric and electric-auxiliary */
  LevelData⟨CurlBox⟩ E, D, S;
      /* output */
  LevelData⟨FArrayBox⟩ Out, OutWrite;
      /* media */
  LevelData⟨EdgeFab⟨int⟩⟩ medium_E;
  LevelData⟨FaceFab⟨int⟩⟩ medium_H;
      /* old/new field index */
  int E_idx[2], D_idx[2], B_idx[2], H_idx[2];
      /* UPMLs */
  Vector⟨Real⟩ C0x;
  Vector⟨Real⟩ C1x;
  Vector⟨Real⟩ C2x;
  Vector⟨Real⟩ C3x;
  Vector⟨Real⟩ C4x;
  Vector⟨Real⟩ C0y;
  Vector⟨Real⟩ C1y;
  Vector⟨Real⟩ C2y;
```

```
    Vector⟨Real⟩ C3y;
    Vector⟨Real⟩ C4y;
    Vector⟨Real⟩ C0z;
    Vector⟨Real⟩ C1z;
    Vector⟨Real⟩ C2z;
    Vector⟨Real⟩ C3z;
    Vector⟨Real⟩ C4z;
  };
```

This code is cited in chunk 22.

This code is used in chunk 240.

12 Chombo Extensions

The extensions are needed to handle face, e.g., $\boldsymbol{B}$, and edge, e.g., $\boldsymbol{E}$, mounted data, also to orchestrate the movement of such data between levels, with special attention paid to level boundaries. Throughout all these operations we must ensure that the condition $\boldsymbol{\nabla} \cdot \boldsymbol{E} = \boldsymbol{\nabla} \cdot \boldsymbol{B} = 0$ is maintained.

The extensions are as follows

SCMParmParse Section 12.1, page 350—a class, similar to Chombo **ParmParse** that parses data from a Scheme environment.

EdgeFab⟨T⟩ Section 12.2, page 398—a class template for handling edge mounted fields.

headers Section 12.2.1, page 400

implementation Section 12.2.2, page 403

FaceFab⟨T⟩ Section 12.3, page 414—a class template for handling face mounted fields.

headers Section 12.3.1, page 414

implementation Section 12.3.2, page 415

CurlBox Section 12.4, page 421—a class for handling edge mounted **FArrayBox** fields.

headers Section 12.4.1, page 421

implementation Section 12.4.2, page 422

12.1 Scheme Parser

This class, **SCMParmParse**, is similar to Chombo **ParmParse**, but instead of parsing stuff from a file, it parses it from Scheme that runs within this program. So, the idea is that we have just one file that contains Scheme defines and code, we read it into the Scheme subprocess right at the beginning and then the program picks up what it wants from Scheme by calling various **SCMParmParse** methods.

Although **SCMParmParse** can be thought of as a Scheme utility, it can be also viewed as a Chombo extension. For this reason, it gets its own separate code file, header file and is placed here in the Chombo extensions section.

12.1.1 SCMParmParse Class Headers

269 ⟨SCMParmParse.hw 269⟩ ≡

```
#ifndef CH_SCMPARMPARSE_H
# define CH_SCMPARMPARSE_H
      /* Standard UNIX includes */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
      /* C++ includes */
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>
#include <fstream>
#include <ios>
#include <vector>
#include <cassert>
#include <cstdlib>
      /* Guile includes */
#include <libguile.h>
      /* Chombo includes */
#ifdef CH_SPACEDIM
#include <MayDay.H>
#include <Misc.H>
#include <Vector.H>
#include <parstream.H>     /* contains pout() */
#include <SPMD.H>     /* includes "mpi.h" if needed */
#endif
      /* Error codes */
# defineERR_PARFILE_NULL 121
# defineERR_PARFILE_NOT_READABLE 122
# defineERR_PARFILE_LOAD_FAILED 123
      /* Common functions are from the std class */
  using std::cout;
  using std::cerr;
  using std::endl;
  using std::flush;
  using std::setw;
  using std::numeric_limits;
  using std::ifstream;
  using std::string;
  using std::vector;
```

```
using std::min;
using std::max;
    /* The class declaration */
class SCMParmParse {
public: ⟨SCMParmParse headers: constructors and destructors 270⟩
  ⟨SCMParmParse headers: queries and extractors 271⟩
protected: ⟨SCMParmParse headers: protected members 272⟩
};
#endif    /* CH_SCMPARMPARSE_H */
```

SCMParmParse Constructors and Destructors

270 ⟨**SCMParmParse** headers: constructors and destructors 270⟩ ≡

```
SCMParmParse(int argc, char **argv, const char *prefix = Λ, const char *parfile = Λ);
SCMParmParse(const char *prefix = Λ);
~SCMParmParse();
```

This code is used in chunk 269.

SCMParmParse Queries and Extractors

271 ⟨**SCMParmParse** headers: queries and extractors 271⟩ ≡

```
bool contains(const char *name);
bool contains(const string &name);
bool isProcedure(const char *name);
bool isClosure(const char *name);
bool isThunk(const char *name);
SCM variable(const char *name);
int countval(const char *name);
    /* getting or putting integers */
void fastget(SCM variable, int &ref);
void fastput(SCM variable, const int ref);
void get(const char *name, int &ref);
void put(const char *name, const int ref);
bool query(const char *name, int &ref);
bool queryput(const char *name, const int ref);
    /* getting or putting floats */
void fastget(SCM variable, float &ref);
void fastput(SCM variable, const float ref);
void get(const char *name, float &ref);
void put(const char *name, const float ref);
bool query(const char *name, float &ref);
bool queryput(const char *name, const float ref);
    /* getting or putting doubles */
void fastget(SCM variable, double &ref);
void fastput(SCM variable, const double ref);
void get(const char *name, double &ref);
void put(const char *name, const double ref);
bool query(const char *name, double &ref);
bool queryput(const char *name, const double ref);
    /* getting or putting strings */
void fastget(SCM variable, string &ref);
void get(const char *name, string &ref);
void fastput(SCM variable, const string ref);
```

```
void fastput(SCM variable, const char *ref);
void put(const char *name, const string ref);
void put(const char *name, const char *ref);
bool query(const char *name, string &ref);
bool queryput(const char *name, const string ref);
bool queryput(const char *name, const char *ref);
    /* vector of ints */
void fastgetarr(SCM variable, vector⟨int⟩ &ref);
void fastputarr(SCM variable, vector⟨int⟩ &ref);
#ifdef CH_SPACEDIM
void fastgetarr(SCM variable, Vector⟨int⟩ &ref);
void fastputarr(SCM variable, Vector⟨int⟩ &ref);
#endif
void getarr(const char *name, vector⟨int⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, vector⟨int⟩ &ref, int start_ix, int num_val);
    /* vector of floats */
void fastgetarr(SCM variable, vector⟨float⟩ &ref);
void fastputarr(SCM variable, vector⟨float⟩ &ref);
#ifdef CH_SPACEDIM
void fastgetarr(SCM variable, Vector⟨float⟩ &ref);
void fastputarr(SCM variable, Vector⟨float⟩ &ref);
#endif
void getarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val);
    /* vector of double */
void fastgetarr(SCM variable, vector⟨double⟩ &ref);
void fastputarr(SCM variable, vector⟨double⟩ &ref);
#ifdef CH_SPACEDIM
void fastgetarr(SCM variable, Vector⟨double⟩ &ref);
void fastputarr(SCM variable, Vector⟨double⟩ &ref);
#endif
void getarr(const char *name, vector⟨double⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, vector⟨double⟩ &ref, int start_ix, int num_val);
    /* vector of strings */
void getarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val);
#ifdef CH_SPACEDIM
void getarr(const char *name, Vector⟨int⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, Vector⟨int⟩ &ref, int start_ix, int num_val);
void getarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val);
void getarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val);
bool queryarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val);
void getarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val);
#endif
void print_prefix();
```

This code is used in chunk 269.

SCMParmParse Protected Members

272 ⟨**SCMParmParse** headers: protected members 272⟩ ≡

```
string prefix;
bool does_scm_symbol_exist(const char *name);
```

```
off_t is_file_readable(const char *file_name);
```

This code is used in chunk 269.

12.1.2 SCMParmParse Class Implementation

273 ⟨ SCMParmParse.c 273 ⟩ ≡

```
#include <SCMParmParse.H>
  ⟨ SCMParmParse implementation: constructors and destructors 274 ⟩
  ⟨ SCMParmParse implementation: queries and extractors 275 ⟩
  ⟨ SCMParmParse implementation: protected members 291 ⟩
```

SCMParmParse Constructors and Destructors

274 ⟨ **SCMParmParse** implementation: constructors and destructors 274 ⟩ ≡

```
  SCMParmParse::SCMParmParse(int argc, char **argv, const char *c_prefix, const char *parfile)
  {
    if (parfile ≡ Λ) {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::SCMParmParse): NULL file name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::SCMParmParse): NULL file name" << endl;
#endif
#ifdef CH_MPI
      MPI_Abort(Chombo_MPI::comm, ERR_PARFILE_NULL);
#else
      abort();
#endif
    }
    if (¬is_file_readable(parfile)) {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::SCMParmParse): file " << parfile <<
          " cannot be opened for reading" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::SCMParmParse): file " << parfile <<
          " cannot be opened for reading" << endl;
#endif
#ifdef CH_MPI
      MPI_Abort(Chombo_MPI::comm, ERR_PARFILE_NOT_READABLE);
#else
      abort();
#endif
    }
    SCM status = scm_c_primitive_load(parfile);
    if (scm_is_false(status)) {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::SCMParmParse): primitive load failed" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::SCMParmParse): primitive load failed" << endl;
#endif
#ifdef CH_MPI
      MPI_Abort(Chombo_MPI::comm, ERR_PARFILE_LOAD_FAILED);
#else
      abort();
```

```
#endif
    }
    else {
      prefix.clear();
      if (c_prefix) prefix += c_prefix;
    }
  }
  SCMParmParse::SCMParmParse(const char *c_prefix)
  {
    prefix.clear();
    if (c_prefix) prefix += c_prefix;
  }
  SCMParmParse::~SCMParmParse()
  {
    prefix.clear();
  }
```

This code is cited in chunk 6.

This code is used in chunk 273.

SCMParmParse Queries and Extractors

275 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ ≡

```
  bool SCMParmParse::contains(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::contains): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::contains): NULL name" << endl;
#endif
      return false;
    }
    return does_scm_symbol_exist(fullname.c_str());
  }
  bool SCMParmParse::contains(const string &name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (¬name.empty()) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::contains): empty name" << endl;
```

```
#else
      cerr << "\n\tERROR(SCMParmParse::contains): empty name" << endl;
#endif
      return false;
    }
    return does_scm_symbol_exist(fullname.c_str());
  }
```

See also chunks 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, and 290.
This code is used in chunk 273.

¶ Here we go beyond what Chombo does and check, for certain things Schemish.

276 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
  bool SCMParmParse::isProcedure(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isProcedure): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isProcedure): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      return scm_to_bool(scm_procedure_p(variable));
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isProcedure): symbol does not exist" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isProcedure): symbol does not exist" << endl;
#endif
      return false;
    }
  }
  bool SCMParmParse::isClosure(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
```

```
      pout() << "\n\tERROR(SCMParmParse::isClosure): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isClosure): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      return scm_to_bool(scm_closure_p(variable));
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isClosure): symbol does not exist" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isClosure): symbol does not exist" << endl;
#endif
      return false;
    }
  }
  bool SCMParmParse::isThunk(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isThunk): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isThunk): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      return scm_to_bool(scm_thunk_p(variable));
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#endif
      return false;
    }
  }
  SCM SCMParmParse::variable(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
```

```
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::variable): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::variable): NULL name" << endl;
#endif
      return SCM_UNDEFINED;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      return scm_c_lookup(fullname.c_str());
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#endif
      return SCM_UNDEFINED;
    }
  }
```

277 ¶⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
  int SCMParmParse::countval(const char *name)
  {
    int count = 0;
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::countval): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::countval): NULL name" << endl;
#endif
      return −1;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_true(scm_number_p(variable)) ∨ scm_is_true(scm_string_p(variable))) {
        count = 1;
      }
      else    /* some compound variables */
      {
          /* We test for
```

1. list

2. vector

3. uniform vector

4. array—inclucdes typed arrays

```
*/
        if (scm_is_true(scm_list_p(variable))) count = scm_to_int(scm_length(variable));
        else if (scm_is_true(scm_vector_p(variable))) count = scm_to_int(scm_vector_length(variable));
        else if (scm_is_true(scm_uniform_vector_p(variable)))
          count = scm_to_int(scm_uniform_vector_length(variable));
        else if (scm_is_true(scm_array_p(variable, SCM_UNDEFINED))) {
          int rank = (int) scm_c_array_rank(variable);
          count = 1;
          SCM dimensions = scm_array_dimensions(variable);
          for (int dim = 0; dim < rank; dim++) {
            SCM item = scm_list_ref(dimensions, scm_from_int(dim));
            if (scm_is_true(scm_number_p(item))) count = count * scm_to_int(item);
            else {
              int lo = scm_to_int(scm_list_ref(item, scm_from_int(0)));
              int hi = scm_to_int(scm_list_ref(item, scm_from_int(1)));
              count = count * (hi - lo + 1);
            }
          }
        }
      }
    }
    return count;
  }
```

¶ *fastget*, *get*, *fastput*, *put*, *query* and *queryput* on simple variables. **int** case first.

278 ⟨**SCMParmParse** implementation: queries and extractors 275⟩ +≡

```
  void SCMParmParse::fastget(SCM variable, int &ref)
  {
    ref = scm_to_int(scm_variable_ref(variable));
  }
  void SCMParmParse::get(const char *name, int &ref)
  {
      /* Checks, but does not tell. Prevents crash. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#endif
      return;
```

```
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_integer(variable)) ref = (int) scm_to_double(variable);
    }
  }
  void SCMParmParse::fastput(SCM variable, const int ref)
  {
    scm_variable_set_x(variable, scm_from_int(ref));
  }
  void SCMParmParse::put(const char *name, const int ref)
  {     /* Put ref into name if the Scheme variable named so exists and is an integer. Bail out silently
          otherwise. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_integer(scm_variable_ref(variable))) scm_variable_set_x(variable, scm_from_int(ref));
    }
  }
  bool SCMParmParse::query(const char *name, int &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_integer(variable)) {
```

```
        ref = (int) scm_to_double(variable);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const int ref)
  {
      /* Puts the ref value into name. Checks and returns false if problems are encountered. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_integer(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_int(ref));
        return true;
      }
      else {
        return false;
      }
    }
    else {
      return false;
    }
  }
```

¶ This is the same as above, but **int** is replaced with **float**.

279 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
  void SCMParmParse::fastget(SCM variable, float &ref)
  {
    ref = (float) scm_to_double(scm_variable_ref(variable));
  }
  void SCMParmParse::get(const char *name, float &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
```

```
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_real(variable)) ref = (float) scm_to_double(variable);
    }
  }
  void SCMParmParse::fastput(SCM variable, const float ref)
  {
    scm_variable_set_x(variable, scm_from_double((double) ref));
  }
  void SCMParmParse::put(const char *name, const float ref)
  {     /* Put ref into name if the Scheme variable named so exists and is an integer. Bail out silently
          otherwise. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_real(scm_variable_ref(variable)))
        scm_variable_set_x(variable, scm_from_double((double) ref));
    }
  }
  bool SCMParmParse::query(const char *name, float &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
```

```
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_real(variable)) {
        ref = (float) scm_to_double(variable);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const float ref)
  {
      /* Puts the ref value into name. Checks and returns false if problems are encountered. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_real(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_double((double) ref));
        return true;
      }
      else {
        return false;
      }
    }
    else {
      return false;
    }
  }
```

¶ This is the same as above, but **float** is replaced with **double**.

280 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::fastget(SCM variable, double &ref)
{
  ref = scm_to_double(scm_variable_ref(variable));
}
void SCMParmParse::get(const char *name, double &ref)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_real(variable)) ref = scm_to_double(variable);
  }
}
void SCMParmParse::fastput(SCM variable, const double ref)
{
  scm_variable_set_x(variable, scm_from_double(ref));
}
void SCMParmParse::put(const char *name, const double ref)
{   /* Put ref into name if the Scheme variable named so exists and is an integer. Bail out silently
       otherwise. */
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_c_lookup(fullname.c_str());
    if (scm_is_real(scm_variable_ref(variable))) scm_variable_set_x(variable, scm_from_double(ref));
  }
```

```
  }
  bool SCMParmParse::query(const char *name, double &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_real(variable)) {
        ref = scm_to_double(variable);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const double ref)
  {
      /* Puts the ref value into name. Checks and returns false if problems are encountered. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_real(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_double(ref));
        return true;
      }
      else {
        return false;
```

```
      }
    }
    else {
      return false;
    }
  }
```

¶ This is similar to the above, but this time we get or put a string.

281 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
  void SCMParmParse::fastget(SCM variable, string &ref)
  {
    char *buffer = scm_to_locale_string(scm_variable_ref(variable));
    ref.clear();
    ref += buffer;
    free(buffer);
  }
  void SCMParmParse::get(const char *name, string &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::get): NULL name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::get): NULL name" ≪ endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_string(variable)) {
        char *buffer = scm_to_locale_string(variable);
        ref.clear();
        ref += buffer;
        free(buffer);
      }
    }
  }
  void SCMParmParse::fastput(SCM variable, const string ref)
  {
    scm_variable_set_x(variable, scm_from_locale_string(ref.c_str()));
  }
  void SCMParmParse::fastput(SCM variable, const char *ref)
  {
    scm_variable_set_x(variable, scm_from_locale_string(ref));
  }
```

```
void SCMParmParse::put(const char *name, const string ref)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_c_lookup(fullname.c_str());
    if (scm_is_string(scm_variable_ref(variable))) {
      scm_variable_set_x(variable, scm_from_locale_string(ref.c_str()));
    }
  }
}
void SCMParmParse::put(const char *name, const char *ref)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_c_lookup(fullname.c_str());
    if (scm_is_string(scm_variable_ref(variable))) {
      scm_variable_set_x(variable, scm_from_locale_string(ref));
    }
  }
}
bool SCMParmParse::query(const char *name, string &ref)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
```

```
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_string(variable)) {
        char *buffer = scm_to_locale_string(variable);
        ref.clear();
        ref += buffer;
        free(buffer);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const string ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_string(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_locale_string(ref.c_str()));
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const char *ref)
  {
```

```
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::query): NULL name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::query): NULL name" ≪ endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_string(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_locale_string(ref));
        return true;
      }
      else return false;
    }
    else return false;
  }
```

¶ Process lists, vectors and uniform vectors of **int**s.

282 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
  void SCMParmParse::fastgetarr(SCM variable, vector⟨int⟩ &ref)
  {    /* Transfer data from a uniform Scheme vector to vector⟨int⟩. Both vectors are assumed to be of the
          same size. All vector content is transferred. Nothin is checked. Will crash if anything is wrong. */
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      ref[count] = scm_to_int(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
    }
  }
  void SCMParmParse::fastputarr(SCM variable, vector⟨int⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_int(ref[count]));
    }
  }
#ifdef CH_SPACEDIM
  void SCMParmParse::fastgetarr(SCM variable, Vector⟨int⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      ref[count] = scm_to_int(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
```

```
    }
  }
  void SCMParmParse::fastputarr(SCM variable, Vector<int> &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_int(ref[count]));
    }
  }
#endif
  void SCMParmParse::getarr(const char *name, vector<int> &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::getarr): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): NULL name" << endl;
#endif
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::getarr): list value not an integer" << endl;
#else
            cerr << "\n\tERROR(SCMParmParse::getarr): list value not an integer" << endl;
#endif
```

```
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else
#ifdef CH_SPACEDIM
            pout() ≪ "\n\tERROR(SCMParmParse::getarr): vector value not an integer" ≪ endl;
#else
          cerr ≪ "\n\tERROR(SCMParmParse::getarr): vector value not an integer" ≪ endl;
#endif
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else
#ifdef CH_SPACEDIM
            pout() ≪ "\n\tERROR(SCMParmParse::getarr): uniform vector value not an integer" ≪
                endl;
#else
          cerr ≪ "\n\tERROR(SCMParmParse::getarr): uniform vector value not an integer" ≪
              endl;
#endif
        }
      }
    }
    else
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" ≪
          endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" ≪ endl;
#endif
  }
  exit: return;
}
bool SCMParmParse::queryarr(const char *name, vector⟨int⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
```

```
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): list value not an integer" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): list value not an integer" << endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
```

```
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not an integer" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not an integer" << endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not\
                  an integer" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not an integer" <<
                  endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else {
#ifdef CH_SPACEDIM
        pout() << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
            endl;
#else
        cerr << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
            endl;
#endif
        return_value = false;
        goto exit;
      }
    }
    else return_value = false;
  exit: return return_value;
  }
```

¶ Process lists, vectors and uniform vectors of **float**s.

283 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::fastgetarr(SCM variable, vector⟨float⟩ &ref)
{    /* Transfer data from a uniform Scheme vector to vector⟨float⟩. Both vectors are assumed to be of
        the same size. All vector content is transferred. Nothin is checked. Will crash if anything is wrong. */
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    ref[count] = (float) scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
  }
}
void SCMParmParse::fastputarr(SCM variable, vector⟨float⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double((double) ref[count]));
  }
}
#ifdef CH_SPACEDIM
void SCMParmParse::fastgetarr(SCM variable, Vector⟨float⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    ref[count] = (float) scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
  }
}
void SCMParmParse::fastputarr(SCM variable, Vector⟨float⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double((double) ref[count]));
  }
}
#endif
void SCMParmParse::getarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#endif
    goto exit;
  }
```

```
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() ≪ "\n\tERROR(SCMParmParse::getarr): list value not a float" ≪ endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::getarr): list value not a float" ≪ endl;
#endif
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() ≪ "\n\tERROR(SCMParmParse::getarr): vector value not a float" ≪ endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::getarr): vector value not a float" ≪ endl;
#endif
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
```

```
            if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a float" <<
                  endl;
#else
            cerr << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a float" << endl;
#endif
          }
        }
      }
      else
#ifdef CH_SPACEDIM
        pout() << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" <<
            endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" << endl;
#endif
    }
  exit: return;
  }
  bool SCMParmParse::queryarr(const char *name, vector<float> &ref, int start_ix, int num_val)
  {
    string fullname;
    bool return_value = true;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
```

```
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
              else {
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a float" << endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): list value not a float" << endl;
#endif
                return_value = false;
                goto exit;
              }
            }
          }
        }
        else if (scm_is_vector(variable)) {
          int length = scm_to_int(scm_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
              else {
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not a float" << endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not a float" << endl;
#endif
                return_value = false;
                goto exit;
              }
            }
          }
        }
        else if (scm_is_true(scm_uniform_vector_p(variable))) {
          int length = scm_to_int(scm_uniform_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
              else {
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a float" <<
                    endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a float" <<
                    endl;
#endif
                return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
          endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
          endl;
#endif
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

¶ Process lists, vectors and uniform vectors of **double**s.

284 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::fastgetarr(SCM variable, vector⟨double⟩ &ref)
{   /* Transfer data from a uniform Scheme vector to vector⟨double⟩. Both vectors are assumed to be of
       the same size. All vector content is transferred. Nothin is checked. Will crash if anything is wrong. */
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    ref[count] = scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
  }
}
void SCMParmParse::fastputarr(SCM variable, vector⟨double⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double(ref[count]));
  }
}
#ifdef CH_SPACEDIM
void SCMParmParse::fastgetarr(SCM variable, Vector⟨double⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    ref[count] = scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
  }
}
void SCMParmParse::fastputarr(SCM variable, Vector⟨double⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
```

```
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double(ref[count]));
    }
  }
#endif
  void SCMParmParse::getarr(const char *name, vector<double> &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#endif
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::getarr): list value not a double" << endl;
#else
            cerr << "\n\tERROR(SCMParmParse::getarr): list value not a double" << endl;
#endif
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
```

```
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = scm_to_double(item);
              else
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::getarr): vector value not a double" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::getarr): vector value not a double" << endl;
#endif
            }
          }
        }
        else if (scm_is_true(scm_uniform_vector_p(variable))) {
          int length = scm_to_int(scm_uniform_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = scm_to_double(item);
              else
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a double" <<
                    endl;
#else
              cerr << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a double" << endl;
#endif
            }
          }
        }
        else
#ifdef CH_SPACEDIM
          pout() << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" <<
              endl;
#else
        cerr << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" << endl;
#endif
      }
    exit: return;
    }
    bool SCMParmParse::queryarr(const char *name, vector<double> &ref, int start_ix, int num_val)
    {
      string fullname;
      bool return_value = true;
      fullname.clear();
      if (¬prefix.empty()) {
        fullname += prefix;
        fullname += ".";
      }
      if (name) fullname += name;
      else {
#ifdef CH_SPACEDIM
        pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
```

```
#else
        cerr << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#endif
        return_value = false;
        goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
        SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
            /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
        if (scm_is_true(scm_list_p(variable))) {
            int length = scm_to_int(scm_length(variable));
            if ((0 <= start_ix) ∧ (start_ix <= length - 1)) {
                int num_val_available = length - start_ix;
                ref.resize(min(num_val_available, num_val));
                for (int count = 0; count < min(num_val_available, num_val); count++) {
                    SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
                    if (scm_is_real(item)) ref[count] = scm_to_double(item);
                    else {
#ifdef CH_SPACEDIM
                        pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a double" << endl;
#else
                        cerr << "\n\tERROR(SCMParmParse::queryarr): list value not a double" << endl;
#endif
                        return_value = false;
                        goto exit;
                    }
                }
            }
        }
        else if (scm_is_vector(variable)) {
            int length = scm_to_int(scm_vector_length(variable));
            if ((0 <= start_ix) ∧ (start_ix <= length - 1)) {
                int num_val_available = length - start_ix;
                ref.resize(min(num_val_available, num_val));
                for (int count = 0; count < min(num_val_available, num_val); count++) {
                    SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
                    if (scm_is_real(item)) ref[count] = scm_to_double(item);
                    else {
#ifdef CH_SPACEDIM
                        pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not a double" << endl;
#else
                        cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not a double" << endl;
#endif
                        return_value = false;
                        goto exit;
                    }
```

```
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
#ifdef CH_SPACEDIM
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a double" ≪
                endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a double" ≪
                endl;
#endif
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" ≪
          endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" ≪
          endl;
#endif
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

¶ Process lists and vectors of **string**s.

285 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
```

```
#ifdef CH_SPACEDIM
        pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#else
        cerr << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#endif
        goto exit;
      }
      ref.clear();
      if (does_scm_symbol_exist(fullname.c_str())) {
        SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
            /* This may be either

   1. a list

   2. a vector

   */
        if (scm_is_true(scm_list_p(variable))) {
          int length = scm_to_int(scm_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_string(item)) {
                char *buffer = scm_to_locale_string(item);
                ref[count].clear();
                ref[count] += buffer;
                free(buffer);
              }
              else
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::getarr): list value not a string" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::getarr): list value not a string" << endl;
#endif
            }
          }
        }
        else if (scm_is_vector(variable)) {
          int length = scm_to_int(scm_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_string(item)) {
                char *buffer = scm_to_locale_string(item);
                ref[count].clear();
                ref[count] += buffer;
                free(buffer);
              }
              else
#ifdef CH_SPACEDIM
```

```
            pout() << "\n\tERROR(SCMParmParse::getarr): vector value not a string" << endl;
#else
            cerr << "\n\tERROR(SCMParmParse::getarr): vector value not a string" << endl;
#endif
          }
        }
      }
      else
#ifdef CH_SPACEDIM
        pout() << "\n\tERROR(SCMParmParse::getarr): not a list or vector" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): not a list or vector" << endl;
#endif
    }
  exit: return;
  }
  bool SCMParmParse::queryarr(const char *name, vector<string> &ref, int start_ix, int num_val)
  {
    string fullname;
    bool return_value = true;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_string(item)) {
              char *buffer = scm_to_locale_string(item);
```

```
                ref[count].clear();
                ref[count] += buffer;
                free(buffer);
              }
              else {
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a string" << endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): list value not a string" << endl;
#endif
                return_value = false;
                goto exit;
              }
            }
          }
        }
        else if (scm_is_vector(variable)) {
          int length = scm_to_int(scm_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_string(item)) {
                char *buffer = scm_to_locale_string(item);
                ref[count].clear();
                ref[count] += buffer;
                free(buffer);
              }
              else {
#ifdef CH_SPACEDIM
                pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not a string" << endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not a string" << endl;
#endif
                return_value = false;
                goto exit;
              }
            }
          }
        }
        else {
#ifdef CH_SPACEDIM
          pout() << "\n\tERROR(SCMParmParse::queryarr): not a list or vector" << endl;
#else
          cerr << "\n\tERROR(SCMParmParse::queryarr): not a list or vector" << endl;
#endif
          return_value = false;
          goto exit;
        }
      }
      else return_value = false;
    exit: return return_value;
    }
```

¶ Trivial overload: **vector** → **Vector** and *cerr* → *pout*( ).

286 ⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
#ifdef CH_SPACEDIM
  void SCMParmParse::getarr(const char *name, Vector<int> &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
      pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else pout() << "\n\tERROR(SCMParmParse::getarr): list value not an integer" << endl;
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else
              pout() << "\n\tERROR(SCMParmParse::getarr): vector value not an integer" << endl;
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
```

```
            else
              pout() << "\n\tERROR(SCMParmParse::getarr): uniform vector value not an integer" <<
                  endl;
          }
        }
      }
      else pout() << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" <<
            endl;
    }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector<int> &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
    return_value = false;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else {
            pout() << "\n\tERROR(SCMParmParse::queryarr): list value not an integer" << endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
```

```
            else {
              pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not an integer" << endl;
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
              pout() << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not\
                  an integer" << endl;
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else {
        pout() << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
            endl;
        return_value = false;
        goto exit;
      }
    }
    else return_value = false;
  exit: return return_value;
  }
```

287 ¶⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
```

```
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣list␣value␣not␣a␣float" ≪ endl;
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣a␣float" ≪ endl;
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else
            pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣uniform␣vector␣value␣not␣a␣float" ≪
                endl;
        }
      }
    }
    else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
  }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
```

```
if (name) fullname += name;
else {
  pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
  return_value = false;
  goto exit;
}
ref.clear();
if (does_scm_symbol_exist(fullname.c_str())) {
  SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
  if (scm_is_true(scm_list_p(variable))) {
    int length = scm_to_int(scm_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
        else {
          pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a float" << endl;
          return_value = false;
          goto exit;
        }
      }
    }
  }
  else if (scm_is_vector(variable)) {
    int length = scm_to_int(scm_vector_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
        else {
          pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not a float" << endl;
          return_value = false;
          goto exit;
        }
      }
    }
  }
  else if (scm_is_true(scm_uniform_vector_p(variable))) {
    int length = scm_to_int(scm_uniform_vector_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
        else {
          pout() << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a float" <<
              endl;
          return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else {
      pout() << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
          endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

288 ¶⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else pout() << "\n\tERROR(SCMParmParse::getarr): list value not a double" << endl;
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
```

```
            if (scm_is_real(item)) ref[count] = scm_to_double(item);
            else pout() << "\n\tERROR(SCMParmParse::getarr): vector value not a double" << endl;
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = scm_to_double(item);
            else
              pout() << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a double" <<
                    endl;
          }
        }
      }
      else pout() << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" <<
            endl;
    }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
    return_value = false;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
            pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a double" << endl;
            return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣vector␣value␣not␣a␣double" ≪ endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣uniform␣vector␣value␣not␣a␣double" ≪
                endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else {
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

289 ¶⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val)
{
  string fullname;
```

```
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣name␣empty" ≪ endl;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_string(item)) {
            char *buffer = scm_to_locale_string(item);
            ref[count].clear();
            ref[count] += buffer;
            free(buffer);
          }
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣list␣value␣not␣a␣string" ≪ endl;
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_string(item)) {
            char *buffer = scm_to_locale_string(item);
            ref[count].clear();
            ref[count] += buffer;
            free(buffer);
          }
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣a␣string" ≪ endl;
        }
      }
    }
    else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list␣or␣vector" ≪ endl;
  }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val)
{
```

```
string fullname;
bool return_value = true;
fullname.clear();
if (¬prefix.empty()) {
  fullname += prefix;
  fullname += ".";
}
if (name) fullname += name;
else {
  pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
  return_value = false;
  goto exit;
}
ref.clear();
if (does_scm_symbol_exist(fullname.c_str())) {
  SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
  if (scm_is_true(scm_list_p(variable))) {
    int length = scm_to_int(scm_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_string(item)) {
          char *buffer = scm_to_locale_string(item);
          ref[count].clear();
          ref[count] += buffer;
          free(buffer);
        }
        else {
          pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣a␣string" ≪ endl;
          return_value = false;
          goto exit;
        }
      }
    }
  }
  else if (scm_is_vector(variable)) {
    int length = scm_to_int(scm_vector_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_string(item)) {
          char *buffer = scm_to_locale_string(item);
          ref[count].clear();
          ref[count] += buffer;
          free(buffer);
        }
        else {
          pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣vector␣value␣not␣a␣string" ≪ endl;
          return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else {
      pout() << "\n\tERROR(SCMParmParse::queryarr): not a list or vector" << endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
#endif    /* CH_SPACEDIM */
```

290 ¶⟨ **SCMParmParse** implementation: queries and extractors 275 ⟩ +≡

```
void SCMParmParse::print_prefix()
{
#ifdef CH_SPACEDIM
    pout() << "SCMParmParse::prefix = " << prefix << endl;
#else
    cout << "SCMParmParse::prefix = " << prefix << endl;
#endif
}
```

SCMParmParse Protected Members

Function *does_scm_symbol_exist*, borrowed from [9] with one small modification that fixes the guile-1.8.1 note about deprecated features, does a job that is similar to *scm_c_lookup*( )[15] with one difference: it does not bomb out if the symbol is not found, just returns *false*.

Implementation notes:

scm_str2symbol This function is defined on `"libguile/discouraged.h"`. I have replaced it with *scm_string_to_symbol*.

scm_sym2var This is a Guile internal, defined on `"libguile/modules.h"`.

scm_current_module_lookup_closure( ) Ditto.

291 ⟨ **SCMParmParse** implementation: protected members 291 ⟩ ≡

```
    /* This function comes from [9], with some modifications to account for guile-1.8.1 and our own needs. */
bool SCMParmParse::does_scm_symbol_exist(const char *name)
{
  SCM symbol;
  SCM variable;
  assert(name ≠ (const char *) Λ);
  symbol = scm_string_to_symbol(scm_from_locale_string(name));
      /* Lookup this symbol in the current module lookup closure, but do not create it automatically if it
      does not exist. This is what the third argument to this function is about. Return SCM_BOOL_F if this is
      the case. */
  variable = scm_sym2var(symbol, scm_current_module_lookup_closure(), SCM_BOOL_F);
```

[15] Defined in "Accessing Modules from C" in "The Guile Module System" section of the "Modules" chapter.

```
    return (variable ≠ SCM_BOOL_F);
  }
```

See also chunk 292.

This code is used in chunk 273.

¶ This is a simple function that returns the file size in bytes if the file named in its argument is readable and *false* (zero) otherwise. If it so happens that the file is readable, but its size is zero, *false* is returned as well.

We first call the UNIX *stat* function on the file, and if this fails, we return *false*.

Otherwise we check if the file is regular, i.e., not a device file and not a directory either.[16] If the file is not regular, we return *false*.

Otherwise we attempt to open the file with the C++ **ifstream**::*open*( ). If it fails we return *false*.

Otherwise, we close the file and return its size in bytes.

Why not try **ifstream**::*open*( ) right away? This is because the *open*( ) would work on a directory or a device file, and we don't want this.

This function is an almost exact copy of *is_file_readable* declared and defined in the FORMS code, with the exception that it uses *cerr* instead of *pout*( ) if CH_SPACEDIM is not defined, because we want it to work outside Chombo as well. Also, we use *false* instead of NO.

292 ⟨ **SCMParmParse** implementation: protected members 291 ⟩ +≡

```
  off_t SCMParmParse::is_file_readable(const char *file_name)
  {
    struct stat file_status;
    off_t exit_status;
    ifstream file_stream;
    if (stat(file_name, &file_status) < 0) {
#ifdef CH_SPACEDIM
      pout( ) << "\n\tERROR(is_file_readable): cannot stat " << file_name << endl;
      pout( ) << "\t\tError number " << errno << endl;
#else
      cerr << "\n\tERROR(is_file_readable): cannot stat " << file_name << endl;
      cerr << "\t\tError number " << errno << endl;
#endif
      exit_status = (off_t) false;
      goto exit;
    }
    if (¬(S_ISREG(file_status.st_mode))) {
#ifdef CH_SPACEDIM
      pout( ) << "\n\tERROR(is_file_readable): file " << file_name << " is not regular" << endl;
#else
      cerr << "\n\tERROR(is_file_readable): file " << file_name << " is not regular" << endl;
#endif
      exit_status = (off_t) false;
      goto exit;
    }
    file_stream.open(file_name);
    if (file_stream.fail( )) {
#ifdef CH_SPACEDIM
      pout( ) << "\n\tERROR(is_file_readable): file " << file_name <<
          " cannot be opened for reading" << endl;
#else
      cerr << "\n\tERROR(is_file_readable): file " << file_name <<
          " cannot be opened for reading" << endl;
#endif
```

[16] (Will a soft link show as a regular file? Hopefully not.

```
    exit_status = (off_t) false;
    goto exit;
  }
  file_stream.close();
  exit_status = file_status.st_size;
exit: return exit_status;
}
```

12.2 EdgeFab⟨T⟩ Class Template

The **EdgeFab⟨T⟩** class template [17] is similar to **FluxBox**, with the exception that fields are mounted on cell edges, not on faces, and that the fields can be of various numeric types, **Real**, **int**, or anything else that the **BaseFab** template can accept.

Figure 13 shows the geometric difference between a **FluxBox** and an **EdgeFab⟨T⟩** cell.

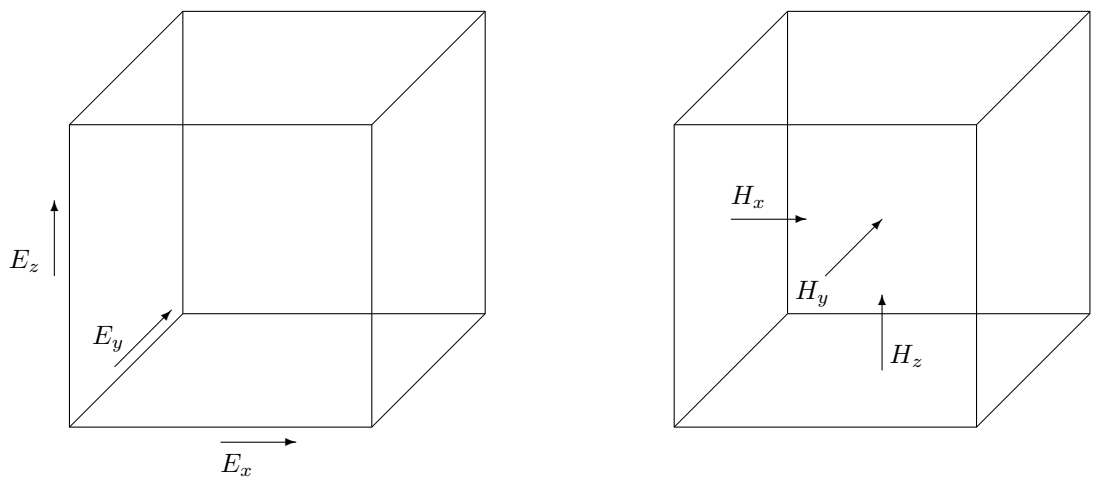


Figure 13: An **EdgeFab⟨T⟩** cell on the left versus a **FluxBox** cell on the right.

In a **FluxBox**, fields are fluxes through the faces of each cell. And so,

H_x is attached in the middle of the $\boldsymbol{e}_y \times \boldsymbol{e}_z$ face, and is of type (`NODE`, `CELL`, `CELL`),

H_y is attached in the middle of the $\boldsymbol{e}_z \times \boldsymbol{e}_y$ face, and is of type (`CELL`, `NODE`, `CELL`), and

H_z is attached in the middle of the $\boldsymbol{e}_x \times \boldsymbol{e}_y$ face, and is of type (`CELL`, `CELL`, `NODE`).

On the other hand, in an **EdgeFab⟨T⟩**, fields are defined on the edges of each cell, so that curls, which in themselves are fluxes, can be evaluated for each cell face. And so,

E_x is attached in the middle of the $\boldsymbol{e}_x$ edge, and is of type (`CELL`, `NODE`, `NODE`),

E_y is attached in the middle of the $\boldsymbol{e}_y$ edge, and is of type (`NODE`, `CELL`, `NODE`), and

E_z is attached in the middle of the $\boldsymbol{e}_z$ edge, and is of type (`NODE`, `NODE`, `CELL`).

In summary, we find that in a **FluxBox** fields are `NODE` mounted in their directions, and (`CELL`, `CELL`) mounted in the other two directions, whereas in an **EdgeFab⟨T⟩** it's the other way round: fields are `CELL` mounted in their directions, and (`NODE`, `NODE`) mounted in the remaining two directions.

This has obvious ramifications in terms of how data is to be moved between levels for these two complementary classes, because their geometric alignment between the two levels will be different—and different from cell-mounted fields, too. At the same time, we will have to ensure that divergence is not accidentally generated neither for the $\boldsymbol{B}$, nor for the $\boldsymbol{E}$ fields in the process.

We can easily make use of the `CELL` and `NODE` designation, to find the exact location of field attachment in the physical space. The **IntVect** numbering of the cells corresponds to cell centers. At the same time we have that `CELL` $= 0$ and `NODE` $= 1$, and the specific indexing type of a box is returned by **IntVect Box**::*type*( ).

For example, for the H_x field, the box type should be (`NODE`, `CELL`, `CELL`) $= (1, 0, 0)$. The point of attachment of a field in a cell (i, j, k) of type *type* is

$$\boldsymbol{r}_{\text{field}} = \boldsymbol{r}_0 \quad + \left(i - \frac{\text{type}_x}{2}\right) \Delta_x \boldsymbol{e}_x \quad + \left(j - \frac{\text{type}_y}{2}\right) \Delta_y \boldsymbol{e}_y \quad + \left(k - \frac{\text{type}_z}{2}\right) \Delta_z \boldsymbol{e}_z \tag{466}$$

[17] This template derives from the **FluxBox** class. I have also consulted the **EdgeDataBox** code by Dan Martin.

And for H_x and $\boldsymbol{r}_0 = (0,0,0)$ this becomes

$$\boldsymbol{r}_{H_x}(i,j,k) = \left(i - \frac{1}{2}\right)\Delta_x \boldsymbol{e}_x + j\Delta_y \boldsymbol{e}_y + k\Delta_z \boldsymbol{e}_z$$

At the same time for E_x and $\boldsymbol{r}_0 = (0,0,0)$ this becomes

$$\boldsymbol{r}_{E_x}(i,j,k) = i\Delta_x \boldsymbol{e}_x + \left(j - \frac{1}{2}\right)\Delta_y \boldsymbol{e}_y + \left(k - \frac{1}{2}\right)\Delta_z \boldsymbol{e}_z$$

These simple formulae will become useful when defining media layouts and movements of data between levels.

Although both **FluxBox** and **EdgeFab**$\langle$**T**$\rangle$ are defined on a cell centered box of some dimensions, the fields, (H_x, H_y, H_z) for **FluxBox** and (E_x, E_y, E_z) for **EdgeFab**$\langle$**T**$\rangle$ each have their own boxes associated with them. The boxes are typed, depending on the field, as above, and they are also *sized* so as to include fluxes or curls over the whole box, which means adding layers of data "on the other side" of the box. These would not normally be included, if we were to define fields simply on the original cells of a cell centered box, merely re-typed to mount data on faces or edges, with the effect that we could not evaluate full fluxes or full curls associated with the box using the box data alone. [18]

For example, consider a cell centered box, let us call it *cell_centered_box*: $((0,0,0)(2,2,2)(0,0,0))$. This is how Chombo would write the box out to *cout* or *pout*( ). The box has its cells numbered from $(0,0,0)$ through $(2,2,2)$ and the field mounting points in all its cells are in cell centres, $(0,0,0)$.

In order to include fluxes through all its surfaces, we need to add a layer of cells on the right hand side for the H_x field, a layer of cells on the far side for the H_y field and a layer of cells on the high side for the H_z field.

If we were to define **FluxBox** B(*cell_centered_box*), this would result in packing three **FArrayBox**es into B. One for H_x, one for H_y and one for H_z. Each of these **FArrayBox**es can be extracted from B by calling a **FluxBox**::*getFlux*(*dir*) method. If we were to inspect the actual boxes that would be returned with these fields, we would find that the boxes have been resized and typed so as to incorporate data on the other side and so as to attach the data to cell faces.

And so,

B.*getFlux*(0).*box*( ) returns $((0,0,0)(3,2,2)(1,0,0))$,

B.*getFlux*(1).*box*( ) returns $((0,0,0)(2,3,2)(0,1,0))$, and

B.*getFlux*(2).*box*( ) returns $((0,0,0)(2,2,3)(0,0,1))$.

Similarly for **EdgeFab** the internal boxes associated with fields (E_x, E_y, E_z) must be resized and typed so as to let us calculate curls over the box's surface.

If we were to define **EdgeFab**$\langle$**Real**$\rangle$ E(*cell_centered_box*), this would result in packing three **BaseFab**$\langle$**Real**$\rangle$s into E. One for E_x, one for E_y and one for E_z. Each of these **BaseFab**$\langle$**Real**$\rangle$s can be extracted from E by calling an **EdgeFab**$\langle$**Real**$\rangle$::*getEdgeData*(*dir*) method. If we were to inspect the actual boxes that would be returned with these fields, we would find that the boxes have been resized and typed so as to incorporate data "on the other edge" and so that the fields would be attached to the edges as well.

And so,

E.*getEdgeData*(0).*box*( ) returns $((0,0,0)(2,3,3)(0,1,1))$,

E.*getEdgeData*(1).*box*( ) returns $((0,0,0)(3,2,3)(1,0,1))$, and

E.*getEdgeData*(2).*box*( ) returns $((0,0,0)(3,3,2)(1,1,0))$.

Let us note here that the first box, $((0,0,0)(2,3,3)(0,1,1))$ contains *all* four $\boldsymbol{e}_x$ edges of the box. They are $([0..2],0,0)$, $([0..2],3,0)$, $([0..2],0,3)$, and $([0..2],3,3)$. Similarly for the other boxes. It is therefore possible to evaluate curls over all six faces of the original cell-centered box.

The face and edge data are *shared* with adjacent boxes. A programmer, i.e., me, must ensure that the shared data is always identical.

[18] This is another difference between, say, **FArrayBox**, which *could* be defined on edge or face centered cells as well, and **FluxBox** or **EdgeFab**$\langle$**T**$\rangle$. This fundamental difference implies that, unfortunately, the internal boxes of **FluxBox** and **EdgeFab**$\langle$**T**$\rangle$ cannot be arranged into a **DisjointBoxLayout**, because they aren't disjoint. They must overlap on the box-box boundaries. But the basic cell-centered boxes of both classes, of course, can be arranged into a **DisjointBoxLayout**.

12.2.1 EdgeFab⟨T⟩ Headers

This is the header file, that should be written on `"EdgeFab.H"` eventually. Ctangle will write it on `"EdgeFab.hw"`, for further processing with `sed` and `astyle`.

The whole content of the file is enclosed in the # **ifdef** .. # **endif** brackets, to ensure that the declarations aren't read into the compilation stream more than once.

The implementation is treated like a header file as well and is included after the declarations.

We do not include `"FArrayBox.H"`, because the class is based on the **FArrayBox**'s parent **BaseFab**⟨**T**⟩. Here we lose C++ arithmetic that we can do directly on **FArrayBox**es, but nothing else. In particular, **BaseFab**⟨**T**⟩ provides the same Fortran-style data layout as **FArrayBox**.

294 ⟨ `EdgeFab.hw` 294 ⟩ ≡

```
#ifndef CH_EDGEFAB_H
# define CH_EDGEFAB_H
#include <Box.H>
#include <Vector.H>
#include <BaseFab.H>
#include <REAL.H>
  ⟨ EdgeFab: class template declaration 295 ⟩
#include <EdgeFabImplem.H>
#endif    /* CH_EDGEFAB_H */
```

This code is cited in chunk 303.

¶ The class template **EdgeFab**⟨**T**⟩ is declared with public, protected and private members. These are discussed below in their own chunks.

295 ⟨ **EdgeFab**: class template declaration 295 ⟩ ≡

```
  template⟨class T⟩ class EdgeFab {
  public: ⟨ EdgeFab: public members 296 ⟩
  protected: ⟨ EdgeFab: protected members 301 ⟩
  private: ⟨ EdgeFab: private members 302 ⟩
  };
```

This code is used in chunk 294.

Public Members

This is the public image of the class template and it is here that we are going to discuss what various methods are supposed to do, then in the implementation part, we'll discuss how they go about it.

EdgeFab( ) This is a default constructor, which in this case is not going to construct much, because it doesn't have enough data. But it sets an internal class parameter *m_nComp* to -1, which flags to all other class methods that the object is not fully defined.

EdgeFab(const Box &*box*, **int** *nComp* = 1) This form of the constructor builds an **EdgeFab** on a *cell centered box*, also referred to as the object's *domain,* attaching an *uninitialized nComp* component field at each edge site of the *box*, including its far edges. The default value for *nComp* is 1. The electrodynamic $\boldsymbol{E} = (E_x, E_y, E_z)$ field has $nComp = 1$, because there is just one component, E_x, E_y, or E_z attached to the edge of each cell. Values of *nComp* greater than 1 are for tensor fields. But we could, for example, make use of multiple components here to encode both $\boldsymbol{D}$ and $\boldsymbol{E}$ on the same **EdgeFab**⟨**Real**⟩. We could even pack auxiliary fields $\boldsymbol{S}_n, n = 1, \ldots, k$ that arise from time integration of $\boldsymbol{E}$ (these are various induced "currents"). Or we could choose to keep $\boldsymbol{D}$ and $\boldsymbol{E}$ separate, but lump all $\boldsymbol{S}_n$ together.

~EdgeFab( ) The default destructor. It calls *clear*( ), which deletes all data associated with the **EdgeFab** fields, resets internal pointers to Λ, and resets *m_nComp* to -1, so that all other class methods know that the object is no longer fully defined.

296 ⟨ **EdgeFab**: public members 296 ⟩ ≡
EdgeFab( );
EdgeFab(**const Box** &*box*, **int** *nComp* = 1);
∼**EdgeFab**( );

See also chunks 297, 298, 299, and 300.

This code is used in chunk 295.

Constructors and Destructors

The actual constructors and destructors of the class:

resize(**const Box** &*box*, **int** *nComp* = 1) This function either allocates all space required for the **BaseFab**⟨**T**⟩ fields that are going to live within the **EdgeFab**, or resizes the object's domain, and its internal boxes together with it, and, if requested, also the number of field components for an existing **EdgeFab**, always making sure that there is enough space available for data. *box* is the possibly new cell-centered domain over which to span the field, and *nComp* is the number of the field's components.

define(**const Box** &*box*, **int** *nComp* = 1) This is the real constructor method of the class. It creates *SpaceDim* multicomponent (*nComp*) edge mounted *uninitialized* **BaseFab**⟨**T**⟩s over the object's domain *box*, providing each **BaseFab**⟨**T**⟩ with its own specially sized and typed box, so as to include fields attached to all edges of the domain, including the far ones.

clear( ) This is the effective destructor of the class. It wipes everything out, deallocates space, resets internal pointers to Λ, and resets *m_nComp* to −1.

297 ⟨ **EdgeFab**: public members 296 ⟩ +≡
void *resize*(**const Box** &*box*, **int** *nComp* = 1);
void *define*(**const Box** &*box*, **int** *nComp* = 1);
void *clear*( );

Extractors

Now we have a range of *extractors,* i.e., functions that extract data from **EdgeFab**:

nComp( ) This function returns the number of field components.

box( ) This function returns the cell-centered box that the **EdgeFab** is built on.

getEdgeData(**const int** *dir*) This function returns the **BaseFab**⟨**T**⟩ that is associated with the edge in direction *dir*. The two versions have an identical code inside. The **const** one doesn't allow the programmer to modify the returned field, which is enforced by the declaration. This same function can be also invoked simply by using the [ ] operator, which makes an **EdgeFab**⟨**T**⟩ look like an array of three **BaseFab**⟨**T**⟩s.

298 ⟨ **EdgeFab**: public members 296 ⟩ +≡
int *nComp*( ) **const**;
const Box &*box*( ) **const**;
BaseFab⟨**T**⟩ &*getEdgeData*(**const int** *dir*);
const BaseFab⟨**T**⟩ &*getEdgeData*(**const int** *dir*) **const**;
BaseFab⟨**T**⟩ &**operator**[ ](**const int** *dir*);
const BaseFab⟨**T**⟩ &**operator**[ ](**const int** *dir*) **const**;

Modifiers

The modifiers of the class fall in two groups:

setVal( ) These methods set the value of the **EdgeFab**⟨**T**⟩ fields, either all components of all of them, to some real value, or all components in some specified direction *dir*, or some specified range of the components in a specified direction, on the whole domain of the defined object or on a *box* that is contained by the domain.

The *setVal*( ) methods invoke the **BaseFab**⟨**T**⟩::*setVal*( ) methods internally.

copy( ) These methods *copy* data from an argument **EdgeFab**⟨**T**⟩ to *this* **EdgeFab**⟨**T**⟩. As their semantics are somewhat subtle, here we discuss them in more detail.

copy(**const EdgeFab**⟨**T**⟩ &*src*) This method copies the whole content of the argument **EdgeFab**⟨**T**⟩ into this **EdgeFab**⟨**T**⟩, on the intersection of the two **BaseFab**⟨**T**⟩ domains.

copy(**const EdgeFab**⟨**T**⟩ &*src*, **const int** *srcComp*, **const int** *destComp*, **const int** *numComp*) This method copies as above, but the copy is restricted to the specified range of components.

copy(**const EdgeFab**⟨**T**⟩ &*src*, **const int** *dir*, **const int** *srcComp*, **const int** *destComp*, **const int** *numComp*) This method copies as above, but the copy is additionally restricted to a specified direction, for example, E_x only.

copy(**const Box** &*RegionFrom*, **const Interval** &*Cdest*, **const Box** &*RegionTo*, **const EdgeFab**⟨**T**⟩ &*src*, **const In**
This method copies edge data that is sourced from the intersection of the *RegionFrom* box with the source **EdgeFab**⟨**T**⟩ domain to the intersection of the *RegionTo* box with the domain of *this* box. The copy moves data between the *Csrc* and *Cdest* component intervals.

copy(**const Box** &*Region*, **const Interval** &*Cdest*, **const EdgeFab**⟨**T**⟩ &*src*, **const Interval** &*Csrc*) This method copies as above with the difference that the destination box is the same as the source box, here referred to as *Region*. The data movement is restricted to the intersection of *Region* with the domains of the source and *this* **EdgeFab**⟨**T**⟩s.

shift(**const IntVect** &*v*) This function does nothing to the field data. Instead, it shifts all boxes within the **EdgeFab**⟨**T**⟩ by *v*, which is a lightweight operation. Yet, this has the effect of pushing the whole **EdgeFab**⟨**T**⟩ by *v*, because the way the field data is associated with an actual grid point is through the definition of boxes on which the data is spanned. Chombo does not define the actual grid and does not bind the data to specific grid points other than through the data layout within its internal arrays, which is Fortran-like rather than C-like.

299 ⟨**EdgeFab**: public members 296⟩ +≡

```
void setVal(const T val);
void setVal(const T val, const int dir);
void setVal(const T val, const int dir, const int startComp, const int nComp);
void setVal(const T val, const Box &box);
void setVal(const T val, const Box &box, const int dir, const int startComp, const int nComp);
void copy(const EdgeFab<T> &src);
void copy(const EdgeFab<T> &src, const int srcComp, const int destComp, const int numComp);
void copy(const EdgeFab<T> &src, const int dir, const int srcComp, const int destComp, const int
    numComp);
void copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const EdgeFab<T>
    &src, const Interval &Csrc);
void copy(const Box &Region, const Interval &Cdest, const EdgeFab<T> &src, const Interval &Csrc);
EdgeFab<T> &shift(const IntVect &v);
```

Transfer Helpers

These four functions are used in transferring data between MPI processes. The way it's done, generally, is as follows: the data from box edges are packed into a buffer by calling *linearOut*. The buffer is then MPI transmitted to the destination process (the **LevelData**⟨ ⟩ . *exchange*( ) does this). There, it is unpacked back onto box edges by calling *linearIn*.

size(**const Box** &*box*, **const Interval** &*comps*) This method returns the size, in bytes, of data in components *comps* attached to all cell edges associated with the cell centered box *box*. It's purpose is to provide sizing for transfer buffers.

linearOut(**void** ∗*buf*, **const Box** &*R*, **const Interval** &*comps*) This method writes all edge data associated with all cells of **Box** *R* and resident in components *comps* onto a preallocated and presized buffer *buf*.

linearIn(**void** ∗*buf*, **const Box** &*R*, **const Interval** &*comps*) This method unpacks edge data that has arrived in buffer *buf* onto all cells of **Box** *R* and writes them into components *comps*.

preAllocatable( ) A small helper function used by the above.

300 ⟨ **EdgeFab**: public members 296 ⟩ +≡
```
int size(const Box &box, const Interval &comps) const;
void linearOut(void *buf, const Box &R, const Interval &comps) const;
void linearIn(void *buf, const Box &R, const Interval &comps);
static int preAllocatable( )
{
  return 0;
}
```

Protected Members

m_box This is the cell centered box, also referred to as the object's *domain*, the **EdgeFab**⟨**T**⟩ is built on.

m_nComp This is the number of components mounted on each cell edge.

m_edgeData This is a vector of *SpaceDim* **BaseFab**⟨**T**⟩s that stores the field data.

301 ⟨ **EdgeFab**: protected members 301 ⟩ ≡
```
Box m_box;
int m_nComp;
Vector<BaseFab<T> *> m_edgeData;
```
This code is used in chunk 295.

Private Members

These two are for copying the whole **EdgeFab** onto another one, as in *EdgeFab_1* = *EdgeFab_2*. They are declared here, but they're not defined in the code, so they can't be used.

302 ⟨ **EdgeFab**: private members 302 ⟩ ≡
```
EdgeFab(const EdgeFab<T> &);
EdgeFab<T> &operator=(const EdgeFab<T> &);
```
This code is used in chunk 295.

12.2.2 EdgeFab⟨T⟩ Implementation

Now, we get to implement the function templates declared in Section 12.2.1. Because the whole class is defined as a template, this file is also a header file, to be included in `"EdgeFab.H"`, as discussed in module ⟨ `EdgeFab.hw` 294 ⟩, page 400.

Its own structure is bracketed by # **ifdef** .. # **endif** clauses that ensure that the definitions will not be read into the compilation stream more than once. The file must ultimately become `"EdgeFabImplem.H"`.

303 ⟨ `EdgeFabImplem.hw` 303 ⟩ ≡
```
#ifndef CH_EDGEFABIMPLEM_H
# define CH_EDGEFABIMPLEM_H
  ⟨ EdgeFab template implementation 304 ⟩
#endif     /* CH_EDGEFABIMPLEM_H */
```

Extractors

We begin by implementing the two trivial methods, *nComp*( ) and *box*( ). The first one returns the value of *m_nComp*, which stores the number of components to be attached at any given edge. The second one returns the value of *m_box*, which stores the cell centered box the **EdgeFab** is built on.

304 ⟨ **EdgeFab** template implementation 304 ⟩ ≡

```
template⟨class T⟩ int EdgeFab⟨T⟩::nComp( ) const
{
  return m_nComp;
}
template⟨class T⟩ const Box &EdgeFab⟨T⟩::box( ) const
{
  return m_box;
}
```

See also chunks 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 319, 320, and 321.

This code is used in chunk 303.

Get Edge Data

The next step is to define the *getEdgeData*( ) methods, i.e., functions that return the **BaseFab**⟨**T**⟩s that constitute the **EdgeFab**⟨**T**⟩.

The first form returns a pointer to the corresponding **BaseFab**⟨**T**⟩ associated with a given direction *dir*. It begins, in the style of other Chombo functions, with *assert*s that check if

1. the number of components is positive (if it is not, it means that the **EdgeFab**⟨**T**⟩ is empty),
2. the *dir* value fits between zero and *SpaceDim*, and
3. the *m_edgeData*[*dir*], which is itself a pointer to **BaseFab**⟨**T**⟩, is not Λ (i.e., not `NULL`).

Using *assert*, which is a function from the C Library, is a somewhat crude method of handling errors. When an expression, that is an argument of *assert*, evaluates to *false*, the function prints a message, which is quite informative, so this is good, and then calls *abort*. In short, *assert* exits are rather unceremonious and cannot be caught. ◀

Chombo used to disable *assert*s unless it was explicitly compiled with the `DEBUG` switch. In its latest version Chombo divorces *assert*s from `DEBUG` and binds them instead to `OPT`.

Having *assert*ed that everything is OK, *getEdgeData* returns the object that *m_edgeData*[*dir*] points to, that is, the corresponding **BaseFab**⟨**T**⟩. The return is *by reference,* which means that no data is copied.

The second form of *getEdgeData* has exactly the same stuff inside, but is declared differently. The **const** after the declaration says that this is an *inspector,* i.e., a function that does not modify the state of an **EdgeFab**⟨**T**⟩. The **const** in front of **BaseFab**⟨**T**⟩ & says, more specifically, that this **BaseFab**⟨**T**⟩ must not change. In this form the *getEdgeData* method may be used, for example, inside the *copy* method.

The third and the fourth forms are the same as calling *getEdgeData* in one of the two forms above, but a [ ] operator is used instead. So, if *a_curl_box* is an **EdgeFab**⟨**T**⟩, we can extract its **BaseFab**⟨**T**⟩ components simply by *a_curl_box*[*dir*] instead of *a_curl_box*.*getEdgeData*(*dir*).

305 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ BaseFab⟨T⟩ &EdgeFab⟨T⟩::getEdgeData(const int dir)
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
```

```
template⟨class T⟩ const BaseFab⟨T⟩ &EdgeFab⟨T⟩::getEdgeData(const int dir) const
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
template⟨class T⟩ BaseFab⟨T⟩ &EdgeFab⟨T⟩::operator[](const int dir)
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
template⟨class T⟩ const BaseFab⟨T⟩ &EdgeFab⟨T⟩::operator[](const int dir) const
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
```

Constructors and Destructors

In this section we discuss constructors and destructors of the class.

The simplest default constructor is **EdgeFab⟨T⟩::EdgeFab**( ). Because it knows nothing of the box or the number of components, all it does is to allocate a **Vector** *m_edgeData*, make it *SpaceDim* entries long and set its entries to Λ. The other internal variable of the class, *m_nComp* is set to −1, which flags to other class methods that the object is not fully defined.

When **EdgeFab⟨T⟩::EdgeFab**( ) is called with a box and a number of components, *m_edgeData* is prepared as before, but this time we call *define*(*a_box*, *a_nComp*) to complete the construction.

306 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ EdgeFab⟨T⟩::EdgeFab( )
: m_edgeData(SpaceDim, Λ) {
  m_nComp = −1;
}
template⟨class T⟩ EdgeFab⟨T⟩::EdgeFab(const Box &a_box, int a_nComp)
: m_edgeData(SpaceDim, Λ) {
  define(a_box, a_nComp);
}
```

¶ The destructor of the class is **EdgeFab⟨T⟩::∼EdgeFab**( ). It simply calls *clear*( ), which completes the job.

Function *clear*( ), in turn, deletes all the *m_edgeData*[*dir*] **BaseFab⟨T⟩**s, if they're not Λ, and resets the *m_edgeData*[*dir*] pointers to Λ. *m_nComp* is reset to −1, as it was in the **EdgeFab**( ) constructor, which flags to all class methods that this object is now empty. Finally, the box itself, *m_box*, is reset to an empty box, which is what **Box**( ) returns when called without arguments.

307 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ EdgeFab⟨T⟩::∼EdgeFab( )
{
  clear( );
}
```

```
template⟨class T⟩ void EdgeFab⟨T⟩::clear()
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    if (m_edgeData[dir] ≠ Λ) {
      delete m_edgeData[dir];
      m_edgeData[dir] = Λ;
    }
  }
  m_nComp = −1;
  m_box = Box();
}
```

¶ **EdgeFab⟨T⟩**::*define*( ) is the actual constructor of the class, behind the wrappers. It takes *a_box* and *a_nComp* as its arguments.

The first thing that happens after a sanity check for *a_nComp* is that *a_box* gets copied on the internal variable *m_box* and *a_nComp* is copied on *m_nComp*. And so, these two variables are now accessible to all other class methods. The currently empty vector of **BaseFab⟨T⟩**s, is resized to *SpaceDim* in preparation for space allocations.

Now we enter a loop over space directions, we copy *m_box* onto *edgeBox* and adjust the *edgeBox* for this particular **BaseFab⟨T⟩**. The first adjustment is to convert the original cell-centered *m_box* to a node box in *all* directions. Function *surroundingNodes* not only changes the type of the box, but also increases *all* components of the upper corner of the box by one. But we don't want the box to be stretched and typed so in *all* directions. We only want it modified in directions that are perpendicular to *dir*. So, the next adjustment, *enclosedCells*(*dir*), returns the box to cell centered in the *dir* direction and reduces its high corner *dir* coordinate by one too, thus delivering us a box exactly as is needed for the edge mounted field in this direction.

For comparison, the **FluxBox** code in this place performs just one operation, which is ◀

```
Box edgeBox(surroundingNodes(m_box, dir));
```

When converting the **FluxBox** code to the **EdgeFab⟨T⟩** code we need to replace every occurrence of the above with

```
Box edgeBox(surroundingNodes(m_box)); edgeBox.enclosedCells(dir);
```

We shall flag in the following every time we do so with the ◀ sign, because this is important and a failure to do so would produce an incorrect code.

Having constructed the right box, we now create a new **BaseFab⟨T⟩** with *m_nComp* components on this box, and place a pointer to it in the *m_edgeData* array.

The method creates and structures space for data, and hooks it into the **EdgeFab⟨T⟩** object, but it does not initialize the data.

308 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::define(const Box &a_box, int a_nComp)
{
  assert(a_nComp > 0);
  m_box = a_box;
  m_nComp = a_nComp;
  m_edgeData.resize(SpaceDim);
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box edgeBox(surroundingNodes(m_box));
    edgeBox.enclosedCells(dir);
    BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(edgeBox, m_nComp);
    m_edgeData[dir] = newFabPtr;
  }
}
```

¶ **EdgeFab**⟨**T**⟩::*resize*( ) is quite similar to *define*. It could be called *redefine* instead. Basically, it lets us substitute a new box in place of the old one, and/or change the number of components on a field that may have already been defined.

But we first check if *m_nComp* is less than zero, which would be the sign that this object hasn't been constructed yet. In this case, we simply refer the matter to *define*.

If the object has been defined, we check that the new *a_nComp* is sound, then transfer the new *a_box* and *a_nComp* to the internal class variables *m_box* and *m_nComp* respectively. Then we enter a loop over directions.

And the first thing we do within this loop is to adjust the cell centered *a_box* for the edge-mounted field, as we did in *define*. ◀

With the box suitably adjusted we now have two possible cases.

1. There is already a **BaseFab**⟨**T**⟩ defined for this direction. In this case we simply ask it to **BaseFab**⟨**T**⟩::*resize*( ) itself, and give it the new *edgeBox* and the new *m_nComp*.

2. There is no **BaseFab**⟨**T**⟩ defined for this direction, in which case we define it and place the pointer to it in the *m_edgeData* vector.

> If an **EdgeFab** had been cleared, *m_nComp* would be reset to -1, so in this case *define* would be called. This situation assumes that *m_nComp* > 0, but, for some reason, the **BaseFab**⟨**T**⟩ alone has been cleared, e.g., by a programmer, and reset to Λ.

In case (1) the **BaseFab**⟨**T**⟩::*resize*( ) always results in the memory allocated for this **BaseFab** to *grow* and never shrink. The Chombo documentation also recommends that this method be used for managing temporary space, in situations when calling for a new allocation would be too costly.

309 ⟨**EdgeFab** template implementation 304⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::resize(const Box &a_box, int a_nComp)
{
  if (m_nComp < 0) {
    define(a_box, a_nComp);
  }
  else {
    assert(a_nComp > 0);
    m_box = a_box;
    m_nComp = a_nComp;
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box edgeBox(surroundingNodes(m_box));
      edgeBox.enclosedCells(dir);
      if (m_edgeData[dir] ≠ Λ) {
        m_edgeData[dir]→resize(edgeBox, m_nComp);
      }
      else {
        BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(edgeBox, m_nComp);
        m_edgeData[dir] = newFabPtr;
      }
    }
  }
}
```

Modifiers

Now we have two basic modifiers of the class. The first one, *setVal*, just sets the whole **EdgeFab** to some value. This can be trimmed a little, to set only a specific range of components or only the fields associated with a specific direction or both.

The first three forms are pretty trivial

setVal(**const T** *val*) Set all components of all **EdgeFab**⟨**T**⟩ fields to *val*.

setVal(**const T** *val*, **int** *dir*) Set all components of **EdgeFab**⟨**T**⟩ fields associated with direction *dir* to *val*.

setVal(**const T** *val*, **const int** *dir*, **const int** *startComp*, **const int** *nComp*) Set the range of components of **EdgeFab**⟨**T**⟩ fields associated with direction *dir* to *val*.

They all work by first testing the sanity of parameters passed with *assert*s, and then by calling the **BaseFab**⟨**T**⟩::*setVal*( ) method to do the job.

310 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val)
{
  assert(m_nComp > 0);
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]→setVal(val);
  }
}
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, int dir)
{
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  m_edgeData[dir]→setVal(val);
}
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, const int dir, const int startComp, const
      int nComp)
{
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  for (int comp = startComp; comp < startComp + nComp; comp++) {
    m_edgeData[dir]→setVal(val, comp);
  }
}
```

¶ Now we get into a somewhat trickier version of *setVal*, because this time we set it on a **Box** *box*. For this to work the **Box** in question must be a sub-box of *m_box*. We could relax this condition and simply set the **EdgeFab** to *val* on an overlap of the *box* and *m_box*, if it is not empty, but Chombo semantics do not go this far. So the programmer has to check for this explicitly before calling *setVal*, as we did in module ⟨ Function *draw_box* 135 ⟩ of Section 8.1. We check whether *box* ⊂ *m_box* with an *assert*.

Then we get into a loop over directions. For each direction we make sure that there is a corresponding **BaseFab**⟨**T**⟩ defined, and then we modify the **Box** passed to make it into an edge-**Box** that is shaped specifically for this direction. Once the *box* has been tweaked, we call the **BaseFab**⟨**T**⟩::*setVal*( ) method to carry out the operation. ◀

In the second form of *setVal*( ) the programmer may specify the direction and the range of components additionally. In this case we additionally check if the direction provided and the components range are within the bounds.

311 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, const Box &box)
{
  assert(m_box.contains(box));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box edgeBox(surroundingNodes(box));
```

```
      edgeBox.enclosedCells(dir);
      m_edgeData[dir]→setVal(val, edgeBox, 0, m_nComp);
    }
  }
  template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, const Box &box, const int dir, const int
          startComp, const int nComp)
  {
    assert(m_box.contains(box));
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    assert(startComp > −1);
    assert(startComp + nComp ≤ m_nComp);
    Box edgeBox(surroundingNodes(box));
    edgeBox.enclosedCells(dir);
    m_edgeData[dir]→setVal(val, edgeBox, startComp, nComp);
  }
```

Copy

EdgeFab⟨**T**⟩::*copy*( ) is an important group of modifiers that deserves its own subsection. As is the case with *setVal*( ), here we also defer the actual action to **BaseFab**⟨**T**⟩::*copy*( ), but tweaking input, and checking with *assert*s still has to be done before **BaseFab**⟨**T**⟩::*copy*( ) can be invoked.

The first version of **EdgeFab**⟨**T**⟩::*copy*( ) copies data from one **EdgeFab**⟨**T**⟩ to another one. Both have to have the same number of components. The **BaseFab**⟨**T**⟩::*copy*( ) invoked here performs this operation on the *intersection* of the domains of both **EdgeFab**⟨**T**⟩s.

The original **FluxBox** code had an *assert* here that checked if the source and *this* **EdgeFab** domains were identical, meaning of the same size, location and type. This was too strong an assert and it would fail unnecessarily making it impossible to copy between **EdgeFab**⟨**T**⟩s that are defined on overlapping domains, so I deleted it. I also extended the remaining *assert* to include a check for both **EdgeFab**⟨**T**⟩s being fully defined.

312 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
  template⟨class T⟩ void EdgeFab⟨T⟩::copy(const EdgeFab⟨T⟩ &src)
  {
    assert((src.nComp( ) > 0) ∧ (m_nComp > 0) ∧ (src.nComp( ) ≡ m_nComp));
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_edgeData[dir] ≠ Λ);
      m_edgeData[dir]→copy(src[dir]);
    }
  }
```

¶ This second form of **EdgeFab**⟨**T**⟩::*copy*( ) copies data from a range of source components into a range of destination components on *this* **EdgeFab**, and on the overlap of the two **EdgeFab**⟨**T**⟩ domains, using **BaseFab**⟨**T**⟩::*copy*( ).

I have modifed the *assert* checks here, because the original ones ◀

```
assert(srcComp * destComp > −1); assert(srcComp + numComp ≤ src.nComp( ));
assert(destComp + numComp ≤ m_nvar);
```

inherited from **FluxBox** didn't do enough checking. The first check here was strange, because it would not detect an error if *srcComp* and *destComp* were both negative.

The new *assert*s check if both **EdgeFab**⟨**T**⟩s are fully defined, then check if both *srcComp* and *destComp* are positive and finally check if we'll stay within the component range within both **EdgeFab**⟨**T**⟩s.

Having run our sanity checks, we loop over the directions. For each direction we check that the destination **BaseFab**⟨**T**⟩ exists and invoke **BaseFab**⟨**T**⟩::*copy*( ) on *src*[*dir*].

313 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const EdgeFab⟨T⟩ &src, const int srcComp, const int
        destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    m_edgeData[dir]→copy(srcFab, srcComp, destComp, numComp);
  }
}
```

¶ In this version of *copy*( ) we let a programmer additionally specify the direction. Otherwise it is like the version above. I have carried the same *assert* checks as above and added a check for the *dir* as well.

314 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const EdgeFab⟨T⟩ &src, const int dir, const int
        srcComp, const int destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  const BaseFab⟨T⟩ &srcFab = src[dir];
  m_edgeData[dir]→copy(srcFab, srcComp, destComp, numComp);
}
```

¶ This version of *copy*( ) lets a programmer copy data between two component intervals and two subregions of the **EdgeFab**s' domains. The copy operation is carried out using **BaseFab**⟨**T**⟩::*copy*( ) and for all directions. The order of arguments is the same as in a similar **BaseFab**⟨**T**⟩::*copy*( ) method.

The source and destination boxes are resized for edge mounted data. There is also a change in the semantics of this code. The original **FluxBox** code had destination and source intervals swapped around both in the argument list and in the call to **BaseFab**⟨**T**⟩::*copy*( ), which was a bug–currently fixed in the latest release of Chombo. This code adheres to the original **BaseFab**⟨**T**⟩::*copy*( ) order, as stated above. There are no new *assert*s since **BaseFab**⟨**T**⟩::*copy*( ) does its own checks here. ◀

The checks carried out by **BaseFab**⟨**T**⟩::*copy*( ) are several and important to fully understand the semantics of this operation:

1. If *srcFab* is the same as **this** and both boxes are the same, the operation returns without doing anything, because nothing needs to be done.
2. The size of both **Interval**s must be the same, and they must both fit in the source and destination component range respectively.
3. *RedgeFrom* and *RedgeTo* must be of the same size.
4. *RedgeFrom* must be contained in the source domain, and *RedgeTo* must be contained in the destination domain.

315 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const Box &RegionFrom, const Interval &Cdest, const
        Box &RegionTo, const EdgeFab⟨T⟩ &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box RedgeFrom(surroundingNodes(RegionFrom));
    RedgeFrom.enclosedCells(dir);
    Box RedgeTo(surroundingNodes(RegionTo));
    RedgeTo.enclosedCells(dir);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    RedgeTo &= (m_edgeData[dir]→box());
    m_edgeData[dir]→copy(RedgeFrom, Cdest, RedgeTo, srcFab, Csrc);
  }
}
```

¶ This is another variant of a *copy*( ) method that copies data within a specified region, which must be a shared sub-box of both *src.box*( ) and *m_box*. The copy operation is carried out using **BaseFab**⟨**T**⟩::*copy*( ) for a range of components specified by **Interval**s.

Within the loop over all directions we check that the fields of *this* **EdgeFab**⟨**T**⟩ are fully defined, then grab the *Region* box and stretch it for the edge-mounted field in this direction. ◀

Having done so, we take its intersection with the source field box and then with the destination field box. If there is a non-empty overlap of the three, we perform the **BaseFab**⟨**T**⟩::*copy*( ) operation on this overlap.

316 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const Box &Region, const Interval &Cdest, const
        EdgeFab⟨T⟩ &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box edgeRegion(surroundingNodes(Region));
    edgeRegion.enclosedCells(dir);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    edgeRegion &= srcFab.box();
    edgeRegion &= m_edgeData[dir]→box();
    if (¬edgeRegion.isEmpty()) {
      m_edgeData[dir]→copy(edgeRegion, Cdest, edgeRegion, srcFab, Csrc);
    }
  }
}
```

Shift

This function just shifts *m_box* and the boxes of the **BaseFab**⟨**T**⟩ fields by an **IntVect** *iv*. Although it does not touch the data arrays, it modifies the domain of the **EdgeFab**.

317 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ EdgeFab⟨T⟩ &EdgeFab⟨T⟩::shift(const IntVect &iv)
{
  m_box.shift(iv);
  for (int dir = 0; dir < SpaceDim; dir++) {
    m_edgeData[dir]→shift(iv);
```

```
    }
    return *this;
  }
```

Transfer Helpers

These methods are used in transferring **EdgeFab**⟨**T**⟩ data between MPI processes. The first one, *size*, calculates the size of the buffer that will be needed for the transfer. Chombo functions use it to allocate the buffer space. Then the buffer is filled by calling *linearOut*, the data is transferred to other processes, which call *linearIn* to unpack it and put it back into their local **EdgeFab**⟨**T**⟩s.

As with everything else here, these methods are basically wrappers around **BaseFab**⟨**T**⟩ functions with identical names.

Size

This function calls **BaseFab**⟨**T**⟩::*size*(**Box** &*box*, **Interval** *comps*), which returns the size of the linear representation of the **BaseFab**⟨**T**⟩ data for all components in *comps* and built on this *box*. The size does *not* include the size of the box itself or of the components structure.

The function creates a temporary and undefined **BaseFab**⟨**T**⟩, just so as to provide data typing to **BaseFab**⟨**T**⟩::*size*( ), then enters the loop over directions. For each direction we stretch the original *box* for the edge data placement, as was first done in *define*( ), then call **BaseFab**⟨**T**⟩::*size*( ) on this stretched box and with the components in the **Interval** specified. The returns are then accumulated on *totalSize*, which is returned when the function completes. ◀

319 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
  template⟨class T⟩ int EdgeFab⟨T⟩::size(const Box &box, const Interval &comps) const
  {
    int totalSize = 0;
    BaseFab⟨T⟩ tempFab;
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box edgeBox(surroundingNodes(box));
      edgeBox.enclosedCells(dir);
      const int dirSize = tempFab.size(edgeBox, comps);
      totalSize += dirSize;
    }
    return totalSize;
  }
```

linearOut

This function writes out onto buffer *buf* the content of all components within the **Interval** *comps* of *all three* fields that constitute an **EdgeFab**⟨**T**⟩ for all cells contained by the *box*. The *box* in question does not have to be the same as *m_box*. Data transfer functions will usually build their own boxes around data that needs to be transferred into the ghost regions of adjacent boxes.

We begin our peregrinations by casting *buf* onto a pointer to **T**. This then becomes a pointer to the **T** buffer, called *buffer*. As we push each **BaseFab**⟨**T**⟩ onto the buffer in turn we are going to advance the pointer *buffer*, to prepare the buffer for the next write.

And so, we enter the loop over directions and, the first thing we check is if there is anything in the data portion of the **EdgeFab**⟨**T**⟩ with an *assert*. Then we take the input **Box** *box* and we stretch it into an edge mounted box for this direction. This new stretched box is called *dirBox*. ◀

Now we are ready to use **BaseFab**⟨**T**⟩::*linearOut*( ) method to just write the data for this **BaseFab**⟨**T**⟩ and this *dirBox*. But before we do this, we find how much data is going to be written, in bytes, and store this number on *dirSize*. We do this, because immediately after the write we are going to advance the **T** **buffer* pointer by *dirSize*/**sizeof**(**T**).

320 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::linearOut(void *buf, const Box &box, const Interval &comps)
        const
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearOut(buffer, dirBox, comps);
    buffer += dirSize/sizeof(T);
  }
}
```

linearIn

And here we do exactly as above, but we replace *linearOut* with *linearIn*. All other comments made in the *linearOut* section pertain to this function as well.

321 ⟨ **EdgeFab** template implementation 304 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::linearIn(void *buf, const Box &box, const Interval &comps)
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearIn(buffer, dirBox, comps);
    buffer += dirSize/sizeof(T);
  }
}
```

12.3 FaceFab⟨T⟩ Class Template

The **FaceFab⟨T⟩** template is even more like **FluxBox**. It is **FluxBox** made into a template, so that we can mount fields of various types on its faces.

The source listed below was produced by running automatic renames on `"EdgeFab.H"` and `"EdgeFabImplem.H"` files, and replacing constructs such as

Box *dirBox*(*surroundingNodes*(*box*)); *dirBox*.*enclosedCells*(*dir*);

with

Box *dirBox*(*surroundingNodes*(*box*, *dir*));

This means that this code inherits all little mods (mostly *assert* constructions) from **EdgeFab⟨T⟩**.

Otherwise it is presented here without additional comments.

12.3.1 FaceFab⟨T⟩ Headers

323 ⟨ FaceFab.hw 323 ⟩ ≡

```
#ifndef CH_FACEFAB_H
# define CH_FACEFAB_H
#include <Box.H>
#include <Vector.H>
#include <BaseFab.H>
#include <REAL.H>
  template⟨class T⟩ class FaceFab {
  public: FaceFab( );
    FaceFab(const Box &box, int nComp = 1);
    ~FaceFab( );
    void resize(const Box &box, int nComp = 1);
    void define(const Box &box, int nComp = 1);
    void clear( );
    int nComp( ) const;
    const Box &box( ) const;
    BaseFab⟨T⟩ &getFaceData(const int dir);
    const BaseFab⟨T⟩ &getFaceData(const int dir) const;
    BaseFab⟨T⟩ &operator[](const int dir);
    const BaseFab⟨T⟩ &operator[](const int dir) const;
    void setVal(const T val);
    void setVal(const T val, const int dir);
    void setVal(const T val, const int dir, const int startComp, const int nComp);
    void setVal(const T val, const Box &box);
    void setVal(const T val, const Box &box, const int dir, const int startComp, const int nComp);
    void copy(const FaceFab⟨T⟩ &src);
    void copy(const FaceFab⟨T⟩ &src, const int srcComp, const int destComp, const int numComp);
    void copy(const FaceFab⟨T⟩ &src, const int dir, const int srcComp, const int destComp, const int
        numComp);
    void copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const FaceFab⟨T⟩
        &src, const Interval &Csrc);
    void copy(const Box &Region, const Interval &Cdest, const FaceFab⟨T⟩ &src, const Interval
        &Csrc);
    FaceFab⟨T⟩ &shift(const IntVect &v);
    int size(const Box &box, const Interval &comps) const;
    void linearOut(void *buf, const Box &R, const Interval &comps) const;
    void linearIn(void *buf, const Box &R, const Interval &comps);
```

```
    static int preAllocatable()
    {
      return 0;
    }
  protected: Box m_box;
    int m_nComp;
    Vector<BaseFab<T> *> m_faceData;
  private: FaceFab(const FaceFab<T> &);
    FaceFab<T> &operator=(const FaceFab<T> &);
  };
#include <FaceFabImplem.H>
#endif
```

12.3.2 FaceFab⟨T⟩ Implementation

Extractors

325 ⟨FaceFabImplem.hw 325⟩ ≡

```
#ifndef CH_FACEFABIMPLEM_H
# define CH_FACEFABIMPLEM_H
  template<class T> int FaceFab<T>::nComp() const
  {
    return m_nComp;
  }
  template<class T> const Box &FaceFab<T>::box() const
  {
    return m_box;
  }
  template<class T> BaseFab<T> &FaceFab<T>::getFaceData(const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
  template<class T> const BaseFab<T> &FaceFab<T>::getFaceData(const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
  template<class T> BaseFab<T> &FaceFab<T>::operator[](const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
  template<class T> const BaseFab<T> &FaceFab<T>::operator[](const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
```

```
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
```

See also chunks 326, 327, and 328.

Constructors and Destructors

326 ⟨FaceFabImplem.hw 325⟩ +≡

```
  template⟨class T⟩ FaceFab⟨T⟩::FaceFab()
  : m_faceData(SpaceDim, Λ) {
    m_nComp = −1;
  }
  template⟨class T⟩ FaceFab⟨T⟩::FaceFab(const Box &a_box, int a_nComp)
  : m_faceData(SpaceDim, Λ) {
    define(a_box, a_nComp);
  }
  template⟨class T⟩ FaceFab⟨T⟩::~FaceFab()
  {
    clear();
  }
  template⟨class T⟩ void FaceFab⟨T⟩::clear()
  {
    for (int dir = 0; dir < SpaceDim; dir++) {
      if (m_faceData[dir] ≠ Λ) {
        delete m_faceData[dir];
        m_faceData[dir] = Λ;
      }
    }
    m_nComp = −1;
    m_box = Box();
  }
  template⟨class T⟩ void FaceFab⟨T⟩::define(const Box &a_box, int a_nComp)
  {
    assert(a_nComp > 0);
    m_box = a_box;
    m_nComp = a_nComp;
    m_faceData.resize(SpaceDim);
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box faceBox(surroundingNodes(m_box, dir));
      BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(faceBox, m_nComp);
      m_faceData[dir] = newFabPtr;
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::resize(const Box &a_box, int a_nComp)
  {
    if (m_nComp < 0) {
      define(a_box, a_nComp);
    }
    else {
      assert(a_nComp > 0);
      m_box = a_box;
      m_nComp = a_nComp;
      for (int dir = 0; dir < SpaceDim; dir++) {
        Box faceBox(surroundingNodes(m_box, dir));
```

```
        if (m_faceData[dir] ≠ Λ) {
          m_faceData[dir]→resize(faceBox, m_nComp);
        }
        else {
          BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(faceBox, m_nComp);
          m_faceData[dir] = newFabPtr;
        }
      }
    }
  }
```

Modifiers

327 ⟨FaceFabImplem.hw 325⟩ +≡

```
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val)
  {
    assert(m_nComp > 0);
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_faceData[dir] ≠ Λ);
      m_faceData[dir]→setVal(val);
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, int dir)
  {
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    m_faceData[dir]→setVal(val);
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, const int dir, const int startComp, const
        int nComp)
  {
    assert(startComp > −1);
    assert(startComp + nComp ≤ m_nComp);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    for (int comp = startComp; comp < startComp + nComp; comp++) {
      m_faceData[dir]→setVal(val, comp);
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, const Box &box)
  {
    assert(m_box.contains(box));
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_faceData[dir] ≠ Λ);
      Box faceBox(surroundingNodes(box, dir));
      m_faceData[dir]→setVal(val, faceBox, 0, m_nComp);
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, const Box &box, const int dir, const int
        startComp, const int nComp)
  {
    assert(m_box.contains(box));
```

```
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_faceData[dir] ≠ Λ);
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
  Box faceBox(surroundingNodes(box, dir));
  m_faceData[dir]→setVal(val, faceBox, startComp, nComp);
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const FaceFab⟨T⟩ &src)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0) ∧ (src.nComp( ) ≡ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    m_faceData[dir]→copy(src[dir]);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const FaceFab⟨T⟩ &src, const int srcComp, const int
        destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    m_faceData[dir]→copy(srcFab, srcComp, destComp, numComp);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const FaceFab⟨T⟩ &src, const int dir, const int
        srcComp, const int destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_faceData[dir] ≠ Λ);
  const BaseFab⟨T⟩ &srcFab = src[dir];
  m_faceData[dir]→copy(srcFab, srcComp, destComp, numComp);
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const Box &RegionFrom, const Interval &Cdest, const
        Box &RegionTo, const FaceFab⟨T⟩ &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    Box RfaceFrom(surroundingNodes(RegionFrom, dir));
    Box RfaceTo(surroundingNodes(RegionTo, dir));
    const BaseFab⟨T⟩ &srcFab = src[dir];
    RfaceTo &= (m_faceData[dir]→box( ));
    m_faceData[dir]→copy(RfaceFrom, Cdest, RfaceTo, srcFab, Csrc);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const Box &Region, const Interval &Cdest, const
        FaceFab⟨T⟩ &src, const Interval &Csrc)
```

```
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    Box faceRegion(surroundingNodes(Region, dir));
    const BaseFab<T> &srcFab = src[dir];
    faceRegion &= srcFab.box();
    faceRegion &= m_faceData[dir]→box();
    if (¬faceRegion.isEmpty()) {
      m_faceData[dir]→copy(faceRegion, Cdest, faceRegion, srcFab, Csrc);
    }
  }
}
template<class T> FaceFab<T> &FaceFab<T>::shift(const IntVect &iv)
{
  m_box.shift(iv);
  for (int dir = 0; dir < SpaceDim; dir++) {
    m_faceData[dir]→shift(iv);
  }
  return *this;
}
```

Transfer Helpers

328 ⟨FaceFabImplem.hw 325⟩ +≡

```
template<class T> int FaceFab<T>::size(const Box &box, const Interval &comps) const
{
  int totalSize = 0;
  BaseFab<T> tempFab;
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box faceBox(surroundingNodes(box, dir));
    const int dirSize = tempFab.size(faceBox, comps);
    totalSize += dirSize;
  }
  return totalSize;
}
template<class T> void FaceFab<T>::linearOut(void *buf, const Box &box, const Interval &comps)
        const
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box, dir));
    int dirSize = m_faceData[dir]→size(dirBox, comps);
    m_faceData[dir]→linearOut(buffer, dirBox, comps);
    buffer += dirSize/sizeof(T);
  }
}
template<class T> void FaceFab<T>::linearIn(void *buf, const Box &box, const Interval &comps)
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
```

```
        Box dirBox(surroundingNodes(box, dir));
        int dirSize = m_faceData[dir]->size(dirBox, comps);
        m_faceData[dir]->linearIn(buffer, dirBox, comps);
        buffer += dirSize/sizeof(T);
      }
    }
#endif
```

12.4 CurlBox Class

The **CurlBox** class is basically the **EdgeFab**⟨**T**⟩ template specified for **T** = **Real**. But we do this on the **BaseFab**⟨**T**⟩ level, replacing it with **FArrayBox**, which gives us **FArrayBox** arithmetic and direct access to Chombo HDF5/IO routines.

12.4.1 CurlBox Class Headers

330 ⟨ CurlBox.hw 330 ⟩ ≡

```
      /* Chombo includes */
#ifndef CH_CURLBOX_H
# define CH_CURLBOX_H
#include <Box.H>
#include <Vector.H>
#include <FArrayBox.H>
#include <REAL.H>
      /* C++ includes */
#include <cassert>
  class CurlBox {
  public: CurlBox( );
    CurlBox(const Box &box, int nComp = 1);
    ~CurlBox( );
    void resize(const Box &box, int nComp = 1);
    void define(const Box &box, int nComp = 1);
    void clear( );
    int nComp( ) const;
    const Box &box( ) const;
    FArrayBox &getEdgeData(const int dir);
    const FArrayBox &getEdgeData(const int dir) const;
    FArrayBox &operator[](const int dir);
    const FArrayBox &operator[](const int dir) const;
    void setVal(const Real val);
    void setVal(const Real val, const int dir);
    void setVal(const Real val, const int dir, const int startComp, const int nComp);
    void setVal(const Real val, const Box &box);
    void setVal(const Real val, const Box &box, const int dir, const int startComp, const int nComp);
    void copy(const CurlBox &src);
    void copy(const CurlBox &src, const int srcComp, const int destComp, const int numComp);
    void copy(const CurlBox &src, const int dir, const int srcComp, const int destComp, const int
        numComp);
    void copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const CurlBox
        &src, const Interval &Csrc);
    void copy(const Box &Region, const Interval &Cdest, const CurlBox &src, const Interval &Csrc);
    CurlBox &shift(const IntVect &v);
    int size(const Box &box, const Interval &comps) const;
    void linearOut(void *buf, const Box &R, const Interval &comps) const;
    void linearIn(void *buf, const Box &R, const Interval &comps);
    static int preAllocatable( )
    {
      return 0;
    }
  protected: Box m_box;
    int m_nComp;
    Vector⟨FArrayBox *⟩ m_edgeData;
```

```
  private: CurlBox(const CurlBox &);
    CurlBox &operator=(const CurlBox &);
  };
#endif
```

12.4.2 CurlBox Class Implementation

Extractors

332 ⟨ CurlBox.c 332 ⟩ ≡

```
#include <CurlBox.H>
  int CurlBox::nComp() const
  {
    return m_nComp;
  }
  const Box &CurlBox::box() const
  {
    return m_box;
  }
  FArrayBox &CurlBox::getEdgeData(const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
  const FArrayBox &CurlBox::getEdgeData(const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
  FArrayBox &CurlBox::operator[](const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
  const FArrayBox &CurlBox::operator[](const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
```

See also chunks 333, 334, and 335.

Constructors and Destructors

333 ⟨CurlBox.c 332⟩ +≡

```
CurlBox::CurlBox()
: m_edgeData(SpaceDim, Λ) {
  m_nComp = −1;
}
CurlBox::CurlBox(const Box &a_box, int a_nComp)
: m_edgeData(SpaceDim, Λ) {
  define(a_box, a_nComp);
}
CurlBox::~CurlBox()
{
  clear();
}
void CurlBox::clear()
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    if (m_edgeData[dir] ≠ Λ) {
      delete m_edgeData[dir];
      m_edgeData[dir] = Λ;
    }
  }
  m_nComp = −1;
  m_box = Box();
}
void CurlBox::define(const Box &a_box, int a_nComp)
{
  assert(a_nComp > 0);
  m_box = a_box;
  m_nComp = a_nComp;
  m_edgeData.resize(SpaceDim);
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box edgeBox(surroundingNodes(m_box));
    edgeBox.enclosedCells(dir);
    FArrayBox *newFabPtr = new FArrayBox(edgeBox, m_nComp);
    m_edgeData[dir] = newFabPtr;
  }
}
void CurlBox::resize(const Box &a_box, int a_nComp)
{
  if (m_nComp < 0) {
    define(a_box, a_nComp);
  }
  else {
    assert(a_nComp > 0);
    m_box = a_box;
    m_nComp = a_nComp;
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box edgeBox(surroundingNodes(m_box));
      edgeBox.enclosedCells(dir);
      if (m_edgeData[dir] ≠ Λ) {
        m_edgeData[dir]→resize(edgeBox, m_nComp);
      }
      else {
```

```
            FArrayBox *newFabPtr = new FArrayBox(edgeBox, m_nComp);
            m_edgeData[dir] = newFabPtr;
          }
        }
      }
    }
```

Modifiers

334 ⟨ CurlBox.c 332 ⟩ +≡

```
void CurlBox::setVal(const Real val)
{
  assert(m_nComp > 0);
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]→setVal(val);
  }
}
void CurlBox::setVal(const Real val, int dir)
{
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  m_edgeData[dir]→setVal(val);
}
void CurlBox::setVal(const Real val, const int dir, const int startComp, const int nComp)
{
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  for (int comp = startComp; comp < startComp + nComp; comp++) {
    m_edgeData[dir]→setVal(val, comp);
  }
}
void CurlBox::setVal(const Real val, const Box &box)
{
  assert(m_box.contains(box));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box edgeBox(surroundingNodes(box));
    edgeBox.enclosedCells(dir);
    m_edgeData[dir]→setVal(val, edgeBox, 0, m_nComp);
  }
}
void CurlBox::setVal(const Real val, const Box &box, const int dir, const int startComp, const int
        nComp)
{
  assert(m_box.contains(box));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
```

```
  Box edgeBox(surroundingNodes(box));
  edgeBox.enclosedCells(dir);
  m_edgeData[dir]→setVal(val, edgeBox, startComp, nComp);
}
void CurlBox::copy(const CurlBox &src)
{
  assert((src.nComp() > 0) ∧ (m_nComp > 0) ∧ (src.nComp() ≡ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]→copy(src[dir]);
  }
}
void CurlBox::copy(const CurlBox &src, const int srcComp, const int destComp, const int numComp)
{
  assert((src.nComp() > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp()) ∧ (destComp + numComp ≤ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    const FArrayBox &srcFab = src[dir];
    m_edgeData[dir]→copy(srcFab, srcComp, destComp, numComp);
  }
}
void CurlBox::copy(const CurlBox &src, const int dir, const int srcComp, const int destComp, const
        int numComp)
{
  assert((src.nComp() > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp()) ∧ (destComp + numComp ≤ m_nComp));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  const FArrayBox &srcFab = src[dir];
  m_edgeData[dir]→copy(srcFab, srcComp, destComp, numComp);
}
void CurlBox::copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const
        CurlBox &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box RedgeFrom(surroundingNodes(RegionFrom));
    RedgeFrom.enclosedCells(dir);
    Box RedgeTo(surroundingNodes(RegionTo));
    RedgeTo.enclosedCells(dir);
    const FArrayBox &srcFab = src[dir];
    RedgeTo &= (m_edgeData[dir]→box());
    m_edgeData[dir]→copy(RedgeFrom, Cdest, RedgeTo, srcFab, Csrc);
  }
}
void CurlBox::copy(const Box &Region, const Interval &Cdest, const CurlBox &src, const Interval
        &Csrc)
{
```

```
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_edgeData[dir] ≠ Λ);
      Box edgeRegion(surroundingNodes(Region));
      edgeRegion.enclosedCells(dir);
      const FArrayBox &srcFab = src[dir];
      edgeRegion &= srcFab.box();
      edgeRegion &= m_edgeData[dir]→box();
      if (¬edgeRegion.isEmpty()) {
        m_edgeData[dir]→copy(edgeRegion, Cdest, edgeRegion, srcFab, Csrc);
      }
    }
  }
  CurlBox &CurlBox::shift(const IntVect &iv)
  {
    m_box.shift(iv);
    for (int dir = 0; dir < SpaceDim; dir++) {
      m_edgeData[dir]→shift(iv);
    }
    return *this;
  }
```

Transfer Helpers

335 ⟨CurlBox.c 332⟩ +≡

```
  int CurlBox::size(const Box &box, const Interval &comps) const
  {
    int totalSize = 0;
    FArrayBox tempFab;
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box edgeBox(surroundingNodes(box));
      edgeBox.enclosedCells(dir);
      const int dirSize = tempFab.size(edgeBox, comps);
      totalSize += dirSize;
    }
    return totalSize;
  }
  void CurlBox::linearOut(void *buf, const Box &box, const Interval &comps) const
  {
    Real *buffer = (Real *) buf;
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_edgeData[dir] ≠ Λ);
      Box dirBox(surroundingNodes(box));
      dirBox.enclosedCells(dir);
      int dirSize = m_edgeData[dir]→size(dirBox, comps);
      m_edgeData[dir]→linearOut(buffer, dirBox, comps);
      buffer += dirSize/sizeof(Real);
    }
  }
  void CurlBox::linearIn(void *buf, const Box &box, const Interval &comps)
  {
    Real *buffer = (Real *) buf;
```

```
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearIn(buffer, dirBox, comps);
    buffer += dirSize/sizeof(Real);
  }
}
```

13 Fortran Subroutines

The Fortran subroutines presented in this section come with little comments, as they are not part of the Web system. Instead they have been included by the `\lstinputlisting` command of the *listings* package.

13.1 Subroutines `update_d` and `update_d_upml`

Update the ***D*** field by applying Maxwell equations. Both UPML and normal versions are provided on this file.

```fortran
C Fortran subroutines update_d and update_d_upml
C Update the D field by taking curl H − corrected for UPML
C if needed.
C
C $Id: update_d.f,v 1.8 2007/08/31 16:00:53 gustav Exp $
C
      subroutine update_d(
     & SpaceDim, margin, D_index, H_index, dt_by_dg,
     & imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx, Dx,
     & imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy, Dy,
     & imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz, Dz,
     & imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx, Hx,
     & imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy, Hy,
     & imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz, Hz,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, D_index(0:1), H_index(0:1),
     & fortran_verbose, return_status
C
      double precision dt_by_dg
C
      integer imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx
      double precision
     & Dx(imin_Dx:imax_Dx, jmin_Dx:jmax_Dx, kmin_Dx:kmax_Dx, 0:1)
C
      integer imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy
      double precision
     & Dy(imin_Dy:imax_Dy, jmin_Dy:jmax_Dy, kmin_Dy:kmax_Dy, 0:1)
C
      integer imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz
      double precision
     & Dz(imin_Dz:imax_Dz, jmin_Dz:jmax_Dz, kmin_Dz:kmax_Dz, 0:1)
C
      integer imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx
      double precision
     & Hx(imin_Hx:imax_Hx, jmin_Hx:jmax_Hx, kmin_Hx:kmax_Hx, 0:1)
C
      integer imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy
      double precision
     & Hy(imin_Hy:imax_Hy, jmin_Hy:jmax_Hy, kmin_Hy:kmax_Hy, 0:1)
C
      integer imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz
      double precision
     & Hz(imin_Hz:imax_Hz, jmin_Hz:jmax_Hz, kmin_Hz:kmax_Hz, 0:1)
C
      integer i, j, k, new_D, old_D, new_H
      integer ERR_ADVANCE_E
      parameter (ERR_ADVANCE_E = 5)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_d_): bad SpaceDim'
         return_status = ERR_ADVANCE_E
         goto 100
      end if
```

```
      new_D = D_index(0)
      old_D = D_index(1)
      new_H = H_index(0)

      do k = kmin_Dx + margin, kmax_Dx - margin
         do j = jmin_Dx + margin, jmax_Dx - margin
            do i = imin_Dx + margin, imax_Dx - margin
               Dx(i, j, k, new_D) = Dx(i, j, k, old_D) + dt_by_dg *
     & (Hz(i, j, k, new_H) - Hz(i, j - 1, k, new_H)
     & - Hy(i, j, k, new_H) + Hy(i, j, k - 1, new_H))
            end do
         end do
      end do

      do k = kmin_Dy + margin, kmax_Dy - margin
         do j = jmin_Dy + margin, jmax_Dy - margin
            do i = imin_Dy + margin, imax_Dy - margin
              Dy(i, j, k, new_D) = Dy(i, j, k, old_D) + dt_by_dg *
     & (Hx(i, j, k, new_H) - Hx(i, j, k - 1, new_H)
     & - Hz(i, j, k, new_H) + Hz(i - 1, j, k, new_H))
            end do
         end do
      end do

      do k = kmin_Dz + margin, kmax_Dz - margin
         do j = jmin_Dz + margin, jmax_Dz - margin
            do i = imin_Dz + margin, imax_Dz - margin
               Dz(i, j, k, new_D) = Dz(i, j, k, old_D) + dt_by_dg *
     & (Hy(i, j, k, new_H) - Hy(i - 1, j, k, new_H)
     & - Hx(i, j, k, new_H) + Hx(i, j - 1, k, new_H))
            end do
         end do
      end do

  100 continue

      return
      end
C
C
C
      subroutine update_d_upml(
     & SpaceDim, margin, D_index, H_index,
     & imin_Cx, imax_Cx, C1x, C2x,
     & jmin_Cy, jmax_Cy, C1y, C2y,
     & kmin_Cz, kmax_Cz, C1z, C2z,
     & imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx, Dx,
     & imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy, Dy,
     & imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz, Dz,
     & imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx, Hx,
     & imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy, Hy,
     & imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz, Hz,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, D_index(0:1), H_index(0:1),
     & fortran_verbose, return_status
C
```

```
      integer imin_Cx, imax_Cx
      double precision C1x(imin_Cx:imax_Cx), C2x(imin_Cx:imax_Cx)
C
      integer jmin_Cy, jmax_Cy
      double precision C1y(jmin_Cy:jmax_Cy), C2y(jmin_Cy:jmax_Cy)
C
      integer kmin_Cz, kmax_Cz
      double precision C1z(kmin_Cz:kmax_Cz), C2z(kmin_Cz:kmax_Cz)
C
      integer imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx
      double precision
     & Dx(imin_Dx:imax_Dx, jmin_Dx:jmax_Dx, kmin_Dx:kmax_Dx, 0:1)
C
      integer imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy
      double precision
     & Dy(imin_Dy:imax_Dy, jmin_Dy:jmax_Dy, kmin_Dy:kmax_Dy, 0:1)
C
      integer imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz
      double precision
     & Dz(imin_Dz:imax_Dz, jmin_Dz:jmax_Dz, kmin_Dz:kmax_Dz, 0:1)
C
      integer imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx
      double precision
     & Hx(imin_Hx:imax_Hx, jmin_Hx:jmax_Hx, kmin_Hx:kmax_Hx, 0:1)
C
      integer imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy
      double precision
     & Hy(imin_Hy:imax_Hy, jmin_Hy:jmax_Hy, kmin_Hy:kmax_Hy, 0:1)
C
      integer imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz
      double precision
     & Hz(imin_Hz:imax_Hz, jmin_Hz:jmax_Hz, kmin_Hz:kmax_Hz, 0:1)
C
      integer i, j, k, new_D, old_D, new_H
      integer ERR_ADVANCE_E
      parameter (ERR_ADVANCE_E = 5)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_d_upml_): Bad SpaceDim'
         return_status = ERR_ADVANCE_E
         goto 100
      end if

      new_D = D_index(0)
      old_D = D_index(1)
      new_H = H_index(0)

      do k = kmin_Dx + margin, kmax_Dx - margin
         do j = jmin_Dx + margin, jmax_Dx - margin
            do i = imin_Dx + margin, imax_Dx - margin
               Dx(i, j, k, new_D) = C1y(j) * Dx(i, j, k, old_D)
     & + C2y(j) * (Hz(i, j, k, new_H)
     & - Hz(i, j - 1, k, new_H)
     & - Hy(i, j, k, new_H)
     & + Hy(i, j, k - 1, new_H))
            end do
         end do
```

```
      end do

      do k = kmin_Dy + margin, kmax_Dy - margin
         do j = jmin_Dy + margin, jmax_Dy - margin
            do i = imin_Dy + margin, imax_Dy - margin
               Dy(i, j, k, new_D) = C1z(k) * Dy(i, j, k, old_D)
     & + C2z(k) * (Hx(i, j, k, new_H)
     & - Hx(i, j, k - 1, new_H)
     & - Hz(i, j, k, new_H) + Hz(i - 1, j, k, new_H))
            end do
         end do
      end do

      do k = kmin_Dz + margin, kmax_Dz - margin
         do j = jmin_Dz + margin, jmax_Dz - margin
            do i = imin_Dz + margin, imax_Dz - margin
               Dz(i, j, k, new_D) = C1x(i) * Dz(i, j, k, old_D)
     & + C2x(i) * (Hy(i, j, k, new_H)
     & - Hy(i - 1, j, k, new_H)
     & - Hx(i, j, k, new_H) + Hx(i, j - 1, k, new_H))
            end do
         end do
      end do

  100 continue

      return
      end

C
C $Log: update_d.f,v $
C Revision 1.8 2007/08/31 16:00:53 gustav
C Combined update_d_ and update_d_upml_ on this single file.
C Added handling of the new/old-field index to both.
C
C
```

13.2 Subroutines `update_b` and `update_b_upml`

Update the $\boldsymbol{B}$ field by applying Maxwell equations. Both UPML and normal versions are provided on this file.

```
C Fortran subroutine update_b
C
C Update the B field, both pure and UPML version.
C
C $Id: update_b.f,v 1.5 2007/09/03 19:04:02 gustav Exp $
C
C Pure Maxwell first.
C
      subroutine update_b(
     & SpaceDim, margin, B_index, E_index, dt_by_dg,
     & imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx, Bx,
     & imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By, By,
     & imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz, Bz,
     & imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex, Ex,
     & imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey, Ey,
     & imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez, Ez,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, B_index(0:1), E_index(0:1),
     & fortran_verbose, return_status
C
      double precision dt_by_dg
C
      integer imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx
      double precision
     & Bx(imin_Bx:imax_Bx, jmin_Bx:jmax_Bx, kmin_Bx:kmax_Bx, 0:1)
C
      integer imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By
      double precision
     & By(imin_By:imax_By, jmin_By:jmax_By, kmin_By:kmax_By, 0:1)
C
      integer imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz
      double precision
     & Bz(imin_Bz:imax_Bz, jmin_Bz:jmax_Bz, kmin_Bz:kmax_Bz, 0:1)
C
      integer imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex
      double precision
     & Ex(imin_Ex:imax_Ex, jmin_Ex:jmax_Ex, kmin_Ex:kmax_Ex, 0:1)
C
      integer imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey
      double precision
     & Ey(imin_Ey:imax_Ey, jmin_Ey:jmax_Ey, kmin_Ey:kmax_Ey, 0:1)
C
      integer imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez
      double precision
     & Ez(imin_Ez:imax_Ez, jmin_Ez:jmax_Ez, kmin_Ez:kmax_Ez, 0:1)
C
      integer i, j, k, new_B, old_B, new_E
      integer ERR_ADVANCE_H
      parameter (ERR_ADVANCE_H = 6)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_b_): bad SapceDim'
         return_status = ERR_ADVANCE_H
```

```
          goto 100
      end if

      new_B = B_index(0)
      old_B = B_index(1)
      new_E = E_index(0)

      do k = kmin_Bx + margin, kmax_Bx − margin
         do j = jmin_Bx + margin, jmax_Bx − margin
            do i = imin_Bx + margin, imax_Bx − margin
               Bx(i, j, k, new_B) = Bx(i, j, k, old_B) + dt_by_dg ∗
     & (Ey(i, j, k + 1, new_E) − Ey(i, j, k, new_E)
     & − Ez(i, j + 1, k, new_E) + Ez(i, j, k, new_E))
            end do
         end do
      end do

      do k = kmin_By + margin, kmax_By − margin
         do j = jmin_By + margin, jmax_By − margin
            do i = imin_By + margin, imax_By − margin
               By(i, j, k, new_B) = By(i, j, k, old_B) + dt_by_dg ∗
     & (Ez(i + 1, j, k, new_E) − Ez(i, j, k, new_E)
     & − Ex(i, j, k + 1, new_E) + Ex(i, j, k, new_E))
            end do
         end do
      end do

      do k = kmin_Bz + margin, kmax_Bz − margin
         do j = jmin_Bz + margin, jmax_Bz − margin
            do i = imin_Bz + margin, imax_Bz − margin
               Bz(i, j, k, new_B) = Bz(i, j, k, old_B) + dt_by_dg ∗
     & (Ex(i, j + 1, k, new_E) − Ex(i, j, k, new_E)
     & − Ey(i + 1, j, k, new_E) + Ey(i, j, k, new_E))
            end do
         end do
      end do

  100 continue

      return
      end
C
C The UPML version.
C
      subroutine update_b_upml(
     & SpaceDim, margin, B_index, E_index,
     & imin_Cx, imax_Cx, C1x, C2x,
     & jmin_Cy, jmax_Cy, C1y, C2y,
     & kmin_Cz, kmax_Cz, C1z, C2z,
     & imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx, Bx,
     & imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By, By,
     & imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz, Bz,
     & imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex, Ex,
     & imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey, Ey,
     & imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez, Ez,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, B_index(0:1), E_index(0:1),
```

```
     & fortran_verbose, return_status
C
      integer imin_Cx, imax_Cx
      double precision C1x(imin_Cx:imax_Cx), C2x(imin_Cx:imax_Cx)
C
      integer jmin_Cy, jmax_Cy
      double precision C1y(jmin_Cy:jmax_Cy), C2y(jmin_Cy:jmax_Cy)
C
      integer kmin_Cz, kmax_Cz
      double precision C1z(kmin_Cz:kmax_Cz), C2z(kmin_Cz:kmax_Cz)
C
      integer imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx
      double precision
     & Bx(imin_Bx:imax_Bx, jmin_Bx:jmax_Bx, kmin_Bx:kmax_Bx, 0:1)
C
      integer imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By
      double precision
     & By(imin_By:imax_By, jmin_By:jmax_By, kmin_By:kmax_By, 0:1)
C
      integer imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz
      double precision
     & Bz(imin_Bz:imax_Bz, jmin_Bz:jmax_Bz, kmin_Bz:kmax_Bz, 0:1)
C
      integer imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex
      double precision
     & Ex(imin_Ex:imax_Ex, jmin_Ex:jmax_Ex, kmin_Ex:kmax_Ex, 0:1)
C
      integer imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey
      double precision
     & Ey(imin_Ey:imax_Ey, jmin_Ey:jmax_Ey, kmin_Ey:kmax_Ey, 0:1)
C
      integer imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez
      double precision
     & Ez(imin_Ez:imax_Ez, jmin_Ez:jmax_Ez, kmin_Ez:kmax_Ez, 0:1)
C
      integer i, j, k, new_B, old_B, new_E
      integer ERR_ADVANCE_H
      parameter (ERR_ADVANCE_H = 6)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_b_upml_): Bad SpaceDim'
         return_status = ERR_ADVANCE_H
         goto 100
      end if

      new_B = B_index(0)
      old_B = B_index(1)
      new_E = E_index(0)

      do k = kmin_Bx + margin, kmax_Bx - margin
         do j = jmin_Bx + margin, jmax_Bx - margin
            do i = imin_Bx + margin, imax_Bx - margin
               Bx(i, j, k, new_B) = C1y(j) * Bx(i, j, k, old_B)
     & + C2y(j) * (Ey(i, j, k + 1, new_E)
     & - Ey(i, j, k, new_E)
     & - Ez(i, j + 1, k, new_E) + Ez(i, j, k, new_E))
            end do
```

```
      end do
    end do

    do k = kmin_By + margin, kmax_By − margin
      do j = jmin_By + margin, jmax_By − margin
        do i = imin_By + margin, imax_By − margin
          By(i, j, k, new_B) = C1z(k) ∗ By(i, j, k, old_B)
     & + C2z(k) ∗ (Ez(i + 1, j, k, new_E)
     & − Ez(i, j, k, new_E)
     & − Ex(i, j, k + 1, new_E) + Ex(i, j, k, new_E))
        end do
      end do
    end do

    do k = kmin_Bz + margin, kmax_Bz − margin
      do j = jmin_Bz + margin, jmax_Bz − margin
        do i = imin_Bz + margin, imax_Bz − margin
          Bz(i, j, k, new_B) = C1x(i) ∗ Bz(i, j, k, old_B)
     & + C2x(i) ∗ (Ex(i, j + 1, k, new_E)
     & − Ex(i, j, k, new_E)
     & − Ey(i + 1, j, k, new_E) + Ey(i, j, k, new_E))
        end do
      end do
    end do

100 continue

    return
    end
C
C $Log: update_b.f,v $
C Revision 1.5 2007/09/03 19:04:02 gustav
C Forgot to initialize new_B, old_B and new_E in update_b_upml_.
C
C Revision 1.4 2007/09/03 17:45:44 gustav
C Combined non−UPML and UPML versions on a single file.
C Added handling of the new/old−index switch.
C
C
```

13.3 *D*-to-*E* Conversion Subroutines

```
C Convert D to E within the region provided, taking care of UPMLs,
C for level 0 and otherwise for levels other than zero.
C
C $Id: d_to_e.f,v 1.10 2007/09/28 00:41:26 gustav Exp $
C
C Level zero case
C
C C++ declaration
C
C This first function assumes no auxiliary fields.
C
C extern"C"
C {
C void d_to_e_0_0_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const E_idx,
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_Cx,
C const int*const imax_Cx,
C const Real*const C0x,
C const Real*const C1x,
C const Real*const C3x,
C const Real*const C4x,
C const int*const imin_Cy,
C const int*const imax_Cy,
C const Real*const C0y,
C const Real*const C1y,
C const Real*const C3y,
C const Real*const C4y,
C const int*const imin_Cz,
C const int*const imax_Cz,
C const Real*const C0z,
C const Real*const C1z,
C const Real*const C3z,
C const Real*const C4z,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_0_0(
     & SpaceDim, dir, margin,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_Cx, imax_Cx, C0x, C1x, C3x, C4x,
```

```
     & imin_Cy, imax_Cy, C0y, C1y, C3y, C4y,
     & imin_Cz, imax_Cz, C0z, C1z, C3z, C4z,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
      integer imin_Cx, imax_Cx
      double precision
     & C0x(imin_Cx:imax_Cx),
     & C1x(imin_Cx:imax_Cx),
     & C3x(imin_Cx:imax_Cx),
     & C4x(imin_Cx:imax_Cx)
      integer imin_Cy, imax_Cy
      double precision
     & C0y(imin_Cy:imax_Cy),
     & C1y(imin_Cy:imax_Cy),
     & C3y(imin_Cy:imax_Cy),
     & C4y(imin_Cy:imax_Cy)
      integer imin_Cz, imax_Cz
      double precision
     & C0z(imin_Cz:imax_Cz),
     & C1z(imin_Cz:imax_Cz),
     & C3z(imin_Cz:imax_Cz),
     & C4z(imin_Cz:imax_Cz)
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, new_E, new_D, old_E, old_D
      double precision C0, C1, C3, C4
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external function
C
      double precision scm_d_to_e_0
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_0_0_): bad SpaceDim'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
```

```
            write(*,*) 'ERROR(d_to_e_0_0_): bad dir'
            return_status = ERROR_IN_D_TO_E
            goto 100
        end if
C
        new_E = E_idx(0)
        old_E = E_idx(1)
        new_D = D_idx(0)
        old_D = D_idx(1)
C
        do k = kmin_D + margin, kmax_D - margin
            do j = jmin_D + margin, jmax_D - margin
                do i = imin_D + margin, imax_D - margin
                    select case (medium_E(i, j, k))
                        case (-2)
                            select case (dir)
                                case (0)
                                    C0 = C0z(k)
                                    C1 = C1z(k)
                                    C3 = C3x(i)
                                    C4 = C4x(i)
                                case (1)
                                    C0 = C0x(i)
                                    C1 = C1x(i)
                                    C3 = C3y(j)
                                    C4 = C4y(j)
                                case (2)
                                    C0 = C0y(j)
                                    C1 = C1y(j)
                                    C3 = C3z(k)
                                    C4 = C4z(k)
                            end select
                            E(i, j, k, new_E) = C1 * E(i, j, k, old_E)
     & + C3 * C0 * D(i, j, k, new_D)
     & - C4 * C0 * D(i, j, k, old_D)
                        case (-1:0)
                            E(i,j,k, new_E) = D(i,j,k, new_D)
                        case default
                            E(i,j,k, new_E) = scm_d_to_e_0(
     & dir, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & time_e, dt)
                    end select
                end do
            end do
        end do
C
 100 continue
        return
        end
C
C C++ declaration
C
C This function is similar to the one above, but it makes
C provision for passing of auxiliary fields.
C
C extern"C"
C {
```

```
C void d_to_e_0_n_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const number_of_auxiliary_fields,
C const int*const E_idx,
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_Cx,
C const int*const imax_Cx,
C const Real*const C0x,
C const Real*const C1x,
C const Real*const C3x,
C const Real*const C4x,
C const int*const imin_Cy,
C const int*const imax_Cy,
C const Real*const C0y,
C const Real*const C1y,
C const Real*const C3y,
C const Real*const C4y,
C const int*const imin_Cz,
C const int*const imax_Cz,
C const Real*const C0z,
C const Real*const C1z,
C const Real*const C3z,
C const Real*const C4z,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const Real*const S,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_0_n(
     & SpaceDim, dir, margin, n_aux_fields,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_Cx, imax_Cx, C0x, C1x, C3x, C4x,
     & imin_Cy, imax_Cy, C0y, C1y, C3y, C4y,
     & imin_Cz, imax_Cz, C0z, C1z, C3z, C4z,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, S, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, n_aux_fields,
     & E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
```

```
      integer imin_Cx, imax_Cx
      double precision
     & C0x(imin_Cx:imax_Cx),
     & C1x(imin_Cx:imax_Cx),
     & C3x(imin_Cx:imax_Cx),
     & C4x(imin_Cx:imax_Cx)
      integer imin_Cy, imax_Cy
      double precision
     & C0y(imin_Cy:imax_Cy),
     & C1y(imin_Cy:imax_Cy),
     & C3y(imin_Cy:imax_Cy),
     & C4y(imin_Cy:imax_Cy)
      integer imin_Cz, imax_Cz
      double precision
     & C0z(imin_Cz:imax_Cz),
     & C1z(imin_Cz:imax_Cz),
     & C3z(imin_Cz:imax_Cz),
     & C4z(imin_Cz:imax_Cz)
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & S(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, n_aux_fields)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, n, n_max, new_E, old_E, new_D, old_D
      parameter (n_max = 100)
      double precision C0, C1, C3, C4
      double precision S_vector(n_max)
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external functions
C
      double precision scm_d_to_e_n
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_0_n_): bad SpaceDim'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(d_to_e_0_n_): bad dir'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if (n_aux_fields .gt. n_max) then
         write(*,*) 'ERROR(d_to_e_0_n_): too many fields'
         return_status = ERROR_IN_D_TO_E
```

```
            goto 100
         end if
C
         new_E = E_idx(0)
         old_E = E_idx(1)
         new_D = D_idx(0)
         old_D = D_idx(1)
C
         do k = kmin_D + margin, kmax_D − margin
            do j = jmin_D + margin, jmax_D − margin
               do i = imin_D + margin, imax_D − margin
                  select case (medium_E(i, j, k))
                     case (−2)
                        select case (dir)
                           case (0)
                              C0 = C0z(k)
                              C1 = C1z(k)
                              C3 = C3x(i)
                              C4 = C4x(i)
                           case (1)
                              C0 = C0x(i)
                              C1 = C1x(i)
                              C3 = C3y(j)
                              C4 = C4y(j)
                           case (2)
                              C0 = C0y(j)
                              C1 = C1y(j)
                              C3 = C3z(k)
                              C4 = C4z(k)
                        end select
                        E(i, j, k, new_E) = C1 ∗ E(i, j, k, old_E)
     & + C3 ∗ C0 ∗ D(i, j, k, new_D)
     & − C4 ∗ C0 ∗ D(i, j, k, old_D)
                     case (−1:0)
                        E(i,j,k,new_E) = D(i,j,k,new_D)
                     case default
                        do n = 1, n_aux_fields
                           S_vector(n) = S(i,j,k,n)
                        end do
                        E(i,j,k,new_E) = scm_d_to_e_n(
     & dir, n_aux_fields, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & S_vector, time_e, dt)
                        do n = 1, n_aux_fields
                           S(i,j,k,n) = S_vector(n)
                        end do
                  end select
               end do
            end do
         end do
C
 100 continue
         return
         end
C
C Higher levels case
C
C C++ declaration
```

```
C
C This first function for higher levels assumes no auxiliary fields.
C
C extern"C"
C {
C void d_to_e_n_0_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const E_idx,
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_n_0(
     & SpaceDim, dir, margin,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, new_E, old_E, new_D, old_D
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external function
C
      double precision scm_d_to_e_0
C
C action
```

```
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_n_0_): bad SpaceDim'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(d_to_e_n_0_): bad dir'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      new_E = E_idx(0)
      old_E = E_idx(1)
      new_D = D_idx(0)
      old_D = D_idx(1)
C
      do k = kmin_D + margin, kmax_D - margin
         do j = jmin_D + margin, jmax_D - margin
            do i = imin_D + margin, imax_D - margin
               select case (medium_E(i, j, k))
                  case (-2)
                     write(*,*)
     & 'ERROR(d_to_e_n_0_): no media handler'
                     return_status = ERROR_IN_D_TO_E
                     goto 100
                  case (-1:0)
                     E(i,j,k,new_E) = D(i,j,k,new_D)
                  case default
                     E(i,j,k,new_E) = scm_d_to_e_0(
     & dir, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & time_e, dt)
               end select
            end do
         end do
      end do
C
  100 continue
      return
      end
C
C C++ declaration
C
C This function is similar to the one above, but it makes
C provision for passing of auxiliary fields.
C
C extern"C"
C {
C void d_to_e_n_n_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const number_of_auxiliary_fields,
C const int*const E_idx,
```

```
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const Real*const S,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_n_n(
     & SpaceDim, dir, margin, n_aux_fields,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, S, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, n_aux_fields,
     & E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & S(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, n_aux_fields)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, n, n_max, new_E, old_E, new_D, old_D
      parameter (n_max = 100)
      double precision S_vector(n_max)
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external functions
C
      double precision scm_d_to_e_n
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_n_n_): bad SpaceDim'
```

```
      return_status = ERROR_IN_D_TO_E
      goto 100
    end if
C
    if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
      write(*,*) 'ERROR(d_to_e_n_n_): bad dir'
      return_status = ERROR_IN_D_TO_E
      goto 100
    end if
C
    if (n_aux_fields .gt. n_max) then
      write(*,*) 'ERROR(d_to_e_n_n_): too many fields'
      return_status = ERROR_IN_D_TO_E
      goto 100
    end if
C
    new_E = E_idx(0)
    old_E = E_idx(1)
    new_D = D_idx(0)
    old_D = D_idx(1)
C
    do k = kmin_D + margin, kmax_D - margin
      do j = jmin_D + margin, jmax_D - margin
        do i = imin_D + margin, imax_D - margin
          select case (medium_E(i, j, k))
            case (-2)
              write(*,*)
     & 'ERROR(d_to_e_n_n_): no media handler'
              return_status = ERROR_IN_D_TO_E
              goto 100
            case (-1:0)
              E(i,j,k,new_E) = D(i,j,k,new_D)
            case default
              do n = 1, n_aux_fields
                S_vector(n) = S(i,j,k,n)
              end do
              E(i,j,k,new_E) = scm_d_to_e_n(
     & dir, n_aux_fields, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & S_vector, time_e, dt)
              do n = 1, n_aux_fields
                S(i,j,k,n) = S_vector(n)
              end do
          end select
        end do
      end do
    end do
C
 100 continue
    return
    end
C
C $Log: d_to_e.f,v $
C Revision 1.10 2007/09/28 00:41:26 gustav
C + C4 -> - C4.
C
C Revision 1.9 2007/08/31 20:56:08 gustav
C Changed the order of parameters as passed to the functions with
```

```
C Auxiliary fields. There is now D, E, S, medium_E.
C
C Revision 1.8 2007/08/31 20:46:04 gustav
C Added handling of E_idx and D_idx and deleted any references to
C D_old and E_old.
C
C Revision 1.7 2007/08/28 20:32:27 gustav
C Fixed a bug in the declaration of scm_d_to_e_0. It was declared as
C integer, which caused a significant slow-down of the code. Probably
C generating NANs.
C
C Revision 1.6 2007/08/28 18:08:13 gustav
C Added handlers for functions for media with auxiliary fields.
C changed error exit to 33 everywhere.
C Added error messages on errors encountered and caught.
C Simplified logic, because errors are handled up front.
C
C Revision 1.5 2007/08/27 21:45:26 gustav
C added dir to the call to scm_d_to_e_0.
C
C Revision 1.4 2007/08/27 21:35:02 gustav
C Function scm_d_to_e_0 now returns E, not return_status.
C
C Revision 1.3 2007/08/27 18:10:32 gustav
C Added calls to an external function scm_d_to_e_0, which is going to
C handle Scheme driven conversion of D to E at this point. Only for the
C case with no auxiliary fields for the time being.
C
C Revision 1.2 2007/08/23 21:02:39 gustav
C Added subroutines with auxiliary fields.
C
C Revision 1.1 2007/08/23 20:32:32 gustav
C Initial revision
C
C
```

13.4 *B*-to-*H* Conversion Subroutines

```
C Convert B to H within the region provided, taking care of UPMLs
C for level 0 and otherwise for levels other than zero.
C
C $Id: b_to_h.f,v 1.3 2007/09/28 00:41:39 gustav Exp $
C
C Level zero case
C
C C++ declaration
C
C extern"C"
C {
C void b_to_h_0_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const H_idx,
C const int*const B_idx,
C const Real*const time_h,
C const Real*const dt,
C const int*const imin_Cx,
C const int*const imax_Cx,
C const Real*const C0x,
C const Real*const C1x,
C const Real*const C3x,
C const Real*const C4x,
C const int*const imin_Cy,
C const int*const imax_Cy,
C const Real*const C0y,
C const Real*const C1y,
C const Real*const C3y,
C const Real*const C4y,
C const int*const imin_Cz,
C const int*const imax_Cz,
C const Real*const C0z,
C const Real*const C1z,
C const Real*const C3z,
C const Real*const C4z,
C const int*const imin_B,
C const int*const jmin_B,
C const int*const kmin_B,
C const int*const imax_B,
C const int*const jmax_B,
C const int*const kmax_B,
C const Real*const B,
C const Real*const H,
C const int*const medium_H,
C const int*const watch_b_to_h,
C const int*const return_status
C );
C }
C
      subroutine b_to_h_0(
     & SpaceDim, dir, margin, H_idx, B_idx,
     & time_h, dt,
     & imin_Cx, imax_Cx, C0x, C1x, C3x, C4x,
     & imin_Cy, imax_Cy, C0y, C1y, C3y, C4y,
     & imin_Cz, imax_Cz, C0z, C1z, C3z, C4z,
     & imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B,
```

```
     & B, H, medium_H,
     & watch_b_to_h, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, H_idx(0:1), B_idx(0:1)
      double precision time_h, dt
      integer imin_Cx, imax_Cx
      double precision
     & C0x(imin_Cx:imax_Cx),
     & C1x(imin_Cx:imax_Cx),
     & C3x(imin_Cx:imax_Cx),
     & C4x(imin_Cx:imax_Cx)
      integer imin_Cy, imax_Cy
      double precision
     & C0y(imin_Cy:imax_Cy),
     & C1y(imin_Cy:imax_Cy),
     & C3y(imin_Cy:imax_Cy),
     & C4y(imin_Cy:imax_Cy)
      integer imin_Cz, imax_Cz
      double precision
     & C0z(imin_Cz:imax_Cz),
     & C1z(imin_Cz:imax_Cz),
     & C3z(imin_Cz:imax_Cz),
     & C4z(imin_Cz:imax_Cz)
      integer imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B
      double precision
     & B(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1),
     & H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1)
      integer
     & medium_H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B)
      integer watch_b_to_h, return_status
C
C internal variables
C
      integer i, j, k, new_H, old_H, new_B, old_B
      double precision C0, C1, C3, C4
      integer ERROR_IN_B_TO_H, NO_ERROR
      parameter (ERROR_IN_B_TO_H = 9, NO_ERROR = 0)
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(b_to_h_0_): bad SpaceDim'
         return_status = ERROR_IN_B_TO_H
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(b_to_h_0_): bad dir'
         return_status = ERROR_IN_B_TO_H
         goto 100
      end if
C
      new_H = H_idx(0)
      old_H = H_idx(1)
```

```
      new_B = B_idx(0)
      old_B = B_idx(1)
C
      do k = kmin_B + margin, kmax_B − margin
         do j = jmin_B + margin, jmax_B − margin
            do i = imin_B + margin, imax_B − margin
               select case (medium_H(i, j, k))
                  case (−2)
                     select case (dir)
                        case (0)
                           C0 = C0z(k)
                           C1 = C1z(k)
                           C3 = C3x(i)
                           C4 = C4x(i)
                        case (1)
                           C0 = C0x(i)
                           C1 = C1x(i)
                           C3 = C3y(j)
                           C4 = C4y(j)
                        case (2)
                           C0 = C0y(j)
                           C1 = C1y(j)
                           C3 = C3z(k)
                           C4 = C4z(k)
                     end select
                     H(i, j, k, new_H) = C1 ∗ H(i, j, k, old_H)
     & + C3 ∗ C0 ∗ B(i, j, k, new_B)
     & − C4 ∗ C0 ∗ B(i, j, k, old_B)
                  case default
                     H(i, j, k, new_H) = B(i, j, k, new_B)
               end select
            end do
         end do
      end do
C
 100 continue
      return
      end
C
C Higher levels case
C
C C++ declaration
C
C extern"C"
C {
C void b_to_h_n_(
C const int∗const space_dim,
C const int∗const dir,
C const int∗const ghost_margin,
C const int∗const H_idx,
C const int∗const B_idx,
C const Real∗const time_h,
C const Real∗const dt,
C const int∗const imin_B,
C const int∗const jmin_B,
C const int∗const kmin_B,
C const int∗const imax_B,
C const int∗const jmax_B,
C const int∗const kmax_B,
```

```
C const Real*const B,
C const Real*const H,
C const int*const medium_H,
C const int*const watch_b_to_h,
C const int*const return_status
C );
C }
C
      subroutine b_to_h_n(
     & SpaceDim, dir, margin, H_idx, B_idx,
     & time_h, dt,
     & imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B,
     & B, H, medium_H,
     & watch_b_to_h, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, H_idx(0:1), B_idx(0:1)
      double precision time_h, dt
      integer imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B
      double precision
     & B(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1),
     & H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1)
      integer
     & medium_H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B)
      integer watch_b_to_h, return_status
C
C internal variables
C
      integer i, j, k, new_H, new_B
      integer ERROR_IN_B_TO_H, NO_ERROR
      parameter (ERROR_IN_B_TO_H = 9, NO_ERROR = 0)
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(b_to_h_n_): bad SpaceDim'
         return_status = ERROR_IN_B_TO_H
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(b_to_h_n_): bad dir'
         return_status = ERROR_IN_B_TO_H
         goto 100
      end if
C
      new_H = H_idx(0)
      new_B = B_idx(0)
C
      do k = kmin_B + margin, kmax_B - margin
         do j = jmin_B + margin, jmax_B - margin
            do i = imin_B + margin, imax_B - margin
               select case (medium_H(i, j, k))
                  case(-2)
                     write(*,*)
```

```
     & 'ERROR(b_to_h_n_): no media handler'
                        return_status = ERROR_IN_B_TO_H
                        goto 100
                    case default
                        H(i,j,k, new_H) = B(i,j,k, new_B)
                end select
            end do
        end do
    end do
C
 100 continue
    return
    end
C
C $Log: b_to_h.f,v $
C Revision 1.3 2007/09/28 00:41:39 gustav
C + C4 -> - C4
C
C Revision 1.2 2007/09/03 18:22:19 gustav
C Implemented the new/old index switch.
C
C Revision 1.1 2007/08/23 19:07:04 gustav
C Initial revision
C
C
```

13.5 Fortran Subroutines for Drawing on Chombo Data Structures

```
C
C $Id: draw.f,v 1.22 2009/08/20 20:26:02 gustav Exp $
C
C Auxiliary vector algebra functions
C
C We call them my_dot_product and my_norm, because Fortran
C has its own special intrinsics with these names.
C
      double precision function my_dot_product(dim, a, b)
      implicit none

      integer dim
      double precision a(dim), b(dim)

      double precision sum
      integer i

      sum = 0.0
      do i = 1, dim
         sum = sum + a(i) * b(i)
      end do

      my_dot_product = sum

      return
      end
C
      subroutine my_cross_product (dim, a, b, c)
      implicit none

      integer dim
      double precision a(dim), b(dim), c(dim)

      if (dim .ne. 3) then
         write(*,*) 'ERROR(my_cross_product): bad dimension'
         return
      end if

      c(1) = a(2) * b(3) - a(3) * b(2)
      c(2) = a(3) * b(1) - a(1) * b(3)
      c(3) = a(1) * b(2) - a(2) * b(1)

      return
      end
C
      double precision function my_norm(dim, a)
      implicit none

      integer dim
      double precision a(dim)
C
C external function--defined above
C
      double precision my_dot_product

      my_norm = sqrt(my_dot_product(dim, a, a))

      return
```

```
      end
C
      subroutine normalize(dim, a, a_normalized)
      implicit none

      integer dim
      double precision a(dim), a_normalized(dim)

      integer i
      double precision norm_a
C
C external function
C
      double precision my_norm

      norm_a = my_norm(dim, a)

      if (norm_a .eq. 0.0) then
         write(*,*) 'ERROR(normalize): zero norm'
         do i = 1, dim
            a_normalized(i) = 0.0
         end do
         return
      end if

      do i = 1, dim
         a_normalized(i) = a(i)/norm_a
      end do

      return
      end
C
      subroutine perpendicular_to(dim, a, a_perpendicular)
      implicit none

      integer dim
      double precision a(dim), a_perpendicular(dim)

      integer i

      if (dim .ne. 3) then
         write(*,*) 'ERROR(perpendicular_to): bad dimension'
         return
      end if

      do i = 1, 3
         a_perpendicular(i) = 0.0
      end do

      if ((a(1) .eq. 0.0) .and. (a(2) .eq. 0.0)) then
         if (a(3) .eq. 0.0) then
            return
         else
            a_perpendicular(1) = a(3)
            return
         end if
      else
         a_perpendicular(1) = - a(2)
         a_perpendicular(2) = a(1)
```

```
      return
    end if

    end
C
    double precision function volume (dim, v_1, v_2, v_3)
    implicit none

    integer dim
    double precision v_1(dim), v_2(dim), v_3(dim)

    if (dim .ne. 3) then
      write(*,*) 'ERROR(volume): bad dimension'
      return
    end if

    volume =
   & v_1(1) * v_2(2) * v_3(3)
   & + v_2(1) * v_3(2) * v_1(3)
   & + v_3(1) * v_1(2) * v_2(3)
   & - v_2(1) * v_1(2) * v_3(3)
   & - v_1(1) * v_3(2) * v_2(3)
   & - v_3(1) * v_2(2) * v_1(3)

    return
    end
C
C Auxiliary function determining if a point is within a cylinder
C
    logical function in_cylinder(dim, x_0, end_1, end_2, radius)
    implicit none

    integer dim
    double precision x_0(dim), end_1(dim), end_2(dim), radius

    double precision r(3), x(3), x_cross_r(3), norm_of_r, x_on_r,
   & x_on_r_perp
    integer i
C
C external functions--defined above
C
    double precision my_norm
    double precision my_dot_product

    in_cylinder = .false.

    if (dim .ne. 3) then
      write(*,*) 'ERROR(in_cylinder): bad dimension'
      return
    end if

    do i = 1, 3
      r(i) = end_2(i) - end_1(i)
      x(i) = x_0(i) - end_1(i)
    end do

    norm_of_r = my_norm(dim, r)
    x_on_r = my_dot_product(dim, x, r) / norm_of_r
```

```
      if ((0 .le. x_on_r) .and. (x_on_r .le. norm_of_r)) then
         call my_cross_product (dim, x, r, x_cross_r)
         x_on_r_perp = my_norm(dim, x_cross_r) / norm_of_r
         if (x_on_r_perp .le. radius) then
            in_cylinder = .true.
            return
         end if
      end if

      return

      end
C
C Auxiliary function determining if a point is within a cone
C
      logical function in_cone(dim, x_0, end_1, end_2, radius)
      implicit none

      integer dim
      double precision x_0(dim), end_1(dim), end_2(dim), radius

      double precision r(3), x(3), x_cross_r(3), norm_of_r, x_on_r,
     & x_on_r_perp, cone_radius
      integer i
C
C external functions--defined above
C
      double precision my_norm
      double precision my_dot_product
C
C Assume the answer is .false. unless the in_cone condition
C below checks.
C
      in_cone = .false.

      if (dim .ne. 3) then
         write(*,*) 'ERROR(in_cone): bad dimension'
         return
      end if

      do i = 1, 3
         r(i) = end_2(i) - end_1(i)
         x(i) = x_0(i) - end_1(i)
      end do
C
C x_on_r computed here is the projection of x on r, or
C x cos alpha.
C
      norm_of_r = my_norm(dim, r)
      x_on_r = my_dot_product(dim, x, r) / norm_of_r

      if ((0 .le. x_on_r) .and. (x_on_r .le. norm_of_r)) then
         call my_cross_product (dim, x, r, x_cross_r)
C
C x_on_r_perp computed here is the projection of x on the
C direction perpendicular to r, or x sin alpha.
C
         x_on_r_perp = my_norm(dim, x_cross_r) / norm_of_r
         cone_radius = radius * (1.0 - x_on_r/norm_of_r)
```

```
            if (x_on_r_perp .le. cone_radius) then
               in_cone = .true.
               return
            end if
         end if

         return

         end
C
C Auxiliary function determining if a point is within an arbitrary
C ellipsoid
C
         logical function in_ellipsoid(dim, r, r_0, e_1, c_1,
       & e_2, c_2, e_3, c_3)
         implicit none
C
C passed variables
C
         integer dim
         double precision r(dim), r_0(dim), e_1(dim), e_2(dim), e_3(dim)
         double precision c_1, c_2, c_3
C
C internal variables
C
         double precision dr(3), dr_1, dr_2, dr_3, sum
         double precision n_1(3), n_2(3), n_3(3)
         integer i
C
C external functions--defined above
C
         double precision my_dot_product, volume
C
C Assume the answer is .false. unless the in_ellipsoid condition
C below checks.
C
         in_ellipsoid = .false.

         if (dim .ne. 3) then
            write(*,*) 'ERROR(in_ellipsoid): bad dimension'
            return
         end if

         if (volume(dim, e_1, e_2, e_3) .eq. 0.0) then
            write(*,*) 'ERROR(in_ellipsoid): bad volume'
            return
         end if

         call normalize(dim, e_1, n_1)
         call normalize(dim, e_2, n_2)
         call normalize(dim, e_3, n_3)

         do i = 1, 3
            dr(i) = r(i) - r_0(i)
         end do

         dr_1 = my_dot_product(dim, dr, n_1)
         dr_2 = my_dot_product(dim, dr, n_2)
         dr_3 = my_dot_product(dim, dr, n_3)
```

```
      sum = dr_1 * dr_1 / (c_1 * c_1) + dr_2 * dr_2 / (c_2 * c_2)
     & + dr_3 * dr_3 / (c_3 * c_3)

      if (sum .le. 1.0) then
         in_ellipsoid = .true.
         return
      end if

      return
      end

C
C Auxiliary function determining if a point is within an arbitrarily
C oriented disk---based on in_ellipsoid.
C
      logical function in_disk(dim, r, r_0, r_n, radius, thickness)
      implicit none
C
C passed variables
C
      integer dim
      double precision r(dim), r_0(dim), r_n(dim), radius, thickness
C
C internal variables
C
      double precision r_z(dim), r_x(dim), r_y(dim)
C
C external functions (defined above)
C
      logical in_ellipsoid
C
      in_disk = .false.
C
      if (dim .ne. 3) then
         write(*,*) 'ERROR(in_disk): bad dimension'
         return
      end if
C
      call normalize (dim, r_n, r_z)
      call perpendicular_to (dim, r_z, r_x)
      call my_cross_product (dim, r_z, r_x, r_y)
C
      if (in_ellipsoid(dim, r, r_0,
     & r_z, thickness/2.0,
     & r_x, radius,
     & r_y, radius)) then
         in_disk = .true.
         return
      end if

      return
      end
C
C Auxiliary function to determine if a given point is
C in a parallelepiped.
C
      logical function in_parallelepiped(dim, r, r_0, e_1, e_2, e_3)
      implicit none
```

```
C
C passed variables
C
      integer dim
      double precision
     & r(dim), r_0(dim), e_1(dim), e_2(dim), e_3(dim)
C
C internal variables
C
      integer i
      double precision dr(3), n_1(3), n_2(3), n_3(3)
      double precision length_1, length_2, length_3
      double precision dr_1, dr_2, dr_3
C
C external functions
C
      double precision my_norm, my_dot_product, volume
C
      in_parallelepiped = .false.

      if (dim .ne. 3) then
         write(*,*) 'ERROR(in_parallelepiped): bad dimension'
         return
      end if

      if (volume(dim, e_1, e_2, e_3) .eq. 0.0) then
         write(*,*) 'ERROR(in_parallelepiped): bad volume'
         return
      end if

      length_1 = my_norm(dim, e_1)
      length_2 = my_norm(dim, e_2)
      length_3 = my_norm(dim, e_3)

      call normalize(dim, e_1, n_1)
      call normalize(dim, e_2, n_2)
      call normalize(dim, e_3, n_3)

      do i = 1, 3
         dr(i) = r(i) - r_0(i)
      end do

      dr_1 = my_dot_product(dim, dr, n_1)
      dr_2 = my_dot_product(dim, dr, n_2)
      dr_3 = my_dot_product(dim, dr, n_3)

      if ((0.0 .le. dr_1) .and. (dr_1 .le. length_1)
     & .and. (0.0 .le. dr_2) .and. (dr_2 .le. length_2)
     & .and. (0.0 .le. dr_3) .and. (dr_3 .le. length_3)) then
         in_parallelepiped = .true.
         return
      end if

      return
      end
C
C Auxiliary function to check if a point is in a ball.
C
      logical function in_ball(dim, point, c, r)
```

```
      implicit none

      integer dim
      double precision point(dim), c(dim), r

      integer i
      double precision dp(3), dp_length

      double precision my_norm

      in_ball = .false.

      if (dim .ne. 3) then
         write(*,*) 'ERROR(in_ball): bad dimension'
         return
      end if

      do i = 1, 3
         dp(i) = point(i) - c(i)
      end do

      dp_length = my_norm(dim, dp)

      if (dp_length .le. r) then
         in_ball = .true.
      end if

      return

      end

C
C Auxiliary function to check if a point is in a lens.
C
      logical function in_lens(dim, r, c_1, r_1, c_2, r_2)
      implicit none
C
C passed variables
C
      integer dim
      double precision r(dim), c_1(dim), c_2(dim), r_1, r_2
C
C internal variables
C
      integer i
      double precision dc(3), dc_length, sum, diff
C
C external functions
C
      logical in_ball
      double precision my_norm
C
      in_lens = .false.

      if (dim .ne. 3) then
         write(*,*) 'ERROR(in_lens): bad dimension'
         return
      end if
```

```
      do i = 1, 3
         dc(i) = c_2(i) - c_1(i)
      end do

      dc_length = my_norm(dim, dc)
      sum = r_1 + r_2
      diff = abs(r_1 - r_2)

      if ((diff .lt. dc_length)
     & .and. (dc_length .lt. sum)) then
         if (in_ball(dim, r, c_1, r_1)
     & .and. in_ball(dim, r, c_2, r_2)) then
            in_lens = .true.
            return
         end if
      else
         write(*,*) 'ERROR(in_lens): bad lens'
      end if

      return

      end
C
C Auxiliary function to check if a point resides within
C a torus.
C
      logical function in_torus(dim, point, center, normal, r_1, r_2)
      implicit none
C
C passed variables
C
      integer dim
      double precision point(dim), center(dim), normal(dim), r_1, r_2
C
C internal variables
C
      integer i
      double precision r_min, r_max, r_c, r_t, n_dot_xc
      double precision n(3), x_c(3), x_k(3), k(3), center_t(3)
      double precision x_t(3), x_t_length
C
C external functions
C
      double precision my_norm
      double precision my_dot_product
C
      if (dim .ne. 3) then
         write(*,*) 'ERROR(in_torus): bad dimension'
         in_torus = .false.
         return
      end if

      if ((point(1) .eq. center(1))
     & .and. (point(2) .eq. center(2))
     & .and. (point(3) .eq. center(3))) then
         in_torus = .false.
         return
      end if

```

```
      if ((normal(1) .eq. 0.0)
     & .and. (normal(2) .eq. 0.0)
     & .and. (normal(3) .eq. 0.0)) then
         write(*,*) 'ERROR(in_torus): bad normal'
         in_torus = .false.
         return
      end if

      if ((r_1 .le. 0.0)
     & .or. (r_2 .le. 0.0)
     & .or. (r_1 .eq. r_2)) then
         write(*,*) 'ERROR(in_torus): bad radius'
         in_torus = .false.
         return
      end if

      r_min = min(r_1, r_2)
      r_max = max(r_1, r_2)
      r_c = (r_min + r_max) / 2.0
      r_t = (r_max - r_min) / 2.0
      call normalize(dim, normal, n)

      do i = 1, 3
         x_c(i) = point(i) - center(i)
      end do

      n_dot_xc = my_dot_product (dim, n, x_c)

      do i = 1, 3
         x_k(i) = x_c(i) - n_dot_xc * n(i)
      end do
      call normalize(dim, x_k, k)

      do i = 1, 3
         center_t(i) = center(i) + r_c * k(i)
      end do

      do i = 1, 3
         x_t(i) = point(i) - center_t(i)
      end do
      x_t_length = my_norm(dim, x_t)

      if (x_t_length .le. r_t) then
         in_torus = .true.
      else
         in_torus = .false.
      end if

      return
      end
C
      subroutine out_standard_params (spacedim, margin, delta,
     & origin, grid_type, box_low, box_hi)
      implicit none
C
      integer spacedim, margin
      double precision delta, origin(0:spacedim-1)
      integer grid_type(0:spacedim-1)
      integer box_low(0:spacedim-1), box_hi(0:spacedim-1)
```

```
C
C action
C
      write(*,*) ' spacedim = ', spacedim
      write(*,*) ' margin = ', margin
      write(*,*) ' delta = ', delta
      write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
      write(*,*) ' type = ',
     & grid_type(0), ', ',
     & grid_type(1), ', ',
     & grid_type(2)
      write(*,*) ' box_low = ',
     & box_low(0), ', ',
     & box_low(1), ', ',
     & box_low(2)
      write(*,*) ' box_hi = ',
     & box_hi(0), ', ',
     & box_hi(1), ', ',
     & box_hi(2)
C
      return
      end
C
C
C
C Draw a box on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_box_(
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
C const Real*const delta,
C const int*const iv_lower_corner,
C const int*const iv_upper_corner,
C const Real*const lower_corner,
C const Real*const upper_corner,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
      subroutine f_draw_box(
     & spacedim, margin,
```

```
     & origin, delta,
     & iv_low_corner, iv_high_corner,
     & low_corner, high_corner, color,
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
     & medium,
     & verbosity, return_status)
      implicit none
C
      integer spacedim, margin
      double precision origin(0:spacedim - 1), delta
      integer
     & iv_low_corner(0:spacedim - 1),
     & iv_high_corner(0:spacedim - 1)
      double precision
     & low_corner(0:spacedim - 1),
     & high_corner(0:spacedim - 1)
      integer color
      integer
     & box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x, y, z
      integer count
C
C Action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_box_): bad dimension'
         return_status = 0
         goto 100
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_box_:'
         write(*,*) ' spacedim = ', spacedim
         write(*,*) ' margin = ', margin
         write(*,*) ' delta = ', delta
         write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(*,*) ' iv_low = ',
     & iv_low_corner(0), ', ',
     & iv_low_corner(1), ', ',
     & iv_low_corner(2)
         write(*,*) ' iv_hi = ',
     & iv_high_corner(0), ', ',
     & iv_high_corner(1), ', ',
     & iv_high_corner(2)
         write(*,*) ' low = ',
     & low_corner(0), ', ',
```

```
     & low_corner(1), ', ',
     & low_corner(2)
         write(*,*) ' hi = ',
     & high_corner(0), ', ',
     & high_corner(1), ', ',
     & high_corner(2)
         write(*,*) ' color = ', color
         write(*,*) ' type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      i_min = max(i_lo + margin, iv_low_corner(0))
      j_min = max(j_lo + margin, iv_low_corner(1))
      k_min = max(k_lo + margin, iv_low_corner(2))
C
      i_max = min(i_hi - margin, iv_high_corner(0))
      j_max = min(j_hi - margin, iv_high_corner(1))
      k_max = min(k_hi - margin, iv_high_corner(2))
C
      count = 0
C
      do k = k_min, k_max
         z = origin(2) + (k - 0.5 * box_type(2)) * delta
         if ((low_corner(2) .le. z) .and.
     & (z .le. high_corner(2))) then
            do j = j_min, j_max
               y = origin(1) + (j - 0.5 * box_type(1)) * delta
               if ((low_corner(1) .le. y) .and.
     & (y .le. high_corner(1))) then
                  do i = i_min, i_max
                     x = origin(0) + (i - 0.5 * box_type(0)) * delta
                     if ((low_corner(0) .le. x) .and.
     & (x .le. high_corner(0))) then
                        medium(i,j,k) = color
                        count = count + 1
                     end if
                  end do
               end if
            end do
         end if
      end do
C
      if (verbosity .gt. 0) then
         write(*,*) ' marked ', count, ' cells'
      end if
C
  100 continue
C
      end
C
C Draw a ball on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_ball_(
```

```
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
C const Real*const delta,
C const int*const iv_lower_corner,
C const int*const iv_upper_corner,
C const Real*const center,
C const Real*const radius,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
      subroutine f_draw_ball(
     & spacedim, margin,
     & origin, delta,
     & iv_low_corner, iv_high_corner,
     & center, radius, color,
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
     & medium,
     & verbosity, return_status)
      implicit none
C
      integer spacedim, margin
      double precision origin(0:spacedim - 1), delta
      integer
     & iv_low_corner(0:spacedim - 1),
     & iv_high_corner(0:spacedim - 1)
      double precision
     & center(0:spacedim - 1)
      double precision radius
      integer color
      integer
     & box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x, y, z
      double precision dx2, dy2, dz2, d2, r2
      integer count
C
C Action
C
```

```
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_ball_): bad dimension'
         return_status = 0
         goto 100
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_ball_:'
         write(*,*) ' spacedim = ', spacedim
         write(*,*) ' margin = ', margin
         write(*,*) ' delta = ', delta
         write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(*,*) ' iv_low = ',
     & iv_low_corner(0), ', ',
     & iv_low_corner(1), ', ',
     & iv_low_corner(2)
         write(*,*) ' iv_hi = ',
     & iv_high_corner(0), ', ',
     & iv_high_corner(1), ', ',
     & iv_high_corner(2)
         write(*,*) ' center = ',
     & center(0), ', ',
     & center(1), ', ',
     & center(2)
         write(*,*) ' radius = ', radius
         write(*,*) ' color = ', color
         write(*,*) ' type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      i_min = max(i_lo + margin, iv_low_corner(0))
      j_min = max(j_lo + margin, iv_low_corner(1))
      k_min = max(k_lo + margin, iv_low_corner(2))
C
      i_max = min(i_hi - margin, iv_high_corner(0))
      j_max = min(j_hi - margin, iv_high_corner(1))
      k_max = min(k_hi - margin, iv_high_corner(2))
C
      count = 0
      r2 = radius**2
C
      do k = k_min, k_max
         z = origin(2) + (k - 0.5 * box_type(2)) * delta
         dz2 = (center(2) - z)**2
         do j = j_min, j_max
            y = origin(1) + (j - 0.5 * box_type(1)) * delta
            dy2 = (center(1) - y)**2
            do i = i_min, i_max
               x = origin(0) + (i - 0.5 * box_type(0)) * delta
               dx2 = (center(0) - x)**2
               d2 = (dx2 + dy2 + dz2)
               if (d2 .le. r2) then
                  medium(i,j,k) = color
                  count = count + 1
```

```
                end if
              end do
            end do
          end do
C
          if (verbosity .gt. 0) then
            write(*,*) ' marked ', count, ' cells'
          end if
C
  100 continue
C
          end
C
C Draw an ellipsoid on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_ellipsoid_(
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
C const Real*const delta,
C const int*const iv_lower_corner,
C const int*const iv_upper_corner,
C const Real*const focus_1,
C const Real*const focus_2,
C const Real*const sum,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
          subroutine f_draw_ellipsoid(
         & spacedim, margin,
         & origin, delta,
         & iv_low_corner, iv_high_corner,
         & focus_1, focus_2, sum, color,
         & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
         & medium,
         & verbosity, return_status)
          implicit none
C
          integer spacedim, margin
          double precision origin(0:spacedim - 1), delta
          integer
         & iv_low_corner(0:spacedim - 1),
         & iv_high_corner(0:spacedim - 1)
          double precision
```

```
     & focus_1(0:spacedim − 1),
     & focus_2(0:spacedim − 1)
      double precision sum
      integer color
      integer
     & box_type(0:spacedim − 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x, y, z
      double precision dx1sqr, dy1sqr, dz1sqr
      double precision dx2sqr, dy2sqr, dz2sqr
      double precision dd
      double precision interfocal, eccentricity
      integer count
C
C Action
C
      if (spacedim .ne. 3) then
         write(∗,∗) 'ERROR(f_draw_ellipsoid_): bad dimension'
         return_status = 0
         goto 100
      end if
C
      interfocal =
     & sqrt((focus_2(0) − focus_1(0))∗∗2
     & + (focus_2(1) − focus_1(1))∗∗2
     & + (focus_2(2) − focus_1(2))∗∗2)
      eccentricity = interfocal/sum
C
      if (verbosity .gt. 0) then
         write(∗,∗) ' f_draw_ellipsoid_:'
         write(∗,∗) ' spacedim = ', spacedim
         write(∗,∗) ' margin = ', margin
         write(∗,∗) ' delta = ', delta
         write(∗,∗) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(∗,∗) ' iv_low = ',
     & iv_low_corner(0), ', ',
     & iv_low_corner(1), ', ',
     & iv_low_corner(2)
         write(∗,∗) ' iv_hi = ',
     & iv_high_corner(0), ', ',
     & iv_high_corner(1), ', ',
     & iv_high_corner(2)
         write(∗,∗) ' focus_1 = ',
     & focus_1(0), ', ',
     & focus_1(1), ', ',
     & focus_1(2)
         write(∗,∗) ' focus_2 = ',
```

```
     & focus_2(0), ', ',
     & focus_2(1), ', ',
     & focus_2(2)
        write(*,*) ' sum = ', sum
        write(*,*) ' interfcl = ', interfocal
        write(*,*) ' eccentr = ', eccentricity
        write(*,*) ' color = ', color
        write(*,*) ' type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      if (eccentricity .ge. 1) then
        write(*,*) 'ERROR(f_draw_ellipsoid_): ',
     & 'bad eccentricity'
        return_status = 0
        goto 100
      end if
C
      i_min = max(i_lo + margin, iv_low_corner(0))
      j_min = max(j_lo + margin, iv_low_corner(1))
      k_min = max(k_lo + margin, iv_low_corner(2))
C
      i_max = min(i_hi - margin, iv_high_corner(0))
      j_max = min(j_hi - margin, iv_high_corner(1))
      k_max = min(k_hi - margin, iv_high_corner(2))
C
      count = 0
C
      do k = k_min, k_max
        z = origin(2) + (k - 0.5 * box_type(2)) * delta
        dz1sqr = (focus_1(2) - z)**2
        dz2sqr = (focus_2(2) - z)**2
        do j = j_min, j_max
          y = origin(1) + (j - 0.5 * box_type(1)) * delta
          dy1sqr = (focus_1(1) - y)**2
          dy2sqr = (focus_2(1) - y)**2
          do i = i_min, i_max
            x = origin(0) + (i - 0.5 * box_type(0)) * delta
            dx1sqr = (focus_1(0) - x)**2
            dx2sqr = (focus_2(0) - x)**2
            dd = sqrt(dz1sqr + dy1sqr + dx1sqr)
     & + sqrt(dz2sqr + dy2sqr + dx2sqr)
            if (dd .le. sum) then
              medium(i,j,k) = color
              count = count + 1
            end if
          end do
        end do
      end do
C
      if (verbosity .gt. 0) then
        write(*,*) ' marked ', count, ' cells'
      end if
C
  100 continue
C
      end
```

```
C
C Draw a cylinder on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_cylinder_(
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
C const Real*const delta,
C const int*const cylinder_box_lo,
C const int*const cylinder_box_hi,
C const Real*const end_1,
C const Real*const end_2,
C const Real*const radius,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
      subroutine f_draw_cylinder(
     & spacedim, ghost_margin,
     & origin, delta,
     & cylinder_box_lo, cylinder_box_hi,
     & end_1, end_2, radius, color,
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
     & medium,
     & verbosity, return_status)
      implicit none
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim - 1), delta
      integer
     & cylinder_box_lo(0:spacedim - 1),
     & cylinder_box_hi(0:spacedim - 1)
      double precision
     & end_1(0:spacedim - 1),
     & end_2(0:spacedim - 1)
      double precision radius
      integer color
      integer
     & box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
```

```
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x(3)
      integer count
C
C external function--defined above
C
      logical in_cylinder
C
C Action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_cylinder_): bad dimension'
         return_status = 0
         goto 100
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_cylinder_:'
         write(*,*) ' spacedim = ', spacedim
         write(*,*) ' margin = ', ghost_margin
         write(*,*) ' delta = ', delta
         write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(*,*) ' c_box_lo = ',
     & cylinder_box_lo(0), ', ',
     & cylinder_box_lo(1), ', ',
     & cylinder_box_lo(2)
         write(*,*) ' c_box_hi = ',
     & cylinder_box_hi(0), ', ',
     & cylinder_box_hi(1), ', ',
     & cylinder_box_hi(2)
         write(*,*) ' end_1 = ',
     & end_1(0), ', ',
     & end_1(1), ', ',
     & end_1(2)
         write(*,*) ' end_2 = ',
     & end_2(0), ', ',
     & end_2(1), ', ',
     & end_2(2)
         write(*,*) ' radius = ', radius
         write(*,*) ' color = ', color
         write(*,*) ' box_type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      i_min = max(i_lo + ghost_margin, cylinder_box_lo(0))
      j_min = max(j_lo + ghost_margin, cylinder_box_lo(1))
      k_min = max(k_lo + ghost_margin, cylinder_box_lo(2))
C
      i_max = min(i_hi - ghost_margin, cylinder_box_hi(0))
      j_max = min(j_hi - ghost_margin, cylinder_box_hi(1))
```

```
      k_max = min(k_hi − ghost_margin, cylinder_box_hi(2))
C
      count = 0
C
      do k = k_min, k_max
         x(3) = origin(2) + (k − 0.5 ∗ box_type(2)) ∗ delta
         do j = j_min, j_max
            x(2) = origin(1) + (j − 0.5 ∗ box_type(1)) ∗ delta
            do i = i_min, i_max
               x(1) = origin(0) + (i − 0.5 ∗ box_type(0)) ∗ delta
               if (in_cylinder(spacedim, x, end_1, end_2, radius)) then
                  medium(i,j,k) = color
                  count = count + 1
               end if
            end do
         end do
      end do
C
      if (verbosity .gt. 0) then
         write(∗,∗) ' marked ', count, ' cells'
      end if
C
 100 continue
C
      end

C
C Draw a cone on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_cone_(
C const int∗const SpaceDim,
C const int∗const margin,
C const Real∗const origin,
C const Real∗const delta,
C const int∗const cone_box_lo,
C const int∗const cone_box_hi,
C const Real∗const end_1,
C const Real∗const end_2,
C const Real∗const radius,
C const int∗const color,
C const int∗const box_type,
C const int∗const i_lo,
C const int∗const j_lo,
C const int∗const k_lo,
C const int∗const i_hi,
C const int∗const j_hi,
C const int∗const k_hi,
C const int∗const medium,
C const int∗const verbosity,
C const int∗const return_status
C );
C }
C
      subroutine f_draw_cone(
     & spacedim, ghost_margin,
```

```
     & origin, delta,
     & cone_box_lo, cone_box_hi,
     & end_1, end_2, radius, color,
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
     & medium,
     & verbosity, return_status)
      implicit none
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim - 1), delta
      integer
     & cone_box_lo(0:spacedim - 1),
     & cone_box_hi(0:spacedim - 1)
      double precision
     & end_1(0:spacedim - 1),
     & end_2(0:spacedim - 1)
      double precision radius
      integer color
      integer
     & box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x(3)
      integer count
C
C external function--defined above
C
      logical in_cone
C
C Action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_cone_): bad dimension'
         return_status = 0
         goto 100
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_cone_:'
         write(*,*) ' spacedim = ', spacedim
         write(*,*) ' margin = ', ghost_margin
         write(*,*) ' delta = ', delta
         write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(*,*) ' c_box_lo = ',
     & cone_box_lo(0), ', ',
     & cone_box_lo(1), ', ',
     & cone_box_lo(2)
         write(*,*) ' c_box_hi = ',
```

```
     & cone_box_hi(0), ', ',
     & cone_box_hi(1), ', ',
     & cone_box_hi(2)
         write(*,*) ' end_1 = ',
     & end_1(0), ', ',
     & end_1(1), ', ',
     & end_1(2)
         write(*,*) ' end_2 = ',
     & end_2(0), ', ',
     & end_2(1), ', ',
     & end_2(2)
         write(*,*) ' radius = ', radius
         write(*,*) ' color = ', color
         write(*,*) ' box_type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      i_min = max(i_lo + ghost_margin, cone_box_lo(0))
      j_min = max(j_lo + ghost_margin, cone_box_lo(1))
      k_min = max(k_lo + ghost_margin, cone_box_lo(2))
C
      i_max = min(i_hi - ghost_margin, cone_box_hi(0))
      j_max = min(j_hi - ghost_margin, cone_box_hi(1))
      k_max = min(k_hi - ghost_margin, cone_box_hi(2))
C
      count = 0
C
      do k = k_min, k_max
         x(3) = origin(2) + (k - 0.5 * box_type(2)) * delta
         do j = j_min, j_max
            x(2) = origin(1) + (j - 0.5 * box_type(1)) * delta
            do i = i_min, i_max
               x(1) = origin(0) + (i - 0.5 * box_type(0)) * delta
               if (in_cone(spacedim, x, end_1, end_2, radius)) then
                  medium(i,j,k) = color
                  count = count + 1
               end if
            end do
         end do
      end do
C
      if (verbosity .gt. 0) then
         write(*,*) ' marked ', count, ' cells'
      end if
C
  100 continue
C
      end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
      subroutine f_draw_disk(
C
C grid specs
C
     & spacedim, ghost_margin, origin, delta,
C
```

```
C disk specs
C
     & disk_box_lo, disk_box_hi,
     & center, normal, radius, thickness, color,
C
C canvas specs
C
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi, medium,
C
C subroutine specs
C
     & verbosity, return_status)
      implicit none
C
C passed variables
C
C grid specs
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim - 1), delta
C
C canvas specs
C
      integer box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
C
C subroutine specs
C
      integer verbosity, return_status
C
C disk specs
C
      integer
     & disk_box_lo(0:spacedim - 1),
     & disk_box_hi(0:spacedim - 1)
      double precision
     & center(0:spacedim - 1),
     & normal(0:spacedim - 1)
      double precision radius, thickness
      integer color
C
C internal variables and external functions
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer count, i, j, k
      double precision x(3)
      logical in_disk
C
C action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_disk_): bad dimension'
         return_status = 0
         return
      end if
C
      if (verbosity .gt. 0) then
```

```
            write(*,*) ' f_draw_disk_:'
            call out_standard_params (spacedim, ghost_margin, delta,
     & origin, box_type, disk_box_lo, disk_box_hi)
            write(*,*) ' center = ',
     & center(0), ', ', center(1), ', ', center(2)
            write(*,*) ' normal = ',
     & normal(0), ', ', normal(1), ', ', normal(2)
            write(*,*) ' radius = ', radius
            write(*,*) ' thicknss = ', thickness
            write(*,*) ' color = ', color
         end if
C
         i_min = max(i_lo + ghost_margin, disk_box_lo(0))
         j_min = max(j_lo + ghost_margin, disk_box_lo(1))
         k_min = max(k_lo + ghost_margin, disk_box_lo(2))
C
         i_max = min(i_hi - ghost_margin, disk_box_hi(0))
         j_max = min(j_hi - ghost_margin, disk_box_hi(1))
         k_max = min(k_hi - ghost_margin, disk_box_hi(2))
C
         count = 0
C
         do k = k_min, k_max
            x(3) = origin(2) + (k - 0.5 * box_type(2)) * delta
            do j = j_min, j_max
               x(2) = origin(1) + (j - 0.5 * box_type(1)) * delta
               do i = i_min, i_max
                  x(1) = origin(0) + (i - 0.5 * box_type(0)) * delta
                  if (in_disk
     & (spacedim, x, center, normal, radius, thickness))
     & then
                     medium(i,j,k) = color
                     count = count + 1
                  end if
               end do
            end do
         end do
C
         if (verbosity .gt. 0) then
            write(*,*) ' marked ', count, ' cells'
         end if
C
         return
         end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
         subroutine f_draw_ary_ellipsoid(
C
C grid specs
C
     & spacedim, ghost_margin, origin, delta,
C
C ellipsoid specs
C
     & ellipsoid_box_lo, ellipsoid_box_hi,
     & center, e_1, r_1, e_2, r_2, e_3, r_3, color,
C
C canvas specs
```

```
C
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi, medium,
C
C subroutine specs
C
     & verbosity, return_status)
      implicit none
C
C passed variables
C
C grid specs
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim - 1), delta
C
C canvas specs
C
      integer box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
C
C subroutine specs
C
      integer verbosity, return_status
C
C ellipsoid specs
C
      integer
     & ellipsoid_box_lo(0:spacedim - 1),
     & ellipsoid_box_hi(0:spacedim - 1)
      double precision
     & center(0:spacedim - 1),
     & e_1(0:spacedim - 1),
     & e_2(0:spacedim - 1),
     & e_3(0:spacedim - 1)
      double precision r_1, r_2, r_3
      integer color
C
C internal variables and external functions
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer count, i, j, k
      double precision x(3)
      logical in_ellipsoid
C
C action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_ary_ellipsoid_): bad dimension'
         return_status = 0
         return
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_ary_ellipsoid_:'
         call out_standard_params (spacedim, ghost_margin, delta,
     & origin, box_type, ellipsoid_box_lo, ellipsoid_box_hi)
         write(*,*) ' center = ',
```

```
     & center(0), ', ', center(1), ', ', center(2)
         write(*,*) ' e_1 = ',
     & e_1(0), ', ', e_1(1), ', ', e_1(2)
         write(*,*) ' e_2 = ',
     & e_2(0), ', ', e_2(1), ', ', e_2(2)
         write(*,*) ' e_3 = ',
     & e_3(0), ', ', e_3(1), ', ', e_3(2)
         write(*,*) ' r_1 = ', r_1
         write(*,*) ' r_2 = ', r_2
         write(*,*) ' r_3 = ', r_3
         write(*,*) ' color = ', color
      end if
C
      i_min = max(i_lo + ghost_margin, ellipsoid_box_lo(0))
      j_min = max(j_lo + ghost_margin, ellipsoid_box_lo(1))
      k_min = max(k_lo + ghost_margin, ellipsoid_box_lo(2))
C
      i_max = min(i_hi - ghost_margin, ellipsoid_box_hi(0))
      j_max = min(j_hi - ghost_margin, ellipsoid_box_hi(1))
      k_max = min(k_hi - ghost_margin, ellipsoid_box_hi(2))
C
      count = 0
C
      do k = k_min, k_max
         x(3) = origin(2) + (k - 0.5 * box_type(2)) * delta
         do j = j_min, j_max
            x(2) = origin(1) + (j - 0.5 * box_type(1)) * delta
            do i = i_min, i_max
               x(1) = origin(0) + (i - 0.5 * box_type(0)) * delta
               if (in_ellipsoid
     & (spacedim, x, center, e_1, r_1, e_2, r_2, e_3, r_3))
     & then
                  medium(i,j,k) = color
                  count = count + 1
               end if
            end do
         end do
      end do
C
      if (verbosity .gt. 0) then
         write(*,*) ' marked ', count, ' cells'
      end if
C
      return
      end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
      subroutine f_draw_parallelepiped(
C
C grid specs
C
     & spacedim, ghost_margin, origin, delta,
C
C ellipsoid specs
C
     & parallelepiped_box_lo, parallelepiped_box_hi,
     & corner, e_1, e_2, e_3, color,
C
```

```
C canvas specs
C
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi, medium,
C
C subroutine specs
C
     & verbosity, return_status)
      implicit none
C
C passed variables
C
C grid specs
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim - 1), delta
C
C canvas specs
C
      integer box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
C
C subroutine specs
C
      integer verbosity, return_status
C
C parallelepiped specs
C
      integer
     & parallelepiped_box_lo(0:spacedim - 1),
     & parallelepiped_box_hi(0:spacedim - 1)
      double precision
     & corner(0:spacedim - 1),
     & e_1(0:spacedim - 1),
     & e_2(0:spacedim - 1),
     & e_3(0:spacedim - 1)
      integer color
C
C internal variables and external functions
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer count, i, j, k
      double precision x(3)
      logical in_parallelepiped
C
C action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_parallelepiped_): bad dimension'
         return_status = 0
         return
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_parallelepiped_:'
         call out_standard_params (spacedim, ghost_margin, delta,
     & origin, box_type, parallelepiped_box_lo,
     & parallelepiped_box_hi)
```

```
          write(*,*) ' corner = ',
     & corner(0), ', ', corner(1), ', ', corner(2)
          write(*,*) ' e_1 = ',
     & e_1(0), ', ', e_1(1), ', ', e_1(2)
          write(*,*) ' e_2 = ',
     & e_2(0), ', ', e_2(1), ', ', e_2(2)
          write(*,*) ' e_3 = ',
     & e_3(0), ', ', e_3(1), ', ', e_3(2)
          write(*,*) ' color = ', color
       end if
C
       i_min = max(i_lo + ghost_margin, parallelepiped_box_lo(0))
       j_min = max(j_lo + ghost_margin, parallelepiped_box_lo(1))
       k_min = max(k_lo + ghost_margin, parallelepiped_box_lo(2))
C
       i_max = min(i_hi - ghost_margin, parallelepiped_box_hi(0))
       j_max = min(j_hi - ghost_margin, parallelepiped_box_hi(1))
       k_max = min(k_hi - ghost_margin, parallelepiped_box_hi(2))
C
       count = 0
C
       do k = k_min, k_max
          x(3) = origin(2) + (k - 0.5 * box_type(2)) * delta
          do j = j_min, j_max
             x(2) = origin(1) + (j - 0.5 * box_type(1)) * delta
             do i = i_min, i_max
                x(1) = origin(0) + (i - 0.5 * box_type(0)) * delta
                if (in_parallelepiped
     & (spacedim, x, corner, e_1, e_2, e_3))
     & then
                   medium(i,j,k) = color
                   count = count + 1
                end if
             end do
          end do
       end do
C
       if (verbosity .gt. 0) then
          write(*,*) ' marked ', count, ' cells'
       end if
C
       return
       end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
       subroutine f_draw_torus(
C
C grid specs
C
     & spacedim, ghost_margin, origin, delta,
C
C ellipsoid specs
C
     & torus_box_lo, torus_box_hi,
     & center, normal, r_1, r_2, color,
C
C canvas specs
C
```

```
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi, medium,
C
C subroutine specs
C
     & verbosity, return_status)
      implicit none
C
C passed variables
C
C grid specs
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim − 1), delta
C
C canvas specs
C
      integer box_type(0:spacedim − 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
C
C subroutine specs
C
      integer verbosity, return_status
C
C torus specs
C
      integer
     & torus_box_lo(0:spacedim − 1),
     & torus_box_hi(0:spacedim − 1)
      double precision
     & center(0:spacedim − 1),
     & normal(0:spacedim − 1)
      double precision r_1, r_2
      integer color
C
C internal variables and external functions
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer count, i, j, k
      double precision x(3)
      logical in_torus
C
C action
C
      if (spacedim .ne. 3) then
         write(∗,∗) 'ERROR(f_draw_torus_): bad dimension'
         return_status = 0
         return
      end if
C
      if (verbosity .gt. 0) then
         write(∗,∗) ' f_draw_torus_:'
         call out_standard_params (spacedim, ghost_margin, delta,
     & origin, box_type, torus_box_lo, torus_box_hi)
         write(∗,∗) ' center = ',
     & center(0), ', ', center(1), ', ', center(2)
         write(∗,∗) ' normal = ',
     & normal(0), ', ', normal(1), ', ', normal(2)
```

```
1770          write(*,*) ' r_1 = ', r_1
              write(*,*) ' r_2 = ', r_2
              write(*,*) ' color = ', color
           end if
C
           i_min = max(i_lo + ghost_margin, torus_box_lo(0))
           j_min = max(j_lo + ghost_margin, torus_box_lo(1))
           k_min = max(k_lo + ghost_margin, torus_box_lo(2))
C
           i_max = min(i_hi − ghost_margin, torus_box_hi(0))
1780       j_max = min(j_hi − ghost_margin, torus_box_hi(1))
           k_max = min(k_hi − ghost_margin, torus_box_hi(2))
C
           count = 0
C
           do k = k_min, k_max
              x(3) = origin(2) + (k − 0.5 ∗ box_type(2)) ∗ delta
              do j = j_min, j_max
                 x(2) = origin(1) + (j − 0.5 ∗ box_type(1)) ∗ delta
                 do i = i_min, i_max
1790                x(1) = origin(0) + (i − 0.5 ∗ box_type(0)) ∗ delta
                    if (in_torus
         & (spacedim, x, center, normal, r_1, r_2))
         & then
                       medium(i,j,k) = color
                       count = count + 1
                    end if
                 end do
              end do
           end do
1800 C
           if (verbosity .gt. 0) then
              write(*,*) ' marked ', count, ' cells'
           end if
C
           return
           end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
           subroutine f_draw_lens(
C
C grid specs
C
         & spacedim, ghost_margin, origin, delta,
C
C ellipsoid specs
C
         & lens_box_lo, lens_box_hi,
         & center_1, r_1, center_2, r_2, color,
1820 C
C canvas specs
C
         & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi, medium,
C
C subroutine specs
C
         & verbosity, return_status)
           implicit none
```

```
C
C passed variables
C
C grid specs
C
      integer spacedim, ghost_margin
      double precision origin(0:spacedim - 1), delta
C
C canvas specs
C
      integer box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
C
C subroutine specs
C
      integer verbosity, return_status
C
C lens specs
C
      integer
     & lens_box_lo(0:spacedim - 1),
     & lens_box_hi(0:spacedim - 1)
      double precision
     & center_1(0:spacedim - 1),
     & center_2(0:spacedim - 1)
      double precision r_1, r_2
      integer color
C
C internal variables and external functions
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer count, i, j, k
      double precision x(3)
      logical in_lens
C
C action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_lens_): bad dimension'
         return_status = 0
         return
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_lens_:'
         call out_standard_params (spacedim, ghost_margin, delta,
     & origin, box_type, lens_box_lo, lens_box_hi)
         write(*,*) ' center_1 = ',
     & center_1(0), ', ', center_1(1), ', ', center_1(2)
         write(*,*) ' center_2 = ',
     & center_2(0), ', ', center_2(1), ', ', center_2(2)
         write(*,*) ' r_1 = ', r_1
         write(*,*) ' r_2 = ', r_2
         write(*,*) ' color = ', color
      end if
C
      i_min = max(i_lo + ghost_margin, lens_box_lo(0))
```

```
      j_min = max(j_lo + ghost_margin, lens_box_lo(1))
      k_min = max(k_lo + ghost_margin, lens_box_lo(2))
C
      i_max = min(i_hi − ghost_margin, lens_box_hi(0))
      j_max = min(j_hi − ghost_margin, lens_box_hi(1))
      k_max = min(k_hi − ghost_margin, lens_box_hi(2))
C
      count = 0
C
      do k = k_min, k_max
         x(3) = origin(2) + (k − 0.5 ∗ box_type(2)) ∗ delta
         do j = j_min, j_max
            x(2) = origin(1) + (j − 0.5 ∗ box_type(1)) ∗ delta
            do i = i_min, i_max
               x(1) = origin(0) + (i − 0.5 ∗ box_type(0)) ∗ delta
               if (in_lens
     & (spacedim, x, center_1, r_1, center_2, r_2))
     & then
                  medium(i,j,k) = color
                  count = count + 1
               end if
            end do
         end do
      end do
C
      if (verbosity .gt. 0) then
         write(∗,∗) ' marked ', count, ' cells'
      end if
C
      return
      end
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
```

13.6 Subroutine `inject_d`

Inject the $\boldsymbol{D}$ field into the total field region, on a specified face.

```
C
C Fortran subroutine inject_d
C
C $Id: inject_d.f,v 1.5 2007/10/02 21:04:24 gustav Exp $
C
      subroutine inject_d(
     & dir, side, margin, new_component,
     & x_0, y_0, z_0, delta,
     & dt_by_dg, time_h,
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi,
     & i_D1_lo, j_D1_lo, k_D1_lo,
     & i_D1_hi, j_D1_hi, k_D1_hi, D1,
     & i_D2_lo, j_D2_lo, k_D2_lo,
     & i_D2_hi, j_D2_hi, k_D2_hi, D2,
     & watch_inject_d, return_status)
      implicit none
C
C passed variables
C
      integer dir, side, margin, new_component
      double precision x_0, y_0, z_0, delta
      double precision dt_by_dg, time_h
      integer
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi
      integer
     & i_D1_lo, j_D1_lo, k_D1_lo,
     & i_D1_hi, j_D1_hi, k_D1_hi
      double precision
     & D1(i_D1_lo:i_D1_hi, j_D1_lo:j_D1_hi, k_D1_lo:k_D1_hi, 0:1)
      integer
     & i_D2_lo, j_D2_lo, k_D2_lo,
     & i_D2_hi, j_D2_hi, k_D2_hi
      double precision
     & D2(i_D2_lo:i_D2_hi, j_D2_lo:j_D2_hi, k_D2_lo:k_D2_hi, 0:1)
      integer watch_inject_d, return_status
C
C local variables
C
      integer
     & i1_min, j1_min, k1_min,
     & i1_max, j1_max, k1_max,
     & i2_min, j2_min, k2_min,
     & i2_max, j2_max, k2_max
      integer i_min, i_max, j_min, j_max, k_min, k_max
      integer i, j, k, i_D, j_D, k_D, factor
      integer i_H, j_H, k_H
      double precision x_H, y_H, z_H
      integer lo, hi
      parameter (lo = 0, hi = 1)
      integer ERROR_IN_INJECT_D
      parameter (ERROR_IN_INJECT_D = 10)
      integer Hx_box_type(3), Hy_box_type(3), Hz_box_type(3)
      data (Hx_box_type(i), i = 1,3) /1, 0, 0/
      data (Hy_box_type(i), i = 1,3) /0, 1, 0/
      data (Hz_box_type(i), i = 1,3) /0, 0, 1/
```

```
C
C external function
C
      double precision Hx_inc, Hy_inc, Hz_inc
C
C action
C
C 1. Find the real index ranges for the two fields provided.
C
      i1_min = i_D1_lo + margin;
      j1_min = j_D1_lo + margin;
      k1_min = k_D1_lo + margin;
      i1_max = i_D1_hi - margin;
      j1_max = j_D1_hi - margin;
      k1_max = k_D1_hi - margin;

      i2_min = i_D2_lo + margin;
      j2_min = j_D2_lo + margin;
      k2_min = k_D2_lo + margin;
      i2_max = i_D2_hi - margin;
      j2_max = j_D2_hi - margin;
      k2_max = k_D2_hi - margin;
C
C 2. Switch the action depending on dir
C a. Select the fixed index and afix it
C b. Select the sign of the correction
C c. Select the loop ranges for this field
C d. Loop over all cells of the wall and tweak the field
C
      select case (dir)
C
C West--East
C
      case (0)

         select case (side)
         case (lo)
            i_D = i_sgnl_lo
            i_H = i_sgnl_lo
            factor = 1
         case (hi)
            i_D = i_sgnl_hi + 1
            i_H = i_sgnl_hi
            factor = -1
         case default
            write(*,*) "Bad_side:_", side
            return_status = ERROR_IN_INJECT_D
            goto 100
         end select

         j_min = max(j1_min, j_sgnl_lo)
         j_max = min(j1_max, j_sgnl_hi)
         k_min = max(k1_min, k_sgnl_lo + 1)
         k_max = min(k1_max, k_sgnl_hi)
         x_H = x_0 + (i_H - 0.5 * Hz_box_type(1)) * delta
         do k = k_min, k_max
            z_H = z_0 + (k - 0.5 * Hz_box_type(3)) * delta
            do j = j_min, j_max
               y_H = y_0 + (j - 0.5 * Hz_box_type(2)) * delta
```

```
                D1(i_D, j, k, new_component)
      & = D1(i_D, j, k, new_component)
      & + factor * dt_by_dg * Hz_inc(x_H, y_H, z_H, time_h)
            end do
         end do

         j_min = max(j2_min, j_sgnl_lo + 1)
         j_max = min(j2_max, j_sgnl_hi)
         k_min = max(k2_min, k_sgnl_lo)
         k_max = min(k2_max, k_sgnl_hi)
         x_H = x_0 + (i_H - 0.5 * Hy_box_type(1)) * delta
         do k = k_min, k_max
            z_H = z_0 + (k - 0.5 * Hy_box_type(3)) * delta
            do j = j_min, j_max
               y_H = y_0 + (j - 0.5 * Hy_box_type(2)) * delta
               D2(i_D, j, k, new_component)
      & = D2(i_D, j, k, new_component)
      & - factor * dt_by_dg * Hy_inc(x_H, y_H, z_H, time_h)
            end do
         end do
C
C Front--Back
C
      case (1)

         select case (side)
         case (lo)
            j_D = j_sgnl_lo
            j_H = j_sgnl_lo
            factor = -1
         case (hi)
            j_D = j_sgnl_hi + 1
            j_H = j_sgnl_hi
            factor = 1
         case default
            write(*,*) "Bad side: ", side
            return_status = ERROR_IN_INJECT_D
            goto 100
         end select

         i_min = max(i1_min, i_sgnl_lo)
         i_max = min(i1_max, i_sgnl_hi)
         k_min = max(k1_min, k_sgnl_lo + 1)
         k_max = min(k1_max, k_sgnl_hi)
         y_H = y_0 + (j_H - 0.5 * Hz_box_type(2)) * delta
         do k = k_min, k_max
            z_H = z_0 + (k - 0.5 * Hz_box_type(3)) * delta
            do i = i_min, i_max
               x_H = x_0 + (i - 0.5 * Hz_box_type(1)) * delta
               D1(i, j_D, k, new_component)
      & = D1(i, j_D, k, new_component)
      & + factor * dt_by_dg * Hz_inc(x_H, y_H, z_H, time_h)
            end do
         end do

         i_min = max(i2_min, i_sgnl_lo + 1)
         i_max = min(i2_max, i_sgnl_hi)
         k_min = max(k2_min, k_sgnl_lo)
         k_max = min(k2_max, k_sgnl_hi)
```

```
            y_H = y_0 + (j_H − 0.5 ∗ Hx_box_type(2)) ∗ delta
            do k = k_min, k_max
                z_H = z_0 + (k − 0.5 ∗ Hx_box_type(3)) ∗ delta
                do i = i_min, i_max
                    x_H = x_0 + (i − 0.5 ∗ Hx_box_type(1)) ∗ delta
                    D2(i, j_D, k, new_component)
        & = D2(i, j_D, k, new_component)
        & − factor ∗ dt_by_dg ∗ Hx_inc(x_H, y_H, z_H, time_h)
                end do
            end do
C
C Bottom−−Top
C
        case (2)

            select case (side)
            case (lo)
                k_D = k_sgnl_lo
                k_H = k_sgnl_lo
                factor = 1
            case (hi)
                k_D = k_sgnl_hi + 1
                k_H = k_sgnl_hi
                factor = −1
            case default
                write(∗,∗) "Bad_side:_", side
                return_status = ERROR_IN_INJECT_D
                goto 100
            end select

            i_min = max(i1_min, i_sgnl_lo)
            i_max = min(i1_max, i_sgnl_hi)
            j_min = max(j1_min, j_sgnl_lo + 1)
            j_max = min(j1_max, j_sgnl_hi)
            z_H = z_0 + (k_H − 0.5 ∗ Hy_box_type(3)) ∗ delta
            do j = j_min, j_max
                y_H = y_0 + (j − 0.5 ∗ Hy_box_type(2)) ∗ delta
                do i = i_min, i_max
                    x_H = x_0 + (i − 0.5 ∗ Hy_box_type(1)) ∗ delta
                    D1(i, j, k_D, new_component)
        & = D1(i, j, k_D, new_component)
        & + factor ∗ dt_by_dg ∗ Hy_inc(x_H, y_H, z_H, time_h)
                end do
            end do

            i_min = max(i2_min, i_sgnl_lo + 1)
            i_max = min(i2_max, i_sgnl_hi)
            j_min = max(j2_min, j_sgnl_lo)
            j_max = min(j2_max, j_sgnl_hi)
            z_H = z_0 + (k_H − 0.5 ∗ Hx_box_type(3)) ∗ delta
            do j = j_min, j_max
                y_H = y_0 + (j − 0.5 ∗ Hx_box_type(2)) ∗ delta
                do i = i_min, i_max
                    x_H = x_0 + (i − 0.5 ∗ Hx_box_type(1)) ∗ delta
                    D2(i, j, k_D, new_component)
        & = D2(i, j, k_D, new_component)
        & − factor ∗ dt_by_dg ∗ Hx_inc(x_H, y_H, z_H, time_h)
                end do
            end do
```

```
C
C Error: no such direction
C
      case default
239      write(*,*) "Bad direction: ", dir
         return_status = ERROR_IN_INJECT_D
         goto 100
      end select

  100 continue

      end
```

13.7 Subroutine `inject_b`

Inject the $\boldsymbol{B}$ field into the total field region, on a specified face.

```
C
C Fortran subroutine inject_b
C
C $Id: inject_b.f,v 1.5 2007/10/02 21:13:08 gustav Exp $
C
      subroutine inject_b(
     & dir, side, margin, new_component,
     & x_0, y_0, z_0, delta,
     & dt_by_dg, time_e,
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi,
     & i_B1_lo, j_B1_lo, k_B1_lo,
     & i_B1_hi, j_B1_hi, k_B1_hi, B1,
     & i_B2_lo, j_B2_lo, k_B2_lo,
     & i_B2_hi, j_B2_hi, k_B2_hi, B2,
     & watch_inject_b, return_status)
      implicit none
C
C passed variables
C
      integer dir, side, margin, new_component
      double precision x_0, y_0, z_0, delta
      double precision dt_by_dg, time_e
      integer
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi
      integer
     & i_B1_lo, j_B1_lo, k_B1_lo,
     & i_B1_hi, j_B1_hi, k_B1_hi
      double precision
     & B1(i_B1_lo:i_B1_hi, j_B1_lo:j_B1_hi, k_B1_lo:k_B1_hi, 0:1)
      integer
     & i_B2_lo, j_B2_lo, k_B2_lo,
     & i_B2_hi, j_B2_hi, k_B2_hi
      double precision
     & B2(i_B2_lo:i_B2_hi, j_B2_lo:j_B2_hi, k_B2_lo:k_B2_hi, 0:1)
      integer watch_inject_b, return_status
C
C local variables
C
      integer
     & i1_min, j1_min, k1_min,
     & i1_max, j1_max, k1_max,
     & i2_min, j2_min, k2_min,
     & i2_max, j2_max, k2_max
      integer i_min, i_max, j_min, j_max, k_min, k_max
      integer i, j, k, i_B, j_B, k_B, factor
      integer i_E, j_E, k_E
      double precision x_E, y_E, z_E
      integer lo, hi
      parameter (lo = 0, hi = 1)
      integer ERROR_IN_INJECT_B
      parameter (ERROR_IN_INJECT_B = 10)
      integer Ex_box_type(3), Ey_box_type(3), Ez_box_type(3)
      data (Ex_box_type(i), i = 1,3) /0, 1, 1/
      data (Ey_box_type(i), i = 1,3) /1, 0, 1/
      data (Ez_box_type(i), i = 1,3) /1, 1, 0/
```

```
C
C external function
C
      double precision Ex_inc, Ey_inc, Ez_inc
C
C action
C
C 1. Find the real index ranges for the two fields provided.
C
      i1_min = i_B1_lo + margin;
      j1_min = j_B1_lo + margin;
      k1_min = k_B1_lo + margin;
      i1_max = i_B1_hi − margin;
      j1_max = j_B1_hi − margin;
      k1_max = k_B1_hi − margin;

      i2_min = i_B2_lo + margin;
      j2_min = j_B2_lo + margin;
      k2_min = k_B2_lo + margin;
      i2_max = i_B2_hi − margin;
      j2_max = j_B2_hi − margin;
      k2_max = k_B2_hi − margin;
      select case (dir)
C
C West−−East
C
      case (0)

         select case (side)
         case (lo)
            i_B = i_sgnl_lo
            i_E = i_sgnl_lo
            factor = −1
         case (hi)
            i_B = i_sgnl_hi
            i_E = i_sgnl_hi + 1
            factor = 1
         case default
            write(∗,∗) "Bad_side:_", side
            return_status = ERROR_IN_INJECT_B
            goto 100
         end select

         j_min = max(j1_min, j_sgnl_lo + 1)
         j_max = min(j1_max, j_sgnl_hi)
         k_min = max(k1_min, k_sgnl_lo)
         k_max = min(k1_max, k_sgnl_hi)
         x_E = x_0 + (i_E − 0.5 ∗ Ez_box_type(1)) ∗ delta
         do k = k_min, k_max
            z_E = z_0 + (k − 0.5 ∗ Ez_box_type(3)) ∗ delta
            do j = j_min, j_max
               y_E = y_0 + (j − 0.5 ∗ Ez_box_type(2)) ∗ delta
               B1(i_B, j, k, new_component)
     & = B1(i_B, j, k, new_component)
     & + factor ∗ dt_by_dg ∗ Ez_inc(x_E, y_E, z_E, time_e)
            end do
         end do

         j_min = max(j2_min, j_sgnl_lo)
```

```
        j_max = min(j2_max, j_sgnl_hi)
        k_min = max(k2_min, k_sgnl_lo + 1)
        k_max = min(k2_max, k_sgnl_hi)
        x_E = x_0 + (i_E − 0.5 ∗ Ey_box_type(1)) ∗ delta
        do k = k_min, k_max
           z_E = z_0 + (k − 0.5 ∗ Ey_box_type(3)) ∗ delta
           do j = j_min, j_max
              y_E = y_0 + (j − 0.5 ∗ Ey_box_type(2)) ∗ delta
              B2(i_B, j, k, new_component)
     & = B2(i_B, j, k, new_component)
     & − factor ∗ dt_by_dg ∗ Ey_inc(x_E, y_E, z_E, time_e)
           end do
        end do
C
C Front−Back
C
      case (1)

        select case (side)
        case (lo)
           j_B = j_sgnl_lo
           j_E = j_sgnl_lo
           factor = 1
        case (hi)
           j_B = j_sgnl_hi
           j_E = j_sgnl_hi + 1
           factor = −1
        case default
           write(∗,∗) "Bad side: ", side
           return_status = ERROR_IN_INJECT_B
           goto 100
        end select

        i_min = max(i1_min, i_sgnl_lo + 1)
        i_max = min(i1_max, i_sgnl_hi)
        k_min = max(k1_min, k_sgnl_lo)
        k_max = min(k1_max, k_sgnl_hi)
        y_E = y_0 + (j_E − 0.5 ∗ Ez_box_type(2)) ∗ delta
        do k = k_min, k_max
           z_E = z_0 + (k − 0.5 ∗ Ez_box_type(3)) ∗ delta
           do i = i_min, i_max
              x_E = x_0 + (i − 0.5 ∗ Ez_box_type(1)) ∗ delta
              B1(i, j_B, k, new_component)
     & = B1(i, j_B, k, new_component)
     & + factor ∗ dt_by_dg ∗ Ez_inc(x_E, y_E, z_E, time_e)
           end do
        end do

        i_min = max(i2_min, i_sgnl_lo)
        i_max = min(i2_max, i_sgnl_hi)
        k_min = max(k2_min, k_sgnl_lo + 1)
        k_max = min(k2_max, k_sgnl_hi)
        y_E = y_0 + (j_E − 0.5 ∗ Ex_box_type(2)) ∗ delta
        do k = k_min, k_max
           z_E = z_0 + (k − 0.5 ∗ Ex_box_type(3)) ∗ delta
           do i = i_min, i_max
              x_E = x_0 + (i − 0.5 ∗ Ex_box_type(1)) ∗ delta
              B2(i, j_B, k, new_component)
     & = B2(i, j_B, k, new_component)
```

```
     & − factor ∗ dt_by_dg ∗ Ex_inc(x_E, y_E, z_E, time_e)
            end do
         end do
C
C Bottom−−Top
C
      case (2)

         select case (side)
         case (lo)
            k_B = k_sgnl_lo
            k_E = k_sgnl_lo
            factor = −1
         case (hi)
            k_B = k_sgnl_hi
            k_E = k_sgnl_hi + 1
            factor = 1
         case default
            write(∗,∗) "Bad␣side:␣", side
            return_status = ERROR_IN_INJECT_B
            goto 100
         end select

         i_min = max(i1_min, i_sgnl_lo + 1)
         i_max = min(i1_max, i_sgnl_hi)
         j_min = max(j1_min, j_sgnl_lo)
         j_max = min(j1_max, j_sgnl_hi)
         z_E = z_0 + (k_E − 0.5 ∗ Ey_box_type(3)) ∗ delta
         do j = j_min, j_max
            y_E = y_0 + (j − 0.5 ∗ Ey_box_type(2)) ∗ delta
            do i = i_min, i_max
               x_E = x_0 + (i − 0.5 ∗ Ey_box_type(1)) ∗ delta
               B1(i, j, k_B, new_component)
     & = B1(i, j, k_B, new_component)
     & + factor ∗ dt_by_dg ∗ Ey_inc(x_E, y_E, z_E, time_e)
            end do
         end do

         i_min = max(i2_min, i_sgnl_lo)
         i_max = min(i2_max, i_sgnl_hi)
         j_min = max(j2_min, j_sgnl_lo + 1)
         j_max = min(j2_max, j_sgnl_hi)
         z_E = z_0 + (k_E − 0.5 ∗ Ex_box_type(3)) ∗ delta
         do j = j_min, j_max
            y_E = y_0 + (j − 0.5 ∗ Ex_box_type(2)) ∗ delta
            do i = i_min, i_max
               x_E = x_0 + (i − 0.5 ∗ Ex_box_type(1)) ∗ delta
               B2(i, j, k_B, new_component)
     & = B2(i, j, k_B, new_component)
     & − factor ∗ dt_by_dg ∗ Ex_inc(x_E, y_E, z_E, time_e)
            end do
         end do
C
C Error: no such direction
C
      case default
         write(∗,∗) "Bad_direction:␣", dir
         return_status = ERROR_IN_INJECT_B
         goto 100
```

```
      end select
100 continue
      end
```

13.8 Output Subroutines

Five subroutines live on this file so far. More will be added.

```
C Fortran subroutines used in output operations.
C
C $Id: output.f,v 1.6 2009/08/20 18:35:38 gustav Exp $
C
      subroutine from_edge_to_center(
     & spacedim, dir, n_out, count, idx,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi, fld,
     & watch, status)
      implicit none
C
      integer spacedim, dir, n_out, count, idx, watch, status
C
      integer
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
      double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out - 1)
C
      integer
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi
      double precision fld(
     & i_fld_lo:i_fld_hi,
     & j_fld_lo:j_fld_hi,
     & k_fld_lo:k_fld_hi, 0:1)
C
      integer i, j, k
      integer ERROR_IN_EDGE
      parameter (ERROR_IN_EDGE = 83)
C
      if ((count .gt. n_out - 1) .or. (count .lt. 0)) then
         write(*,*) 'Bad output field count'
         status = ERROR_IN_EDGE
         goto 100
      end if
      if ((idx .lt. 0) .or. (idx .gt. 1)) then
         write(*,*) 'Bad field index'
         status = ERROR_IN_EDGE
         goto 100
      end if
      if ((i_out_lo .lt. i_fld_lo)
     & .or. (j_out_lo .lt. j_fld_lo)
     & .or. (k_out_lo .lt. k_fld_lo)) then
         write(*,*) 'Bad low index on field'
         status = ERROR_IN_EDGE
         goto 100
      end if
C
      select case (dir)
      case (0)
         if ((j_out_hi + 1 .gt. j_fld_hi)
     & .or. (k_out_hi + 1 .gt. k_fld_hi)) then
```

```
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_EDGE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,idx) + fld(i,j+1,k,idx)
     & + fld(i,j,k+1,idx) + fld(i,j+1,k+1,idx))
                end do
            end do
        end do
      case (1)
        if ((i_out_hi + 1 .gt. i_fld_hi)
     & .or. (k_out_hi + 1 .gt. k_fld_hi)) then
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_EDGE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,idx) + fld(i+1,j,k,idx)
     & + fld(i,j,k+1,idx) + fld(i+1,j,k+1,idx))
                end do
            end do
        end do
      case (2)
        if ((i_out_hi + 1 .gt. i_fld_hi)
     & .or. (j_out_hi + 1 .gt. j_fld_hi)) then
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_EDGE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,idx) + fld(i+1,j,k,idx)
     & + fld(i,j+1,k,idx) + fld(i+1,j+1,k,idx))
                end do
            end do
        end do
      end select

  100 continue

      end
C
C
C
      subroutine from_int_edge_to_center(
     & spacedim, dir, n_out, count,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi, fld,
```

```
     & watch, status)
      implicit none
C
      integer spacedim, dir, n_out, count, watch, status
C
      integer
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
      double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out − 1)
C
      integer
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi
      integer fld(
     & i_fld_lo:i_fld_hi,
     & j_fld_lo:j_fld_hi,
     & k_fld_lo:k_fld_hi)
C
      integer i, j, k
      integer ERROR_IN_EDGE
      parameter (ERROR_IN_EDGE = 83)
C
      if ((count .gt. n_out − 1) .or. (count .lt. 0)) then
         write(∗,∗) 'Bad output field count'
         status = ERROR_IN_EDGE
         goto 100
      end if
      if ((i_out_lo .lt. i_fld_lo)
     & .or. (j_out_lo .lt. j_fld_lo)
     & .or. (k_out_lo .lt. k_fld_lo)) then
         write(∗,∗) 'Bad low index on field'
         status = ERROR_IN_EDGE
         goto 100
      end if
C
      select case (dir)
      case (0)
         if ((j_out_hi + 1 .gt. j_fld_hi)
     & .or. (k_out_hi + 1 .gt. k_fld_hi)) then
            write(∗,∗) 'Bad high index on field'
            status = ERROR_IN_EDGE
            goto 100
         end if
         do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
               do i = i_out_lo, i_out_hi
                  out(i,j,k,count) =
     & 0.25 ∗ (fld(i,j,k) + fld(i,j+1,k)
     & + fld(i,j,k+1) + fld(i,j+1,k+1))
               end do
            end do
         end do
      case (1)
         if ((i_out_hi + 1 .gt. i_fld_hi)
     & .or. (k_out_hi + 1 .gt. k_fld_hi)) then
            write(∗,∗) 'Bad high index on field'
```

```
            status = ERROR_IN_EDGE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
     & 0.25 * (fld(i,j,k) + fld(i+1,j,k)
     & + fld(i,j,k+1) + fld(i+1,j,k+1))
                end do
            end do
        end do
     case (2)
        if ((i_out_hi + 1 .gt. i_fld_hi)
     & .or. (j_out_hi + 1 .gt. j_fld_hi)) then
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_EDGE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
     & 0.25 * (fld(i,j,k) + fld(i+1,j,k)
     & + fld(i,j+1,k) + fld(i+1,j+1,k))
                end do
            end do
        end do
     end select

  100 continue

     end
C
C
C
     subroutine from_face_to_center(
     & spacedim, dir, n_out, count, idx,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi, fld,
     & watch, status)
     implicit none
C
     integer spacedim, dir, n_out, count, idx, watch, status
C
     integer
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
     double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out - 1)
C
     integer
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi
     double precision fld(
```

```
     & i_fld_lo:i_fld_hi,
     & j_fld_lo:j_fld_hi,
     & k_fld_lo:k_fld_hi, 0:1)
C
      integer i, j, k
      integer ERROR_IN_FACE
      parameter (ERROR_IN_FACE = 84)
C
      if ((count .gt. n_out − 1) .or. (count .lt. 0)) then
         write(∗,∗) 'Bad output field count'
         status = ERROR_IN_FACE
         goto 100
      end if
      if ((idx .lt. 0) .or. (idx .gt. 1)) then
         write(∗,∗) 'Bad field index'
         status = ERROR_IN_FACE
         goto 100
      end if
      if ((i_out_lo .lt. i_fld_lo)
     & .or. (j_out_lo .lt. j_fld_lo)
     & .or. (k_out_lo .lt. k_fld_lo)) then
         write(∗,∗) 'Bad low index on field'
         status = ERROR_IN_FACE
         goto 100
      end if
C
C Face centered fields are magnetic and we interpolate
C them both in space and time. So we do not actually make
C any use in computations of the idx index. But we keep it
C here, so as not to change the calling interface.
C
      select case (dir)
      case (0)
         if (i_out_hi + 1 .gt. i_fld_hi) then
            write(∗,∗) 'Bad high index on field'
            status = ERROR_IN_FACE
            goto 100
         end if
         do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
               do i = i_out_lo, i_out_hi
                  out(i,j,k,count) =
     & 0.25 ∗ (fld(i,j,k,0) + fld(i+1,j,k,0)
     & + fld(i,j,k,1) + fld(i+1,j,k,1))
               end do
            end do
         end do
      case (1)
         if (j_out_hi + 1 .gt. j_fld_hi) then
            write(∗,∗) 'Bad high index on field'
            status = ERROR_IN_FACE
            goto 100
         end if
         do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
               do i = i_out_lo, i_out_hi
                  out(i,j,k,count) =
     & 0.25 ∗ (fld(i,j,k,0) + fld(i,j+1,k,0)
     & + fld(i,j,k,1) + fld(i,j+1,k,1))
```

```
            end do
         end do
      end do
   case (2)
      if (k_out_hi + 1 .gt. k_fld_hi) then
         write(*,*) 'Bad high index on field'
         status = ERROR_IN_FACE
         goto 100
      end if
      do k = k_out_lo, k_out_hi
         do j = j_out_lo, j_out_hi
            do i = i_out_lo, i_out_hi
               out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,0) + fld(i,j,k+1,0)
     & + fld(i,j,k,1) + fld(i,j,k+1,1))
            end do
         end do
      end do
   end select

100 continue

   end
C
C
C
   subroutine from_int_face_to_center(
     & spacedim, dir, n_out, count,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi, fld,
     & watch, status)
   implicit none
C
   integer spacedim, dir, n_out, count, watch, status
C
   integer
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
   double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out - 1)
C
   integer
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi
   integer fld(
     & i_fld_lo:i_fld_hi,
     & j_fld_lo:j_fld_hi,
     & k_fld_lo:k_fld_hi)
C
   integer i, j, k
   integer ERROR_IN_FACE
   parameter (ERROR_IN_FACE = 84)
C
   if ((count .gt. n_out - 1) .or. (count .lt. 0)) then
      write(*,*) 'Bad output field count'
```

```
        status = ERROR_IN_FACE
        goto 100
    end if
    if ((i_out_lo .lt. i_fld_lo)
   & .or. (j_out_lo .lt. j_fld_lo)
   & .or. (k_out_lo .lt. k_fld_lo)) then
        write(*,*) 'Bad low index on field'
        status = ERROR_IN_FACE
        goto 100
    end if
C
    select case (dir)
    case (0)
        if (i_out_hi + 1 .gt. i_fld_hi) then
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_FACE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
   & 0.5 * (fld(i,j,k) + fld(i+1,j,k))
                end do
            end do
        end do
    case (1)
        if (j_out_hi + 1 .gt. j_fld_hi) then
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_FACE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
   & 0.5 * (fld(i,j,k) + fld(i,j+1,k))
                end do
            end do
        end do
    case (2)
        if (k_out_hi + 1 .gt. k_fld_hi) then
            write(*,*) 'Bad high index on field'
            status = ERROR_IN_FACE
            goto 100
        end if
        do k = k_out_lo, k_out_hi
            do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                    out(i,j,k,count) =
   & 0.5 * (fld(i,j,k) + fld(i,j,k+1))
                end do
            end do
        end do
    end select

100 continue

    end
```

```
C
C Evaluate energy density
C
      subroutine evaluate_energy(
     & spacedim, n_out, count,
     & Ex, Ey, Ez, Bx, By, Bz,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & watch, status)
      implicit none
C
      integer spacedim, n_out, count
      integer Ex, Ey, Ez, Bx, By, Bz
      integer i_out_lo, j_out_lo, k_out_lo
      integer i_out_hi, j_out_hi, k_out_hi
      double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out - 1)
      integer watch, status
C
      integer i, j, k, ERROR_IN_ENERGY
      parameter (ERROR_IN_ENERGY = 85)
C
      if ((Ex .lt. 0) .or. (Ex .gt. n_out - 1)
     & .or. (Ey .lt. 0) .or. (Ey .gt. n_out - 1)
     & .or. (Ez .lt. 0) .or. (Ez .gt. n_out - 1)
     & .or. (Bx .lt. 0) .or. (Bx .gt. n_out - 1)
     & .or. (By .lt. 0) .or. (By .gt. n_out - 1)
     & .or. (Bz .lt. 0) .or. (Bz .gt. n_out - 1)) then
         write(*,*) 'Bad field index in evaluate_energy_'
         status = ERROR_IN_ENERGY
         goto 100
      end if
      do k = k_out_lo, k_out_hi
         do j = j_out_lo, j_out_hi
            do i = i_out_lo, i_out_hi
               out(i,j,k,count) =
     & 0.5 * (out(i,j,k,Ex)**2
     & + out(i,j,k,Ey)**2
     & + out(i,j,k,Ez)**2
     & + out(i,j,k,Bx)**2
     & + out(i,j,k,By)**2
     & + out(i,j,k,Bz)**2)
            end do
         end do
      end do

  100 continue

      end
C
C Evaluate energy flux
C
      subroutine evaluate_flow(
     & spacedim, n_out, count,
     & Ey, Bz, Ez, By,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
```

```
     & watch, status)
      implicit none
C
      integer spacedim, n_out, count
      integer Ey, Bz, Ez, By
      integer i_out_lo, j_out_lo, k_out_lo
      integer i_out_hi, j_out_hi, k_out_hi
      double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out - 1)
      integer watch, status
C
      integer i, j, k, ERROR_IN_FLOW
      parameter (ERROR_IN_FLOW = 86)
C
      if ((Ey .lt. 0) .or. (Ey .gt. n_out - 1)
     & .or. (Bz .lt. 0) .or. (Bz .gt. n_out - 1)
     & .or. (Ez .lt. 0) .or. (Ez .gt. n_out - 1)
     & .or. (By .lt. 0) .or. (By .gt. n_out - 1)) then
         write(*,*) 'Bad field index in evaluate_flow_'
         status = ERROR_IN_FLOW
         goto 100
      end if
      do k = k_out_lo, k_out_hi
         do j = j_out_lo, j_out_hi
            do i = i_out_lo, i_out_hi
               out(i,j,k,count) =
     & out(i,j,k,Ey) * out(i,j,k,Bz)
     & - out(i,j,k,Ez) * out(i,j,k,By)
            end do
         end do
      end do

  100 continue

      end
C
C
C
      subroutine output_field_copy(
     & spacedim, n_pcmp, idx_pcmp, n_outp, idx_outp,
     & i_out_lo, j_out_lo, k_out_lo, i_out_hi, j_out_hi, k_out_hi,
     & pcmp, outp,
     & watch, status)
      implicit none
C
      integer spacedim, n_pcmp, idx_pcmp, n_outp, idx_outp
      integer i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
      double precision
     & pcmp(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_pcmp - 1),
     & outp(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_outp - 1)
```

```
      integer watch, status
C
      integer i, j, k, ERROR_IN_COPY
      parameter (ERROR_IN_COPY = 87)
C
      if ((idx_pcmp .lt. 0) .or. (idx_pcmp .gt. n_pcmp − 1)) then
         write(*,*) 'Bad source index in output_field_copy_'
         status = ERROR_IN_COPY
         goto 100
      end if
C
      if ((idx_outp .lt. 0) .or. (idx_outp .gt. n_outp −1)) then
         write(*,*) 'Bad destination index in output_field_copy_'
         status = ERROR_IN_COPY
         goto 100
      end if
C
      do k = k_out_lo, k_out_hi
         do j = j_out_lo, j_out_hi
            do i = i_out_lo, i_out_hi
               outp(i, j, k, idx_outp) = pcmp(i, j, k, idx_pcmp)
            end do
         end do
      end do

  100 continue

      end
```

14 Makefile and srcMakefile

The `Makefile` is simplified now to make Chombo happy.

```
# Makefile for a 3D FDTD Chombo Program "Forms".
#
# $Id: Makefile,v 1.72 2009/12/23 15:38:40 gustav Exp $
#
# My Cygwin locations: on Crawley
#
# CHOMBO_HOME = /home/gustav/Chombo-3.0-Apr2009/lib3d.CYGWIN.g++.g77.OPT
# CHOMBO_HOME = /home/gustav/Chombo-3.0-Apr2009/lib3d.CYGWIN.g++.g77.DEBUG
# CHOMBO_HOME = /home/gustav/Chombo-3.0-Apr2009/lib3d.CYGWIN.mpicxx.g77.OPT.MPI
CHOMBO_HOME = /home/gustav/Chombo-3.0-Apr2009/lib3d.CYGWIN.mpicxx.g77.DEBUG.MPI
#
# My Linux locations
#
# Quarry (IU):
#
# CHOMBO_HOME = /N/u/gustav/Quarry/Chombo-3.0-Apr2009/lib3d.Linux.64.g++.gfortran.OPT
# CHOMBO_HOME = /N/u/gustav/Quarry/Chombo-3.0-Apr2009/lib3d.Linux.64.g++.gfortran.DEBUG
# CHOMBO_HOME = /N/u/gustav/Quarry/Chombo-3.0-Apr2009/lib3d.Linux.64.mpicxx.gfortran.OPT.MPI
# CHOMBO_HOME = /N/u/gustav/Quarry/Chombo-3.0-Apr2009/lib3d.Linux.64.mpicxx.gfortran.DEBUG.MPI
#
# Steele (Purdue):
#
# CHOMBO_HOME = /home/ba01/u108/gustav/Chombo-3.0-Apr2009/lib3d.Linux.64.g++.gfortran.DEBUG
# CHOMBO_HOME = /home/ba01/u108/gustav/Chombo-3.0-Apr2009/lib3d.Linux.64.g++.gfortran.OPT
# CHOMBO_HOME = /home/ba01/u108/gustav/Chombo-3.0-Apr2009/lib3d.Linux.64.mpicxx.gfortran.OPT.MPI
# CHOMBO_HOME = /home/ba01/u108/gustav/Chombo-3.0-Apr2009/lib3d.Linux.64.mpicxx.gfortran.DEBUG.MPI
#
# Abe (NCSA):
#
# CHOMBO_HOME = /u/ac/meglicki/Chombo-3.0-Apr2009/lib3d.Linux.64.icpc.ifort.OPT
# CHOMBO_HOME = /u/ac/meglicki/Chombo-3.0-Apr2009/lib3d.Linux.64.icpc.ifort.DEBUG
# CHOMBO_HOME = /u/ac/meglicki/Chombo-3.0-Apr2009/lib3d.Linux.64.mpicxx.ifort.OPT.MPI
# CHOMBO_HOME = /u/ac/meglicki/Chombo-3.0-Apr2009/lib3d.Linux.64.mpicxx.ifort.DEBUG.MPI
#
ebase = Forms
#
LibNames = AMRTools BoxTools
#
base_dir = ./
src_dirs = ./src
XTRALIBFLAGS = `guile-config link`
XTRACXXFLAGS = `guile-config compile`
INPUT = Input.scm

include $(CHOMBO_HOME)/mk/Make.example
```

I have offloaded all source management and documentation generation to `srcMakefile` listed below.

```
# Makefile for the Forms source.
#
# $Id: srcMakefile,v 1.94 2010/02/05 22:13:09 gustav Exp $
#
# Split parameters
#
TO = 191
FROM = 192
```

```
BIBLIO = 516−517
#
SRC_DEST = ./src
DOC_DEST = ./doc
CXX_SOURCES = Forms.cpp Initialize.cpp Iteration.cpp Scheme.cpp Auxiliary.cpp \
                SCMParmParse.cpp IO.cpp CurlBox.cpp
C_SOURCES = Injection.c Conversion.c
F_SOURCES = update_d.f update_b.f d_to_e.f b_to_h.f inject_d.f inject_b.f output.f draw.f
PY_SOURCES = script.py
SH_SOURCES = make_movie.sh
EXAMPLES = Examples/dielectric.scm Examples/distribution.scm Examples/signal.scm \
                Examples/waveform.gnplt Examples/dielectric.c Examples/include_graphic.tex \
                Examples/auxiliary−fields.cpp Examples/d−to−e−conversion.f \
                Examples/d−to−e−conversion.cpp Examples/d−to−e−conversion.c \
                Examples/create−s−array.cpp Examples/drude.c Examples/drude.scm \
                Examples/TestBall.scm Examples/inject.c Examples/TestBallC.scm \
                Examples/jazz_submit.sh Examples/Dust.scm Examples/domain_partition.cpp \
                Examples/Flux_1.scm Examples/FourierFlux.scm Examples/interpolate.scm \
                Examples/TestBallD.scm Examples/TestBallE.scm Examples/vinterpolate.scm \
                Examples/PlaneWave.scm Examples/Accuracy.txt Examples/PlaneWaveB.scm \
                Examples/Job−128.txt Examples/Job−256.txt Examples/WaveFlux.scm \
                Examples/FluxLib.scm \
                Examples/FluxOutput.txt Examples/PowerFlux.scm Examples/PowerFlux−out−128.txt \
                Examples/FourierFlux.scm Examples/signal.eps Examples/FourierOutput.txt \
                Examples/FourierOutput−256.txt Examples/signal_lib.scm Examples/media_lib.scm \
                Examples/drude−lib.scm Examples/WaveFlux−Lib.scm Examples/WaveFlux−halfLib.scm \
                Examples/PowerFlux−Lib.scm Examples/FourierFlux−Lib.scm \
                Examples/Snell−no−media.scm Examples/Snell.scm Examples/Mie−1.scm \
                Examples/mie−power−map.scm Examples/fdtd−power−map.scm \
                Examples/fdtd−fourier−map.scm Examples/Drude−Input.scm \
                Examples/Parallel−Drude−Input.scm Examples/merge.scm
LOCAL_HEADERS = Forms.H EdgeFab.H EdgeFabImplem.H FaceFab.H \
                FaceFabImplem.H CurlBox.H SCMParmParse.H Injection.h Conversion.h
SOURCES = $(CXX_SOURCES) $(C_SOURCES) $(F_SOURCES) $(LOCAL_HEADERS)
WEB_FILES = Forms.w Introduction.w Numerics.w Mainlines.w Initialize.w \
                Iteration.w Scheme.w IO.w Auxiliary.w Includes.w Extensions.w Fortran.w \
                Makefile.w Closing.w Bibliography.w
EPS_FILES = signal.eps exp−hahn.eps linear−gauss.eps linear−hann.eps \
                shrink−exp−hahn.eps shrink−linear−hahn.eps visit−example−1.eps \
                visit−example−2.eps Snell−2.eps power−map.eps mie−power−map.eps \
                Ex_0911_xz.eps Ex_0911_yz.eps mie_0000_xz.eps mie_0000_yz.eps \
                fourier−power−map.eps FDTD_Ag_0_89_Ex_zx_0907.eps FDTD_Ag_0_89_Ex_zy_0907.eps \
                Mie_Ag_0_89_Ex_zx_0059.eps Mie_Ag_0_89_Ex_zy_0059.eps
WEB_OUTPUTS = Forms.c Initialize.c Iteration.c Scheme.c Auxiliary.c IO.c \
                CurlBox.c SCMParmParse.c \
                Forms.hw Initialize.hw Iteration.hw Scheme.hw Auxiliary.hw IO.hw \
                CurlBox.hw SCMParmParse.hw \
                Conversion.c_std Injection.c_std \
                Conversion.h_std Injection.h_std \
                EdgeFab.hw EdgeFabImplem.hw FaceFab.hw FaceFabImplem.hw
TEX_OUTPUTS = Forms.aux Forms.idx Forms.log Forms.scn Forms.toc Forms.tex Forms.dvi

INPUT = Input.scm

ASTYLE = astyle −−style=ansi −−break−blocks −−pad=oper


all: raw doc src clean
```

```
big-all: raw big-doc src clean

src-only: raw src clean

raw: $(WEB_FILES) $(F_SOURCES)
        ctangle Forms.w

src: $(SOURCES)
        [ \! -d $(SRC_DEST) ] && mkdir $(SRC_DEST)
        mv $(SOURCES) $(SRC_DEST)
        mv ./src/Forms.cpp .
        mv ./src/Forms.H .

big-doc: $(EXAMPLES) $(F_SOURCES) $(EPS_FILES) $(PY_SOURCES) $(SH_SOURCES) Forms.pdf Forms_1.pdf Forms_2.pdf
        [ \! -d $(DOC_DEST) ] && mkdir $(DOC_DEST)
        mv Forms.pdf Forms_1.pdf Forms_2.pdf $(DOC_DEST)

doc: $(EXAMPLES) $(F_SOURCES) $(EPS_FILES) $(PY_SOURCES) $(SH_SOURCES) Forms.pdf
        [ \! -d $(DOC_DEST) ] && mkdir $(DOC_DEST)
        mv Forms.pdf $(DOC_DEST)


Forms.pdf: $(WEB_FILES) $(EXAMPLES) $(F_SOURCES) $(PY_SOURCES) $(SH_SOURCES) Makefile

Forms_1.pdf: $(WEB_FILES) $(EXAMPLES) $(F_SOURCES) $(PY_SOURCES) $(SH_SOURCES) Makefile
        dvipdfm -p letter -s -$(TO),$(BIBLIO) -o Forms_1.pdf Forms.dvi

Forms_2.pdf: $(WEB_FILES) $(EXAMPLES) $(F_SOURCES) $(PY_SOURCES) $(SH_SOURCES) Makefile
        dvipdfm -p letter -s $(FROM)- -o Forms_2.pdf Forms.dvi

%.tex: %.w
        cweave $<

%.dvi: %.tex
        latex $<
        latex $<
        latex $<

%.pdf: %.dvi
        dvipdfm -p letter $<

%.cpp: %.c
        sed -e '/^#line/d' \
            -e '/^\/\*/d' \
            -e '/^$$/d' $< > $@
        $(ASTYLE) $@
        rm $@.orig

%.H: %.hw
        sed -e '/^#line/d' \
            -e '/^\/\*/d' \
            -e '/^$$/d' $< > $@
        $(ASTYLE) $@
        rm $@.orig

%.h: %.h_std
        sed -e '/^#line/d' \
            -e '/^\/\*/d' \
            -e '/^$$/d' $< > $@
```

```
        $(ASTYLE) $@
        rm $@.orig

%.c: %.c_std
        sed -e '/^#line/d' \
            -e '/^\/\*/d' \
            -e '/^$$/d' $< > $@
        $(ASTYLE) $@
        rm $@.orig

clean:
        rcsclean
        rm -f $(WEB_OUTPUTS)
        rm -f $(TEX_OUTPUTS)
        rm -f *.orig
        ( cd Examples ; rcsclean )
        co Makefile

clobber: clean
        rcsclean
        rm -rf $(SRC_DEST)
        rm -rf $(DOC_DEST)
        rm -f Forms.pdf
        rm -f Forms.cpp Forms.H

#
# Special dependencies
#
SCMParmParse.hw: Extensions.w

SCMParmParse.c: Extensions.w

EdgeFab.hw: Extensions.w

EdgeFabImplem.hw: Extensions.w

FaceFab.hw: Extensions.w

FaceFabImplem.hw: Extensions.w

CurlBox.hw: Extensions.w

CurlBox.c: Extensions.w

Injection.c_std: Iteration.w

Injection.h_std: Iteration.w

Conversion.c_std: Iteration.w

Conversion.h_std: Iteration.w

Forms.hw: Includes.w

#
# $Log: srcMakefile,v $
# Revision 1.94 2010/02/05 22:13:09 gustav
# Changed the split.
#
```

```
# Revision 1.93 2010/02/05 21:02:58 gustav
# Added merge.scm to the Examples.
#
# Revision 1.92 2010/02/01 20:12:40 gustav
# Added parallel Drude input example.
#
# Revision 1.91 2009/11/02 19:18:17 gustav
# Changed the split point.
#
# Revision 1.90 2009/11/02 19:10:03 gustav
# Added Examples/Drude-Input.scm to examples.
#
# Revision 1.89 2009/11/02 16:42:10 gustav
# Added images for Mie scattering on a silver ball.
#
# Revision 1.88 2009/10/23 21:58:38 gustav
# Changed the split again.
#
# Revision 1.87 2009/10/23 20:46:33 gustav
# Changed the location of the split.
#
# Revision 1.86 2009/10/14 20:13:00 gustav
# Added a new split point.
#
# Revision 1.85 2009/10/10 23:19:22 gustav
# Changed the split.
#
# Revision 1.84 2009/10/09 21:34:15 gustav
# Added fourier-power-map.eps to eps files.
#
# Revision 1.83 2009/10/08 19:03:14 gustav
# Changed the split location.
#
# Revision 1.82 2009/10/07 22:27:17 gustav
# Changed Bibliography numbering.
#
# Revision 1.81 2009/10/07 22:12:23 gustav
# Changed the split position.
#
# Revision 1.80 2009/10/07 19:32:04 gustav
# Added power-map scripts and images.
# .;
#
# Revision 1.79 2009/10/06 22:18:25 gustav
# changed the split for the document.
#
# Revision 1.78 2009/09/29 20:04:29 gustav
# changed src_only to src-only. This makes it the same as, e.g., big-doc.
#
# Revision 1.77 2009/09/17 20:29:05 gustav
# Replaced the Mie test EPS files.
#
# Revision 1.76 2009/09/15 19:09:43 gustav
# Changed the divide.
#
# Revision 1.75 2009/09/14 16:47:18 gustav
# Added Ball-4, Ball-5, Mie-1 and Mie-2 to EPS files.
#
# Revision 1.74 2009/09/11 22:45:27 gustav
```

```
# Changed the split location and the biblio.
#
# Revision 1.73 2009/09/11 22:07:09 gustav
# added Mie-1.scm to examples.
#
# Revision 1.72 2009/09/03 20:53:54 gustav
# Added BIBLIO
#
# Revision 1.71 2009/09/03 20:40:48 gustav
# Added split params.
#
# Revision 1.70 2009/09/01 21:59:54 gustav
# Changed the location of the split.
#
# Revision 1.69 2009/08/31 21:34:03 gustav
# Changed the split point.
#
# Revision 1.68 2009/08/31 18:21:18 gustav
# Changed the order of big-all and all.
#
# Revision 1.67 2009/08/31 18:03:08 gustav
# Introduced big-all and all, also big-doc and doc.
#
# Revision 1.66 2009/08/25 23:55:05 gustav
# A minor bug in dvipdfm options.
#
# Revision 1.65 2009/08/25 23:18:49 gustav
# Added clauses for making Forms_1.pdf and Forms_2.pdf.
#
# Revision 1.64 2009/08/25 21:10:29 gustav
# added Sness-2.eps to EPS_files.
#
# Revision 1.63 2009/08/25 18:09:13 gustav
# Added Snell.scm to the list of examples.
#
# Revision 1.62 2009/08/04 20:10:05 gustav
# added a target src_only. It doesn't make the doc.
#
# Revision 1.61 2009/07/28 19:08:31 gustav
# Added visit-example-2.eps to EPS files.
#
# Revision 1.60 2009/07/27 22:07:11 gustav
# Added visit-example-1.eps to EPS files.
#
# Revision 1.59 2009/07/27 19:41:39 gustav
# Added Snell-no-media.scm to Examples.
#
# Revision 1.58 2009/06/11 18:59:07 gustav
# Added FourierFlux-Lib.scm to examples.
#
# Revision 1.57 2009/05/29 20:05:46 gustav
# Added PowerFlux-Lib.scm to examples.
#
# Revision 1.56 2009/05/27 14:46:18 gustav
# Added WaveFlux-Lib.scm and WaveFlux-halfLib.scm to examples.
#
# Revision 1.55 2009/05/11 19:30:55 gustav
# Added drude-lib.scm to Examples.
#
```

```
# Revision 1.54 2009/04/22 19:12:46 gustav
# Added media_lib.scm to Examples
#
# Revision 1.53 2009/04/21 18:34:03 gustav
# Added signal_lib.scm to examples.
# .l
#
# Revision 1.52 2009/03/15 15:06:20 gustav
# Added FluxLib.scm to examples.
#
# Revision 1.51 2009/01/13 22:53:41 gustav
# add more eps files.
#
# Revision 1.50 2008/06/09 01:38:40 gustav
# Added FourierOutput-256.txt to Examples.
#
# Revision 1.49 2008/06/08 21:28:12 gustav
# signal.scm -> signal.eps
#
# Revision 1.48 2008/06/08 21:23:21 gustav
# Added signal.eps to examples and FourierOutput, too.
#
# Revision 1.47 2008/06/08 20:34:08 gustav
# Added FourierFlux.scm to Examples.
#
# Revision 1.46 2008/06/07 19:23:36 gustav
# Added PowerFlux-out-128.txt to the examples.
#
# Revision 1.45 2008/06/07 19:01:22 gustav
# Added PowerFlux.scm to examples.
#
# Revision 1.44 2008/06/06 16:13:37 gustav
# Added WaveFlux.scm and FluxOutput.scm to Examples.
#
# Revision 1.43 2008/06/03 00:53:29 gustav
# Added Job-128.txt and Job-256.txt to Examples.
#
# Revision 1.42 2008/06/03 00:24:22 gustav
# Added PlaneWaveB.scm to Examples.
#
# Revision 1.41 2008/05/30 20:16:10 gustav
# Added PlaneWave.scm to Examples.
#
# Revision 1.40 2008/05/30 20:08:10 gustav
# Added accuracy.txt to examples.
#
# Revision 1.39 2008/05/29 20:36:43 gustav
# Added vinterpolate.scm to examples.
#
# Revision 1.38 2008/05/24 17:07:50 gustav
# added TestBallE.scm to examples.
#
# Revision 1.37 2008/05/16 20:33:04 gustav
# Added TestBallD.scm to examples.
#
# Revision 1.36 2008/05/09 22:23:32 gustav
# Added interpolate.scm to examples.
#
# Revision 1.35 2008/04/30 21:37:17 gustav
```

```
# Added FourierFlux.scm to Examples
#
# Revision 1.34 2008/04/30 17:44:26 gustav
# Added Flux_1.scm to Examples.
#
# Revision 1.33 2008/04/22 17:40:57 gustav
# Added domain_partition.cpp to examples.
#
# Revision 1.32 2008/04/21 18:45:43 gustav
# Added jazz_submit.sh and Dust.scm to Examples.
#
# Revision 1.31 2008/04/04 14:50:09 gustav
# Added rcsclean op to clobber: this should remove Makefile that's been put
# in the top directory by "clean".
#
# Revision 1.30 2008/04/04 14:48:50 gustav
# Moved extraction of Makefile to the clean target.
#
# Revision 1.29 2008/04/04 14:46:09 gustav
# Added the rule for getting Makefile out.
#
# Revision 1.28 2008/04/04 14:44:21 gustav
# Added Makefile to all target.
#
# Revision 1.27 2008/04/03 19:28:31 gustav
# Added inject.c and TestBallC.scm to Examples.
#
# Revision 1.26 2008/04/03 14:15:08 gustav
# Added TestBall.scm to the Examples list.
#
# Revision 1.25 2008/04/02 21:12:56 gustav
# Added drude.scm to Examples.
#
# Revision 1.24 2008/04/02 19:37:31 gustav
# Added drude.c to the Examples list.
#
# Revision 1.23 2008/04/02 17:32:04 gustav
# Added an rcsclean in the Examples directory.
#
# Revision 1.22 2008/04/02 16:48:39 gustav
# Added d-to-e-conversion.c and create-s-array.cpp to Examples.
#
# Revision 1.21 2008/04/02 15:45:22 gustav
# Added d-to-e-conversion.cpp to Examples.
#
# Revision 1.20 2008/04/02 15:31:03 gustav
# Added d-to-e-conversion.f to Examples.
#
# Revision 1.19 2008/04/02 14:55:41 gustav
# Added auxiliary-fields.cpp to Examples.
#
# Revision 1.18 2008/03/27 19:55:29 gustav
# Expanded the list of examples.
#
# Revision 1.17 2008/03/26 20:09:16 gustav
# Defined EXAMPLES.
#
# Revision 1.16 2008/03/20 18:54:25 gustav
# Added PY_SOURCES and SH_SOURCES.
```

```
#
# Revision 1.15 2008/03/14 21:05:50 gustav
# EPS_SOURCES had to be moved to the doc target together with F_FILES.
#
# Revision 1.14 2008/03/14 21:02:07 gustav
# Added EPS_FILES and made .tex depend on them and F_SOURCES.
#
# Revision 1.13 2008/02/28 19:28:40 gustav
# Changed the order of targets in all.
#
# Revision 1.12 2008/02/28 19:26:01 gustav
# Added a doc target to all.
#
# Revision 1.11 2008/02/14 20:12:59 gustav
# Added dependencies for Injection.h_std, Conversion.h_std, and Forms.hw.
#
# Revision 1.10 2008/02/14 20:03:59 gustav
# Added dependencies for Injection.c and Conversion.c.
#
# Revision 1.9 2008/02/14 19:57:10 gustav
# Added specific dependencies for files defined on Extensions.w
#
# Revision 1.8 2007/10/04 22:56:57 gustav
# Added draw.f to Fortran sources.
#
# Revision 1.7 2007/10/04 15:17:44 gustav
# Added a clause for making the doc directory and cleaning it away.
# Cleaned up the "move" clause.
#
# Revision 1.6 2007/09/21 14:17:01 gustav
# Replaced cweave with ctangle in the "raw" clause.
#
# Revision 1.5 2007/09/21 14:09:32 gustav
# added target "move" to all
# added "move" clause that moves Forms.cpp and Form.H back to the top directory,
# because this is how Chombo Makefiles want it.
# added clean up of Forms.cpp and Forms.H from the top directory.
#
# Revision 1.4 2007/09/20 18:46:35 gustav
# more minor changes.
#
# Revision 1.3 2007/09/20 18:36:44 gustav
# Fixed clean up problems.
#
# Revision 1.2 2007/09/20 18:34:14 gustav
# Had to add dependencies for making a doc.
#
# Revision 1.1 2007/09/20 18:25:42 gustav
# Initial revision
#
#
```

15 Closing Comments

Corea Septentrionalis nuntiavit se primum experimentum nucleare fecisse. Ita illa dictatura Stalinistica, pauper et inclusa, futura est nona in orbe terrarum potestas, cui arma nuclearia esse constat.

Experimentum Coreæ Septentrionalis a moderatoribus orbis terrarum et a Consilio Securitatis Nationum Unitarum severissime condemnatum est.

Americani novas contra Coream Septentrionalem sanctiones a Nationibus Unitis flagitaverunt, Coreani Meridiani nuntiaverunt se potestatem nuclearem sibi propinquam pati non posse.

Etiam Sinenses, qui fœderati fideles Coreanorum Septentrionalium esse solent, eos reprobaverunt.

Minister primarius Iaponiæ dixit illo experimento nucleari stabilitatem Asiæ Orientalis mutari.

Sinenses anno vergente plus miliardum trecenti miliones numerabantur.

Ex eis quadraginta quattuor centesimæ in urbibus vivebant. Politica unius infantis æquilibrium sexuum perturbavit, nam plures quam dimidium Sinensium (51.5%) sunt viri.

Magistratus censent post quindecim annos numerum virorum cælibum triginta milionibus maiorem quam feminarum apud Sinenses futurum esse.

References

[1] Dinshaw S. Balsara, "Divergence-Free Adaptive Mesh Refinement for Magnetohydrodynamics," Journal of Computational Physics, vol. 174, pp. 614–648, 2001.

[2] F. I. Baida and D. Van Labeke, "Three-dimensional structures for enhanced transmission through a metallic film: Annular aperture arrays," Physical Review B, Vol. 67, No. 15, p. 155314(7), 25 April 2003

[3] Craig F. Bohren and Donald R. Huffman, "Absorption and Scattering of Light by Small Particles", Wiley-VCH 2004

[4] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini and B. Van Straalen, "Chombo Software Package for AMR Applications Design Document" Applied Numerical Algorithms Group, NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, September 12, 2003.

[5] R. Courant, K. Friedrichs and H. Lewy, "Über die partiellen Differenzengleichungen der mathematischen Physik," Mathematische Annalen, vol. 100, no. 1, pp. 32–74, 1928.

[6] David Drysdale, "Tutorial Introduction to Guile," `http://www.gnu.org/software/guile/docs/guile-tut/tutorial.html`, 2000.

[7] Richard P. Feynman, Robert B. Leighton and Matthew Sands, "The Feynman Lectures on Physics–Mainly Electromagnetism and Matter," Volume II, Addison-Wesley Publishing Company, Fourteenth Printing, 1981

[8] Stephen D. Gedney, "An Anisotropic Perfectly Matched Layer—Absorbing Medium for the Truncation of FDTD Lattices," IEEE Transactions on Antennas and Propagation, Vol. 44, No. 12, pp. 1630–1639, December 1996.

[9] Michael Gran, "How to extend C programs with Guile," `http://lonelycactus.com/guilebook/book1.html`, 2004

[10] Stephen K. Gray and Teobald Kupka, "Propagation of light in metallic nanowire arrays: Finite-difference time-domain studies of silver cylinders," Physical Review B, Vol. 68, No. 4, p. 045415(11), 16 July 2003

[11] Guile: "Project GNU's extension language," `http://www.gnu.org/software/guile/guile.html`

[12] P. B. Johnson and R. W. Christy, "Optical Constants of the Noble Metals," Physical Review B, Vol. 6, No. 12, pp. 4370–4379, 15 December 1972

[13] Tae-Woo Lee and Stephen K. Gray, "Subwavelength light bending by metal slit structures," Optics Express, Vol. 13, No. 24, p. 9652(8), 28 November 2005

[14] D. W. Lynch and W. R. Hunter, in "Handbook of Optical Constants of Solids," ed. E. D. Palic, pp. 350–357, Academic, Orlando 1985

[15] Dan Martin and Gustav Meglicki, "Chombo Class PiecewiseLinearFillPatchFace," Code Weave Report, Argonne National Laboratory, October 13, 2006, `http://www-unix.mcs.anl.gov/~meglicki/Shapes/Prolongate.pdf`

[16] Dan Martin and Gustav Meglicki, "Chombo Class LevelFluxRegisterEdge," Code Weave Report, Argonne National Laboratory, November 29, 2006, `http://www-unix.mcs.anl.gov/~meglicki/Shapes/Reflux.pdf`

[17] Jeffrey M. McMahon, Joel Henzie, Teri W. Odom, George C. Schatz, and Stephen K. Gray, "Tailoring the sensing capabilities of nanohole arrays in gold films with Rayleigh anomaly-surface plasmon polaritons," Optics Express, Vol. 15, No. 26, pp. 18119–18129, 24 December 2007

[18] Zdzisław Meglicki, Stephen K. Gray and Boyana Norris, "Multigrid FDTD with Chombo," Computer Physics Communications, vol. 176, no. 2, pp. 109–120, January 2007.

[19] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, "Numerical Recipes", Cambridge University Press, 1987

[20] Dennis M. Sullivan, "Electromagnetic Simulation Using the FDTD Method," IEEE Press Series on RF and Microwave Technology, IEEE Press, 2000.

[21] Allen Taflove and Susan C. Hagness, "Computational Electrodynamics: The Finite-Difference Time-Domain Method," Second Edition, Artech House, 2000.

[22] A. R. Zakharian, M. Brio, C. Dineen, and J. V. Moloney, "Second-Order Accurate FDTD Space and Time Grid Refinement Method in Three Space Dimensions," IEEE Photonics Technology Letters, Vol. 18, No. 11, pp. 1237–1239, June 1, 2006

[23] A. R. Zakharian, M. Brio, and J. V. Moloney, "FDTD based local mesh refinement method for Maxwell's equations in two space dimensions," Communications in Mathematical Sciences, vol. 2, no. 3, pp. 497–513, September 2004

[24] A. R. Zakharian, M. Brio, C. Dineen, and J. V. Moloney, "Stability of 2-D FDTD algorithms with local mesh refinement for Maxwell's equations," Communications in Mathematical Sciences, vol. 4, no. 2, pp. 345–374, June 2006

Index

Here is an index of various terms and identifiers used, and the chunks, *not* page numbers, where they appear. Underlined entries indicate locations of definitions.

LaTeX: 1.
`_FORMS_H_`: 237.
_lib_config: 110.
a_box: 306, 308, 309, 326, 333.
a_curl_box: 305.
a_nComp: 306, 308, 309, 326, 333.
abort: 274, 305.
actual_number_of_levels: 78, 79.
advance_b: 80, 125, 240.
advance_b_n: 125, 127, 240.
advance_b_0: 125, 126, 240.
advance_clock: 80.
advance_clock1: 80.
advance_clock2: 80.
advance_d: 80, 105, 240.
advance_d_n: 105, 107, 240.
advance_d_0: 105, 106, 240.
argc: 72, 73, 74, 270, 274.
Argonne National Laboratory: 1.
argv: 72, 73, 74, 270, 274.
assert: 196, 291, 305, 308, 309, 310, 311, 312, 313, 314, 315, 316, 320, 321, 322, 325, 326, 327, 328, 332, 333, 334, 335.
astyle: 3.
auxiliaries: 94, 95, 96, 97.
B: 129, 132, 133, 201, 259, 260, 266, 293.
`b_to_h`: 340.
B_dir: 132, 133.
B_idx: 97, 126, 127, 129, 132, 133, 198, 201, 204, 208, 254, 259, 260, 266.
b_to_h: 80, 131, 240.
b_to_h_n: 131, 133, 240.
*b_to_h_n_*: 133, 260.
b_to_h_0: 131, 132, 240.
*b_to_h_0_*: 132, 259.
barrier: 66, 197.
BaseFab: 92, 93, 117, 118, 120, 121, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 200, 222, 223, 227, 231, 266, 293, 294, 297, 298, 299, 301, 305, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 323, 325, 326, 327, 328, 329.
`BASISV`: 90, 198.
big_end: 234.
bigEnd: 109, 129, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 219, 234.
black triangle: 2.
bounding_box: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
bounding_box_hi: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
bounding_box_lo: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
box: 106, 126, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 219, 227, 231, 293, 296, 297, 298, 299, 300, 304, 311, 315, 316, 319, 320, 321, 322, 323, 325, 327, 328, 330, 332, 334, 335.
Box: 87, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 219, 227, 231, 233, 234, 266, 293, 296, 297, 298, 299, 300, 301, 304, 306, 307, 308, 309, 311, 315, 316, 319, 320, 321, 322, 323, 325, 326, 327, 328, 330, 332, 333, 334, 335.
box_iterator: 227, 231.
box_layout: 90, 92, 93, 96, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 217, 219, 224, 233, 234, 266.
box_type: 198, 231, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
BoxIterator: 227, 231.
BoxLayoutData: 96.
BRMeshRefine: 87, 92, 93, 222.
broadcast: 66, 197, 228.
buf: 300, 320, 321, 323, 328, 330, 335.
buffer: 198, 281, 285, 289, 320, 321, 328, 335.
buffer_size: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
`BUFSIZ`: 199, 201.
build_levels: 73, 77, 78, 82, 240.
Bx: 126, 127, 129, 254, 261.
Bx_hi: 129.
Bx_idx: 201, 204, 205, 264.
Bx_lo: 129.
By: 126, 127, 129, 254, 261.
By_hi: 129.
By_idx: 201, 204, 205, 264.
By_lo: 129.
Bz: 126, 127, 129, 254.
Bz_hi: 129.
Bz_idx: 201, 204, 205, 264.
Bz_lo: 129.
C++: 3, 72, 92, 199, 213, 292, 294.
C_center: 141, 143, 144, 145, 146, 165, 167, 168, 169, 170, 171, 173, 174, 175, 176, 183, 185, 186, 187, 188.
C_center_1: 189, 191, 192, 193, 194.
C_center_2: 189, 191, 192, 193, 194.
C_color: 135, 137, 138, 139, 141, 143, 144, 145, 147, 149, 150, 151, 153, 155, 156, 157, 159, 161, 162, 163, 165, 167, 168, 169, 171, 173,

174, 175, 177, 179, 180, 181, 183, 185, 186, 187, 189, 191, 192, 193.
C_cone_1: 164.
C_cone_2: 164.
C_corner: 177, 179, 180, 181, 182.
C_cylinder_1: 158.
C_cylinder_2: 158.
C_e_1: 171, 173, 174, 175, 176, 177, 179, 180, 181.
C_e_2: 171, 173, 174, 175, 176, 177, 179, 180, 181.
C_e_3: 171, 173, 174, 175, 176, 177, 179, 180, 181.
C_end_1: 153, 155, 156, 157, 159, 161, 162, 163, 169, 187.
C_end_2: 153, 155, 156, 157, 159, 161, 162, 163, 169, 187.
C_focus_1: 147, 149, 150, 151, 152.
C_focus_1_ext: 151.
C_focus_2: 147, 149, 150, 151, 152.
C_focus_2_ext: 151.
C_hi: 151, 157, 163, 169, 175, 181, 187, 193.
C_lo: 151, 157, 163, 169, 175, 181, 187, 193.
C_lower_corner: 135, 137, 138, 139, 140.
C_normal: 165, 167, 168, 169, 170, 183, 185, 186, 187, 188.
C_normal_norm: 169, 187.
c_prefix: 274.
C_r_1: 171, 173, 174, 175, 176, 188.
C_r_2: 171, 173, 174, 175, 176, 188.
C_r_3: 171, 173, 174, 175, 176.
C_radius: 141, 143, 144, 145, 146, 153, 155, 156, 157, 158, 159, 161, 162, 163, 164, 165, 167, 168, 169, 170.
C_radius_1: 183, 185, 186, 187, 189, 191, 192, 193, 194.
C_radius_2: 183, 185, 186, 187, 189, 191, 192, 193, 194.
c_str: 201, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
C_sum: 147, 149, 150, 151, 152.
C_thickness: 165, 167, 168, 169, 170.
C_upper_corner: 135, 137, 138, 139, 140.
C_v_1: 182.
C_v_2: 182.
C_v_3: 182.
Cdest: 299, 315, 316, 323, 327, 330, 334.
ceil: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
`CELL`: 293.
cell: 227, 231.
cell_box: 231.
cell_centered_box: 293.
center: 141, 143, 165, 167, 171, 173, 183, 185, 244, 248, 249, 251.
center_1: 189, 191, 252.
center_2: 189, 191, 252.
cerr: 237, 269, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 292.
`CH_CURLBOX_H`: 330.
`CH_EDGEFAB_H`: 294.
`CH_EDGEFABIMPLEM_H`: 303.
`CH_FACEFAB_H`: 323.
`CH_FACEFABIMPLEM_H`: 325.
CH_Linux: 237.
`CH_MPI`: 73, 274.
`CH_SCMPARMPARSE_H`: 269.
`CH_SPACEDIM`: 269, 271, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 289, 290, 292.
Chombo: 1, 2, 72, 73, 90, 92, 293, 299, 305, 309, 311, 318.
Chombo, Makefile: 3.
Chombo, mk/Make.rules: 3.
Chombo, staggered: 2.
Chombo, two-dimensional library: 3.
`CHOMBO_HOME`: 110.
Chombo_MPI: 274.
Chombovis: 16.
cin: 237.
clear: 90, 91, 93, 201, 234, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 296, 297, 307, 323, 326, 330, 333.
clock: 80.
`CLOCKS_PER_SEC`: 80.
close: 213, 292.
collection_vector: 197.
color: 135, 137, 141, 143, 147, 149, 153, 155, 159, 161, 165, 167, 171, 173, 177, 179, 183, 185, 189, 191, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
comm: 274.
command: 199.
comp: 310, 327, 334.
complement: 232.
comps: 300, 319, 320, 321, 323, 328, 330, 335.
compute: 197, 225.
cone_box: 163.
cone_box_hi: 163.
cone_box_lo: 163.
contains: 151, 157, 163, 169, 175, 181, 187, 193, 198, 271, 275, 311, 327, 334.
convert_clock: 80.
convert_clock1: 80.
convert_clock2: 80.
copy: 201, 210, 299, 305, 312, 313, 314, 315, 316, 323, 327, 330, 334.
copyright: 1.
corner: 177, 179, 198, 250.
corners: 181.
count: 80, 97, 99, 100, 101, 109, 123, 129, 181, 197, 198, 201, 203, 204, 205, 206, 207, 208, 210,

228, 230, 233, 234, 235, 240, 263, 264, 277, 282, 283, 284, 285, 286, 287, 288, 289.
countval: 96, 201, 271, 277.
cout: 73, 237, 269, 290, 293.
Csrc: 299, 315, 316, 323, 327, 330, 334.
curl refluxing: 2.
CurlBox: 1, 3, 22, 96, 109, 117, 118, 120, 121, 200, 201, 266, 267, 329, 330, 332, 333, 334, 335.
current_component: 106, 107, 126, 127.
current_idx: 198.
current_level_ptr: 72, 86, 92, 93, 94, 102, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
Cweb: 1, 3.
cx: 198.
Cx_hi: 106, 117, 118, 126, 132.
Cx_length: 235.
Cx_lo: 106, 117, 118, 126, 132.
cxy: 198.
cxyz: 198.
cy: 198.
Cy_hi: 106, 117, 118, 126, 132.
Cy_length: 235.
Cy_lo: 106, 117, 118, 126, 132.
cyz: 198.
cz: 198.
Cz_hi: 106, 117, 118, 126, 132.
Cz_length: 235.
Cz_lo: 106, 117, 118, 126, 132.
czx: 198.
c0: 198.
C0x: 99, 117, 118, 132, 235, 256, 259, 266.
C0y: 100, 117, 118, 132, 235, 256, 259, 266.
C0z: 101, 117, 118, 132, 235, 256, 259, 266.
C1x: 99, 106, 117, 118, 126, 132, 235, 254, 256, 259, 266.
C1y: 100, 106, 117, 118, 126, 132, 235, 254, 256, 259, 266.
C1z: 101, 106, 117, 118, 126, 132, 235, 254, 256, 259, 266.
C2x: 99, 106, 126, 235, 254, 266.
C2y: 100, 106, 126, 235, 254, 266.
C2z: 101, 106, 126, 235, 254, 266.
C3x: 99, 117, 118, 132, 235, 256, 259, 266.
C3y: 100, 117, 118, 132, 235, 256, 259, 266.
C3z: 101, 117, 118, 132, 235, 256, 259, 266.
C4x: 99, 117, 118, 132, 235, 256, 259, 266.
C4y: 100, 117, 118, 132, 235, 256, 259, 266.
C4z: 101, 117, 118, 132, 235, 256, 259, 266.
D: 109, 117, 118, 120, 121, 122, 123, 124, 201, 256, 257, 266.
`d_to_e`: 339.
`D_DECL`: 215.
D_dir: 117, 118, 120, 121.
D_idx: 97, 106, 107, 109, 117, 118, 120, 121, 198, 201, 203, 254, 256, 257, 266.
D_old: 122, 123, 124.
D_old_var: 22, 122, 123, 199.
d_to_e: 80, 102, 115, 240.
d_to_e_n: 115, 119, 240.
d_to_e_n_n: 119, 121, 240.
*d_to_e_n_n_*: 121, 257.
d_to_e_n_0: 119, 120, 240.
*d_to_e_n_0_*: 120, 257.
d_to_e_0: 115, 116, 240.
d_to_e_0_n: 22, 116, 118, 240.
*d_to_e_0_n_*: 22, 118, 256.
d_to_e_0_0: 116, 117, 240.
*d_to_e_0_0_*: 117, 256.
D_var: 21, 22, 122, 123, 199.
dangerous bend sign: 2.
data: 72, 73.
data_index: 106, 107, 109, 117, 118, 120, 121, 126, 127, 129, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 233, 234.
data_iterator: 92, 96, 97, 106, 107, 117, 118, 120, 121, 126, 127, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 224, 227, 231, 234, 266.
DataIndex: 106, 107, 109, 117, 118, 120, 121, 126, 127, 129, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 233, 234, 266.
DataIterator: 92, 96, 97, 106, 107, 117, 118, 120, 121, 126, 127, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 224, 231, 234, 266.
dataIterator: 96.
dataPtr: 106, 107, 109, 117, 118, 120, 121, 126, 127, 129, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 203, 204, 205, 206, 207, 208, 210.
datum: 197.
`DEBUG`: 305.
define: 90, 92, 93, 96, 106, 107, 124, 126, 127, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 237, 239, 269, 294, 297, 303, 306, 308, 309, 319, 323, 325, 326, 330, 333.
`DELTA`: 84, 239.
delta: 84, 89, 90, 93, 107, 109, 127, 129, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 220, 230, 231, 235, 240, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 261, 266.
delta_r: 151.
depth: 230.
destComp: 299, 313, 314, 323, 327, 330, 334.
destination: 197.
determinant: 181.
dielectrics: 21.
dim: 277.
dimensions: 277.

dir: 22, 108, 109, 117, 118, 120, 121, 128, 129, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 203, 204, 206, 207, 208, 231, 233, 234, 240, 256, 257, 259, 260, 261, 263, 293, 298, 299, 305, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 319, 320, 321, 322, 323, 325, 326, 327, 328, 330, 332, 333, 334, 335.
dirBox: 320, 321, 322, 328, 335.
direction: 122, 123, 124.
direction_var: 22, 122, 123, 199.
dirSize: 319, 320, 321, 328, 335.
DisjointBoxLayout: 83, 92, 93, 96, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 200, 201, 217, 224, 233, 234, 266, 293.
display: 92.
does_scm_symbol_exist: 21, 196, 241, 272, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 291.
domain: 90, 93, 201, 266.
domainSplit: 90, 214.
draw_ary_ellipsoid: 171, 199, 249.
draw_ball: 3, 141, 199, 244.
draw_box: 3, 20, 135, 199, 243.
draw_cone: 159, 199, 247.
draw_cylinder: 153, 159, 199, 246.
draw_disk: 165, 187, 199, 248.
draw_ellipsoid: 3, 147, 199, 245.
draw_lens: 189, 199, 252.
draw_media_ary_ellipsoid: 174, 175, 249.
draw_media_ball: 144, 145, 244.
draw_media_box: 138, 139, 243.
draw_media_cone: 162, 163, 247.
draw_media_cylinder: 151, 156, 157, 246.
draw_media_disk: 168, 169, 248.
draw_media_ellipsoid: 150, 151, 245.
draw_media_lens: 192, 193, 252.
draw_media_parallelepiped: 180, 181, 250.
draw_media_torus: 186, 187, 251.
draw_on_media: 135, 137, 138, 141, 143, 144, 147, 149, 150, 153, 155, 156, 159, 161, 162, 165, 167, 168, 171, 173, 174, 177, 179, 180, 183, 185, 186, 189, 191, 192.
draw_parallelepiped: 177, 199, 250.
draw_tags_ary_ellipsoid: 174, 176, 249.
draw_tags_ball: 144, 146, 244.
draw_tags_box: 138, 140, 243.
draw_tags_cone: 162, 164, 247.
draw_tags_cylinder: 156, 158, 246.
draw_tags_disk: 168, 170, 248.
draw_tags_ellipsoid: 150, 152, 245.
draw_tags_lens: 192, 194, 252.
draw_tags_parallelepiped: 180, 182, 250.
draw_tags_torus: 186, 188, 251.
draw_torus: 183, 199, 251.
dt: 80, 90, 93, 105, 107, 109, 117, 118, 120, 121, 122, 123, 124, 125, 127, 129, 132, 133, 201, 235, 256, 257, 259, 260, 266.
dt_by_delta: 109, 129, 261.
dt_by_dg: 107, 127, 254.
dt_var: 22, 80, 122, 123, 199.
dump_data: 73, 80, 200, 201, 240.
dx: 198.
Dx: 106, 107, 109, 254, 261.
Dx_hi: 109.
Dx_idx: 201, 203.
Dx_lo: 109.
dy: 198.
Dy: 106, 107, 109, 254, 261.
Dy_hi: 109.
Dy_idx: 201, 203.
Dy_lo: 109.
dz: 198.
Dz: 106, 107, 109, 254.
Dz_hi: 109.
Dz_idx: 201, 203.
Dz_lo: 109.
E: 117, 118, 120, 121, 201, 256, 257, 266, 293.
E_box_dir: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
E_dir: 117, 118, 120, 121.
E_hi: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
E_idx: 97, 117, 118, 120, 121, 126, 127, 198, 201, 203, 207, 254, 256, 257, 266.
e_lambda: 22, 122, 123, 199.
E_lo: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
E_old: 122, 123, 124.
E_old_var: 22, 122, 123, 199.
E_ret: 122, 123.
e_times_n: 112.
E_var: 199.
e_1: 171, 173, 177, 179, 249, 250.
e_2: 171, 173, 177, 179, 249, 250.
e_3: 171, 173, 177, 179, 249, 250.
edgeBox: 308, 309, 311, 319, 333, 334, 335.
EdgeDataBox: 1, 293.
EdgeFab: 1, 3, 96, 117, 118, 120, 121, 201, 231, 234, 266, 267, 293, 295, 296, 297, 298, 299, 301, 302, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 329.
EdgeFab_1: 302.
EdgeFab_2: 302.
edgeRegion: 316, 334.
effective_grid_bounds: 90, 93, 214, 222, 240.
eMdx_idx: 201, 207.
eMdy_idx: 201, 207.
eMdz_idx: 201, 207.

empty: 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
En_idx: 201, 205.
enclosedCells: 198, 308, 309, 311, 315, 316, 319, 320, 321, 322, 333, 334, 335.
end_1: 153, 155, 157, 159, 161, 163, 246, 247.
end_2: 153, 155, 157, 159, 161, 163, 246, 247.
endl: 73, 74, 76, 78, 79, 80, 82, 88, 89, 90, 91, 92, 93, 95, 96, 97, 98, 99, 100, 101, 102, 103, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 135, 136, 137, 139, 140, 141, 142, 143, 145, 146, 147, 148, 149, 151, 152, 153, 154, 155, 157, 158, 159, 160, 161, 163, 164, 165, 166, 167, 169, 170, 171, 172, 173, 175, 176, 177, 178, 179, 181, 182, 183, 184, 185, 187, 188, 189, 190, 191, 193, 194, 197, 198, 199, 201, 203, 204, 205, 206, 210, 212, 213, 218, 219, 220, 221, 226, 227, 228, 229, 231, 232, 233, 234, 235, 237, 269, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 292.
enlarged_box: 234.
eps: 212.
`EPS_1`: 21.
`EPS_2`: 21.
`EPSILON`: 139, 145, 239.
`ERR_ADVANCE_B`: 126, 127, 239.
`ERR_ADVANCE_D`: 106, 107, 239.
`ERR_B_TO_H`: 132, 133, 239.
`ERR_BAD_DIMENSION`: 73, 239.
`ERR_BUILD_LEVELS`: 88, 89, 90, 91, 92, 93, 239.
`ERR_COMPLEMENT`: 232, 239.
`ERR_D_TO_E`: 116, 117, 118, 119, 120, 121, 239.
`ERR_DUMP_DATA`: 205, 206, 210, 239.
`ERR_FILL_LEVELS`: 95, 102, 239.
`ERR_INITIALIZE_SCHEME`: 199, 239.
`ERR_INJECT_B`: 128, 129, 239.
`ERR_INJECT_D`: 108, 109, 239.
`ERR_MARK_REGIONS`: 231, 239.
`ERR_NO_INPUT_FILE`: 74, 239.
`ERR_PARFILE_LOAD_FAILED`: 269, 274.
`ERR_PARFILE_NOT_READABLE`: 269, 274.
`ERR_PARFILE_NULL`: 269, 274.
`ERR_PML_ARRAYS`: 235, 239.
errno: 213, 237, 292.
`evaluate_energy`: 344.
`evaluate_flow`: 344.
*evaluate_energy_*: 205, 264.
*evaluate_flow_*: 206, 264.
ex: 112.
Ex: 126, 127, 254.
Ex_idx: 200, 201, 203, 205, 206, 264.
*ex_inc_*: 111, 112, 113.
ex_lambda: 3, 112, 199.
exchange: 80, 106, 107, 126, 127, 300.
exit: 73, 74, 77, 78, 79, 80, 82, 88, 89, 90, 91, 92, 93, 95, 98, 102, 103, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 199, 201, 203, 204, 205, 206, 210, 213, 231, 233, 235, 282, 283, 284, 285, 286, 287, 288, 289, 292.
exit_status: 73, 74, 77, 78, 79, 80, 213, 292.
`EXIT_SUCCESS`: 72, 73, 83, 94, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 124, 125, 126, 127, 128, 129, 131, 132, 133, 199, 201, 203, 204, 205, 206, 210, 214, 222, 233, 234, 235, 239.
ey: 112.
Ey: 126, 127, 254.
Ey_idx: 201, 203, 205, 206, 264.
*ey_inc_*: 111, 112, 113.
ey_lambda: 3, 112, 199.
ez: 112.
Ez: 126, 127, 254.
Ez_idx: 201, 203, 205, 206, 264.
*ez_inc_*: 111, 112, 113.
ez_lambda: 3, 112, 199.
f: 198.
`f_draw_ary_ellipsoid`: 341.
`f_draw_ball`: 341.
`f_draw_box`: 341.
`f_draw_cone`: 341.
`f_draw_cylinder`: 341.
`f_draw_disk`: 341.
`f_draw_ellipsoid`: 341.
`f_draw_lens`: 341.
`f_draw_parallelepiped`: 341.
`f_draw_torus`: 341.
*f_draw_ary_ellipsoid_*: 175, 249.
*f_draw_ball_*: 145, 146, 244.
*f_draw_box_*: 139, 140, 243.
*f_draw_cone_*: 163, 164, 247.
*f_draw_cylinder_*: 157, 158, 246.
*f_draw_disk_*: 169, 248.
*f_draw_ellipsoid_*: 151, 152, 245.
*f_draw_lens_*: 193, 252.
*f_draw_parallelepiped_*: 181, 250.
*f_draw_torus_*: 187, 251.
f_int: 198.
face_boxes: 233, 234.
faceBox: 326, 327, 328.
FaceFab: 1, 3, 96, 132, 133, 201, 231, 234, 266, 267, 322, 323, 325, 326, 327, 328.
faceRegion: 327.
fail: 213, 292.
false: 75, 80, 83, 92, 94, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 135, 137, 139, 140, 141,

143, 145, 146, 147, 149, 151, 152, 153, 155, 157, 158, 159, 161, 163, 164, 165, 167, 169, 170, 171, 173, 175, 176, 177, 179, 181, 182, 183, 185, 187, 188, 189, 191, 193, 194, 196, 198, 199, 201, 214, 222, 231, 233, 234, 235, 275, 276, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 291, 292, 305.
FArrayBox: 22, 106, 107, 109, 117, 118, 120, 121, 126, 127, 129, 132, 133, 198, 200, 201, 210, 266, 267, 293, 294, 329, 330, 332, 333, 334, 335.
fastget: 271, 278, 279, 280, 281.
fastgetarr: 271, 282, 283, 284.
fastput: 80, 271, 278, 279, 280, 281.
fastputarr: 271, 282, 283, 284.
FDTD: 2.
field: 198, 201, 203, 204, 205, 206, 207, 208, 210.
file_name: 201, 213, 240, 272, 292.
file_status: 213, 292.
file_stream: 213, 292.
fill_levels: 77, 79, 94, 200, 240.
fill_PML_arrays: 98, 235, 240.
fld: 263.
floor: 90, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 212.
flush: 96, 106, 126, 199, 201, 237, 269.
flux: 13.
FluxBox: 1, 96, 129, 132, 133, 200, 201, 266, 293, 308, 312, 313, 315, 322.
focus_1: 147, 149, 245.
focus_2: 147, 149, 245.
format: 201.
forms: 75.
Forms library: 2.
Forms, binary name: 3.
Forms, required utilities: 3.
Forms, source files: 3.
Forms.cpp: 3.
Forms.H: 3.
Forms.pdf: 3.
`FORMS_OUTPUT`: 201, 239.
forms_parser: 199.
Fortran: 2, 294, 299.
fortran_verbose: 254.
Fourier transform: 13.
free: 281, 285, 289.
`from_edge_to_center`: 344.
`from_face_to_center`: 344.
*from_edge_to_center_*: 200, 203, 263.
*from_face_to_center_*: 200, 204, 263.
*from_int_edge_to_center_*: 207, 263.
*from_int_face_to_center_*: 208, 263.
full_box: 198.
fullname: 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
F77: 3.
garbage_collect: 108, 115, 128, 131.
gather: 66, 197, 228.
get: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 219, 233, 234, 271, 278, 279, 280, 281.
getarr: 201, 271, 282, 283, 284, 285, 286, 287, 288, 289.
getEdgeData: 293, 298, 305, 330, 332.
getFaceData: 323, 325.
getFlux: 293.
gh_enter: 72.
gh_new_procedure: 199.
ghost_margin: 94, 96, 106, 107, 109, 117, 118, 120, 121, 126, 127, 129, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 230, 235, 240, 243, 254, 256, 257, 259, 260.
Goldberg, Jeffrey: 1.
grid_parameter_parser: 85, 88, 91, 94, 95.
grow: 198.
Guile: 2, 5, 20, 33, 48, 72, 92.
Guile, linking: 3.
Guile, version: 3.
H: 132, 133, 201, 259, 260, 266.
H_box_dir: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
H_dir: 132, 133.
H_hi: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
H_idx: 97, 106, 107, 132, 133, 198, 201, 204, 254, 259, 260, 266.
H_lo: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
have_post_all_lambda: 80, 199.
have_post_iteration_lambda: 80, 199.
HDF5, version: 3.
hi: 277.
hi_vector: 219.
high_corner: 243.
hiVect: 106, 107, 117, 118, 120, 121, 126, 127, 132, 133, 203, 204, 205, 206, 207, 208, 210.
hMdx_idx: 201, 208.
hMdy_idx: 201, 208.
hMdz_idx: 201, 208.
hold: 236.
Hx: 106, 107, 254.
Hx_idx: 201, 204, 206.
*hx_inc_*: 111, 113.
Hy: 106, 107, 254.
Hy_idx: 201, 204, 206.
*hy_inc_*: 111, 113.
Hz: 106, 107, 254.
Hz_idx: 201, 204, 206.
*hz_inc_*: 111, 113.
i: 89, 90, 93, 137, 139, 143, 145, 149, 151, 155, 157, 161, 163, 167, 169, 173, 175, 179, 181, 185, 187, 191, 193, 198, 210, 219, 220, 231, 232, 234.

i_Bx_hi: 261.
i_Bx_lo: 261.
i_By_hi: 261.
i_By_lo: 261.
i_Dx_hi: 261.
i_Dx_lo: 261.
i_Dy_hi: 261.
i_Dy_lo: 261.
i_fld_hi: 263.
i_fld_lo: 263.
i_hi: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
i_lo: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
i_out_hi: 263, 264, 265.
i_out_lo: 263, 264, 265.
i_pml_hi: 261.
i_pml_lo: 261.
idx: 263.
idx_array: 236, 240.
idx_outp: 265.
idx_pcmp: 265.
ifstream: 213, 237, 269, 292.
ijk_hi: 109, 129.
ijk_lo: 109, 129.
ijk_max: 90, 215, 216, 219, 220, 235, 266.
ijk_min: 90, 215, 216, 219, 220, 235, 266.
image_frequency: 75, 76, 80.
`IMAGE_FREQUENCY`: 75, 239.
imax_B: 259, 260.
imax_Bx: 254.
imax_By: 254.
imax_Bz: 254.
imax_Cx: 254, 256, 259.
imax_Cy: 256, 259.
imax_Cz: 256, 259.
imax_D: 256, 257.
imax_Dx: 254.
imax_Dy: 254.
imax_Dz: 254.
imax_Ex: 254.
imax_Ey: 254.
imax_Ez: 254.
imax_Hx: 254.
imax_Hy: 254.
imax_Hz: 254.
imin_B: 259, 260.
imin_Bx: 254.
imin_By: 254.
imin_Bz: 254.
imin_Cx: 254, 256, 259.
imin_Cy: 256, 259.
imin_Cz: 256, 259.
imin_D: 256, 257.
imin_Dx: 254.
imin_Dy: 254.
imin_Dz: 254.
imin_Ex: 254.
imin_Ey: 254.
imin_Ez: 254.
imin_Hx: 254.
imin_Hy: 254.
imin_Hz: 254.
`in_ball`: 341.
`in_cone`: 341.
`in_cylinder`: 341.
`in_disk`: 341.
`in_ellipsoid`: 341.
`in_lens`: 341.
`in_parallelepiped`: 341.
`in_torus`: 341.
in_cone: 159.
in_cylinder: 159.
index: 210, 230.
Indiana University: 1.
`INITIAL_LABEL`: 75, 239.
initial_label: 75, 76, 80.
Initialize: 81.
initialize_scheme: 3, 20, 22, 66, 73, 199, 241.
`inject_b`: 343.
`inject_d`: 342.
inject_b: 80, 128, 129, 240.
*inject_b_*: 129, 261.
inject_clock: 80.
inject_clock1: 80.
inject_clock2: 80.
inject_d: 80, 108, 109, 240.
*inject_d_*: 109, 111, 261.
inner_main: 72, 73, 75, 76.
input_file_name: 74.
interpolate: 69.
intersects: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
Interval: 106, 107, 126, 127, 299, 300, 315, 316, 319, 320, 321, 323, 327, 328, 330, 334, 335.
IntVect: 60, 90, 92, 93, 94, 96, 109, 129, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 215, 216, 219, 220, 227, 231, 233, 234, 235, 266, 293, 299, 317, 323, 327, 330, 334.
IntVects: 157, 163.
IntVectSet: 87, 93, 222, 223, 225, 266.
is_a_pwr_of_two: 88, 89, 212, 240.
is_file_readable: 213, 240, 272, 274, 292.
is_in: 234.
isClosure: 271, 276.
isDefined: 92.
isDisjoint: 90, 93.
isEmpty: 139, 151, 157, 163, 169, 175, 181, 187, 193, 316, 327, 334.
isProcedure: 91, 102, 199, 271, 276.

isThunk: 271, 276.
item: 197, 277, 282, 283, 284, 285, 286, 287, 288, 289.
iterate: 75.
iterate_count_var: 80, 199.
iterate_parameter_parser: 75, 85, 89.
Iteration: 104.
iv: 317, 327, 334.
iv_lower_corner: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
iv_upper_corner: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
j: 232.
j_Bx_hi: 261.
j_Bx_lo: 261.
j_By_hi: 261.
j_By_lo: 261.
j_Dx_hi: 261.
j_Dx_lo: 261.
j_Dy_hi: 261.
j_Dy_lo: 261.
j_fld_hi: 263.
j_fld_lo: 263.
j_hi: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
j_lo: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
j_out_hi: 263, 264, 265.
j_out_lo: 263, 264, 265.
j_pml_hi: 261.
j_pml_lo: 261.
jmax_B: 259, 260.
jmax_Bx: 254.
jmax_By: 254.
jmax_Bz: 254.
jmax_Cy: 254.
jmax_D: 256, 257.
jmax_Dx: 254.
jmax_Dy: 254.
jmax_Dz: 254.
jmax_Ex: 254.
jmax_Ey: 254.
jmax_Ez: 254.
jmax_Hx: 254.
jmax_Hy: 254.
jmax_Hz: 254.
jmin_B: 259, 260.
jmin_Bx: 254.
jmin_By: 254.
jmin_Bz: 254.
jmin_Cy: 254.
jmin_D: 256, 257.
jmin_Dx: 254.
jmin_Dy: 254.
jmin_Dz: 254.
jmin_Ex: 254.
jmin_Ey: 254.
jmin_Ez: 254.
jmin_Hx: 254.
jmin_Hy: 254.
jmin_Hz: 254.
j0: 20.
j0_wrapper: 20.
k_Bx_hi: 261.
k_Bx_lo: 261.
k_By_hi: 261.
k_By_lo: 261.
k_Dx_hi: 261.
k_Dx_lo: 261.
k_Dy_hi: 261.
k_Dy_lo: 261.
k_fld_hi: 263.
k_fld_lo: 263.
k_hi: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
k_lo: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
k_out_hi: 263, 264, 265.
k_out_lo: 263, 264, 265.
k_pml_hi: 261.
k_pml_lo: 261.
kmax_B: 259, 260.
kmax_Bx: 254.
kmax_By: 254.
kmax_Bz: 254.
kmax_Cz: 254.
kmax_D: 256, 257.
kmax_Dx: 254.
kmax_Dy: 254.
kmax_Dz: 254.
kmax_Ex: 254.
kmax_Ey: 254.
kmax_Ez: 254.
kmax_Hx: 254.
kmax_Hy: 254.
kmax_Hz: 254.
kmin_B: 259, 260.
kmin_Bx: 254.
kmin_By: 254.
kmin_Bz: 254.
kmin_Cz: 254.
kmin_D: 256, 257.
kmin_Dx: 254.
kmin_Dy: 254.
kmin_Dz: 254.
kmin_Ex: 254.
kmin_Ey: 254.
kmin_Ez: 254.
kmin_Hx: 254.
kmin_Hy: 254.

kmin_Hz: 254.
Knuth, Donald E.: 1.
label: 80, 201, 240.
lambda: 92, 102, 222.
LAMBDA_MIN_SIZE: 239.
lambda_return: 92, 102.
Lawrence Berkeley National Laboratory: 1.
layout_box: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193.
layout_iterator: 96, 217, 219, 266.
layoutIterator: 96, 217.
LayoutIterator: 96, 214, 217, 266.
length: 199, 282, 283, 284, 285, 286, 287, 288, 289.
level: 72, 78, 82, 86, 90, 93, 94, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 200, 201, 214, 222, 231, 233, 234, 235, 240, 266.
level_box_layout: 96.
level_number: 91, 92, 93, 105, 106, 107, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 131, 132, 133, 201, 235.
level_ptr: 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 214, 217, 220, 222, 223, 224, 228, 231, 233, 234, 235, 240.
level_0_ptr: 86, 90, 198.
LevelData: 22, 80, 92, 93, 109, 129, 200, 201, 223, 231, 266, 300.
levels: 78, 79, 80, 82, 86, 90, 92, 93, 94, 96, 97, 98, 99, 100, 101, 102, 201, 233, 240.
levels_ptr: 72, 78, 80, 90, 198.
level0_parameter_parser: 85, 89.
Levy, Silvio: 1.
linearIn: 300, 318, 321, 323, 328, 330, 335.
linearOut: 300, 318, 320, 321, 323, 328, 330, 335.
listings: 336.
lo: 277.
Lo: 234.
lo_vector: 219.
LoadBalance: 90, 93.
local_maximum: 201.
local_maximum_location: 201.
local_minimum: 201.
local_minimum_location: 201.
local_tag_set: 223, 227, 228.
local_tags: 227.
log: 212.
LoHiSide: 108, 109, 128, 129, 233, 234, 240.
Lorentz medium: 2.
loVect: 106, 107, 117, 118, 120, 121, 126, 127, 132, 133, 203, 204, 205, 206, 207, 208, 210.
low_corner: 198, 243.
lower_corner: 135, 137.
m: 230.
m_box: 301, 304, 307, 308, 309, 311, 316, 317, 320, 323, 325, 326, 327, 330, 332, 333, 334.
m_edgeData: 301, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 320, 321, 330, 332, 333, 334, 335.
m_faceData: 323, 325, 326, 327, 328.
m_nComp: 296, 297, 301, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 323, 325, 326, 327, 330, 332, 333, 334.
m_nvar: 313.
main: 72, 73, 75.
Make: 110.
make: 3.
make_tag_set: 93, 222, 240.
makedepend: 110.
makeEmpty: 227, 228.
Makefile: 3, 345.
margin: 244, 245, 246, 247, 248, 249, 250, 251, 252, 261.
mark_regions: 98, 231, 240.
mark_TFR_faces: 98, 233, 234, 240.
Martin, Dan: 1, 2, 293.
max: 151, 157, 163, 169, 175, 181, 187, 193, 201, 215, 269.
MAX_BOX_SIZE: 83, 239.
max_box_size: 23, 83, 88, 90.
MAX_BOX_SIZE_MAX: 88, 239.
MAX_BOX_SIZE_MIN: 88, 239.
maximum: 201.
maximum_location: 201.
maxIndex: 201.
Maxwell equations: 2.
media_parameter_parser: 75, 94, 95, 102, 115, 116, 117, 118, 119, 120, 121, 131, 199.
medium: 21, 22, 77, 102, 122, 123, 124, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
medium_E: 96, 97, 117, 118, 120, 121, 122, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 201, 207, 231, 256, 257, 266.
medium_E_dir: 117, 118, 120, 121, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 231.
medium_E_dir_box: 231.
medium_H: 96, 97, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 201, 208, 231, 259, 260, 266.
medium_H_dir: 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 231.
medium_H_dir_box: 231.
medium_var: 21, 22, 122, 123, 199.
mesh_refine: 93.
min: 151, 157, 163, 169, 181, 187, 193, 201, 215, 269, 282, 283, 284, 285, 286, 287, 288, 289.
minimum: 201.
minimum_location: 201.
minIndex: 201.

mk: 110.
MPI_Abort: 274.
MPI_Finalize: 73.
MPI_Init: 73.
my_cross_product: 341.
my_dot_product: 341.
my_norm: 341.
my_rank: 197.
n: 96, 97, 102, 151, 169, 187, 199, 201, 212, 240.
n_aux_fields: 123, 124.
N_AUXILIARIES: 75, 94, 239.
N_AUXILIARIES_MAX: 95, 239.
N_AUXILIARIES_MIN: 95, 239.
n_cells: 23, 84, 89, 90.
N_CELLS: 84, 239.
N_CELLS_MAX: 89, 239.
N_CELLS_MIN: 89, 239.
n_compare: 212.
N_GHOST_CELLS: 94, 106, 107, 109, 117, 118, 120, 121, 126, 127, 129, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 235, 239.
N_LEVELS: 83, 94, 239.
N_LEVELS_MAX: 88, 95, 239.
N_LEVELS_MIN: 88, 95, 239.
n_out: 263, 264.
n_outp: 265.
n_pcmp: 265.
name: 196, 198, 241, 271, 272, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 291.
name_length: 198.
nComp: 96, 118, 121, 201, 296, 297, 298, 299, 304, 310, 311, 312, 313, 314, 323, 325, 327, 330, 332, 334.
need_old_field: 198.
new_component: 109, 129, 261.
new_level_ptr: 86, 93.
new_vector_of_vectors_of_boxes: 87, 93.
newFabPtr: 308, 309, 326, 333.
NO: 213, 239, 292.
NODE: 293.
normal: 165, 167, 183, 185, 248, 251.
normalize: 341.
normalized: 112, 199.
ns: 234.
num_val: 271, 282, 283, 284, 285, 286, 287, 288, 289.
num_val_available: 282, 283, 284, 285, 286, 287, 288, 289.
number: 90, 93, 105, 106, 107, 108, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 131, 132, 133, 231, 235, 266.
number_of_auxiliary_fields: 75, 76, 116, 117, 118, 119, 120, 121, 199, 256, 257.
number_of_digits: 201.
number_of_levels: 83, 88, 91, 94, 95, 96, 97, 102, 201.
number_of_outputs: 201, 210.
number_of_precompute_fields: 94, 96, 97.
number_of_precomputed_fields: 200.
number_of_precomputes: 201, 203, 204, 205, 206, 207, 208, 210.
number_of_processes: 197, 225, 228.
NUMBER_OF_STEPS: 75, 239.
number_of_steps: 75, 76, 80.
number_of_write_fields: 94, 96, 97, 200.
numBoxes: 233.
numComp: 299, 313, 314, 323, 327, 330, 334.
numeric_limits: 215, 237, 269.
numProc: 66, 197, 225, 238.
numPts: 228.
nx: 111, 112, 199.
ny: 111, 112, 199.
nz: 111, 112, 199.
off_t: 213, 240, 272, 292.
ok: 92, 97, 106, 107, 108, 117, 118, 120, 121, 126, 127, 128, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 219, 227, 231, 233, 234.
OK: 231, 239.
old_idx: 198.
ONE: 239.
one_by_delta: 198.
one_by_delta_cube: 198.
one_by_delta_square: 198.
one_by_eps_1: 21.
one_by_eps_2: 21.
open: 213, 292.
OPT: 305.
origin: 84, 89, 90, 93, 109, 129, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 220, 230, 231, 235, 240, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 266.
OUCH: 239.
out: 263, 264.
Out: 96, 97, 200, 201, 203, 204, 205, 206, 207, 208, 210, 266.
out_standard_params: 341.
outp: 265.
output_field_copy: 344.
output_clock: 80.
output_clock1: 80.
output_clock2: 80.
*output_field_copy_*: 200, 210, 265.
OUTPUT_NUMBER_OF_DIGITS: 201, 239.
output_parameter_parser: 94, 96, 201.
OutWrite: 96, 97, 200, 201, 210, 266.
parameter_parser: 74.
parfile: 270, 274.
ParmParse: 3, 6, 7, 267, 268.

pcmp: 265.
perpendicular_to: 341.
PML_M: 230, 239.
PML_MARGIN: 90, 239.
PML_MARKER: 231, 239.
pml_marker: 231.
pml_max: 230, 240.
pml_min: 230, 240.
pml_parameter_parser: 85, 89, 94, 98, 235.
PML_SIGMA_MAX: 235, 239.
PML_XYZ_MAX: 84, 239.
pml_xyz_max: 84, 89, 90, 93, 231, 235, 266.
PML_XYZ_MIN: 84, 239.
pml_xyz_min: 84, 89, 90, 93, 231, 235, 266.
pool_size: 197.
position: 198, 230, 231.
post_all_clock: 80.
post_all_clock1: 80.
post_all_clock2: 80.
post_all_lambda: 80, 199.
post_initialize_scheme: 73, 199, 241.
post_iteration_clock: 80.
post_iteration_clock1: 80.
post_iteration_clock2: 80.
post_iteration_lambda: 80, 199.
post_iteration_parser: 80.
post_parser: 199.
pout: 5, 73, 74, 76, 78, 79, 80, 82, 88, 89, 90, 91, 92, 93, 95, 96, 97, 98, 99, 100, 101, 102, 103, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 135, 136, 137, 139, 140, 141, 142, 143, 145, 146, 147, 148, 149, 151, 152, 153, 154, 155, 157, 158, 159, 160, 161, 163, 164, 165, 166, 167, 169, 170, 171, 172, 173, 175, 176, 177, 178, 179, 181, 182, 183, 184, 185, 187, 188, 189, 190, 191, 193, 194, 197, 198, 199, 201, 203, 204, 205, 206, 210, 212, 213, 218, 219, 220, 221, 226, 227, 228, 229, 231, 232, 233, 234, 235, 238, 269, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 292, 293.
pow: 212, 230.
preAllocatable: 300, 323, 330.
prefix: 270, 272, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290.
print_pml_arrays: 94, 98.
print_prefix: 271, 290.
procID: 66, 73, 197, 227, 228, 233, 238.
PROGRAM_AUTHOR: 237.
PROGRAM_VERSION: 73, 237.
proper_box: 198.
push_back: 90, 93, 201, 234.
put: 271, 278, 279, 280, 281.
Px_idx: 201, 206.
Py_idx: 201, 206.
Pz_idx: 200, 201, 206.
query: 75, 88, 89, 91, 95, 98, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 136, 139, 142, 145, 148, 151, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187, 190, 193, 197, 198, 199, 201, 218, 226, 231, 233, 234, 235, 271, 278, 279, 280, 281.
queryarr: 89, 199, 271, 282, 283, 284, 285, 286, 287, 288, 289.
queryput: 271, 278, 279, 280, 281.
R: 300, 323, 330.
r: 151.
r_length: 151.
r_max: 175, 187.
r_min: 187.
r_tube: 187.
r_1: 171, 173, 249, 251, 252.
r_2: 171, 173, 249, 251, 252.
r_3: 171, 173, 249.
radius: 141, 143, 151, 153, 155, 159, 161, 165, 167, 244, 246, 247, 248.
radius_1: 183, 185, 189, 191, 251, 252.
radius_2: 183, 185, 189, 191, 251, 252.
rank: 197, 277.
RCS: 3.
Real: 83, 84, 97, 106, 107, 109, 111, 112, 113, 117, 118, 120, 121, 122, 123, 124, 126, 127, 129, 132, 133, 135, 139, 140, 141, 145, 146, 147, 151, 152, 153, 157, 158, 159, 163, 164, 165, 169, 170, 171, 175, 176, 177, 181, 182, 183, 187, 188, 189, 193, 194, 197, 198, 199, 200, 201, 216, 230, 231, 235, 240, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 254, 256, 257, 259, 260, 261, 263, 264, 265, 266, 293, 296, 329, 330, 334, 335.
receive: 197.
redefine: 309.
RedgeFrom: 315, 334.
RedgeTo: 315, 334.
ref: 271, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
refine: 93.
refine_base_level: 83, 93.
REFINE_BLOCK_FACTOR: 83, 239.
refine_block_factor: 83, 91, 93.
refine_buffer_size: 83, 91, 93.
REFINE_BUFFER_SIZE: 83, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 239.
refine_fill_ratio: 83, 91, 93.
REFINE_FILL_RATIO: 83, 239.
refine_max_size: 83, 91, 93.
REFINE_MAX_SIZE: 83, 239.
refine_ratio: 83, 93.
REFINE_RATIO: 83, 201, 239.
refine_top_level: 83, 93.

Region: 299, 316, 323, 327, 330, 334.
RegionFrom: 299, 315, 323, 327, 330, 334.
RegionTo: 299, 315, 323, 327, 330, 334.
regrid: 93.
replace: 198.
report_outliers: 201.
reset: 92, 97, 106, 107, 108, 117, 118, 120, 121, 126, 127, 128, 132, 133, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 201, 219, 227, 231, 233, 234.
resize: 90, 93, 181, 220, 233, 235, 282, 283, 284, 285, 286, 287, 288, 289, 297, 308, 309, 323, 326, 330, 333.
return_status: 82, 83, 88, 89, 90, 91, 92, 93, 94, 95, 98, 102, 103, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 135, 137, 138, 139, 141, 143, 144, 145, 147, 149, 150, 151, 153, 155, 156, 157, 159, 161, 162, 163, 165, 167, 168, 169, 171, 173, 174, 175, 177, 179, 180, 181, 183, 185, 186, 187, 189, 191, 192, 193, 199, 201, 203, 204, 205, 206, 207, 208, 210, 214, 221, 222, 229, 233, 234, 235, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 254, 256, 257, 259, 260, 261.
return_value: 231, 232, 282, 283, 284, 285, 286, 287, 288, 289.
RfaceFrom: 327.
RfaceTo: 327.
root: 201.
root_process: 225, 228.
root_process_rank: 197.
rules: 110.
S: 118, 121, 123, 124, 256, 257, 266.
S_dir: 118, 121.
`S_ISREG`: 213, 292.
S_ptr: 123.
S_var: 199.
S_var_ref: 22, 123, 199.
`SCATTERED_FIELD_MARKER`: 231, 239.
scattered_field_marker: 231.
scattering coefficient: 13.
Scheme: 2, 72, 82, 83, 86, 92, 93, 199.
Schrod, Joachim: 1.
SCM: 20, 21, 22, 80, 92, 102, 112, 122, 123, 135, 141, 147, 153, 159, 165, 171, 177, 183, 189, 196, 197, 198, 199, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 271, 274, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 291.
scm_array_dimensions: 277.
scm_array_p: 277.
`SCM_BOOL_F`: 69, 196, 197, 198, 291.
`SCM_BOOL_T`: 197.
scm_boot_guile: 72, 73.
scm_c_array_rank: 277.
scm_c_define: 199.
scm_c_define_gsubr: 20, 199.
scm_c_eval_string: 199, 222.
scm_c_lookup: 21, 22, 92, 102, 196, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 291.
scm_c_primitive_load: 6, 274.
scm_c_vector_length: 197.
scm_c_vector_set_x: 197.
scm_call_0: 80, 92, 102, 122, 123.
scm_call_1: 92, 112.
scm_closure_p: 276.
scm_current_module_lookup_closure: 196, 291.
scm_d_to_e_n: 22.
*scm_d_to_e_n_*: 22, 123, 124.
*scm_d_to_e_0_*: 122, 124.
scm_from: 92.
scm_from_bool: 135, 141, 147, 153, 159, 165, 171, 177, 183, 189.
scm_from_double: 112, 122, 123, 197, 198, 279, 280, 283, 284.
scm_from_int: 92, 122, 123, 137, 143, 149, 155, 161, 167, 173, 179, 185, 191, 197, 199, 277, 278, 282, 283, 284, 285, 286, 287, 288, 289.
scm_from_locale_string: 196, 281, 291.
scm_f64vector: 135, 141, 147, 153, 155, 159, 161.
scm_f64vector_length: 137, 143, 149, 155, 161, 167, 173, 179, 185, 191.
scm_f64vector_p: 137, 143, 149, 155, 161, 167, 173, 179, 185, 191.
scm_f64vector_ref: 137, 143, 149, 155, 161, 167, 173, 179, 185, 191.
scm_gc: 108, 115, 128, 131.
scm_init_guile: 72.
scm_int2num: 92.
scm_is_bool: 80, 92, 102.
scm_is_false: 80, 92, 102, 274.
scm_is_integer: 137, 143, 149, 155, 161, 167, 173, 179, 185, 191, 278, 282, 286.
scm_is_real: 143, 149, 155, 161, 167, 173, 185, 191, 197, 198, 279, 280, 283, 284, 287, 288.
scm_is_string: 198, 281, 285, 289.
scm_is_true: 137, 143, 149, 155, 161, 167, 173, 179, 185, 191, 277, 282, 283, 284, 285, 286, 287, 288, 289.
scm_is_vector: 282, 283, 284, 285, 286, 287, 288, 289.
scm_length: 277, 282, 283, 284, 285, 286, 287, 288, 289.
scm_list_p: 277, 282, 283, 284, 285, 286, 287, 288, 289.
scm_list_ref: 277, 282, 283, 284, 285, 286, 287, 288, 289.
scm_name: 198.
scm_number_p: 277.
scm_procedure_p: 276.

scm_string_p: 277.
scm_string_to_symbol: 196, 291.
scm_str2symbol: 196, 291.
scm_sym2var: 196, 291.
scm_thunk_p: 276.
scm_to: 92.
scm_to_bool: 276.
scm_to_double: 112, 122, 123, 137, 143, 149, 155, 161, 167, 173, 179, 185, 191, 197, 198, 278, 279, 280, 282, 283, 284, 286, 287, 288.
scm_to_int: 137, 143, 149, 155, 161, 167, 173, 179, 185, 191, 197, 277, 278, 282, 283, 284, 285, 286, 287, 288, 289.
scm_to_locale_string: 281, 285, 289.
scm_to_locale_stringbuf: 198.
`SCM_UNBNDP`: 20, 137, 143, 149, 155, 161, 167, 173, 179, 185, 191.
`SCM_UNDEFINED`: 276, 277.
scm_uniform_vector_length: 277, 282, 283, 284, 286, 287, 288.
scm_uniform_vector_p: 277, 282, 283, 284, 286, 287, 288.
scm_uniform_vector_ref: 123, 282, 283, 284, 286, 287, 288.
scm_uniform_vector_set_x: 123, 282, 283, 284.
scm_variable_ref: 21, 92, 102, 199, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
scm_variable_set_x: 21, 122, 123, 278, 279, 280, 281.
scm_vector_length: 277, 282, 283, 284, 285, 286, 287, 288, 289.
scm_vector_p: 277.
scm_vector_ref: 282, 283, 284, 285, 286, 287, 288, 289.
scm_x: 198.
scm_y: 198.
scm_z: 198.
SCMbarrier: 197, 199.
SCMbroadcast: 197, 199.
SCMgather: 197, 199.
SCMinterpolate: 69, 198, 199.
SCMnumProc: 197, 199.
SCMParmParse: 3, 6, 74, 75, 80, 85, 91, 94, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 122, 125, 126, 127, 128, 129, 131, 132, 133, 136, 139, 142, 145, 148, 151, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187, 190, 193, 196, 197, 198, 199, 200, 201, 214, 222, 231, 233, 234, 235, 267, 268, 269, 270, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292.
SCMprocID: 197, 199.
SCMuniqueProc: 197, 199.
sed: 3.
select: 122.
SerialTask: 197, 225.
setVal: 92, 97, 231, 299, 310, 311, 312, 323, 327, 330, 334.
setw: 237, 269.
Shapes: 2.
shift: 299, 317, 323, 327, 330, 334.
Side: 108, 109, 128, 129, 233, 234, 240.
side: 108, 109, 128, 129, 233, 234, 240.
side_boxes: 109, 129.
side_iterator: 108, 128, 233.
side_number: 109, 129, 261.
SideIterator: 108, 128, 233.
sigma: 235.
sigma_max: 230, 235, 240.
signal_ijk_hi: 90, 109, 129, 233, 234, 266.
signal_ijk_lo: 90, 109, 129, 233, 234, 266.
`SIGNAL_MARGIN`: 90, 239.
signal_parameter_parser: 85, 89, 94, 108, 128, 199.
`SIGNAL_XYZ_MAX`: 84, 239.
signal_xyz_max: 84, 89, 90, 93, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 231, 266.
signal_xyz_min: 84, 89, 90, 93, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 231, 266.
`SIGNAL_XYZ_MIN`: 84, 239.
size: 78, 79, 90, 93, 99, 100, 101, 106, 109, 117, 118, 126, 129, 132, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 201, 233, 235, 282, 283, 284, 300, 318, 319, 320, 321, 323, 328, 330, 335.
small_end: 234.
smallEnd: 109, 129, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 219, 234.
source: 197.
source_process_rank: 197.
space_dim: 256, 257, 259, 260.
SpaceDim: 73, 84, 89, 90, 93, 106, 107, 108, 109, 117, 118, 120, 121, 126, 127, 128, 129, 132, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 198, 199, 203, 204, 205, 206, 207, 208, 210, 216, 219, 220, 231, 232, 233, 234, 238, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 254, 297, 301, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 319, 320, 321, 325, 326, 327, 328, 332, 333, 334, 335.
spacedim: 263, 264, 265.
sprintf: 199, 201.
sqrt: 151, 169, 187, 199.
src: 299, 312, 313, 314, 315, 316, 323, 327, 330, 334.
srcComp: 299, 313, 314, 323, 327, 330, 334.
srcFab: 313, 314, 315, 316, 327, 334.
srcMakefile: 3, 345.
srcMakefile, targets: 3.
st_mode: 213, 292.
st_size: 213, 292.

staggeredChombo: 2.
start_ix: 271, 282, 283, 284, 285, 286, 287, 288, 289.
startComp: 299, 310, 311, 323, 327, 330, 334.
stat: 213, 292.
status: 80, 263, 264, 265, 274.
std: 237, 269.
stdlib: 72.
STRIDE: 84, 89, 239.
stride: 25, 75, 84, 89, 90.
string: 198, 201, 237, 269, 271, 272, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
sum: 147, 149, 245.
surroundingNodes: 234, 308, 309, 311, 315, 316, 319, 320, 321, 322, 326, 327, 328, 333, 334, 335.
swap_idx: 106, 107, 117, 118, 120, 121, 126, 127, 132, 133, 236, 240.
symbol: 196, 291.
t: 111, 112, 113, 122, 123, 124.
t_e_var: 22, 80, 122, 123, 199.
tag_box: 227.
tag_set: 93, 223, 228, 266.
tags: 83, 86, 92, 93, 200, 222, 223, 227, 266.
tempFab: 319, 328, 335.
TFR_face_boxes: 109, 129, 233, 234, 266.
the_box: 233, 234.
thickness: 165, 167, 248.
time_e: 80, 90, 93, 105, 117, 118, 120, 121, 129, 201, 256, 257, 261, 266.
time_h: 80, 90, 93, 109, 125, 132, 133, 259, 260, 261, 266.
time_to_dump_data: 80.
time_to_dump_var: 80, 199.
total_clock: 80.
total_clock1: 80.
total_clock2: 80.
total_field_box: 151, 157, 163, 169, 175, 181, 187, 193.
total_field_hi: 151, 157, 163, 169, 175, 181, 187, 193.
total_field_lo: 151, 157, 163, 169, 175, 181, 187, 193.
total_field_side_boxes: 233.
totalSize: 319, 328, 335.
true: 112, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 198, 199, 234, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289.
type: 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 198, 231, 293.
t0: 84, 89, 90, 93.
TO: 84, 239.
U. S. Department of Energy: 1.
unique_process: 197.
uniqueProc: 66, 197, 225.
Unit: 92, 94.
University of California: 1.
update_b: 338.
update_b_upml: 338.
update_d: 337.
update_d_upml: 337.
*update_b_*: 127, 254.
*update_b_upml_*: 126, 254.
*update_d_*: 107, 254.
*update_d_upml_*: 106, 254.
upml_sigma: 230, 235, 240.
upper_corner: 135, 137.
v: 299, 323, 330.
val: 299, 310, 311, 323, 327, 330, 334.
variable: 196, 199, 271, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 291.
variable_ref: 282, 283, 284.
Vector: 72, 78, 82, 84, 87, 94, 106, 109, 117, 118, 126, 129, 132, 135, 139, 140, 141, 145, 146, 147, 151, 152, 153, 157, 158, 159, 163, 164, 165, 169, 170, 171, 175, 176, 177, 181, 182, 183, 187, 188, 189, 193, 194, 197, 198, 199, 200, 201, 216, 225, 231, 233, 234, 235, 240, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 266, 271, 282, 283, 284, 286, 287, 288, 289, 301, 306, 323, 330.
vector: 234, 269, 271, 282, 283, 284, 285, 286.
vector_of_box_layouts: 200, 201.
vector_of_boxes: 87, 90, 93, 266.
vector_of_items: 197.
vector_of_outputs: 200, 201.
vector_of_precomputes: 201, 210.
vector_of_processes: 90, 93, 266.
vector_of_ptrs_to_level_data: 200, 201.
vector_of_refinements: 87, 91, 93, 201.
vector_of_refinments: 93.
vector_of_tag_sets: 87, 91, 93, 225, 228.
vector_of_vectors_of_boxes: 87, 91, 93.
verbosity: 243, 244, 245, 246, 247, 248, 249, 250, 251, 252.
VisIt: 18.
volume: 341.
watch: 75, 263, 264, 265.
watch_advance_b: 125, 126, 127.
watch_advance_d: 105, 106, 107.
watch_b_to_h: 131, 132, 133, 259, 260.
watch_build_levels: 82, 83, 88, 89, 90, 91, 92, 93.
watch_d_to_e: 115, 116, 117, 118, 119, 120, 121, 256, 257.
watch_draw_ary_ellipsoid: 171, 172, 173, 175.
watch_draw_ball: 141, 142, 143, 145.
watch_draw_box: 135, 136, 137, 139.
watch_draw_cone: 159, 160, 161, 163.
watch_draw_cylinder: 153, 154, 155, 157.
watch_draw_disk: 165, 166, 167, 169.
watch_draw_ellipsoid: 147, 148, 149, 151.
watch_draw_lens: 189, 190, 191, 193.

watch_draw_parallelepiped: 177, 178, 179, 181.
watch_draw_torus: 183, 184, 185, 187.
watch_dump_data: 201, 203, 204, 205, 206, 207, 208, 210.
watch_effective_grid_bounds: 214, 218, 219, 220, 221.
watch_fill_levels: 94, 95, 96, 97, 103.
watch_fill_PML_arrays: 235.
watch_initialize_scheme: 199.
watch_inject_b: 128, 129, 261.
watch_inject_d: 108, 109, 261.
watch_main: 75, 76, 78, 79, 80.
watch_make_tag_set: 222, 226, 227, 228, 229.
watch_mark_regions: 231.
watch_mark_TFR_faces: 233, 234.
watch_parameter_parser: 75, 85, 88, 94, 95, 105, 106, 107, 108, 109, 115, 116, 117, 118, 119, 120, 121, 125, 126, 127, 128, 129, 131, 132, 133, 136, 139, 142, 145, 148, 151, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187, 190, 193, 197, 198, 201, 214, 218, 222, 226, 231, 233, 234, 235.
watch_parser: 199.
watch_scheme: 197, 198.
watch_update_b: 126, 127.
watch_update_d: 106, 107.
weave: 1.
width: 230.
write: 92.
WriteAMRHierarchyHDF5: 200, 201.
writeAMRHierarchyHDF5: 200.
x: 111, 112, 113.
x_max: 230, 240.
x_min: 230, 240.
x_0: 261.
`XTRACXXFLAGS`: 3.
`XTRALIBFLAGS`: 3.
xyz_max: 90, 216, 220, 235, 266.
xyz_min: 90, 216, 220, 235, 266.
x0: 198.
`XO`: 84, 239.
y: 111, 112, 113.
y_0: 261.
`YES`: 239.
y0: 198.
z: 111, 112, 113.
z_0: 261.
`ZERO`: 97, 239.
Zero: 90, 96, 139, 145, 151, 157, 163, 169, 175, 181, 187, 193, 201.
zeta: 112.
z0: 198.

List of Refinements

⟨ Auxiliary Scheme Procedures 196 ⟩ Used in chunk 134.
⟨ `Auxiliary.c` 211 ⟩
⟨ Build a Multigrid 78 ⟩ Used in chunk 73.
⟨ `Conversion.c_std` 122, 123 ⟩ Cited in chunk 22.
⟨ `Conversion.h_std` 124 ⟩
⟨ `CurlBox.c` 332, 333, 334, 335 ⟩
⟨ `CurlBox.hw` 330 ⟩
⟨ `EdgeFab.hw` 294 ⟩ Cited in chunk 303.
⟨ `EdgeFabImplem.hw` 303 ⟩
⟨ `FaceFab.hw` 323 ⟩
⟨ `FaceFabImplem.hw` 325, 326, 327, 328 ⟩
⟨ Fill the Multigrid 79 ⟩ Used in chunk 73.
⟨ `Forms.hw` 237, 238, 239, 240, 241, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 254, 256, 257, 259, 260, 261, 263, 264, 265 ⟩
⟨ Function *advance_b_0* 126 ⟩ Used in chunk 104.
⟨ Function *advance_b_n* 127 ⟩ Used in chunk 104.
⟨ Function *advance_b* 125 ⟩ Used in chunk 104.
⟨ Function *advance_d_0* 106 ⟩ Used in chunk 104.
⟨ Function *advance_d_n* 107 ⟩ Used in chunk 104.
⟨ Function *advance_d* 105 ⟩ Used in chunk 104.
⟨ Function *b_to_h_0* 132 ⟩ Used in chunk 104.
⟨ Function *b_to_h_n* 133 ⟩ Used in chunk 104.
⟨ Function *b_to_h* 131 ⟩ Used in chunk 104.
⟨ Function *build_levels* 82 ⟩ Cited in chunk 73. Used in chunk 81.
⟨ Function *complement* 232 ⟩ Used in chunk 211.
⟨ Function *d_to_e_0_0* 117 ⟩ Used in chunk 104.
⟨ Function *d_to_e_0_n* 118 ⟩ Cited in chunk 22. Used in chunk 104.
⟨ Function *d_to_e_0* 116 ⟩ Cited in chunk 22. Used in chunk 104.
⟨ Function *d_to_e_n_0* 120 ⟩ Used in chunk 104.
⟨ Function *d_to_e_n_n* 121 ⟩ Used in chunk 104.
⟨ Function *d_to_e_n* 119 ⟩ Used in chunk 104.
⟨ Function *d_to_e* 115 ⟩ Used in chunk 104.
⟨ Function *draw_ary_ellipsoid* 171 ⟩ Used in chunk 134.
⟨ Function *draw_ball* 141 ⟩ Used in chunk 134.
⟨ Function *draw_box* 135 ⟩ Cited in chunk 311. Used in chunk 134.
⟨ Function *draw_cone* 159 ⟩ Cited in chunk 163. Used in chunk 134.
⟨ Function *draw_cylinder* 153 ⟩ Cited in chunk 157. Used in chunk 134.
⟨ Function *draw_disk* 165 ⟩ Used in chunk 134.
⟨ Function *draw_ellipsoid* 147 ⟩ Used in chunk 134.
⟨ Function *draw_lens* 189 ⟩ Used in chunk 134.
⟨ Function *draw_media_ary_ellipsoid* 175 ⟩ Used in chunk 171.
⟨ Function *draw_media_ball* 145 ⟩ Used in chunk 141.
⟨ Function *draw_media_box* 139 ⟩ Used in chunk 135.
⟨ Function *draw_media_cone* 163 ⟩ Used in chunk 159.
⟨ Function *draw_media_cylinder* 157 ⟩ Cited in chunk 151. Used in chunk 153.
⟨ Function *draw_media_disk* 169 ⟩ Used in chunk 165.
⟨ Function *draw_media_ellipsoid* 151 ⟩ Used in chunk 147.
⟨ Function *draw_media_lens* 193 ⟩ Used in chunk 189.
⟨ Function *draw_media_parallelepiped* 181 ⟩ Used in chunk 177.
⟨ Function *draw_media_torus* 187 ⟩ Used in chunk 183.
⟨ Function *draw_parallelepiped* 177 ⟩ Used in chunk 134.
⟨ Function *draw_tags_ary_ellipsoid* 176 ⟩ Used in chunk 171.
⟨ Function *draw_tags_ball* 146 ⟩ Used in chunk 141.
⟨ Function *draw_tags_box* 140 ⟩ Used in chunk 135.

⟨ Function *draw_tags_cone* 164 ⟩ Used in chunk 159.
⟨ Function *draw_tags_cylinder* 158 ⟩ Used in chunk 153.
⟨ Function *draw_tags_disk* 170 ⟩ Used in chunk 165.
⟨ Function *draw_tags_ellipsoid* 152 ⟩ Used in chunk 147.
⟨ Function *draw_tags_lens* 194 ⟩ Used in chunk 189.
⟨ Function *draw_tags_parallelepiped* 182 ⟩ Used in chunk 177.
⟨ Function *draw_tags_torus* 188 ⟩ Used in chunk 183.
⟨ Function *draw_torus* 183 ⟩ Used in chunk 134.
⟨ Function *dump_data* 201 ⟩ Cited in chunk 73. Used in chunk 200.
⟨ Function *effective_grid_bounds* 214, 215, 216, 217, 218, 219, 220, 221 ⟩ Cited in chunks 90 and 222. Used in chunk 211.
⟨ Function *fill_PML_arrays* 235 ⟩ Used in chunk 211.
⟨ Function *fill_levels* 94, 95, 96, 97, 98, 102, 103 ⟩ Cited in chunks 22 and 200. Used in chunk 81.
⟨ Function *inject_b* 128, 129 ⟩ Used in chunk 104.
⟨ Function *inject_d* 108, 109 ⟩ Used in chunk 104.
⟨ Function *is_a_pwr_of_two* 212 ⟩ Used in chunk 211.
⟨ Function *is_file_readable* 213 ⟩ Used in chunk 211.
⟨ Function *make_tag_set* 222, 223, 224, 225, 226, 227, 228, 229 ⟩ Cited in chunk 93. Used in chunk 211.
⟨ Function *mark_TFR_faces* 233, 234 ⟩ Used in chunk 211.
⟨ Function *mark_regions* 231 ⟩ Used in chunk 211.
⟨ Function *swap_idx* 236 ⟩ Used in chunk 211.
⟨ Function *upml_sigma* 230 ⟩ Used in chunk 211.
⟨ `IO.c` 200 ⟩
⟨ Initialize Scheme 199 ⟩ Cited in chunks 22 and 66. Used in chunk 134.
⟨ `Initialize.c` 81 ⟩
⟨ `Injection.c_std` 111, 112 ⟩
⟨ `Injection.h_std` 113 ⟩
⟨ Inner Main 73 ⟩ Cited in chunks 84 and 199. Used in chunk 72.
⟨ Interpolation 198 ⟩ Cited in chunks 13 and 41. Used in chunk 134.
⟨ Iterate 80 ⟩ Cited in chunk 69. Used in chunk 73.
⟨ `Iteration.c` 104 ⟩
⟨ MPI Functions 197 ⟩ Cited in chunks 13 and 66. Used in chunk 134.
⟨ Process an Input File 74, 75, 76 ⟩ Used in chunk 73.
⟨ `SCMParmParse.c` 273 ⟩
⟨ `SCMParmParse.hw` 269 ⟩
⟨ `Scheme.c` 134 ⟩
⟨ Structure level 266 ⟩ Cited in chunk 22. Used in chunk 240.
⟨ print *Ckx* arrays 99 ⟩ Used in chunk 98.
⟨ print *Cky* arrays 100 ⟩ Used in chunk 98.
⟨ print *Ckz* arrays 101 ⟩ Used in chunk 98.
⟨ **EdgeFab** template implementation 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 319, 320, 321 ⟩ Used in chunk 303.
⟨ **EdgeFab**: class template declaration 295 ⟩ Used in chunk 294.
⟨ **EdgeFab**: private members 302 ⟩ Used in chunk 295.
⟨ **EdgeFab**: protected members 301 ⟩ Used in chunk 295.
⟨ **EdgeFab**: public members 296, 297, 298, 299, 300 ⟩ Used in chunk 295.
⟨ **SCMParmParse** headers: constructors and destructors 270 ⟩ Used in chunk 269.
⟨ **SCMParmParse** headers: protected members 272 ⟩ Used in chunk 269.
⟨ **SCMParmParse** headers: queries and extractors 271 ⟩ Used in chunk 269.
⟨ **SCMParmParse** implementation: constructors and destructors 274 ⟩ Cited in chunk 6. Used in chunk 273.
⟨ **SCMParmParse** implementation: protected members 291, 292 ⟩ Used in chunk 273.
⟨ **SCMParmParse** implementation: queries and extractors 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290 ⟩ Used in chunk 273.
⟨ *build_levels*: build higher levels 91 ⟩ Used in chunk 82.
⟨ *build_levels*: build level 0 90 ⟩ Cited in chunks 23 and 89. Used in chunk 82.
⟨ *build_levels*: build next level 92, 93 ⟩ Cited in chunk 91. Used in chunk 91.

⟨ *build_levels*: read grid specifications 88, 89 ⟩ Used in chunk 82.
⟨ *build_levels*: variable definitions 83, 84, 85, 86, 87 ⟩ Cited in chunk 91. Used in chunk 82.
⟨ *draw_ary_ellipsoid*: call appropriate utility 174 ⟩ Used in chunk 171.
⟨ *draw_ary_ellipsoid*: introduce yourself 172 ⟩ Used in chunk 171.
⟨ *draw_ary_ellipsoid*: read, check, and translate arguments 173 ⟩ Used in chunk 171.
⟨ *draw_ball*: call appropriate utility 144 ⟩ Used in chunk 141.
⟨ *draw_ball*: introduce yourself 142 ⟩ Used in chunk 141.
⟨ *draw_ball*: read, check, and translate arguments 143 ⟩ Used in chunk 141.
⟨ *draw_box*: call appropriate utility 138 ⟩ Used in chunk 135.
⟨ *draw_box*: introduce yourself 136 ⟩ Used in chunk 135.
⟨ *draw_box*: read, check, and translate arguments 137 ⟩ Used in chunk 135.
⟨ *draw_cone*: call appropriate utility 162 ⟩ Used in chunk 159.
⟨ *draw_cone*: introduce yourself 160 ⟩ Used in chunk 159.
⟨ *draw_cone*: read, check, and translate arguments 161 ⟩ Used in chunk 159.
⟨ *draw_cylinder*: call appropriate utility 156 ⟩ Used in chunk 153.
⟨ *draw_cylinder*: introduce yourself 154 ⟩ Used in chunk 153.
⟨ *draw_cylinder*: read, check, and translate arguments 155 ⟩ Used in chunk 153.
⟨ *draw_disk*: call appropriate utility 168 ⟩ Used in chunk 165.
⟨ *draw_disk*: introduce yourself 166 ⟩ Used in chunk 165.
⟨ *draw_disk*: read, check, and translate arguments 167 ⟩ Used in chunk 165.
⟨ *draw_ellipsoid*: call appropriate utility 150 ⟩ Used in chunk 147.
⟨ *draw_ellipsoid*: introduce yourself 148 ⟩ Used in chunk 147.
⟨ *draw_ellipsoid*: read, check, and translate arguments 149 ⟩ Used in chunk 147.
⟨ *draw_lens*: call appropriate utility 192 ⟩ Used in chunk 189.
⟨ *draw_lens*: introduce yourself 190 ⟩ Used in chunk 189.
⟨ *draw_lens*: read, check, and translate arguments 191 ⟩ Used in chunk 189.
⟨ *draw_parallelepiped*: call appropriate utility 180 ⟩ Used in chunk 177.
⟨ *draw_parallelepiped*: introduce yourself 178 ⟩ Used in chunk 177.
⟨ *draw_parallelepiped*: read, check, and translate arguments 179 ⟩ Used in chunk 177.
⟨ *draw_torus*: call appropriate utility 186 ⟩ Used in chunk 183.
⟨ *draw_torus*: introduce yourself 184 ⟩ Used in chunk 183.
⟨ *draw_torus*: read, check, and translate arguments 185 ⟩ Used in chunk 183.
⟨ *dump_data*: extract field from precomputes 210 ⟩ Used in chunk 201.
⟨ *dump_data*: precompute field 203, 204, 205, 206, 207, 208, 209 ⟩ Used in chunk 201.