

User Guide*

Zdzisław Meglicki
Indiana University

March 14, 2006

Version \$Id: UG.tex,v 1.76 2006/03/14 23:12:36 meglicki Exp \$

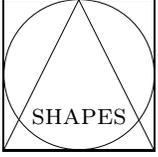
Contents

1	What Is SHAPES	3
2	How SHAPES Works	3
2.1	Maxwell Equations	3
2.2	Solving the Media Equation	5
2.3	Absorbing Boundary Conditions	7
2.4	Signal Injection and Discretization	9
2.5	Spectral Response	16
2.6	Multigrid	16
2.6.1	Multigrid Time Step	18
2.6.2	Building Higher Levels	22
2.7	Parallelization	23
2.8	Output	24
3	How to Use SHAPES	27
3.1	A Simple Jazz Example	27
3.2	A Simple TeraGrid Example	34
3.3	The SHAPES Input File	38
3.3.1	The <i>chat</i> and <i>watch</i> Groups	38
3.3.2	The <i>level0</i> Group	40
3.3.3	The <i>iterate</i> Group	41
3.3.4	The <i>pml</i> Group	43
3.3.5	The <i>signal</i> Group	44
3.3.6	The <i>metal</i> Group	46
3.3.7	The <i>tag</i> Group	59
3.3.8	The <i>refine</i> Group	63
3.3.9	The <i>spectral</i> Group	64
3.3.10	The <i>output</i> Group	64

*SHAPES was developed for the Argonne National Laboratory under a contract with Indiana University

4	Parallel Execution	66
4.1	TeraGrid example	66
4.2	Jazz Example	69
4.3	Working with ChomboVis on Jazz	72
5	Working with Multigrid	78
5.1	A Simple Example without Media	78
5.2	A Classic Example in Parallel	86
5.3	The Cost of High Resolution	97
5.4	Variations	99
	References	103
	Index	105

1 What Is SHAPES



is a finite difference time domain (FDTD) program that simulates scattering of electromagnetic waves on objects of various shapes made of various substances. The program provides a small number of primitive shapes from which more complex shapes can be easily assembled—hence the name.

SHAPES can be run sequentially or in parallel on a cluster. Additionally, SHAPES can perform computations on a multigrid, allowing users to zoom on regions of special interest.

SHAPES produces two types of output: diagnostics that are written on standard output or, in case of parallel execution, on special files associated with MPI processes, and field images that are written on human readable text files or on HDF5 files. The text files are formatted for display with Gnuplot. The HDF5 files are generated in parallel on a parallel file system and images stored on them viewed with a special utility called `ChomboVis`.

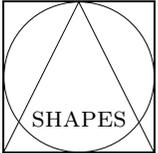
The field images produced by SHAPES also can be assembled into animations.

The SHAPES code comprises two components that interact with each other: a C++ Chombo [1] [2] shell that manages data flow, time stepping, parallelization, multigrid and I/O, and Fortran-77 subroutines that express raw computations and basic algorithms.

Chombo is an object-oriented toolkit for adaptive mesh refinement (AMR) (i.e., flowing multigrid) programming. With SHAPES the flowing of the AMR multigrid is seldom needed, but it is provided. The various objects on which electromagnetic waves scatter in SHAPES simulations do not move. It is therefore more efficient to wrap the objects in static multigrids. The Chombo toolkit works either way.

Currently SHAPES performs its computations in two dimensions only, but three-dimensional simulations will be possible in the near future.

2 How SHAPES Works



uses a number of simple algorithms that are well known in the FDTD community [6], most notably FDTD itself, perfectly matched layer (PML) absorbing boundary conditions (ABCs), total/scattered field regions, and auxiliary differential equations (ADEs) that are used to evaluate material response.

The computations are carried out in *natural units*, defined by setting the speed of light to 1. Both ϵ_0 and μ_0 are absorbed into redefined \mathbf{D} and \mathbf{E} (see Section 2.1). This approach greatly simplifies Maxwell equations, improves computational accuracy (everything is always close to 1), speeds the code's performance, and facilitates interpretation of the results. Input is likewise expected in natural units, and output is written in natural units as well. It is easy to convert data between natural and conventional (for example, SI, CGS) units, and a detailed procedure illustrated by specific examples is discussed in Section 3.

2.1 Maxwell Equations

The specific form of Maxwell equations solved by SHAPES is as follows.

$$\partial_t \mathbf{D} = \nabla \times \mathbf{H}$$

$$\begin{aligned}\mathbf{D}(\omega) &= \epsilon_0 \left(\epsilon_\infty + \sum_k \frac{\epsilon_k}{\alpha_k + i2\delta_k(\omega/\omega_k) - (\omega/\omega_k)^2} \right) \mathbf{E}(\omega) \\ \partial_t \mathbf{H} &= -\frac{1}{\mu_0} \nabla \times \mathbf{E}\end{aligned}$$

First we absorb ϵ_0 and μ_0 into \mathbf{D} and \mathbf{E} ,

$$\begin{aligned}\tilde{\mathbf{E}} &= \sqrt{\frac{\epsilon_0}{\mu_0}} \mathbf{E} \\ \tilde{\mathbf{D}} &= \sqrt{\frac{1}{\epsilon_0 \mu_0}} \mathbf{D} \\ c &= \sqrt{\frac{1}{\epsilon_0 \mu_0}},\end{aligned}$$

where c is the speed of light in a vacuum. The resulting Maxwell equations are now

$$\begin{aligned}\partial_t \tilde{\mathbf{D}} &= c \nabla \times \mathbf{H} \\ \tilde{\mathbf{D}}(\omega) &= \kappa(\omega) \tilde{\mathbf{E}}(\omega) \\ \partial_t \mathbf{H} &= -c \nabla \times \tilde{\mathbf{E}},\end{aligned}$$

where

$$\kappa(\omega) = \epsilon_\infty + \sum_k \frac{\epsilon_k}{\alpha_k + i2\delta_k(\omega/\omega_k) - (\omega/\omega_k)^2}.$$

Dropping the tildes and using $c = 1$, we have

$$\begin{aligned}\partial_t \mathbf{D} &= \nabla \times \mathbf{H} \\ \mathbf{D}(\omega) &= \kappa(\omega) \mathbf{E}(\omega) \\ \partial_t \mathbf{H} &= -\nabla \times \mathbf{E}.\end{aligned}$$

The expression for $\kappa(\omega)$ covers simultaneously the Drude and multiple-resonance Lorentz models. Indeed, the substitutions

$$\begin{aligned}\alpha_k &\leftarrow 0 \\ \delta_k &\leftarrow \frac{\Gamma_D}{2\omega_D} \\ \epsilon_k &\leftarrow 1 \\ \epsilon_\infty &\leftarrow 1\end{aligned}$$

yield the Drude term

$$1 + \frac{\omega_D^2}{-\omega^2 + i\Gamma_D \omega}.$$

The Γ_D coefficient is also written as $1/\tau_D$, where τ_D is the collision time.

The combined Drude/Lorentz model is quite general. It covers substances such as unmagnetized plasma, metals, dielectrics, human muscle tissue, and other materials that have simultaneously dielectric and conducting properties. At the same time, though, this material model is still linear, and as such it is not suitable for modeling of *active* photonic devices, such as switches or amplifiers.

SHAPES lets the user define an arbitrary number of media using an arbitrary number of resonances per medium. The media types can then be assigned to an arbitrary number of figures of various shapes.

We note that α_k , δ_k and ϵ_k are all dimensionless quantities, as is ϵ_∞ . Hence they can be evaluated in any system of units, and the numbers can simply be plugged into the SHAPES input file. No conversions are needed. The only quantity that has to be converted to natural units is ω_k , which must be re-expressed in terms of the inverse of the natural unit of time.

2.2 Solving the Media Equation

The media equation

$$\mathbf{D}(\omega) = \left(\epsilon_\infty + \sum_k \frac{\epsilon_k}{\alpha_k + i2\delta_k(\omega/\omega_k) - (\omega/\omega_k)^2} \right) \mathbf{E}(\omega) \quad (1)$$

can be solved by using the auxiliary differential equation (ADE) method. Let us rewrite the above equation as

$$\mathbf{D}(\omega) = \epsilon_\infty \mathbf{E}(\omega) + \sum_k \mathbf{S}_k(\omega), \quad (2)$$

where

$$\mathbf{S}_k(\omega) = \frac{\epsilon_k}{\alpha_k + i2\delta_k(\omega/\omega_k) - (\omega/\omega_k)^2} \mathbf{E}(\omega). \quad (3)$$

Suppose $\mathbf{S}_k(\mathbf{r}, t) = e^{i\omega t} \mathbf{S}_{k0}(\mathbf{r})$. Then

$$\frac{\partial}{\partial t} \mathbf{S}_k(\mathbf{r}, t) = i\omega \mathbf{S}_k(\mathbf{r}, t)$$

and

$$\frac{\partial^2}{\partial t^2} \mathbf{S}_k(\mathbf{r}, t) = (i\omega)^2 \mathbf{S}_k(\mathbf{r}, t).$$

Equation (3) can then be read as

$$\frac{\partial^2}{\partial t^2} \mathbf{S}_k(\mathbf{r}, t) + 2\omega_k \delta_k \frac{\partial}{\partial t} \mathbf{S}_k(\mathbf{r}, t) + \alpha_k \omega_k^2 \mathbf{S}_k(\mathbf{r}, t) = \epsilon_k \omega_k^2 \mathbf{E}(\mathbf{r}, t). \quad (4)$$

Since we don't have any space derivatives here, we can treat this equation as an ordinary differential equation at each point \mathbf{r} and drop the latter from the notation for simplicity. The partial derivatives then become ordinary derivatives, d/dt and d^2/dt^2 . We introduce the following finite difference approximations for the derivatives at $t = t^{(n)}$:

$$\frac{d^2}{dt^2} \mathbf{S}_k(t) \approx \frac{\frac{\mathbf{S}_k^{(n+1)} - \mathbf{S}_k^{(n)}}{t^{(n+1)} - t^{(n)}} - \frac{\mathbf{S}_k^{(n)} - \mathbf{S}_k^{(n-1)}}{t^{(n)} - t^{(n-1)}}}{t^{(n+1/2)} - t^{(n-1/2)}}.$$

Assuming that $t^{(n+1)} - t^{(n)} = t^{(n)} - t^{(n-1)} = \Delta t$ and $t^{(n+1/2)} - t^{(n-1/2)} = \Delta t$, we get

$$\frac{d^2}{dt^2} \mathbf{S}_k(t) \approx \frac{\mathbf{S}_k^{(n+1)} - 2\mathbf{S}_k^{(n)} + \mathbf{S}_k^{(n-1)}}{\Delta t^2}.$$

We also stretch the first derivative over the $[t^{(n+1)}, t^{(n-1)}]$ segment as follows:

$$\frac{d}{dt} \mathbf{S}_k(t) \approx \frac{\mathbf{S}_k^{(n+1)} - \mathbf{S}_k^{(n-1)}}{2\Delta t},$$

which yields the following finite difference approximation of equation (4):

$$\frac{\mathbf{S}_k^{(n+1)} - 2\mathbf{S}_k^{(n)} + \mathbf{S}_k^{(n-1)}}{\Delta t^2} + 2\omega_k \delta_k \frac{\mathbf{S}_k^{(n+1)} - \mathbf{S}_k^{(n-1)}}{2\Delta t} + \alpha_k \omega_k^2 \mathbf{S}_k^{(n)} = \epsilon_k \omega_k^2 \mathbf{E}^{(n)}.$$

Solving this for $\mathbf{S}_k^{(n+1)}$ yields

$$\mathbf{S}_k^{(n+1)} \left(\frac{1}{\Delta t^2} + \frac{\omega_k \delta_k}{\Delta t} \right) = \mathbf{S}_k^{(n)} \left(\frac{2}{\Delta t^2} - \alpha_k \omega_k^2 \right) + \mathbf{S}_k^{(n-1)} \left(\frac{\omega_k \delta_k}{\Delta t} - \frac{1}{\Delta t^2} \right) + \epsilon_k \omega_k^2 \mathbf{E}^{(n)}.$$

We multiply both sides by Δt^2 :

$$\mathbf{S}_k^{(n+1)} (1 + \omega_k \delta_k \Delta t) = \mathbf{S}_k^{(n)} (2 - \alpha_k \omega_k^2 \Delta t^2) - \mathbf{S}_k^{(n-1)} (1 - \omega_k \delta_k \Delta t) + \mathbf{E}^{(n)} \epsilon_k \omega_k^2 \Delta t^2.$$

Finally we divide both sides by $1 + \omega_k \delta_k \Delta t$ and get

$$\mathbf{S}_k^{(n+1)} = \frac{2 - \alpha_k \omega_k^2 \Delta t^2}{1 + \omega_k \delta_k \Delta t} \mathbf{S}_k^{(n)} - \frac{1 - \omega_k \delta_k \Delta t}{1 + \omega_k \delta_k \Delta t} \mathbf{S}_k^{(n-1)} + \frac{\epsilon_k \omega_k^2 \Delta t^2}{1 + \omega_k \delta_k \Delta t} \mathbf{E}^{(n)}.$$

Renumbering this equation down by one notch, namely, $n+1 \rightarrow n$, $n \rightarrow n-1$, and $n-1 \rightarrow n-2$, yields

$$\mathbf{S}_k^{(n)} = \frac{2 - \alpha_k \omega_k^2 \Delta t^2}{1 + \omega_k \delta_k \Delta t} \mathbf{S}_k^{(n-1)} - \frac{1 - \omega_k \delta_k \Delta t}{1 + \omega_k \delta_k \Delta t} \mathbf{S}_k^{(n-2)} + \frac{\epsilon_k \omega_k^2 \Delta t^2}{1 + \omega_k \delta_k \Delta t} \mathbf{E}^{(n-1)}. \quad (5)$$

Returning to the original equation (2)

$$\mathbf{D}(\omega) = \epsilon_\infty \mathbf{E}(\omega) + \sum_k \mathbf{S}_k(\omega)$$

and using the linearity of the Fourier transform, we can write

$$\mathbf{D}(t) = \epsilon_\infty \mathbf{E}(t) + \sum_k \mathbf{S}_k(t)$$

or

$$\mathbf{E}(t) = \frac{1}{\epsilon_\infty} \left(\mathbf{D}(t) - \sum_k \mathbf{S}_k(t) \right)$$

or, in the discretized form,

$$\mathbf{E}^{(n)} = \frac{1}{\epsilon_\infty} \left(\mathbf{D}^{(n)} - \sum_k \mathbf{S}_k^{(n)} \right), \quad (6)$$

where for each k , $\mathbf{S}_k^{(n)}$ can be evaluated from $\mathbf{S}_k^{(n-1)}$, $\mathbf{S}_k^{(n-2)}$ and $\mathbf{E}^{(n-1)}$ according to equation (5).

The following simple procedure implements this formula within each time step. Apart from \mathbf{E} and \mathbf{S}_k we also need \mathbf{E}_{old} and $\mathbf{S}_{k\text{old}}$ for each k .

$\mathbf{E}_{\text{old}} \leftarrow \mathbf{E}$

$\mathbf{E} \leftarrow \mathbf{D}$

where media is present

repeat for $k = 1$ to *number of resonances*

hold $\leftarrow \mathbf{S}_k$

$\mathbf{S}_k \leftarrow \frac{2 - \alpha_k \omega_k^2 \Delta t^2}{1 + \omega_k \delta_k \Delta t} \mathbf{S}_k - \frac{1 - \omega_k \delta_k \Delta t}{1 + \omega_k \delta_k \Delta t} \mathbf{S}_{k\text{old}} + \frac{\epsilon_k \omega_k^2 \Delta t^2}{1 + \omega_k \delta_k \Delta t} \mathbf{E}_{\text{old}}$

$\mathbf{S}_{k\text{old}} \leftarrow \text{hold}$

$\mathbf{E} \leftarrow \mathbf{E} - \mathbf{S}_k$

end repeat

$\mathbf{E} \leftarrow \mathbf{E} / \epsilon_\infty$

end where

Observe that the number of \mathbf{S}_k and $\mathbf{S}_{k\text{old}}$ fields required is equal to the number of resonances. For example, a formula with three resonances will require six \mathbf{S} fields including the “old” ones. But multiple media can be covered by the same set of \mathbf{S} fields; that is, we do not need to proliferate them if we have multiple media. All that needs to be done is to let the α_k , δ_k , ϵ_k , ω_k , and ϵ_∞ coefficients vary with \mathbf{r} . The C++ Chombo shell of SHAPES dynamically allocates all \mathbf{S} fields and appropriate arrays of media coefficients after reading the input file.

Using ADEs has its limitations. The Courant Friedrichs Lewy (CFL) stability criterion that applies to vacuum Maxwell equations may not necessarily apply to ADEs. Furthermore, even if we can avoid an ADE blowup, we may still introduce inaccuracy into the ADE evaluation by taking too long a time step.

2.3 Absorbing Boundary Conditions

The (E_x, E_y, H_z) mode Maxwell equations in natural units and in the frequency domain look as follows.

$$\begin{aligned} i\omega D_x &= \partial_y H_z \\ i\omega D_y &= -\partial_x H_z \\ D_x &= \epsilon E_x \\ D_y &= \epsilon E_y \\ i\omega H_z &= \partial_y E_x - \partial_x E_y \end{aligned}$$

To absorb an incident signal within a thin boundary layer, we introduce three functions of position, β_x , β_y , and α_z , inserting them in the frequency domain equations

$$\begin{aligned} i\omega D_x \beta_x(x) \beta_y(y) &= \partial_y H_z \\ i\omega D_y \beta_y(x) \beta_x(y) &= -\partial_x H_z \\ D_x &= \epsilon E_x \\ D_y &= \epsilon E_y \\ i\omega H_z \alpha_z(x) \alpha_z(y) &= \partial_y E_x - \partial_x E_y \end{aligned}$$

Functions β_x , β_y , and α_z form perfectly matched layer (PML) absorbing boundary conditions (ABCs) when [3] [5]

$$\begin{aligned} \beta_x &= 1/\beta_y \\ \alpha_z &= \beta_y. \end{aligned}$$

Without much loss in generality we can assume the following form for $\alpha_z = \beta_y = 1/\beta_x$ [4] [5]:

$$\begin{aligned} \alpha_z &= 1 + \frac{\sigma}{i\omega} \\ \beta_x &= \frac{1}{1 + \frac{\sigma}{i\omega}} \\ \beta_y &= 1 + \frac{\sigma}{i\omega}, \end{aligned}$$

where σ is a function of depth *into* the PML layer from within the computational domain. It is equal to zero within the computational domain.

With these in place our frequency domain Maxwell equations become

$$i\omega D_x \left(1 + \frac{\sigma(y)}{i\omega}\right) = \left(1 + \frac{\sigma(x)}{i\omega}\right) \partial_y H_z$$

$$\begin{aligned}
i\omega D_y \left(1 + \frac{\sigma(x)}{i\omega}\right) &= - \left(1 + \frac{\sigma(y)}{i\omega}\right) \partial_x H_z \\
D_x &= \epsilon E_x \\
D_y &= \epsilon E_y \\
i\omega H_z \left(1 + \frac{\sigma(x)}{i\omega}\right) \left(1 + \frac{\sigma(y)}{i\omega}\right) &= \partial_y E_x - \partial_x E_y.
\end{aligned}$$

Converting from the frequency to the time domain yields

$$\begin{aligned}
\partial_t D_x + \sigma(y) D_x &= \partial_y H_z + \sigma(x) \int_0^t \partial_y H_z dt' \\
\partial_t D_y + \sigma(x) D_y &= -\partial_x H_z - \sigma(y) \int_0^t \partial_x H_z dt' \\
D_x &= \epsilon E_x \\
D_y &= \epsilon E_y \\
\partial_t H_z + \sigma(x) H_z + \sigma(y) H_z &= \partial_y E_x - \partial_x E_y - \sigma(x)\sigma(y) \int_0^t H_z dt'.
\end{aligned}$$

The product of two sigmas, $\sigma(x)\sigma(y)$, vanishes everywhere with the exception of the corners, where *both* $\sigma(x)$ and $\sigma(y)$ are different from zero. Furthermore, for an oscillating H_z the integral $\int_0^t H_z dt'$ is going to be zero on average. Consequently, we neglect this term altogether in these computations, so that the last equation simplifies to \diamond

$$\partial_t H_z + \sigma(x) H_z + \sigma(y) H_z = \partial_y E_x - \partial_x E_y.$$

In leap-frog discretization of these time-domain equations, we evaluate non differentiated terms on the left-hand side at the same time slice as the derivatives and as the right-hand side. This approach results in the replacement of, for example, $D_x^{(n)}$ with $(D_x^{(n)} + D_x^{(n-1)})/2$ —with the following effect:

$$\begin{aligned}
D_x^{(n)} &= D_x^{(n-1)} \frac{1 - \sigma(y)\Delta t/2}{1 + \sigma(y)\Delta t/2} + \frac{\Delta t}{1 + \sigma(y)\Delta t/2} \left(\partial_y H_z^{(n-1)} + \frac{\sigma(x)\Delta t}{2} \sum_{k=0}^{n-1} 2\partial_y H_z^{(k)} \right) \\
D_y^{(n)} &= D_y^{(n-1)} \frac{1 - \sigma(x)\Delta t/2}{1 + \sigma(x)\Delta t/2} - \frac{\Delta t}{1 + \sigma(x)\Delta t/2} \left(\partial_x H_z^{(n-1)} + \frac{\sigma(y)\Delta t}{2} \sum_{k=0}^{n-1} 2\partial_x H_z^{(k)} \right) \\
D_x^{(n)} &= \epsilon E_x^{(n)} \\
D_y^{(n)} &= \epsilon E_y^{(n)} \\
H_z^{(n)} &= H_z^{(n-1)} \frac{1 - \sigma(x)\Delta t/2 - \sigma(y)\Delta t/2}{1 + \sigma(x)\Delta t/2 + \sigma(y)\Delta t/2} + \frac{\Delta t}{1 + \sigma(x)\Delta t/2 + \sigma(y)\Delta t/2} \left(\partial_y E_x^{(n)} - \partial_x E_y^{(n)} \right).
\end{aligned}$$

The last equation, for $H_z^{(n)}$, can be rewritten as

$$H_z^{(n)} = H_z^{(n-1)} \left(\frac{1 - \sigma(x)\Delta t/2}{1 + \sigma(x)\Delta t/2} \right) \left(\frac{1 - \sigma(y)\Delta t/2}{1 + \sigma(y)\Delta t/2} \right) + \Delta t \left(\frac{1}{1 + \sigma(x)\Delta t/2} \right) \left(\frac{1}{1 + \sigma(y)\Delta t/2} \right) \left(\partial_y E_x^{(n)} - \partial_x E_y^{(n)} \right)$$

within $\mathcal{O}(\Delta t)^2$ accuracy.

Following [5], we replace $\sigma(x)\Delta t/2$ with

$$\frac{\sigma(x)\Delta t}{2} = f(x) = \frac{1}{3} \left(\frac{\text{depth into the PML layer}}{\text{width of the PML layer}} \right)^3$$

so that the PML equations are

$$D_x^{(n)} = D_x^{(n-1)} \frac{1-f(y)}{1+f(y)} + \frac{1}{1+f(y)} \left(\partial_y H_z^{(n-1)} + f(x) \sum_{k=0}^{n-1} 2\partial_y H_z^{(k)} \right) \Delta t \quad (7)$$

$$D_y^{(n)} = D_y^{(n-1)} \frac{1-f(x)}{1+f(x)} - \frac{1}{1+f(x)} \left(\partial_x H_z^{(n-1)} + f(y) \sum_{k=0}^{n-1} 2\partial_x H_z^{(k)} \right) \Delta t \quad (8)$$

$$D_x^{(n)} = \epsilon E_x^{(n)} \quad (9)$$

$$D_y^{(n)} = \epsilon E_y^{(n)} \quad (10)$$

$$H_z^{(n)} = H_z^{(n-1)} \left(\frac{1-f(x)}{1+f(x)} \right) \left(\frac{1-f(y)}{1+f(y)} \right) + \left(\frac{1}{1+f(x)} \right) \left(\frac{1}{1+f(y)} \right) \left(\partial_y E_x^{(n)} - \partial_x E_y^{(n)} \right) \Delta t. \quad (11)$$

SHAPES implements $f(x)$ functionally, not as a table. The two accumulation terms $\sum_{k=0}^{n-1} 2\partial_y H_z^{(k)}$ and $\sum_{k=0}^{n-1} 2\partial_x H_z^{(k)}$ are implemented as additional fields, but these are restricted to level 0 because PML ABCs are handled within this level only. Generation of subgrids and location of media are restricted to the interior of the total field region, which is far from the boundaries of the computational domain.

2.4 Signal Injection and Discretization

Signals of various types are injected into the total field region. Before we discuss the injection and extraction mechanism, let us focus first on the signal itself.

Suppose $H_z(x, y, t)$ is given by

$$H_z = f(\zeta), \quad \text{where } \zeta = n_x x + n_y y - t,$$

where f is an arbitrary function of one variable, called ζ here, and $n_x^2 + n_y^2 = 1$.

It is easy to build a valid Maxwell equations solution around H_z so defined.

For an arbitrary $f(\zeta)$ we have that

$$\begin{aligned} \partial_t f(\zeta) &= f'(\zeta) \partial_t \zeta = -f'(\zeta) \\ \partial_x f(\zeta) &= f'(\zeta) \partial_x \zeta = n_x f'(\zeta) \\ \partial_y f(\zeta) &= f'(\zeta) \partial_y \zeta = n_y f'(\zeta) \end{aligned}$$

Let

$$\begin{aligned} E_x &= g(\zeta) \\ E_y &= h(\zeta). \end{aligned}$$

Hence the equations

$$\begin{aligned} \partial_t E_x &= -g'(\zeta) = \partial_y H_z = \partial_y f(\zeta) = n_y f'(\zeta) \\ \partial_t E_y &= -h'(\zeta) = -\partial_x H_z = -\partial_x f(\zeta) = -n_x f'(\zeta) \end{aligned}$$

can be satisfied easily by setting

$$\begin{aligned} E_x &= g(\zeta) = -n_y f(\zeta) \\ E_y &= h(\zeta) = n_x f(\zeta). \end{aligned}$$

Indeed, in this case we have

$$\begin{aligned}\partial_t E_x &= -n_y \partial_t f(\zeta) = n_y f'(\zeta) = \partial_y H_z \\ \partial_t E_y &= n_x \partial_t f(\zeta) = -n_x f'(\zeta) = -\partial_x H_z,\end{aligned}$$

which is a trivial restatement of the above, but we also have that

$$\begin{aligned}-(\partial_x E_y - \partial_y E_x) &= -(n_x \partial_x f(\zeta) + n_y \partial_y f(\zeta)) \\ &= -(n_x n_x f'(\zeta) + n_y n_y f'(\zeta)) = -(n_x^2 + n_y^2) f'(\zeta) \\ &= -f'(\zeta) = \partial_t H_z\end{aligned}$$

because $n_x^2 + n_y^2 = 1$.

In summary, for an arbitrarily shaped $f(\zeta)$, we have the following:

$$\begin{aligned}H_z &= f(\zeta) \\ E_x &= -n_y f(\zeta) \\ E_y &= n_x f(\zeta),\end{aligned}$$

where $\zeta = n_x x + n_y y - t$ and $n_x^2 + n_y^2 = 1$ satisfy Maxwell equations.

We can generalize ζ even more by adding constant offsets to x , y , and t :

$$\zeta = n_x(x - x_0) + n_y(y - y_0) - (t - t_0).$$

SHAPES provides numerous signals, listed in Table 1. More can be added trivially on request.

Table 1: Signals provided by SHAPES.

Mode	Description	Formula
0	nothing	
1	harmonic wave	$f(\zeta) = \sin\left(\frac{2\pi}{\lambda}\zeta\right)$
2	step ramped harmonic wave	$f(\zeta) = \theta(-\zeta) \sin\left(\frac{2\pi}{\lambda}\zeta\right)$
3	tanh ramped harmonic wave	$f(\zeta) = \frac{1}{2}(1 - \tanh(\alpha\zeta)) \sin\left(\frac{2\pi}{\lambda}\zeta\right)$
4	Gaussian pulse	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)$
5	Gaussian envelope harmonic wave	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda}\zeta\right)$
6	Gaussian envelope linear chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda + \beta\zeta}\zeta\right)$
7	Gaussian envelope quadratic chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda + \beta\zeta^2}\zeta\right)$
8	Gaussian envelope exp chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda + \alpha \exp(\beta\zeta)}\zeta\right)$
9	Gaussian envelope sin chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda + \alpha \sin(\beta\zeta)}\zeta\right)$
10	Gaussian envelope tanh chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda + \alpha \tanh(\beta\zeta)}\zeta\right)$
11	Gaussian envelope Gaussian chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right) \sin\left(\frac{2\pi}{\lambda + \alpha \exp\left(-\frac{\zeta^2}{2\beta^2}\right)}\zeta\right)$

The user specifies the signal mode on the input file, followed by n_x , n_y , α , β , σ , and λ as required by the chosen formula. The user can specify t_0 , too, in order to delay the injection of the signal into the

total field region. But x_0 and y_0 are chosen by the program automatically, depending on the direction from which the signal is injected.

Of the 11 signals listed above, modes 0 and 1 are provided only for completeness. Mode 6, the Gaussian envelope linear chirp, is somewhat dangerous because it is easy to generate a division by zero in the sine function: as the program iterates, ζ moves toward negative numbers. But quadratic and tanh chirps are useful. The tanh chirp is basically like a linear chirp but restricted to an (almost) linear variation between some minimum and maximum value of the wavelength. One should take care to resolve the signal properly at its shortest wavelength component.

It is generally a good idea to choose the signal parameters carefully and inspect function $f(\zeta)$, for example, with Gnuplot, for the whole span of ζ that is to be covered by `shapes`. It is easy to design a chirp signal so that a complicated beat may result instead of a gradually changing wave. Figure 1 shows

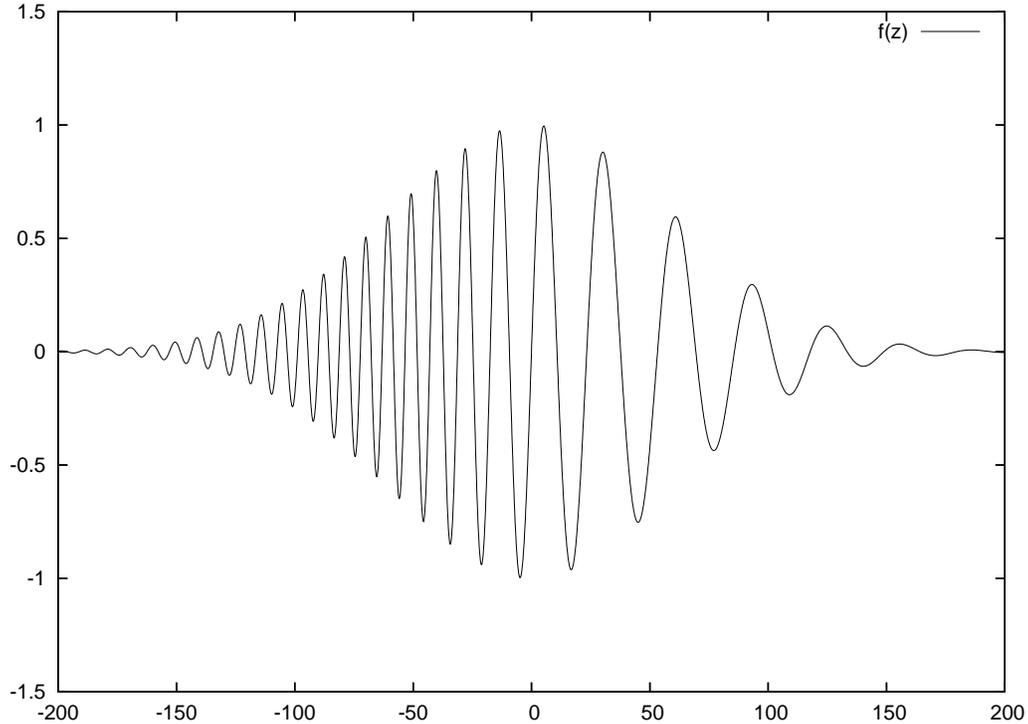


Figure 1: Gaussian envelope tanh chirp for $\alpha = 10$, $\beta = 0.015$, $\sigma = 60$, and $\lambda = 20$. The effective wavelength varies between 10 and 30. The signal moves to the right.

an example signal of a Gaussian envelope tanh chirp for $\alpha = 10$, $\beta = 0.015$, $\sigma = 60$, and $\lambda = 20$.

The incident field is injected into the total field region and then extracted from it by tweaking appropriate field derivatives on the border of the region. One of the signal analytical formulas discussed above, as selected by the user, is used to propagate the field along the boundary, whereas propagation of the field within the total field region is numerical. This strategy has the advantage of letting us observe the effects of numerical dispersion and test the accuracy of the solution method—covering also the accuracy of multilevel computations. On the other hand, it has a disadvantage as well because of numerical dispersion analytically, and numerically propagated pulses eventually diverge, resulting in a small, unsubtracted signal within the scattered field region, especially toward the end of pulse or wave propagation within the total field region. But this signal is absorbed by PML ABCs and does not enter the total field region. It

can be minimized by resolving the injected signal better.

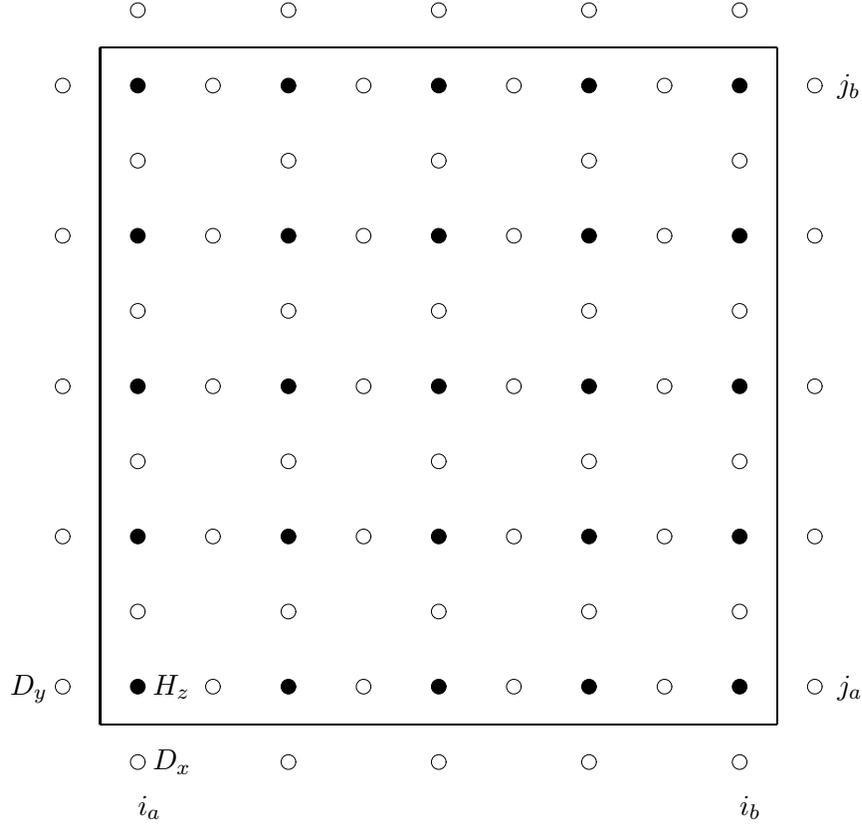


Figure 2: Definition of total and scattered field regions laid out on top of a staggered grid. The total field region is within the rectangle defined by (i_a, j_a) and (i_b, j_b) . Everything outside this rectangle is the scattered field region. Magnetic field H_z is cell-centered (black dots). Field D_y is side-centered to the left of H_z , and field D_x is side-centered below H_z . In other words, $H_z(i, j) \equiv H_z(x_0 + i\Delta x, y_0 + j\Delta y)$, $D_x(i, j) \equiv D_x(x_0 + i\Delta x, y_0 + j\Delta y - \Delta y/2)$ and $D_y(i, j) \equiv D_y(x_0 + i\Delta x - \Delta x/2, y_0 + j\Delta y)$.

Figure 2 illustrates the total and scattered field regions, with the total field region enclosed within a rectangle defined by (i_a, j_a) and (i_b, j_b) .

Maxwell equations in the (E_x, E_y, H_z) mode are

$$\begin{aligned}\partial_t \mathbf{D} &= \nabla \times \mathbf{H} = (\partial_y H_z) \mathbf{e}_x - (\partial_x H_z) \mathbf{e}_y \\ \mathbf{D} &= \epsilon \mathbf{E} \\ \partial_t \mathbf{H} &= -\nabla \times \mathbf{E} = -(\partial_x E_y - \partial_y E_x) \mathbf{e}_z\end{aligned}$$

and are discretized as follows (see Figure 2):

$$\begin{aligned}\frac{D_x^{(n)}(i, j) - D_x^{(n-1)}(i, j)}{\Delta t} &= \frac{H_z^{(n-1)}(i, j) - H_z^{(n-1)}(i, j - 1)}{\Delta y} \\ \frac{D_y^{(n)}(i, j) - D_y^{(n-1)}(i, j)}{\Delta t} &= -\frac{H_z^{(n-1)}(i, j) - H_z^{(n-1)}(i - 1, j)}{\Delta x} \\ D_x^{(n)}(i, j) &= \epsilon E_x^{(n)}(i, j)\end{aligned}$$

$$\begin{aligned}
D_y^{(n)}(i, j) &= \epsilon E_y^{(n)}(i, j) \\
\frac{H_z^{(n)}(i, j) - H_z^{(n-1)}(i, j)}{\Delta t} &= - \left(\frac{E_y^{(n)}(i+1, j) - E_y^{(n)}(i, j)}{\Delta x} - \frac{E_x^{(n)}(i, j+1) - E_x^{(n)}(i, j)}{\Delta y} \right)
\end{aligned}$$

which is solved to yield

$$D_x^{(n)}(i, j) = D_x^{(n-1)}(i, j) + \frac{H_z^{(n-1)}(i, j) - H_z^{(n-1)}(i, j-1)}{\Delta y} \Delta t \quad (12)$$

$$D_y^{(n)}(i, j) = D_y^{(n-1)}(i, j) - \frac{H_z^{(n-1)}(i, j) - H_z^{(n-1)}(i-1, j)}{\Delta x} \Delta t \quad (13)$$

$$D_x^{(n)}(i, j) = \epsilon E_x^{(n)}(i, j) \quad (14)$$

$$D_y^{(n)}(i, j) = \epsilon E_y^{(n)}(i, j) \quad (15)$$

$$H_z^{(n)}(i, j) = H_z^{(n-1)}(i, j) - \left(\frac{E_y^{(n)}(i+1, j) - E_y^{(n)}(i, j)}{\Delta x} - \frac{E_x^{(n)}(i, j+1) - E_x^{(n)}(i, j)}{\Delta y} \right) \Delta t. \quad (16)$$

Now consider the update of $D_y(i_a, j)$ for $j_a \in [j_a, j_b]$:

$$D_y^{(n)}(i_a, j) = D_y^{(n-1)}(i_a, j) - \frac{H_z^{(n-1)}(i_a, j) - H_z^{(n-1)}(i_a-1, j)}{\Delta x} \Delta t.$$

Here $H_z^{(n-1)}(i_a, j)$ is *inside* the total field region, but $D_y^{(n)}(i_a, j)$, $D_y^{(n-1)}(i_a, j)$ and $H_z^{(n-1)}(i_a-1, j)$ are *outside*. The inside value of $H_z^{(n-1)}(i_a, j)$ has the incident field component, which we must subtract from it so that the update within the *scattered* field region is correct. There is no injected signal in the scattered field region. Consequently

$$\begin{aligned}
D_y^{(n)}(i_a, j) &= D_y^{(n-1)}(i_a, j) - \frac{\left(H_z^{(n-1)}(i_a, j) - H_z^{(n-1)}_{\text{inc}}(i_a, j) \right) - H_z^{(n-1)}(i_a-1, j)}{\Delta x} \Delta t \\
&= D_y^{(n)}_{\text{no-inc}}(i_a, j) + \frac{H_z^{(n-1)}_{\text{inc}}(i_a, j)}{\Delta x} \Delta t
\end{aligned} \quad (17)$$

for all $j \in [j_a, j_b]$. Here $D_y^{(n)}_{\text{no-inc}}(i_a, j)$ is the $D_y(i_a, j)$ updated as if there were no incident signal, and $H_z^{(n-1)}_{\text{inc}}(i_a, j)$ is the incident signal. This approach lets us update D_y on the boundary using the same routine as applicable to the total field region and then implement the signal injection as a correction on the boundary.

Similarly for $D_y(i_b+1, j)$ for all $j \in [j_a, j_b]$,

$$D_y^{(n)}(i_b+1, j) = D_x^{(n-1)}(i_b+1, j) - \frac{H_z^{(n-1)}(i_b+1, j) - H_z^{(n-1)}(i_b, j)}{\Delta x} \Delta t$$

Here $D_y^{(n)}(i_b+1, j)$, $D_x^{(n-1)}(i_b+1, j)$ and $H_z^{(n-1)}(i_b+1, j)$ are *outside*, but $H_z^{(n-1)}(i_b, j)$ is *inside* the total field region, and so it carries the incident field component, which we must subtract from it so that the update within the *scattered* field region is correct:

$$\begin{aligned}
D_y^{(n)}(i_b+1, j) &= D_y^{(n-1)}(i_b+1, j) - \frac{H_z^{(n-1)}(i_b+1, j) - \left(H_z^{(n-1)}(i_b, j) - H_z^{(n-1)}_{\text{inc}}(i_b, j) \right)}{\Delta x} \Delta t \\
&= D_y^{(n)}_{\text{no-inc}}(i_b+1, j) - \frac{H_z^{(n-1)}_{\text{inc}}(i_b, j)}{\Delta x} \Delta t
\end{aligned} \quad (18)$$

for all $j \in [j_a, j_b]$.

We note three important points: (1) we correct $D_y^{(n)}(i_a, j)$ on the left-hand side but $D_y^{(n)}(i_b + 1, j)$ on the right-hand side; (2) we *add* the incident signal, which is evaluated at (i_a, j) on the left-hand side, but *subtract* the incident signal, which is evaluated at (i_b, j) on the right hand side; and (3) there is no need for corrections to D_x at $i = i_a$ and $i = i_b$ in general, unless $j = j_b + 1$ or $j = j_a$.

Now let us implement similar corrections to D_x updates at $j = j_a$ and $j = j_b + 1$. For $j = j_a$ and $i \in [i_a, i_b]$ we have

$$D_x^{(n)}(i, j_a) = D_x^{(n-1)}(i, j_a) + \frac{H_z^{(n-1)}(i, j_a) - H_z^{(n-1)}(i, j_a - 1)}{\Delta y} \Delta t.$$

Here $D_x^{(n)}(i, j_a)$, $D_x^{(n-1)}(i, j_a)$, and $H_z^{(n-1)}(i, j_a - 1)$ are in the scattered field region, but $H_z^{(n-1)}(i, j_a)$ is in the total field region. To evaluate a correct update in the scattered field region, we must subtract the incident signal from $H_z^{(n-1)}(i, j_a)$ in the total field region

$$\begin{aligned} D_x^{(n)}(i, j_a) &= D_x^{(n-1)}(i, j_a) + \frac{\left(H_z^{(n-1)}(i, j_a) - H_z^{(n-1)}_{\text{inc}}(i, j_a) \right) - H_z^{(n-1)}(i, j_a - 1)}{\Delta y} \Delta t \\ &= D_x^{(n)}_{\text{no-inc}}(i, j_a) - \frac{H_z^{(n-1)}_{\text{inc}}(i, j_a)}{\Delta y} \Delta t \end{aligned} \quad (19)$$

for all $i \in [i_a, i_b]$.

For $j = j_b + 1$ and $i \in [i_a, i_b]$ we have

$$D_x^{(n)}(i, j_b + 1) = D_x^{(n-1)}(i, j_b + 1) + \frac{H_z^{(n-1)}(i, j_b + 1) - H_z^{(n-1)}(i, j_b)}{\Delta y} \Delta t.$$

Here $D_x^{(n)}(i, j_b + 1)$, $D_x^{(n-1)}(i, j_b + 1)$ and $H_z^{(n-1)}(i, j_b + 1)$ are in the scattered field region, but $H_z^{(n-1)}(i, j_b)$ is in the total field region. To evaluate a correct update in the scattered field region, we must subtract the incident signal from $H_z^{(n-1)}(i, j_b)$ in the total field region

$$\begin{aligned} D_x^{(n)}(i, j_b + 1) &= D_x^{(n-1)}(i, j_b + 1) + \frac{H_z^{(n-1)}(i, j_b + 1) - \left(H_z^{(n-1)}(i, j_b) - H_z^{(n-1)}_{\text{inc}}(i, j_b) \right)}{\Delta y} \Delta t \\ &= D_x^{(n)}_{\text{no-inc}}(i, j_b + 1) + \frac{H_z^{(n-1)}_{\text{inc}}(i, j_b)}{\Delta y} \Delta t \end{aligned} \quad (20)$$

for all $i \in [i_a, i_b]$.

This time we *subtract* the incident signal, which is evaluated at $j = j_a$ at the bottom, but *add* the incident signal, which is evaluated at $j = j_b$ at the top. The corrections are implemented for $D_x^{(n)}(i, j_a)$ at the bottom, but for $D_x^{(n)}(i, j_b + 1)$ at the top. There is no need for corrections to the D_y update at $j = j_a$ and $j = j_b$ except for $i = i_a$ and $i = i_b + 1$.

Signal injection corrections to H_z are more complicated because they have to be evaluated at all total region boundaries.

Let us first consider equation (16) at $i = i_a$:

$$H_z^{(n)}(i_a, j) = H_z^{(n-1)}(i_a, j) - \left(\frac{E_y^{(n)}(i_a + 1, j) - E_y^{(n)}(i_a, j)}{\Delta x} - \frac{E_x^{(n)}(i_a, j + 1) - E_x^{(n)}(i_a, j)}{\Delta y} \right) \Delta t.$$

In this equation $H_z^{(n)}(i_a, j)$, $H_z^{(n-1)}(i_a, j)$, $E_y^{(n)}(i_a + 1, j)$, $E_x^{(n)}(i_a, j + 1)$, and $E_x^{(n)}(i_a, j)$ are all within the *total* field region, unless $j = j_a$ or $j = j_b$ in which case $E_x^{(n)}(i_a, j)$ or $E_x^{(n)}(i_a, j + 1)$ are outside (we take care of this later). The term that is outside, in the *scattered* field region for all $j \in [j_a, j_b]$, is $E_y^{(n)}(i_a, j)$. Since this term has already been processed, it doesn't have the incident field component in it. Hence, to make a correct update to $H_z^{(n-1)}(i_a, j)$, we need to *add* the incident field to $E_y^{(n)}(i_a, j)$:

$$\begin{aligned} H_z^{(n)}(i_a, j) &= H_z^{(n-1)}(i_a, j) \\ &- \left(\frac{E_y^{(n)}(i_a + 1, j) - \left(E_y^{(n)}(i_a, j) + E_y^{(n)}_{\text{inc}}(i_a, j) \right)}{\Delta x} - \frac{E_x^{(n)}(i_a, j + 1) - E_x^{(n)}(i_a, j)}{\Delta y} \right) \Delta t \\ &= H_z^{(n)}_{\text{no-inc}}(i_a, j) + \frac{E_y^{(n)}_{\text{inc}}(i_a, j)}{\Delta x} \Delta t. \end{aligned} \quad (21)$$

On the right-hand side, the equation is

$$H_z^{(n)}(i_b, j) = H_z^{(n-1)}(i_b, j) - \left(\frac{E_y^{(n)}(i_b + 1, j) - E_y^{(n)}(i_b, j)}{\Delta x} - \frac{E_x^{(n)}(i_b, j + 1) - E_x^{(n)}(i_b, j)}{\Delta y} \right) \Delta t.$$

This time all terms are within the *total* field region except $E_y^{(n)}(i_b + 1, j)$, which is in the *scattered* field region. Hence, we must again *add* the incident field to this term:

$$\begin{aligned} H_z^{(n)}(i_b, j) &= H_z^{(n-1)}(i_b, j) \\ &- \left(\frac{\left(E_y^{(n)}(i_b + 1, j) + E_y^{(n)}_{\text{inc}}(i_b + 1, j) \right) - E_y^{(n)}(i_b, j)}{\Delta x} - \frac{E_x^{(n)}(i_b, j + 1) - E_x^{(n)}(i_b, j)}{\Delta y} \right) \Delta t \\ &= H_z^{(n)}_{\text{no-inc}}(i_b, j) - \frac{E_y^{(n)}_{\text{inc}}(i_b + 1, j)}{\Delta x} \Delta t \end{aligned} \quad (22)$$

Now let us look at what happens on the $j = j_a$ and $j = j_b$ lines. At $j = j_a$ we have

$$H_z^{(n)}(i, j_a) = H_z^{(n-1)}(i, j_a) - \left(\frac{E_y^{(n)}(i + 1, j_a) - E_y^{(n)}(i, j_a)}{\Delta x} - \frac{E_x^{(n)}(i, j_a + 1) - E_x^{(n)}(i, j_a)}{\Delta y} \right) \Delta t.$$

Here we find that all terms are within the *total* field region except $E_x^{(n)}(i, j_a)$, which is outside in the *scattered* field region. The E_y terms get outside also at $i = i_a$ and $i = i_b$, but we have already taken care of this above. Hence, to make the update correct, we need to *add* the incident field to $E_x^{(n)}(i, j_a)$:

$$\begin{aligned} H_z^{(n)}(i, j_a) &= H_z^{(n-1)}(i, j_a) \\ &- \left(\frac{E_y^{(n)}(i + 1, j_a) - E_y^{(n)}(i, j_a)}{\Delta x} - \frac{E_x^{(n)}(i, j_a + 1) - \left(E_x^{(n)}(i, j_a) + E_x^{(n)}_{\text{inc}}(i, j_a) \right)}{\Delta y} \right) \Delta t \\ &= H_z^{(n)}_{\text{no-inc}}(i, j_a) - \frac{E_x^{(n)}_{\text{inc}}(i, j_a)}{\Delta y} \Delta t. \end{aligned} \quad (23)$$

Repeating the same procedure for $j = j_b$,

$$H_z^{(n)}(i, j_b) = H_z^{(n-1)}(i, j_b) - \left(\frac{E_y^{(n)}(i + 1, j_b) - E_y^{(n)}(i, j_b)}{\Delta x} - \frac{E_x^{(n)}(i, j_b + 1) - E_x^{(n)}(i, j_b)}{\Delta y} \right) \Delta t,$$

we find that this time we need to add $E_x^{(n)}(i, j_b + 1)$ to the incident field. Hence,

$$\begin{aligned}
H_z^{(n)}(i, j_b) &= H_z^{(n-1)}(i, j_b) \\
&- \left(\frac{E_y^{(n)}(i+1, j_b) - E_y^{(n)}(i, j_b)}{\Delta x} - \frac{\left(E_x^{(n)}(i, j_b + 1) + E_x^{(n)}_{\text{inc}}(i, j_b + 1) \right) - E_x^{(n)}(i, j_b)}{\Delta y} \right) \Delta t \\
&= H_z^{(n)}_{\text{no-inc}}(i, j_b) + \frac{E_x^{(n)}_{\text{inc}}(i, j_b + 1)}{\Delta y} \Delta t.
\end{aligned} \tag{24}$$

Let us summarize the results for the H_z field. Corrections to H_z have to be carried out on all four boundaries. On the $i = i_a$ boundary we *add* the correction, which is evaluated at (i_a, j) . On the $i = i_b$ boundary we *subtract* the correction, which is evaluated at $(i_b + 1, j)$. Then at $j = j_a$ we *subtract* the correction, which is evaluated at (i, j_a) , and at $j = j_b$ we *add* the correction, which is evaluated at $(i, j_b + 1)$. The corner points are handled correctly because here the updates are performed twice.

We note that the \mathbf{D} corrections are evaluated by using the $n - 1$ \mathbf{H} slice, whereas the \mathbf{H} corrections are evaluated by using the n \mathbf{E} slice.

The signal injection routines can be tested easily by submitting runs without any media but with various signals injected into the total field region from various directions.

2.5 Spectral Response

SHAPES will accumulate spectral response integrals, on request, for E_x , E_y , H_z and energy density, $\mathcal{E} = (E_x^2 + E_y^2 + H_z^2) / 2$, for any number of frequencies. The accumulated Fourier integrals for field $f(x, y, t)$ at time step $t_n = n\Delta t$, where Δt is the level 0 (see Section 2.6) time step, are

$$\begin{aligned}
\hat{f}_r(x, y, \omega) &= \sum_{k=1}^n f(x, y, t_k) \cos(\omega t_k) \Delta t \\
\hat{f}_i(x, y, \omega) &= \sum_{k=1}^n f(x, y, t_k) \sin(\omega t_k) \Delta t.
\end{aligned}$$

Currently, spectral response is not calculated at the time granularity of higher resolution levels, if synchronized multistep is used (see Section 2.6.1). But Δt is included in the above formulas so that we can introduce higher resolution time granularity in the future.

The Fourier accumulation terms are then used to evaluate the amplitude and the phase angle:

$$\begin{aligned}
|\hat{f}(x, y, \omega)| &= \sqrt{\hat{f}_r^2(x, y, \omega) + \hat{f}_i^2(x, y, \omega)} \\
\Phi(x, y, \omega) &= \arctan \left(\frac{\hat{f}_i(x, y, \omega)}{\hat{f}_r(x, y, \omega)} \right).
\end{aligned}$$

The frequencies used in the input to and output from SHAPES are *circular* frequencies, $\omega = 2\pi/T$, where T is the vibration period.

Every accumulation term is stored on its own separate Chombo array. The arrays are allocated dynamically depending on the number of frequencies for which the spectral response is to be computed.

2.6 Multigrid

The multigrid is generated on request in specified locations, or in places where the energy density \mathcal{E} changes faster than a certain threshold value, or in places where the energy density exceeds a certain threshold

value, or in places where combinations of the above conditions occur. The user specifies a number of desired multigrid *levels*. Level 0 corresponds to the coarsest grid; level 1 corresponds to a grid that is refined by 2 in both x and y directions:

$$\begin{aligned}\Delta x_1 &= \Delta x_0/2 \\ \Delta y_1 &= \Delta y_0/2;\end{aligned}$$

level 2 corresponds to a grid that is refined by 2 again:

$$\begin{aligned}\Delta x_2 &= \Delta x_1/2 = \Delta x_0/4 \\ \Delta y_2 &= \Delta y_1/2 = \Delta y_0/4\end{aligned}$$

and so on. Finer grids must be thoroughly enclosed within the coarser grids; in other words, the level 1 grid must be enclosed within the level 0 grid, the level 2 grid must be enclosed within the level 1 grid, and so on. Higher-level grids may consist of separate portions, that is, there may be a higher-resolution grid patch here and a higher resolution grid patch there, separated by a lower-resolution grid region. All patches that are characterized by the same refinement number comprise a single, though possibly disconnected, level.

The refinement process also shifts the zero of the refined grid with respect to the coarse grid, as shown in Figure 3. In this figure the black dots correspond to the centers of the coarse-grid cells, and the white

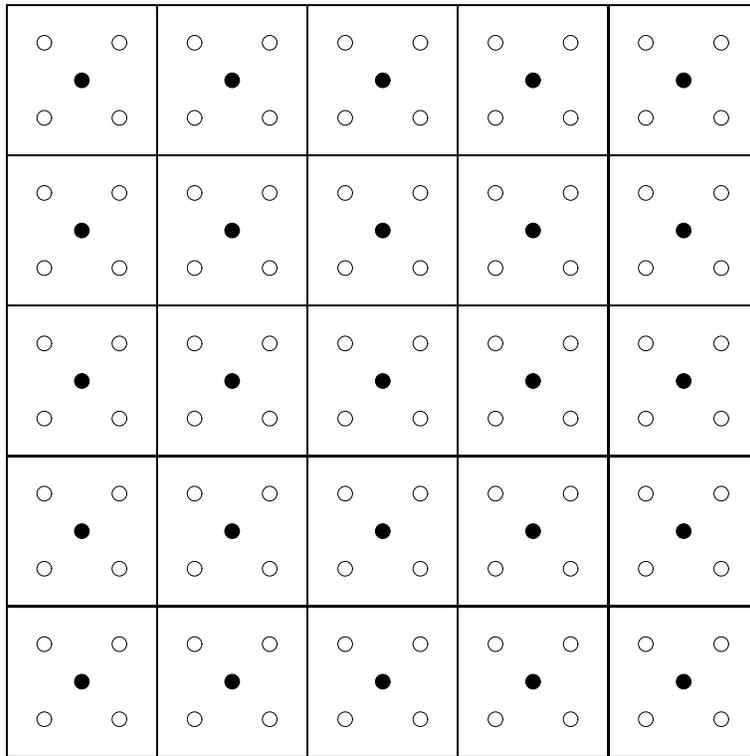


Figure 3: Refinement process. The black dots correspond to the centers of the coarse-grid cells, and the white dots correspond to the centers of the fine-grid cells.

dots correspond to the centers of the fine-grid cells. Each coarse-grid cell is subdivided into four fine-grid cells. The $(0,0)$ grid point of the fine grid is shifted by $(-\Delta x_{\text{coarse}}/4, -\Delta y_{\text{coarse}}/4)$ with respect to (x_0, y_0) of the coarse grid.

Levels like the two shown in Figure 3 are called *adjacent*. For example, levels 2 and 3 are adjacent, but levels 2 and 4 are not. Similarly, levels 2 and 1 are adjacent, but levels 2 and 0 are not.

Data is transferred between adjacent levels *only*. This transfer happens in three ways. First, when the level is created, data from the coarser level is interpolated in space and for some fields also in time onto the new finer level. Second, when the levels are advanced, data from the coarser level is interpolated in space and sometimes in time onto the boundary of the finer level in order to create a boundary condition there. Third, at certain time slices, when data on the finer-level grid is time-aligned with data on the coarser grid, the finer-level data is averaged back onto the coarser-level grid.

Special care must be taken when interpolating and averaging face-centered data. The geometric relationship between coarser- and finer-level data is different for face-centered data and for cell-centered data. Consequently, different utilities must be used to average and interpolate face-centered data and cell-centered data. Chombo provides basic support for cell-centered data only. SHAPES uses additional utilities developed by Dan Martin of Lawrence Berkeley National Laboratory for handling face-centered data.



2.6.1 Multigrid Time Step

A leap-frog time step in a multigrid system may be implemented in two ways. One way is to select a time step that corresponds to the finest level and use it to advance all levels at the same pace. This way works well. A natural question is, where is the saving? The saving is in not having to advance a large number of fine-grid cells, even though we still have to advance all cells of all levels at the same very fine time step that is chosen to satisfy the stability criterion of the finest grid. Sections 5.2 and 5.3 show that the saving is considerable. We call this technique a *synchronized unistep*, which is a short for a *synchronized multilevel leapfrog uniform time-step*.

The other way is to advance each level at its own time step, different from one level to the other, but still maintaining synchronization at shared **E** time slices across all levels and at shared **H** time slices across all levels. For our refinement ratio of 2 this is possible if the time step is refined by 3 between adjacent levels, not by 2, that is,

$$\begin{aligned}\Delta t_1 &= \Delta t_0/3 \\ \Delta t_2 &= \Delta t_1/3 = \Delta t_0/9.\end{aligned}$$

Figure 4 illustrates why this is so. We have only two levels in this figure, but it can be extended by simply relabeling the levels. Observe that the level 0 **E** coincides with the level 1 **E** and that the level 0 **H** coincides with the level 1 **H**. We call this a *synchronized multistep*, which is a short for a *synchronized multilevel leapfrog multi-time-step*.

SHAPES implements both ways and the user is free to choose one or the other.

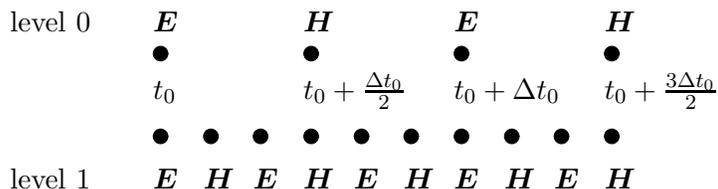


Figure 4: The two-level synchronized multistep leapfrog: $\Delta t_1 = \Delta t_0/3$.

Data is exchanged between adjacent levels in the following three ways.

First, data from a coarser-level grid is interpolated onto a finer-level grid when the finer-level grid is created or moved. When the finer-level grid is moved, then whatever data can be copied from the previous instance of the grid is copied, and where data is not available, it is interpolated from the coarser-level grid. The interpolation takes place in space only for the \mathbf{E} field, but in both space and time for the \mathbf{H} field, because the finer-level grid restructuring or generation is carried out on the \mathbf{E} time slices only.

Second, data from a coarser-level grid is interpolated onto the boundary of a finer-level grid to create the boundary condition for the finer-level grid time step. The interpolation takes place in both time and space for both the \mathbf{E} and \mathbf{H} fields, with the exception of the time slices for which the finer- and the coarser-level \mathbf{E} and \mathbf{H} fields coincide.

Third, data from a finer-level grid is averaged onto a coarser-level grid, as the coarser-level correction, at the time slices for which the finer- and the coarser-level \mathbf{E} and \mathbf{H} fields coincide.

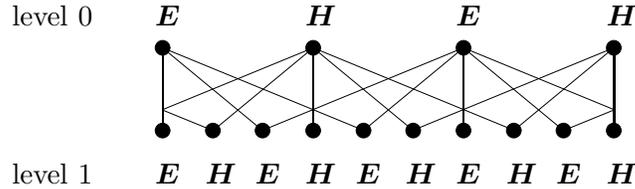


Figure 5: The two-level synchronized multistep leapfrog showing communication lines between the levels. Communication on vertical lines is bidirectional, since these are sync lines and level 1 data can be used to correct (by averaging) level 0 data. Communication on angled lines is from level 0 to level 1. Here level 0 data is used to provide data for the level 1 boundary.

Figure 5 shows interlevel communication for the synchronized multistep. For the level 1 points, where two angled lines meet, time interpolation is implied between the level 0 points from which the lines originate.

Now, let us look at the synchronized multistep procedure more closely. Consider the two-level system shown in Figures 4 and 5. Assume that we are at the following time slices.

$$\begin{array}{lll}
 \mathbf{E}_0 & \text{at} & t_0 \\
 \mathbf{E}_{0\text{old}} & \text{at} & t_0 - \Delta t_0 \\
 \mathbf{H}_0 & \text{at} & t_0 + \Delta t_0/2 \\
 \mathbf{H}_{0\text{old}} & \text{at} & t_0 - \Delta t_0/2 \\
 \mathbf{E}_1 & \text{at} & t_1 = t_0 \\
 \mathbf{H}_1 & \text{at} & t_1 + \Delta t_1/2 = t_0 + \Delta t_0/3/2
 \end{array}$$

We can provide boundary data to $\mathbf{E}_1(t_0)$ because we have $\mathbf{E}_0(t_0)$, and we can provide boundary data to $\mathbf{H}_1(t_0 + \Delta t_0/3/2)$ by time interpolating between $\mathbf{H}_0(t_0 - \Delta t_0/2)$ and $\mathbf{H}_0(t_0 + \Delta t_0/2)$.

The two-level leapfrog now unfolds as follows:

- Advance \mathbf{E}_0 so that we get \mathbf{E}_0 at $t_0 + \Delta t_0$ and $\mathbf{E}_{0\text{old}}$ at t_0 .
 - Advance \mathbf{E}_1 so that we get \mathbf{E}_1 at $t_0 + \Delta t_0/3$. Time interpolate between $\mathbf{E}_{0\text{old}}$ at t_0 and \mathbf{E}_0 at $t_0 + \Delta t_0$ to provide boundary data for \mathbf{E}_1 at $t_0 + \Delta t_0/3$.
 - Advance \mathbf{H}_1 so that we get \mathbf{H}_1 at $t_0 + \Delta t_0/2$. As we are now in sync with level 0 \mathbf{H} -wise, correct \mathbf{H}_0 at this time slice by averaging data from \mathbf{H}_1 . Use \mathbf{H}_0 to provide boundary data to \mathbf{H}_1 at $t_0 + \Delta t_0/2$.

- Advance \mathbf{E}_1 so that we get \mathbf{E}_1 at $t_0 + 2\Delta t_0/3$. Time interpolate between $\mathbf{E}_{0\text{old}}$ at t_0 and \mathbf{E}_0 at $t_0 + \Delta t_0$ to provide boundary data for \mathbf{E}_1 at $t_0 + 2\Delta t_0/3$.
- Advance \mathbf{H}_0 so that we get \mathbf{H}_0 at $t_0 + 3\Delta t_0/2$ and $\mathbf{H}_{0\text{old}}$ at $t_0 + \Delta t_0/2$.
 - Advance \mathbf{H}_1 so that we get \mathbf{H}_1 at $t_0 + \Delta t_0/2 + \Delta t_0/3$. Time interpolate between $\mathbf{H}_{0\text{old}}$ at $t_0 + \Delta t_0/2$ and \mathbf{H}_0 at $t_0 + 3\Delta t_0/2$ to provide boundary data for \mathbf{H}_1 at $t_0 + \Delta t_0/2 + \Delta t_0/3$.
 - Advance \mathbf{E}_1 so that we get \mathbf{E}_1 at $t_0 + \Delta t_0$. As we are now in sync with level 0 \mathbf{E} -wise, correct \mathbf{E}_0 at this time slice by averaging data from \mathbf{E}_1 . Use \mathbf{E}_0 to provide boundary data to \mathbf{E}_1 at $t_0 + \Delta t_0$.
 - Advance \mathbf{H}_1 so that we get \mathbf{H}_1 at $t_0 + \Delta t_0 + \Delta t_1/2 = t_0 + \Delta t_0 + \Delta t_0/3/2$. Time interpolate between $\mathbf{H}_{0\text{old}}$ at $t_0 + \Delta t_0/2$ and \mathbf{H}_0 at $t_0 + 3\Delta t_0/2$ to provide boundary data for \mathbf{H}_1 at $t_0 + \Delta t_0 + \Delta t_0/3/2$.

We are exactly where we started but with $t_0 \rightarrow t_0 + \Delta t_0$.

Let us repeat this sequence below, with interpolation and time slice details stripped off for clarity:

- Update \mathbf{E}_0
 - Update \mathbf{E}_1
 - Update \mathbf{H}_1 , correct \mathbf{H}_0
 - Update \mathbf{E}_1
- Update \mathbf{H}_0
 - Update \mathbf{H}_1
 - Update \mathbf{E}_1 , correct \mathbf{E}_0
 - update \mathbf{H}_1

It is now easy to arrive at, for example, a three-level leapfrog time step by extending the above routine recursively:

- Update \mathbf{E}_0
 - Update \mathbf{E}_1
 - * Update \mathbf{E}_2
 - * Update \mathbf{H}_2 , correct \mathbf{H}_1
 - * Update \mathbf{E}_2
 - Update \mathbf{H}_1 , correct \mathbf{H}_0
 - * Update \mathbf{H}_2
 - * Update \mathbf{E}_2 , correct \mathbf{E}_1
 - * Update \mathbf{H}_2
 - Update \mathbf{E}_1
 - * Update \mathbf{E}_2
 - * Update \mathbf{H}_2 , correct \mathbf{H}_1
 - * Update \mathbf{E}_2
- Update \mathbf{H}_0

- Update \mathbf{H}_1
 - * Update \mathbf{H}_2
 - * Update \mathbf{E}_2 , correct \mathbf{E}_1
 - * Update \mathbf{H}_2
- Update \mathbf{E}_1 , correct \mathbf{E}_0
 - * Update \mathbf{E}_2
 - * Update \mathbf{H}_2 , correct \mathbf{H}_1
 - * Update \mathbf{E}_2
- Update \mathbf{H}_1
 - * Update \mathbf{H}_2
 - * Update \mathbf{E}_2 , correct \mathbf{E}_1
 - * Update \mathbf{H}_2

Clearly, to implement this time step, we need two types of routines that call each other recursively. Let us call the first routine `update_e_n`. It will do the following.

- Update \mathbf{E}_n
- Update \mathbf{H}_n , correct \mathbf{H}_{n-1}
- Update \mathbf{E}_n

Let us call the second one `update_h_n`. It will do the following

- Update \mathbf{H}_n
- Update \mathbf{E}_n , correct \mathbf{E}_{n-1}
- Update \mathbf{H}_n

To implement the recursion, we then do the following

1. For `update_e_n`:
 - Update \mathbf{E}_n , call `update_e_n(n+1)`
 - Update \mathbf{H}_n , call `update_h_n(n+1)`, correct \mathbf{H}_{n-1}
 - Update \mathbf{E}_n , call `update_e_n(n+1)`
2. For `update_h_n`:
 - Update \mathbf{H}_n , call `update_h_n(n+1)`
 - Update \mathbf{E}_n , call `update_e_n(n+1)`, correct \mathbf{E}_{n-1}
 - Update \mathbf{H}_n , call `update_h_n(n+1)`

Of course we must implement various checks to ensure that, for example, a higher level exists before a call is made to `update_h_n` or `update_e_n`. If the next level does not exist, the recursion stops. But we need not check whether a lower level exists, because these two routines can be invoked only from within a coarser-level advance.

SHAPES handles level 0 advance differently, first, because we have to attend to PML ABCs and total and scattered field regions boundaries at this level, and, second, because there is no coarser level to be

corrected. Furthermore, there is no “1, 2, 3” dance step for level 0. Instead, level 0 time steps as usual and drives the recursive “1, 2, 3” dance step of higher levels—if they exist.

This recursive multilevel leapfrog has three major advantages. First, all levels flow always forward, without any back-stepping or look-ahead. Second, the evolution of all levels is always synchronized. Third, each level needs to remember only one previous field state in order to provide data for the higher-level boundary, as the finer levels always lag a little behind the coarser levels.

But the recursive multilevel multistep leapfrog procedure presented here does have some serious disadvantages. The first disadvantage is that computers cannot divide by 3, because they are all binary machines. Hence, a discrepancy is bound to accumulate over a large number of iterations between level 0 and level n times.

The second disadvantage is that because the time step shrinks faster than the space discretization, level n advance is going to be increasingly costly as n goes up. For example, let $\Delta x_0 = 1$ and $\Delta t_0 = 1/2$. Clearly, for a 10-level system the level 9 grid spacing and time step will be $\Delta x_9 = \Delta x_0 \times (1/2)^9 = .0019531250$, but $\Delta t_9 = \Delta t_0 \times (1/3)^9 = .0000254026$. Hence, whereas $\Delta t_0/\Delta x_0 = 1/2$, $\Delta t_9/\Delta x_9 = \Delta t_0 \times (1/3)^9/(\Delta x_0 \times (1/2)^9) = (\Delta t_0/\Delta x_0) \times (2/3)^9 = (1/2) \times 0.0260122948$. This second disadvantage compounds the first one because it results in a very large number of the $1/3$ time steps on multiple levels.

The third disadvantage is the need to time-interpolate boundary conditions for level k from the coarser level, $k - 1$, between $t_{k-1}^{(n)}$ and $t_{k-1}^{(n+1)}$ at $t = t_{k-1}^{(n)} + \Delta t_{k-1}/3$ and $t = t_{k-1}^{(n)} + 2\Delta t_{k-1}/3$, where n is the time step number at level $k - 1$. Apart from having to work with yet another division by 3, this is not going to be as accurate as would be the case if the coarser level had been advanced by using the same small time step as the finer level. This small inaccuracy is enough to introduce high-frequency noise into the solution procedure. The noise then gets trapped within the higher levels because it cannot propagate within the coarser levels, and once trapped, it grows and derails the solution.

The unistep procedure is much cleaner in this respect. There is markedly less noise generation here on level boundaries and experimental tests demonstrated its stability for all configurations investigated so far.

Some of the multigrid-induced problems can be controlled by deploying a judicious multigrid generation strategy. Refined levels should be kept as small as possible. There is no point bothering about multigrid if refined levels overlap with the original computational domain entirely or even largely. In this case, we are better off solving the problem on a sufficiently fine single-level grid or on a smaller number of levels with level 0 having a higher resolution.

2.6.2 Building Higher Levels

Before we can carry out a multilevel leapfrog time step on multilevel data, we have to build a multilevel system of field data. This is done recursively, although since simple tail recursion is deployed in this case, it could be done by using iterative code. The main idea here is that level n must be fully constructed before we can build level $(n + 1)$.

The process of building a new level is split into two steps:

1. We tag cells of level n for refinement based on one of three possible criteria: (1) the cell is located in a static area designated for refinement, (2) energy density in the cell exceeds a certain threshold level, and (3) the speed with which energy density in the cell changes exceeds a certain threshold level. The user may choose one or combine them as needed.

If the resulting set of tagged cells is empty, we finish the building procedure and truncate the vector of levels (there is such a construct in the code) so that level n is the highest level.

2. If the set of tagged cells for level n is not empty, we invoke the same procedure used to build level n , but this time we ask it to build level $(n + 1)$.

Once the level-building process is invoked within level 0, it keeps going either until it reaches a level for which no cells get tagged for further refinement or until it reaches a maximum allowed number of levels. Since building and advancing higher levels is increasingly expensive, a maximum number of levels is read from the input file and applied to this part of the code. But this number can be anything the user wants, limited only by available computational and storage resources.

Such recursive rebuilding (or, as Chombo calls it, regridding) of all levels is carried out at regular time intervals that coincide with \mathbf{E} time slices for level 0—unless the user has requested that cells be tagged on the location criterion only. In this case the multigrid is going to be static. SHAPES detects this, and, having built all levels, it forgoes invoking the regridding routines again.

Since regridding is carried out at \mathbf{E} time slices, current \mathbf{E} fields for multiple levels are simply generated by space interpolation from coarser levels. On the other hand, \mathbf{H} fields and `_old` fields must be both time and space interpolated from coarser levels because the \mathbf{E} time slices do not coincide with the \mathbf{H} time slices and with the `_old` field time slices for the fine level in general.

After data has been generated for a given level by interpolating data from a coarser level, we check whether this level has existed before. If it has, then more accurate high-resolution data is available for this level from the earlier time step. This data will not usually overlap with the current level entirely, but at least some of its content can be overwritten with the more accurate old level data. This overwriting indeed is done every time a new level is generated. To do otherwise would waste high-resolution data of refined levels every time regridding is attempted.

Multigrid is a mixed blessing. On the one hand, it lets us focus on a selected small patch of the system and solve Maxwell equations on this patch more accurately. On the other hand, multigrid may introduce noise and instability into the FDTD method, which is finely tuned for solving Maxwell equations on a *regular* rectangular grid, and we have not yet found a satisfactory way to deal with this problem in case of the synchronized multistep. On the other hand, there is markedly less noise in the solution when synchronized unistep is used, and we were able to carry out very long simulations using this method.

Multigrid computations can be costly, too. Moving data between levels is a communication-heavy process, especially when the code runs on a cluster.

All in all, multigrid is an overkill for most 2D computations, with the notable exception of those that attempt to simulate multiscale systems. We provide a suitable illustration of such a system in Section 5. For example, consider a situation in which we have to condition incident radiation in some way, using a larger device, and then irradiate a very small object with it. To simulate both the conditioning device and the small object at the same time, we may have to resort to multigrid computations. In this specific context multigrid may result in exponential savings, compared to what it would cost to solve the problem on a uniform very high resolution grid.

We expect that in three dimensions there will be more need for multigrid because the problems are going to be a lot larger and closer to real life-systems.

Just as important, it is *not* necessary to use multigrid features when working with SHAPES. SHAPES can do all its work on a single level, and whatever utilities are provided in the code for multi-level operations are not activated unless multigrid construction is explicitly requested by the user. Then the user has a number of strategies available, for example, choice of the time-stepping routine and choices pertaining to the fine-level grid geometry, to tune the multigrid solution method to the problem at hand.

2.7 Parallelization

SHAPES can run in parallel under MPI on whatever systems support MPI. These are most often clusters of small SMPs: two-way or four-way. It is advantageous, then, to invoke two or four MPI processes per node accordingly. SHAPES does not use OpenMP, hence, the only way to take advantage of the SMP architecture of the node is to fork multiple MPI processes on it, which is an efficient method to harness

the SMP architecture anyway. In this case, however, special care must be taken to ensure that HDF5 has been compiled for this architecture with multithreading enabled. Otherwise HDF5 MPI-IO output may hang.

SHAPES will also run on a single SMP. In this case the user should request a number of MPI processes matching the number of SMP CPUs allocated to the run.

Chombo parallelizes its operations by dividing all cells of a given level into rectangles and then distributing the rectangles of cells among the MPI processes. Thus, a Chombo rectangle of cells is a *quantum* of parallelism. But a given MPI process may end up with more than one rectangle for a given level, and then it may have to compute on rectangles covering other levels, too. Chombo attempts to distribute the load evenly. This process, however, depends also on the specific shape of the subgrids. Intricate shapes may require a large number of small rectangles to cover them.

The SHAPES user may specify the smallest size of a rectangle to be used in covering the subgrids, the largest size of a rectangle, and how relaxed the covering is allowed to be. A very relaxed covering may simply slap a single large rectangular patch on the designated area of the grid. A very tight covering may end up tracing the shape of the area closely—generating many small rectangles in the process.

When SHAPES runs in parallel, all I/O is done in parallel, too. SHAPES diagnostics are written by participating MPI processes on their own process-specific files. Field data is written on MPI-IO files by using the HDF5 library. Each snapshot results in a single HDF5-structured MPI-IO file, which is accessed by all MPI processes simultaneously. Such files are best created on a parallel file system, for example, PVFS, GPFS, or Lustre.

Unfortunately, we do not at present have tools that can read and visualize these HDF5 files in parallel. Right now we can only look at them with ChomboVis, which is a sluggish, sequential Python/VTK script. But it still helps when the script can read data *directly* from the parallel file system onto which it was dumped in the first place.

2.8 Output

SHAPES can output field data for D_x , D_y , E_x , E_y , H_z , $\mathcal{E} = (E_x^2 + E_y^2 + H_z^2)/2$, metal distribution, and spectral response associated with E_x , E_y , H_z , and \mathcal{E} . All field data that is written out is cell-centered and time-centered on the level 0 \mathbf{E} time slices. The user can specify which fields to dump and for which levels.

SHAPES can generate output in HDF5 or Gnuplot formats, and the user may choose one or the other or both. The HDF5 format can always be used. The Gnuplot format can be used only in the sequential version of the program; otherwise Gnuplot format directives are ignored.

Disabling both HDF5 and Gnuplot outputs disables all field data generation. The program will still run, though, and it will print various diagnostics on standard output or on the MPI output files if run in parallel and if such diagnostics have been requested. This mode of operation can be used for debugging.

HDF5 data files collate all fields selected for a given snapshot at all levels on a single structured file called `fields_<snapshot_number>.hdf5`. The field data is written first in a special machine independent IEEE approved format, not as text, and it is additionally *compressed*, so the files take much less space than do Gnuplot text files. On the other hand, extracting data from the HDF5 files is a major effort. The ChomboVis tool, mentioned above, must be used to postprocess and visualize data from the HDF5 files dumped by SHAPES.

HDF5 files contain some additional information in the headers, but not as much as Gnuplot files. This situation may change in future versions of SHAPES.

In the Gnuplot output mode the program generates separate files for each field at a given level. The Gnuplot data file names are `<field_name>_<level_number>_<snapshot_number>.dat`, where `<field_name>` corresponds to the field being dumped, for example, E_x , in which case `<field_name>` is `Ex`. For spectral response fields, the frequency enters the field name also, for example, `Ex_amplitude_0.1570`.

Gnuplot files generated by SHAPES are simple text files. They are human readable and their content can be displayed by using the Gnuplot `splot` command, for example, the following.

```
set xrange[10:90]
set yrange[10:90]
splot "Ex_0_074.dat" with lines
```

The header of each Gnuplot file contains extensive annotations, for example, the following.

```
# program: shapes, function: write_gnuplot_data
# header:
#   program author: Zdzislaw (Gustav) Meglicki, Indiana University
#   @Id: shapes.cpp,v 1.21 2005/12/22 18:41:29 gustav Exp @
#   @Id: shapes.h,v 1.64 2005/12/17 23:41:59 gustav Exp @
#   @Id: levels.cpp,v 1.90 2005/12/23 23:07:46 gustav Exp @
#   @Id: io.cpp,v 1.20 2005/12/23 20:50:20 gustav Exp @
#   @Id: update.f,v 1.45 2005/12/24 01:06:58 gustav Exp @
#   system kernel: CYGWIN_NT-5.1.1.5.18(0.132/4/2).2005-07-02 20:30
#   machine:      i686
#   node:        woodlands
#   time of dump: Fri Dec 23 21:36:42 2005
# Signal injection group:
#   x_lo:        20.000000
#   y_lo:        20.000000
#   x_hi:        80.000000
#   y_hi:        80.000000
#   mode:        7 (Gaussian envelope quadratic chirp)
#   t0:          300.000000
#   lambda:      10.000000
#   sigma:       60.000000
#   alpha:       0.000000
#   beta:        0.001200
#   vx:          0.000000
#   vy:          1.000000
# Media group:
#   medium 1:
#   epsilon_infty: 2.364610
#   alpha:         0.000000 1.000000 1.000000
#   omega:         0.662850 0.332290 0.393180
#   delta:         0.004290 0.063930 0.105760
#   epsilon:       1.000000 0.315040 0.868050
#   medium 2:
#   epsilon_infty: 2.364610
#   alpha:         0.000000 0.000000 0.000000
#   omega:         0.662850 0.000000 0.000000
#   delta:         0.004290 0.000000 0.000000
```

```

#   epsilon:      1.000000 0.000000 0.000000
# Media layout group:
#   boxes:
#   x_lo:         30.000000 55.000000
#   y_lo:         30.000000 30.000000
#   x_hi:         45.000000 70.000000
#   y_hi:         70.000000 70.000000
#   medium:       1.000000 2.000000
# Data group:
#   data for:     Hz
#   level:        0
#   label:        238
#   time_e:       476.000000
#   time_h:       476.062500
#   delta_t:      0.125000
#   delta_t_0:    0.125000
#   xmin:         0.000000
#   xmin_0:       0.000000
#   xmax:         99.500000
#   xmax_0:       99.500000
#   ymin:         0.000000
#   ymin_0:       0.000000
#   ymax:         99.500000
#   ymax_0:       99.500000
#   delta_x:      0.500000
#   delta_x_0:    0.500000
#   delta_y:      0.500000
#   delta_y_0:    0.500000
#   data minimum: -0.671084
#   global minimum: -1.843705
#   data maximum: 0.687022
#   global maximum: 1.904647
# data:
# x:      y:      Hz:
# -0.500 -0.500  0.00000
#  0.000 -0.500  0.00000
#  0.500 -0.500  0.00000
# ...

```

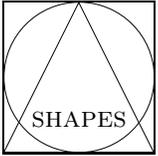
Gnuplot data files can be converted to GIF animations in the following three steps.

1. For each data file from the series to be animated, for example, Hz_0_005.dat set Gnuplot terminal to png, set Gnuplot output to Hz_0_005.png, define a title for the plot, and then `splot` the data file. Gnuplot will write an image in the png format on Hz_0_005.png.
2. Convert each png image to the gd2 image with `pngtogd2`, and then convert it to a gif file with `gd2togif`.
3. Assemble all the gif files into a movie by calling `gifsicle`, for example,

```
gifsicle --delay 20 --loopcount Hz_0_[0-9][0-9][0-9].gif > Hz_0_movie.gif
```

Three scripts are provided with the source in the `scripts` subdirectory: `dattomovie.sh`, `write_gif_files.sh`, and `pngtomovie.sh`. These scripts show how to automate the conversion process.

3 How to Use SHAPES



and auxiliary utilities are installed on the ANL Jazz cluster in the

```
/soft/apps/packages/photonic-packages/bin
```

directory. SHAPES users should therefore add this directory to their command search path on Jazz. It should not be necessary to do anything else to use SHAPES. The program is now invoked through scripts that configure the user environment automatically.

In order to add `photonic-packages` to their command search `PATH` on the Jazz cluster, users can edit the `.soft` file in their `$HOME` directory and insert the `photonic-packages` binary directory in front of their `@default` line. It is also a good idea to define the `WWW_BROWSER` at the same time, because the browser is needed to read ChomboVis documentation. More details are given in Section 4.3, page 72. In effect, the `.soft` file may look as follows:

```
(jlogin2) $ cat .soft
PATH += /soft/apps/packages/photonic-packages/bin
WWW_BROWSER = /soft/apps/packages/photonic-packages/bin/firefox
@default
(jlogin2) $
```

SHAPES also has been installed on the University of Chicago TeraGrid node, and plans are under way to install the package on the Pittsburgh Supercomputer Center's Cray XT3 system, called Bigben. These two systems are well worth using.

The University of Chicago cluster has 62 two-way IA-64/Madison SMP nodes and 8 TB GPFS. It also has access to some 200 TB wide area network GPFS. The cluster is lightly used at present, and one can easily get time on it.

The PSC Bigben is a 2068 CPU MPP with a very fast low-latency custom interconnect and Opteron nodes. This machine has 24 TB of Lustre file system and 2 TB memory. This system is used quite heavily, not so much by a large number of users, but for very large jobs.

3.1 A Simple Jazz Example

The simplest way to use SHAPES on the Jazz cluster is as follows. (Users must have already included the `photonic-packages` binary directory in their command search path. Also, see the note about managing the environment with the `.soft` file above.) The first step is to type

```
(jlogin1) $ interactive
qsub: waiting for job 533050.jmayor5.lcrc.anl.gov to start
qsub: job 533050.jmayor5.lcrc.anl.gov ready
```

```
(j91) $
```

`interactive` is a simple shell script in the `photonic packages` directory that gives the user an interactive session on a PBS-allocated node. All SHAPES runs should be always carried out under PBS and *never* on the front-end nodes. The user may have to wait a little after having issued this command, but eventually a node will be allocated. In this example we got node `j91` and an interactive shell forked on it.

The next step for the user is to go to a data directory on the parallel file system, PVFS. Every user can create a directory there from every node on the Jazz cluster. In this example, we have already created such a directory:

```
(j91) $ cd /pvfs/scratch/meglicki
(j91) $ mkdir test-run
(j91) $ cd test-run
(j91) $
```

We now copy an example input file from the SHAPES source directory to the PVFS data directory:

```
(j91) $ cp /soft/apps/packages/photonic-packages/src/Shapes-2.1/src/shapes.input .
(j91) $ ls
shapes.input
(j91) $
```

The SHAPES input file is discussed in detail in `shapes(5)`, which can be read, as can this document, from our nanophotonics Wiki.

At this stage we are ready to run SHAPES. To make sure we have the right binary in the command search path, we check it by typing

```
(j91) $ which shapes
/soft/apps/packages/photonic-packages/bin/shapes
(j91) $
```

Finally, we type the following.

```
(j91) $ shapes shapes.input
@Id: shapes.cpp,v 2.1 2006/01/20 19:11:34 meglicki Exp @
@Id: shapes.h,v 2.0 2006/01/11 15:24:10 gustav Exp @
@Id: io.cpp,v 2.0 2006/01/11 15:18:01 gustav Exp @
@Id: levels.cpp,v 2.0 2006/01/11 15:18:35 gustav Exp @
@Id: update.f,v 2.0 2006/01/11 15:22:21 gustav Exp @
```

Program developed by Zdzislaw (Gustav) Meglicki, Indiana University

```
print_level: my_level = 0
  levels[0]->domain = Box (0,0) to (199,199) type [(0,0)]
  levels[0]->x0 = 0, levels[0]->y0 = 0
  levels[0]->delta_x = 0.5, levels[0]->delta_y = 0.5
  levels[0]->imin = 0, levels[0]->imax = 199
  levels[0]->jmin = 0, levels[0]->jmax = 199
  levels[0]->xmin = 0, levels[0]->xmax = 99.5
  levels[0]->ymin = 0, levels[0]->ymax = 99.5
  levels[0]->time_e = 0, levels[0]->time_e_old = -0.125
  levels[0]->time_h = 0.0625, levels[0]->time_h_old = -0.0625
  levels[0]->delta_t = 0.125
.....
analyze_levels, level = 0, time_e = 2, time_h = 2.0625
  Dx_min = -4.52283e-07, -4.52283e-07
  Dx_max = 1.01719e-07, 1.01719e-07
  Dy_min = -2.28481e-07, -2.28481e-07
  Dy_max = 2.28481e-07, 2.28481e-07
  Ex_min = -4.52283e-07, -4.52283e-07
  Ex_max = 1.01719e-07, 1.01719e-07
  Ey_min = -2.28481e-07, -2.28481e-07
  Ey_max = 2.28481e-07, 2.28481e-07
  Hz_min = -2.26044e-07, -2.26044e-07
  Hz_max = 6.60138e-07, 6.60138e-07
  Energy_min = 0, 0
  Energy_max = 3.06609e-13, 3.06609e-13
.....
```

The output in this case is verbose. The program introduces itself by printing versions of every source file used in its compilation. This is important: if something changes, the user will be able to see that there is a new version, for example, information of `update.f` in the printout. Then the program builds its first, and in this case the last, level, level 0, and tells the user about it. Finally it commences the iterations. A dot is printed for every level 0 time step. Prior to the data dump, the data is analyzed, and minima and maxima of various fields are printed. This procedure is repeated until the program finishes the execution.

All this output can be switched off and the program made to run silently. But it is good to keep an eye on what the program does and especially on the field minima and maxima. The numbers should not diverge too far from 1. This is the advantage of doing all the computations in natural units.

When SHAPES is done, we get the following message.

```
main: done.

io.cpp:153: 47
io.cpp:142: 53
io.cpp:134: 80
levels.cpp:424: 55
io.cpp:127: 51
```

```

io.cpp:117: 53
es.cpp:127: 55
Total Unfreed from new,malloc, etc: 394 bytes
Vector 3Box: 28 bytes (0 Mb)
Vector 5Entry: 36 bytes (0 Mb)
Vector P7FluxBox: 48 bytes (0 Mb)
Vector P9FArrayBox: 140 bytes (0 Mb)
Vector P9LevelDataI7FluxBoxE: 24 bytes (0 Mb)
Vector d: 192 bytes (0 Mb)
Vector i: 4 bytes (0 Mb)
Vector j: 4 bytes (0 Mb)
BaseFab d: 11622012 bytes (11 Mb)
  IntVectSet allocation 136 bytes (0 Mb)
-----
Total Unfreed : 11623018 bytes (11 Mb)
peak memory usage: 11628974 bytes (11 Mb)
(j91) $

```

The program used only 11 MB memory, which is very little for an application of this type. A listing of the working directory shows us that 300 data files were dumped in it.

```

(j91) $ ls
Hz_0_001.dat  Hz_0_062.dat  Hz_0_123.dat  Hz_0_184.dat  Hz_0_245.dat
Hz_0_002.dat  Hz_0_063.dat  Hz_0_124.dat  Hz_0_185.dat  Hz_0_246.dat
...
Hz_0_055.dat  Hz_0_116.dat  Hz_0_177.dat  Hz_0_238.dat  Hz_0_299.dat
Hz_0_056.dat  Hz_0_117.dat  Hz_0_178.dat  Hz_0_239.dat  Hz_0_300.dat
Hz_0_057.dat  Hz_0_118.dat  Hz_0_179.dat  Hz_0_240.dat  shapes.input
Hz_0_058.dat  Hz_0_119.dat  Hz_0_180.dat  Hz_0_241.dat
Hz_0_059.dat  Hz_0_120.dat  Hz_0_181.dat  Hz_0_242.dat
Hz_0_060.dat  Hz_0_121.dat  Hz_0_182.dat  Hz_0_243.dat
Hz_0_061.dat  Hz_0_122.dat  Hz_0_183.dat  Hz_0_244.dat
(j91) $

```

These are the Gnuplot format data files. Each has a detailed header that tells us about the program that dumped the file, its version, time of dump, what's been dumped, and various run-related parameters. See Section 2.8 and the example provided.

At this stage, we can return to a Jazz front-end node:

```

(j91) $ exit
logout

qsub: job 533050.jmayor5.lcrc.anl.gov completed
(jlogin1) $

```

If the connection was made from an X11 workstation using secure shell, then a DISPLAY variable should be created on the Jazz front-end node:

```
(jlogin1) $ echo $DISPLAY
localhost:23.0
(jlogin1) $
```

If everything works as it should, then issuing the command `xclock` should bring a clock with Chicago time on the user's X11 display. If the clock does not come up, something is wrong. Assuming X11 works as it should, we change to the data directory and then call Gnuplot.

```
(jlogin1) $ xclock
^C
(jlogin1) $ cd /pvfs/scratch/meglicki/test-run
(jlogin1) $ gnuplot
```

```
  G N U P L O T
  Version 3.7 patchlevel 3
  last modified Thu Dec 12 13:00:00 GMT 2002
  System: Linux 2.4.29-rc2
```

```
Copyright(C) 1986 - 1993, 1998 - 2002
Thomas Williams, Colin Kelley and many others
```

```
Type 'help' to access the on-line reference manual
The gnuplot FAQ is available from
http://www.gnuplot.info/gnuplot-faq.html
```

```
Send comments and requests for help to <info-gnuplot@dartmouth.edu>
Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>
```

```
Terminal type set to 'x11'
gnuplot>
```

After defining x and y ranges for the plot, we are ready to view the data itself.

```
gnuplot> set xrange[20:80]
gnuplot> set yrange[20:80]
gnuplot> splot "Hz_0_170.dat" with lines
gnuplot>
```

This should bring a figure similar to that shown in Figure 6. The figure can be rotated by pressing the left mouse button and dragging it left and right. It can be also zoomed in and out by pressing the right mouse button and dragging the mouse.

To get an encapsulated PostScript copy of the figure for inclusion in a TeX document we do the following.

"Hz_0_170.dat" ———

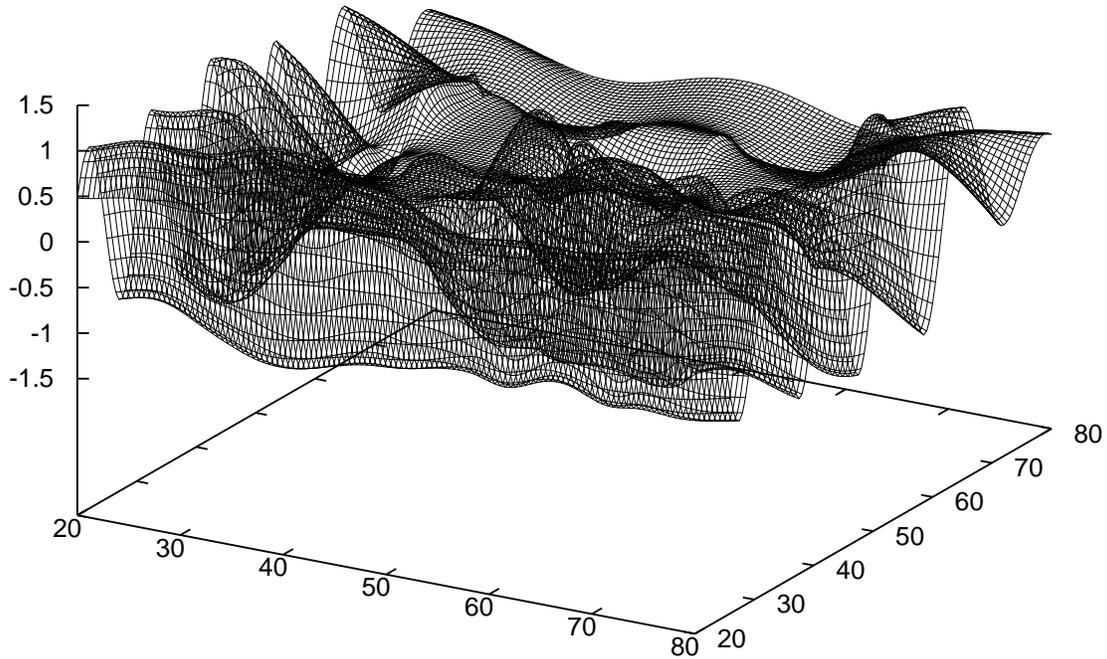


Figure 6: Data snapshot produced by SHAPES.

```
gnuplot> set terminal postscript eps
Terminal type set to 'postscript'
Options are 'eps noenhanced monochrome dashed defaultplex "Helvetica" 14'
gnuplot> set output "Hz_0_170.eps"
gnuplot> replot
gnuplot> exit
(jlogin1) $
```

To convert the images into a GIF animation, we request again a PBS node.

```
(jlogin1) $ interactive
qsub: waiting for job 533065.jmayor5.lcrc.anl.gov to start
qsub: job 533065.jmayor5.lcrc.anl.gov ready

(j91) $
```

We then switch to the data directory

```
(j91) $ cd /pvfs/scratch/meglicki/test-run/  
(j91) $
```

and run a script `write_gif_files.sh` in it:

```
(j91) $ write_gif_files.sh Hz 0 > Hz_0.plt
```

The script takes two arguments: the field name, `Hz` in this case, and the level number. Here we have only one level, 0.

The script reads the headers of `Hz_0_???.dat` data files first and extracts x_{\min} , x_{\max} , y_{\min} , y_{\max} , z_{\min} and z_{\max} , where in this case $z = H_z$, for the whole dumped data set (more precisely, here for all files with names `Hz_0_???.dat`) so that Gnuplot can scale the animation correctly. Then the script generates a Gnuplot command file on standard output. We capture it on `Hz_0.plt`.

The next step is to run Gnuplot on this file:

```
(j91) $ gnuplot Hz_0.plt  
(j91) $
```

But this still won't generate the GIF files. It used to, hence the name of the script (historic), but GIF support has since been removed from Gnuplot. Today the script generates PNG files instead.

To convert the PNG files produced by running Gnuplot on `Hz_0.plt` to GIF and to collate them into a movie we run a `pngtomovie.sh` script in the data directory. No additional command line arguments are needed here.

```
(j91) $ pngtomovie.sh  
Hz_0_001.png to ... Hz_0_001.gd2  
Hz_0_002.png to ... Hz_0_002.gd2  
...  
Hz_0_299.png to ... Hz_0_299.gd2  
Hz_0_300.png to ... Hz_0_300.gd2  
Hz_0_001.gd2 to ... Hz_0_001.gif  
Hz_0_002.gd2 to ... Hz_0_002.gif  
...  
Hz_0_300.gd2 to ... Hz_0_300.gif  
-rw-r--r-- 1 meglicki collab 1469648 Jan 12 2006 Hz_0_movie.gif  
(j91) $
```

And now we exit the shell on the `j91` and come back to `jlogin1`.

```
(j91) $ exit
logout
```

```
qsub: job 533065.jmayor5.lcrc.anl.gov completed
(jlogin1) $
```

Users should not hang idly onto the PBS-allocated nodes because their use of the nodes is measured (against their allowance) according to the wall clock, not according to the CPU time.

If the user has a high bandwidth connection to the Jazz cluster, the animation can be viewed directly from the Jazz front end nodes as follows:

```
(jlogin1) $ gifview --animate Hz_0_movie.gif
```

But it is perilous to try it over ISDN or a telephone connection.

3.2 A Simple TeraGrid Example

TeraGrid accounts come in two varieties: the so-called roaming accounts and machine-specific accounts. Both are acquired by connecting to

```
http://www.teragrid.org
```

... navigating toward “New Allocations” at the bottom of the page and eventually requesting a “Development Allocation” (DAC) account.

TeraGrid managers are liberal when it comes to issuing DAC accounts. One simply must present a brief research proposal to the TeraGrid DAC Committee to get a roaming allocation.

The machine-specific accounts are harder to get. One uses the same TeraGrid WWW machinery, but this time the proposal must specifically focus on the target machine. Special resources are in this category. The PSC Cray XT3, the SDSC IBM SP (called “DataStar”), and the NCSA SGI Altix (called “Cobalt”) are not accessible to the TeraGrid roaming accounts.

The University of Chicago has a 64-node Itanium 2 cluster on the TeraGrid that is normally used by two to six principals only. Hence, a high percentage of queued jobs gets to run as soon as they’re submitted. For example, right now 90% of the requested service units is in execution. For comparison, this percentage on the NCSA Tungsten cluster right now is 25%, and on the SDSC TeraGrid cluster it is 15%.

To run SHAPES on the University of Chicago’s TeraGrid cluster, one does the following after having connected to the cluster’s front-end node:

```
(tg-login2) $ interactive
qsub: waiting for job 236705.tg-master.uc.teragrid.org to start
qsub: job 236705.tg-master.uc.teragrid.org ready
```

```
-----
Begin PBS Prologue Fri Jan 13 14:41:22 CST 2006
Job ID:          236705.tg-master.uc.teragrid.org
Username:       gustav
```

```
Group:          allocate
Nodes:          tg-c040
End PBS Prologue Fri Jan 13 14:41:26 CST 2006
-----
```

```
(tg-c040) $
```

The allocated node is called `tg-c040`. The “c” in its name indicates that the node is IA64 and that it belongs to the computational pool. There are also IA32 nodes on the same cluster, meant to be used for visualization, and these are called something like `tg-v080`, with the “v”. One has to be careful to select the IA64 nodes for all SHAPES runs. This is done by issuing a special instruction to PBS about which more later. The command-script `interactive` selects IA64 nodes automatically.

Once we have connected to the node, the next step is to go to the data directory. On the University of Chicago cluster, the parallel file system is GPFS and the directory where it is mounted is `/disks/scratchgpfs1`. There we create the subdirectory `test-run` and `cd` into it:

```
(tg-c040) $ pwd
/home/gustav
(tg-c040) $ cd /disks/scratchgpfs1/gustav
(tg-c040) $ mkdir test-run
(tg-c040) $ cd test-run
(tg-c040) $
```

Next we copy the SHAPES input file from the source:

```
(tg-c040) $ cp ~/src/Mine/Current/Shapes-Jan12-1819/src/shapes.input .
(tg-c040) $
```

We are almost ready to run. The script that sets the environment and invokes the actual SHAPES binary is located at

```
(tg-c040) $ which shapes
/home/gustav/bin/shapes
(tg-c040) $
```

To run it, we type the following.

```
(tg-c040) $ shapes shapes.input
@Id: shapes.cpp,v 2.1 2006/01/20 19:11:34 meglicki Exp @
@Id: shapes.h,v 2.0 2006/01/11 15:24:10 gustav Exp @
@Id: io.cpp,v 2.0 2006/01/11 15:18:01 gustav Exp @
@Id: levels.cpp,v 2.0 2006/01/11 15:18:35 gustav Exp @
@Id: update.f,v 2.0 2006/01/11 15:22:21 gustav Exp @
Program developed by Zdzislaw (Gustav) Meglicki, Indiana University
```

```

print_level: my_level = 0
    levels[0]->domain = Box (0,0) to (199,199) type [(0,0)]
    levels[0]->x0 = 0, levels[0]->y0 = 0
    levels[0]->delta_x = 0.5, levels[0]->delta_y = 0.5
    levels[0]->imin = 0, levels[0]->imax = 199
    levels[0]->jmin = 0, levels[0]->jmax = 199
    levels[0]->xmin = 0, levels[0]->xmax = 99.5
    levels[0]->ymin = 0, levels[0]->ymax = 99.5
    levels[0]->time_e = 0, levels[0]->time_e_old = -0.125
    levels[0]->time_h = 0.0625, levels[0]->time_h_old = -0.0625
    levels[0]->delta_t = 0.125
.....
analyze_levels, level = 0, time_e = 2, time_h = 2.0625
    Dx_min = -4.52283e-07, -4.52283e-07
    Dx_max = 1.01719e-07, 1.01719e-07
    Dy_min = -2.28481e-07, -2.28481e-07
    Dy_max = 2.28481e-07, 2.28481e-07
    Ex_min = -4.52283e-07, -4.52283e-07
    Ex_max = 1.01719e-07, 1.01719e-07
    Ey_min = -2.28481e-07, -2.28481e-07
    Ey_max = 2.28481e-07, 2.28481e-07
    Hz_min = -2.26044e-07, -2.26044e-07
    Hz_max = 6.60138e-07, 6.60138e-07
    Energy_min = 0, 0
    Energy_max = 3.06609e-13, 3.06609e-13
.....

```

Now everything works the same as on the Jazz cluster, except that fewer diagnostics are written when the program finishes.

```

.....
analyze_levels, level = 0, time_e = 600, time_h = 600.062
    Dx_min = -1.51225, -6.80168
    Dx_max = 1.39867, 4.38859
    Dy_min = -1.30416, -5.56181
    Dy_max = 1.56239, 6.35418
    Ex_min = -0.750809, -2.85311
    Ex_max = 0.654999, 3.77963
    Ey_min = -0.871663, -2.98755
    Ey_max = 0.749242, 3.32821
    Hz_min = -0.208945, -1.8437
    Hz_max = 0.254674, 1.90465
    Energy_min = 0, 0
    Energy_max = 0.171262, 7.83189
.
main: done.

```

```
(tg-c040) $
```

Even though the University of Chicago cluster is an IA64 system and Jazz is an IA32 system, the University of Chicago cluster nodes are a little *slower* than Jazz nodes. But the University of Chicago cluster I/O is faster, so in effect the wall-clock execution time of SHAPES on the University of Chicago system is just a little less than the wall-clock execution time of SHAPES on Jazz. The example job discussed here took 5 minutes and 18 seconds on the Jazz cluster to complete and 4 minutes and 55 seconds on the University of Chicago cluster.

The next step is to build an animation. This is done the same way as on the Jazz cluster. Still on the PBS-allocated node, we do the following:

```
(tg-c040) $ write_gif_files.sh Hz 0 > Hz_0.plt
(tg-c040) $ gnuplot Hz_0.plt
(tg-c040) $ pngtomovie.sh
Hz_0_001.png to ... Hz_0_001.gd2
Hz_0_002.png to ... Hz_0_002.gd2
...
Hz_0_299.png to ... Hz_0_299.gd2
Hz_0_300.png to ... Hz_0_300.gd2
Hz_0_001.gd2 to ... Hz_0_001.gif
Hz_0_002.gd2 to ... Hz_0_002.gif
...
Hz_0_299.gd2 to ... Hz_0_299.gif
Hz_0_300.gd2 to ... Hz_0_300.gif
-rw-r--r--  1 gustav  allocate  1427917 2006-01-13 15:13 Hz_0_movie.gif
(tg-c040) $
```

and we're done.

Small file GPFS I/O on the University of Chicago cluster is exceptional. GPFS caches everything in memory, very aggressively. Because the data files produced by SHAPES in this example are quite small (less than 1.2 MB) and the image files are tiny (4.5 to 7.6 kB), almost all I/O operations are carried out against the memory buffers.

The last step to remember is to exit the shell on the PBS-allocated node.

```
(tg-c040) $ exit
logout

qsub: job 236705.tg-master.uc.teragrid.org completed
(tg-login2) $
```

To view the animation directly from the University of Chicago system one can invoke `gifview`. But a high bandwidth connection is needed for this to work:

```
(tg-login2) $ cd /disks/scratchgpfs1/gustav/test-run/
(tg-login2) $ gifview --animate Hz_0_movie.gif
(tg-login2) $
```

3.3 The SHAPES Input File

SHAPES activities are steered through the input file the name of which is passed to the program on the command line. The stand-alone UNIX man-page `shapes(5)` describes the format of the file, what should go where, and what is the meaning of various entries. Here we basically repeat this document for completeness, but with fewer details and with more examples.

The input file to SHAPES is a simple text file with various keywords, one per line, and their values separated by an equal sign. Lines that begin with the hash character are comments and are neglected by SHAPES.

The keywords may appear in any order, but it is a good practice to organize them in logical groups. And so the following groups are used by the current version of SHAPES:

chat, watch, level0, iterate, pml, signal, metal, tag, refine, spectral, output

The values on the right-hand side may be integer or real numbers or arrays of integers or real numbers. Real numbers may be entered as integers, but integers must not be entered as reals. Arrays are entered as list of integers or reals with entries separated by spaces.

3.3.1 The *chat* and *watch* Groups

The *chat* and *watch* groups specify the verbosity with which SHAPES chats about its activities. It is possible, however, to make SHAPES work silently without switching off possible error messages.

There are eight keywords in the *chat* group at present.

chat.print_versions (**int**) When this parameter is set to 1, the program prints RCS versions of all component source files as well as the name and institution of the author of the program. Setting it to 0 disables this output.

chat.chombo_verbose (**int**) When this parameter is set to 1 or 2, it activates chatting of C++ functions. When set to anything higher than 2, it may trigger Fortran messages too (see below). Setting it to 0 disables the chat.

chat.fortran_verbose (**int**) When set to 1 or higher this parameter activates chatting of Fortran subroutines or disables the chat when set to 0. When the program runs in parallel, setting this to 1 may trigger an enormous amount of output.

chat.print_level_0_domain (**int**) Once the level 0 has been constructed, SHAPES prints its geometry and other level 0-related parameters when this parameter is set to 1 or higher. Setting it to 0 disables this output.

chat.print_levels (**int**) When this parameter is set to 1, SHAPES prints information about every level that has been constructed. Setting this parameter to 1 sets *chat.print_level_0_domain* to 1, too.

chat.print_min_max (**int**) When this parameter is set to 1 or higher, SHAPES prints minimum and maximum values of all fields for all levels for the time slice at which data is dumped. Historical minimum and maximum values for the fields are printed, too. Setting this parameter to 0 disables this output. This is good for keeping an eye on the program in case the solution may diverge.

chat.print_actions (**int**) Setting this parameter to 1 makes SHAPES functions (the Chombo shell ones) print all they do. This can be useful in deciphering recursion, checking synchronization between levels and making sure that data flows between the levels at appropriate times and in the correct direction. This parameter is much the same in practice as *chat.chombo_verbose*.

chat.print_dots (int) When this parameter is set to 1 or higher, SHAPES prints a dot for every iteration. It may be useful to set it for interactive sequential runs, so that the user knows that something happens and how much work has been done so far. Whenever other diagnostic output is enabled, it is best to switch off this parameter.

As an example, let us look at the *chat* group in the input file used to run the job on the Jazz and University of Chicago clusters.

```

chat.chombo_verbose           = 0
chat.fortran_verbose         = 0
chat.print_levels            = 1
chat.print_versions          = 1
chat.print_min_max           = 1
chat.print_actions           = 0
chat.print_dots              = 1

```

Here we have asked SHAPES to print a message whenever it builds a level, to tell us about the RCS versions of the source files used in the compilation of the program, to print field minima and maxima as the computation unfolds, and to print dots for every level 0 iteration.

The keywords of the *chat* group do not discriminate. They make either all functions talk or none. Often we may wish to trace only one or a couple of specific functions, but not all. We can do so by using the *watch* group keywords. These are made of the string “**watch**” followed by a dot and the name of the function. For example, to make function `advance_e` talk, we need only to set

```

watch.advance_e = 1

```

The *watch* keywords take integers as arguments: 0 makes the functions run silently; 1, 2 and more make the functions talk. The higher the number, the more they talk. Setting the value of a *watch* parameter to 3 or higher will activate chatting by Fortran subroutines invoked from the watched functions.

Here is a list of all *watch* parameters that can be used in the SHAPES input file:

<i>watch.advance_e</i>	<i>watch.advance_h</i>	<i>watch.analyze_levels</i>
<i>watch.build_basic_fields</i>	<i>watch.build_cell_centered_fields</i>	<i>watch.build_distributions</i>
<i>watch.build_disjoint_box_layout</i>	<i>watch.build_fourier_fields</i>	<i>watch.build_level</i>
<i>watch.build_media_fields</i>	<i>watch.build_minmax</i>	<i>watch.convert_d_to_e</i>
<i>watch.copy_basic_fields</i>	<i>watch.copy_fourier_fields</i>	<i>watch.copy_media_fields</i>
<i>watch.dump_data</i>	<i>watch.effective_domain_size</i>	<i>watch.evaluate_energy</i>
<i>watch.exchange_basic_fields</i>	<i>watch.exchange_fourier_fields</i>	<i>watch.exchange_media_fields</i>
<i>watch.full_copy</i>	<i>watch.initialize_level_data</i>	<i>watch.inject_d</i>
<i>watch.inject_h</i>	<i>watch.interpolate_basic_fields</i>	<i>watch.interpolate_fourier_fields</i>
<i>watch.interpolate_media_fields</i>	<i>watch.main</i>	<i>watch.patch_basic_fields</i>
<i>watch.patch_fourier_fields</i>	<i>watch.patch_media_fields</i>	<i>watch.push_d</i>
<i>watch.push_h</i>	<i>watch.regrid</i>	<i>watch.tag_cells</i>
<i>watch.time_interpolate</i>	<i>watch.write_box_layout_data</i>	<i>watch.write_gnuplot_data</i>
<i>watch.write_tags_data</i>		

For example, to watch functions *push_d* and *push_h*, and to make fortran subroutines called from these functions talk, we set the following.

```

watch.push_d = 3
watch.push_h = 3

```

3.3.2 The *level0* Group

The *level0* group is used to specify the general geometry of the computational domain. The computational domain is always rectangular. We have to specify its length and width, as well as the number of grid divisions in each direction. Additionally we have to specify how the computational domain is going to be distributed among MPI processes if the program is going to run in parallel.

The geometry of the computational domain is specified in arbitrary units of length, δl . For example, suppose we are going to inject a plane harmonic wave of length 5000 \AA into the computational domain, which is going to be a square. We would like to resolve the wave on, say, 40 grid segments, and we would like to fit up to four full wavelengths into the total field region.

We can do so as follows. Let us make the computational region 100×100 units, and let us make the level 0 grid divisions in both directions $\Delta x_0 = \Delta y_0 = 0.5$. The unit of length is therefore going to be such that

$$2\Delta x_0 = 2\Delta y_0 = 1 \delta l$$

(which is why it is called a *unit*).

SHAPES does not assume that $\Delta x_0 = \Delta y_0$, but, frankly, it has never been tested for $\Delta x_0 \neq \Delta y_0$; there is no computational advantage in making Δx_0 and Δy_0 different. 

This choice of Δx_0 and Δy_0 results in a 200×200 grid. Now we want to resolve our 5000 \AA wave on 40 grid segments, which means that each $\Delta x_0 = \Delta y_0 = 0.5 = 5000 \text{ \AA}/40 = 125 \text{ \AA}$. The unit of length is therefore going to be

$$1 \delta l = 2 \times \Delta x_0 = 250 \text{ \AA},$$

and the wavelength itself is

$$40 \times \Delta x_0 = 40 \times 0.5 \delta l = 20 \delta l.$$

In order to fit four full wavelength in the total field region in each direction, the region must have the length and width of $20 \delta l \times 4 = 80 \delta l$ units. If we were to place the region centrally within the computational domain of $100 \delta l \times 100 \delta l$, its corners would be $(10 \delta l, 10 \delta l)$ and $(90 \delta l, 90 \delta l)$.

Observe that this number

$$1 \delta l = 250 \text{ \AA}$$

does not enter into the computations explicitly at any stage. The reason is that Maxwell electrodynamics is a conformal theory, even in the presence of media. Solutions obtained for any specific geometry and medium can be always reinterpreted, or scaled, to become solutions for something else, of similar shape and proportions but different size and different material properties, properties that are ultimately also described in terms of *our* unit of length. 

SHAPES can perform its computations on a multigrid. Here we define only the so called level 0 grid, that is, the *coarsest* grid. Higher-level grids (i.e., finer grids) are built by the program automatically. The user can specify where finer gridding should be used, if at all, by using keywords of the *tag* group. It is *not* necessary to use the multigrid, though.

We have eight keywords in the *level0* group with which we can convey to SHAPES all the information we have discussed above.

level0.nx (**int**) Number of level 0 cells in the *x* direction (minus 1).

level0.ny (**int**) Number of level 0 cells in the *y* direction (minus 1).

level0.nbx (**int**) Number of boxes in the x direction. SHAPES will group the level 0 grid cells into boxes and will distribute them among MPI processes if executed in parallel. SHAPES will try to make the boxes square, so the number of boxes in the x direction defines the average size of a box *both* in the x and in the y directions. If SHAPES is run sequentially, then it is best to set this number to 1. But it is OK to set it to something higher, say, 2 or 4. In this case SHAPES will group all level 0 grid cells into separate boxes and will go through all the motions of moving ghost data between the boxes, as if it was running on multiple CPUs. But all these operations will take place within memory of a single sequential process. This feature can be used, for example, in test runs before submitting a job to a multicomputer.

level0.x0 (**Real**) Value of the x coordinate (in δl) of the (0,0) point of the level 0 grid (i.e., x_0).

level0.y0 (**Real**) Value of the y coordinate (in δl) of the (0,0) point of the level 0 grid (i.e., y_0).

level0.delta_x (**Real**) Level 0 grid constant in the x direction (i.e., Δx_0 , in δl).

level0.delta_y (**Real**) Level 0 grid constant in the y direction (i.e., Δy_0 , in δl).

level0.time (**Real**) Value of initial time, t_0 (in $\delta t = \delta l$ because $c = 1$) that corresponds to the \mathbf{E} field within the level 0 grid. The \mathbf{H} field is defined on a time slice that is $\Delta t_0/2$ (Δt_0 , the leapfrog time step for level 0 is defined in the iteration group) ahead of the \mathbf{E} time slice.

Let us have a look at the settings in the `shapes.input` file used above.

```
#
# Level 0 geometry group
#
level0.nx           = 200
level0.ny           = 200
level0.nbx         = 1
level0.x0           = 0
level0.y0           = 0
level0.delta_x     = 0.5
level0.delta_y     = 0.5
level0.time        = 0
```

So we have that $x_0 = 0 \delta l$, $y_0 = 0 \delta l$, and $t_0 = 0 \delta t$. We also find that $\Delta x_0 = 0.5 \delta l$ and $\Delta y_0 = 0.5 \delta l$. The grid extends from $(x_0, y_0) = (0 \delta l, 0 \delta l)$ to $(x_0 + n_x \Delta x_0, y_0 + n_y \Delta y_0) = (100 \delta l, 100 \delta l)$ and is grouped into just one box. Let us observe that the actual number of cells in each direction is (201, 201), because the zero point adds one more cell in each direction.

This is the basic cell-centered grid. SHAPES carries out its computations on a staggered grid. Only the H_z field is attached to cell centers. The electric fields are attached to cell sides. Consequently, the electric field grids are going to be larger by one either in the x or in the y direction, depending on whether it's an E_x grid or an E_y grid. SHAPES builds separate, precisely sized, staggered grids for the fields it operates on.

3.3.3 The *iterate* Group

The *iterate* group tells SHAPES how the level 0 should be iterated, that is, what the iteration time step should be, how many iterations should be performed altogether, and how often field images should be dumped.

SHAPES performs its computations in the natural units in which the speed of light in vacuum, c , is 1. This means that in one unit of time, δt , the wave front is going to propagate by one unit of length, δl , in vacuum. If we were to reuse the example discussed in the previous section, where we *decreed* the unit of length to be 250 \AA , we would end up with the unit of time of

$$\delta t = \frac{\delta l}{c} = \frac{250 \text{ \AA}}{c} = \frac{250 \times 10^{-10} \text{ m}}{2.997925 \times 10^8 \text{ m/s}} = 8.3391 \times 10^{-17} \text{ s}. \quad (25)$$

We will need this number to convert resonance frequencies for the media, given in hertz ($1 \text{ Hz} = 1 \text{ s}^{-1}$ to our units, $1/\delta t$), but it is not needed to define iteration parameters of this section.

Instead, the iteration time step Δt_0 is specified by telling SHAPES in how many time steps the wave front is going to traverse a single grid cell in the x direction, namely, how many iterations are needed to push the wavefront through a distance of Δx_0 . This number is called a *level 0 stride*. And so, if we set, say, a level 0 stride to 4, the time step Δt_0 will be chosen so that Δx_0 (in the example discussed above, this was 125 \AA) will be traversed in four level 0 iterations. In other words $\Delta t_0 = \Delta x_0/4$. In traditional units this would be

$$\Delta t_0 = \frac{\Delta x_0}{4c} = \frac{0.5 \delta l}{4c} = \frac{\delta l}{8c} = \frac{\delta t}{8} = \frac{8.3391 \times 10^{-17} \text{ s}}{8} = 1.04 \times 10^{-17} \text{ s}.$$

It is not always the best idea to make the time step as long as the stability criterion for vacuum electrodynamics lets it be, namely, $\Delta t_0 = \Delta x_0/\sqrt{n}$, where n is the number of dimensions. The reason is that this number does not take into account stability conditions for the auxiliary differential equations, or, for this matter, for the PMLs. In the presence of a multigrid, this number may have to be shortened too, because the vacuum electrodynamics CFL criterion brakes on the multigrid boundaries.

There are four keywords in this group:

iterate.level0.stride (**Real**) This is the level 0 *stride* discussed above.

iterate.level0.image_frequency (**int**) This parameter tells SHAPES how often to dump images. SHAPES will dump images every *iterate.level0.stride* times *iterate.level0.image_frequency* time steps. For example, if *iterate.level0.stride* is 4 and *iterate.level0.image_frequency* is 2, the image will be dumped every $4 \times 2 = 8$ time steps. Because in four time steps the wave front propagates by one Δx_0 , the wave front will be shifted by $2 \times \Delta x_0$ between adjacent images. In the example discussed above, the wave front shift between adjacent images will be by one unit of length.

iterate.level0.number_of_steps (**int**) This number tells SHAPES how many time steps to perform altogether.

iterate.use_substep (**int**) This parameter matters only when the computations are carried out on a multigrid. When it is set to 0, SHAPES will use the same time step for each multigrid level. This, of course, implies that in this case the time step should be chosen based on the finest grid-level spacing requested. When it is set to 1, SHAPES will use a shorter time step for finer grid levels, as discussed in Section 2.6.1. The space refinement ratio between adjacent grid levels is 2; and, in this case, the time refinement ratio between adjacent grid levels is 3. This latter ratio is required to synchronize \mathbf{E} and \mathbf{H} time slices between adjacent levels and across the whole multigrid.

Let us look at the *iterate* group in `shapes.input` provided with the program source:

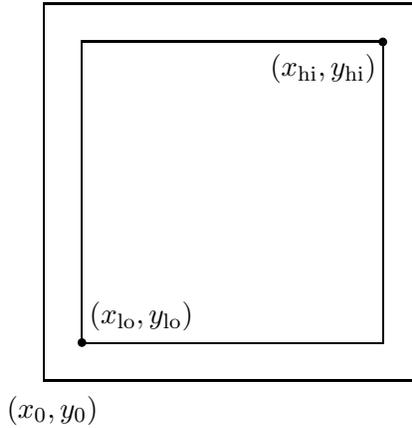


Figure 7: Definition of the PML region boundary. The outer box represents the whole computational domain. The PML region is between the inner and the outer box.

```
#
# Iteration group
#
iterate.level0.stride           = 4
iterate.level0.image_frequency = 4
iterate.level0.number_of_steps = 4800
iterate.use_substep             = 0
```

The stride of 4 means that four time steps will be needed to push the wave front through the distance of $\Delta x_0 = \delta l/2$, hence $\Delta t_0 = \Delta x_0/4 = \delta l/8$. The images will be dumped every $4 \times 4 = 16$ time steps; in other words, the wave front will have moved by $4\Delta x_0 = 2\delta l$ between the snapshots. We are going to make 4,800 time steps altogether, thus pushing the wave front through the distance of $1,200\Delta x_0 = 600\delta l$.

3.3.4 The *pml* Group

The PML ABCs are characterized just by specifying the boundary of the PML region. The user needs to specify the lower-left and the upper-right corners of the boundary box, as shown in Figure 7. SHAPES takes care of all the rest. The keywords are as follows:

pml.x_lo (Real)

pml.y_lo (Real) The *x* and *y* coordinates (in δl) of the lower-left corner of the PML boundary box.

pml.x_hi (Real)

pml.y_hi (Real) The *x* and *y* coordinates (in δl) of the upper-right corner of the PML boundary box.

Consider again the example discussed in Section 3.3.2. There we restricted the total field region to a box defined by the $(10\delta l, 10\delta l)$ and $(90\delta l, 90\delta l)$ corner points. We can therefore define our PML region by the $(5\delta l, 5\delta l)$ and $(95\delta l, 95\delta l)$ corner points (the PML region is *outside* the box so defined). This is still within the $(0\delta l, 0\delta l)$ and $(100\delta l, 100\delta l)$ computational domain, and it yields ten grid cells to attenuate the signal on each boundary, which should be sufficient because the incident signal is usually attenuated within just seven cells or so. The distance between the PML boundary and the total field region boundary is $5\delta l$, which is again ten grid cells:

```

#
# PML group
#
pml.x_lo           = 5
pml.y_lo           = 5
pml.x_hi           = 95
pml.y_hi           = 95

```

File `shapes.input` specifies broader margins. Here we have

```

#
# PML group
#
pml.x_lo           = 10
pml.y_lo           = 10
pml.x_hi           = 90
pml.y_hi           = 90

```

and the total field region is then restricted (in the *signal* group) to the $(20\delta l, 20\delta l)(80\delta l, 80\delta l)$ box. This gives us twenty cells to attenuate the incident signal, which is sufficient, and twenty cells between the total field region boundary and the PML boundary. This corridor is wide enough to carry out analysis of the scattered field, for example, to evaluate the far field limit. SHAPES doesn't do this yet, mostly because we're interested in the near field region within this project, but this feature is definitely on the drawing board.

3.3.5 The *signal* Group

Signals that SHAPES can inject into the total field region have been discussed in Section 2.4, pages 9-16. See, in particular, Table 1 on page 10. The mode numbers are the numbers used in the SHAPES input file to select a particular signal. The signal parameters are then provided as needed, together with the specification for the total field region. The total field region must be located *outside* the PML region, that is, entirely *within* the inner box in Figure 7.

The keywords to do all this are as follows:

signal.x_lo (Real)

signal.y_lo (Real) The x and y coordinates (in δl) of the lower-left corner of the total field region.

signal.x_hi (Real)

signal.y_hi (Real) The x and y coordinates (in δl) of the upper-right corner of the total field region.

signal.mode (int) The type of signal to be injected, see Table 1. Avoid modes 0 and 1, which are here for "historic" reasons only. Be careful with mode 6, and ensure that the effective λ is not going to become 0 (or just too small) anywhere throughout the whole computation. It is better to use a tanh chirp because it provides natural bounds for λ at both ends. Test the signal parameters using Gnuplot, as discussed in Section 2.4, prior to running SHAPES. The following Gnuplot command, for example, will work for the tanh ramped harmonic wave:

```

set sample 800,800
plot [z=-200:200] [-1.5:1.5] \
  f(z) = 0.5 * (1 - tanh(a * z)) * sin(6.2831853 * z / l), \
  a = 0.02, l = 20, f(z)

```

signal.t0 (Real) The signal delay time t_0 (in δt), as in the formula $\zeta = n_x(x - x_0) + n_y(y - y_0) - (t - t_0)$. SHAPES determines x_0 and y_0 automatically, depending on the direction from which the signal is injected and they default to x_{lo} , x_{hi} , y_{lo} , or y_{hi} of the total field region box as the case may be. The user can choose t_0 , though. This parameter should be chosen so that at $t = 0$, the center or the ramp of the signal is sufficiently far from the total field region to make zero a good initial solution within the total field region.

signal.lambda (Real)

signal.sigma (Real)

signal.alpha (Real)

signal.beta (Real) These are the signal parameters λ , σ , α , and β as used in Table 1.

signal.vx (Real)

signal.vy (Real) These are the components v_x and v_y of a vector \mathbf{v} that points in the direction of signal propagation. This vector does not have to be normalized, because **shapes** normalizes it internally anyway, evaluating

$$\begin{aligned}
 n_x &= \frac{v_x}{\sqrt{v_x^2 + v_y^2}} \\
 n_y &= \frac{v_y}{\sqrt{v_x^2 + v_y^2}}.
 \end{aligned}$$

The file `shapes.input` discussed above has the following signal specification:

```

#
# Signal injection group
#
signal.x_lo      = 20
signal.y_lo      = 20
signal.x_hi      = 80
signal.y_hi      = 80
signal.mode      = 7
signal.t0        = 300
signal.lambda    = 10
signal.sigma     = 60
signal.alpha     = 0
signal.beta      = 0.0012
signal.vx        = 0
signal.vy        = 1

```

The total field region is therefore confined to the box defined by the two corners, $(20 \delta l, 20 \delta l)$ and $(80 \delta l, 80 \delta l)$. The signal to be injected is number 7, a Gaussian envelope quadratic chirp. It will be delayed by $300 \delta t$, meaning that the peak of the Gaussian envelope will not enter the total field region until $t = 300 \delta t$. We also find that $\lambda = 10$, $\sigma = 60$, and $\beta = 0.0012$. The signal is injected in the y direction.

If we wanted to inject a signal such as the one shown in Figure 1 on page 11, the specifications could be as follows.

```

signal.mode      = 10
signal.t0        = 300
signal.lambda    = 20
signal.sigma     = 60
signal.alpha     = 10
signal.beta      = 0.15
signal.vx        = 0
signal.vy        = 1

```

3.3.6 The *metal* Group

The *metal* group is at the heart of the input. Here we define the media, their characteristics and layout. This group therefore naturally splits into two subgroups. The first one, called *metal.media*, is used to specify the metals, and the second one, called *metal*, is used to specify the distribution.

The medium, as we discussed in Section 2.2, pages 5-7, is described in terms of a unified Drude-Lorentz model with an arbitrary number of resonances given by equation (1), page 5. Each medium is characterized by ϵ_∞ followed by an array of ϵ_k , α_k , δ_k , and ω_k , where k runs through the resonances. Of these parameters all are *dimensionless* with the exception of ω_k , which should be specified in terms of $1/\delta t$.

This specification is easy to do. Observe that

$$\omega_{\delta t} \frac{1}{\delta t} = \omega_{\delta t} \frac{1}{\delta t} \frac{\text{s}}{\text{s}} = \omega_{\delta t} \frac{\text{s}}{\delta t} \frac{1}{\text{s}} = \omega_s \frac{1}{\text{s}}.$$

Note that both $\omega_{\delta t}$ and ω_s are unitless quantities, just pure numbers. So, we find that

$$\omega_s = \omega_{\delta t} \frac{\text{s}}{\delta t} \quad \text{or} \quad \omega_s \frac{1}{\text{s}} = \omega_{\delta t} \frac{1}{\delta t},$$

which is where we could have started. Hence

$$\omega_{\delta t} = \omega_s \frac{\delta t}{\text{s}}. \tag{26}$$

In other words, to convert an ω_s given in the units of hertz ($1 \text{ Hz} = 1 \text{ s}^{-1}$) to an $\omega_{\delta t}$ given in the units of $1/\delta t$, we need only to multiply ω_s by δt expressed in seconds. For example, we found in Section 3.3.3 that $\delta t = 8.3391 \times 10^{-17} \text{ s}$ (cf. equation (25) on page 42). So, if, say, we have $\omega_D = 1.3257 \times 10^{16} \text{ Hz}$, then ω_D in the units of $1/\delta t$ is going to be

$$\omega_{D\delta t} = \omega_{D_s} \times \frac{\delta t}{\text{s}} = 1.3257 \times 10^{16} \frac{1}{\text{s}} \times 8.3391 \times 10^{-17} \text{ s} = 1.106.$$

This is a much easier number to work with than 1.3257×10^{16} , and it comes out unit-less, too.

Observe that the result would be different if we thought of a principal harmonic wave in Section 3.3.2 as being, say, 4000 \AA rather than 5000 \AA and if we sought to squeeze only two full wavelengths into

the computational domain rather than four. These considerations would yield a different δl , a different $\delta t = \delta l/c$, and hence a different $\omega_D \delta t$.

We again find that since Maxwell equations do not define a unit of length, even in the presence of media, their solutions can always be scaled in various ways to fit various experimental situations.

The keywords used to define the media are as follows

metal.media.number_of_media (**int**) This parameter tells SHAPES how many different metals we are going to have in the system.

metal.media.number_of_terms (**int**) This parameter tells SHAPES how many resonance terms we are going to use for each metal. All metals must be defined by the same number of resonances. If we have two metals and one is defined by 3 resonances, whereas the other one by 2 resonances only, then the second metal must be still entered in terms of 3 resonances, but the last resonance should be set to zero.

The numbers that characterize the metals must now be entered as vectors of reals:

metal.media.epsilon_infty (**vector of Real**) The values of ϵ_∞ for each of the metals must be entered in a single line and separated by spaces.

metal.media.omega (**vector of Real**) The values of ω_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals, enter the numbers for the first metal, then for the second metal, and so on.

metal.media.alpha (**vector of Real**) The values of α_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals, enter the numbers for the first metal, then for the second metal, and so on.

metal.media.delta (**vector of Real**) The values of δ_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals, enter the numbers for the first metal, then for the second metal, and so on.

metal.media.epsilon (**vector of Real**) The values of ϵ_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals, enter the numbers for the first metal, then for the second metal, and so on.

Let us consider the following example. Suppose we get a metal model given by

$$\mathbf{D}(\omega) = \epsilon_0 \left(\epsilon_\infty - \frac{\omega_D^2}{\omega^2 - i\Gamma_D \omega} + \frac{g_1 \omega_1^2 \Delta \epsilon}{\omega_1^2 + 2i\omega \delta_1 - \omega^2} + \frac{g_2 \omega_2^2 \Delta \epsilon}{\omega_2^2 + 2i\omega \delta_2 - \omega^2} \right) \mathbf{E}(\omega). \quad (27)$$

First we need to squeeze this model into formula (1), page 5, namely,

$$\mathbf{D}(\omega) = \left(\epsilon_\infty + \sum_k \frac{\epsilon_k}{\alpha_k + i2\delta_k(\omega/\omega_k) - (\omega/\omega_k)^2} \right) \mathbf{E}(\omega). \quad (28)$$

We already got rid of ϵ_0 by absorbing it into \mathbf{E} and \mathbf{D} . Now consider one of the two Lorentz terms in equation (27). It can be rewritten as follows:

$$\frac{g_k \omega_k^2 \Delta \epsilon}{\omega_k^2 + 2i\omega \delta_k - \omega^2} = \frac{g_k \Delta \epsilon}{1 + 2i \frac{\delta_k}{\omega_k} \frac{\omega}{\omega_k} - \left(\frac{\omega}{\omega_k} \right)^2}. \quad (29)$$

Let us match it against

$$\frac{\epsilon_k}{\alpha_k + 2i\delta_k \frac{\omega}{\omega_k} - \left(\frac{\omega}{\omega_k}\right)^2}. \quad (30)$$

We can transform the term from (29) into the term from (30) with the following substitutions:

$$\begin{aligned} 1 &\rightarrow \alpha_k \\ \frac{\delta_k}{\omega_k} &\rightarrow \delta_k \\ g_k \Delta\epsilon &\rightarrow \epsilon_k, \end{aligned}$$

where the terms on the left-hand side come from equation (27) and the terms on the right-hand side come from equation (28). In the process, the units of δ_k on the left-hand side cancel out, so that all quantities on the right hand-side are unitless.

The Drude term in equation (27) can be rewritten as follows:

$$-\frac{1}{\left(\frac{\omega}{\omega_D}\right)^2 - i\frac{\Gamma_D}{\omega_D} \frac{\omega}{\omega_D}} = \frac{1}{i\frac{\Gamma_D}{\omega_D} \frac{\omega}{\omega_D} - \left(\frac{\omega}{\omega_D}\right)^2}.$$

Let us compare this against

$$\frac{\epsilon_k}{\alpha_k + 2i\delta_k \frac{\omega}{\omega_k} - \left(\frac{\omega}{\omega_k}\right)^2}.$$

Again, we find that we can match both sides with the following substitutions:

$$\begin{aligned} 0 &\rightarrow \alpha_k \\ \frac{\Gamma_D}{2\omega_D} &\rightarrow \delta_k \\ 1 &\rightarrow \epsilon_k \end{aligned}$$

where quantities on the left-hand side are from equation (27) and quantities on the right-hand side are from equation (28).

Given the following values for the parameters found in (27),

$$\begin{aligned} \epsilon_\infty &= 2.36461 \\ \omega_D &= 1.3257 \times 10^{16} \\ \Gamma_D &= 1.136268 \times 10^{14} \\ \omega_1 &= 6.6457856 \times 10^{15} \\ g_1 &= 0.266286 \\ \delta_1 &= 4.2487 \times 10^{14} \\ \omega_2 &= 7.863688 \times 10^{15} \\ g_2 &= 1 - g_1 \\ \delta_2 &= 8.316865 \times 10^{14} \\ \Delta\epsilon &= 1.1830916 \end{aligned}$$

we obtain the following numbers to be typed into the SHAPES input file.

For the Drude term we get

$$\begin{aligned}\alpha &= 0 \\ \delta &= \frac{\Gamma_D}{2\omega_D} = \frac{1.136268 \times 10^{14}}{2 \times 1.3257 \times 10^{16}} = 0.004286 \\ \epsilon &= 1.\end{aligned}$$

For the first Lorentz term we have

$$\begin{aligned}\alpha &= 1 \\ \delta &= \frac{\delta_1}{\omega_1} = \frac{4.2487 \times 10^{14}}{6.6457856 \times 10^{15}} = 0.06393 \\ \epsilon &= g_1 \Delta\epsilon = 0.266286 \times 1.1830916 = 0.31504.\end{aligned}$$

And finally for the second Lorentz term we have

$$\begin{aligned}\alpha &= 1 \\ \delta &= \frac{\delta_2}{\omega_2} = \frac{8.316865 \times 10^{14}}{7.863688 \times 10^{15}} = 0.10576 \\ \epsilon &= g_2 \Delta\epsilon = (1 - 0.266286) \times 1.1830916 = 0.86805.\end{aligned}$$

We now need to evaluate the omegas, or resonance frequencies. We multiply them by $\delta t = 8.3391 \times 10^{-17}$ s, and get the following:

$$\begin{aligned}\omega_D &= 1.3257 \times 10^{16} \frac{1}{\text{s}} \rightarrow 1.3257 \times 10^{16} \frac{1}{\text{s}} \times 8.3391 \times 10^{-17} \text{ s} = 1.106 \\ \omega_1 &= 6.6457856 \times 10^{15} \frac{1}{\text{s}} \rightarrow 6.6457856 \times 10^{15} \frac{1}{\text{s}} \times 8.3391 \times 10^{-17} \text{ s} = 0.554 \\ \omega_2 &= 7.863688 \times 10^{15} \frac{1}{\text{s}} \rightarrow 7.863688 \times 10^{15} \frac{1}{\text{s}} \times 8.3391 \times 10^{-17} \text{ s} = 0.656.\end{aligned}$$

Now let us enter these numbers into the SHAPES input file. The entry for this single metal would look as follows:

```
#
# The media group
#
metal.media.number_of_media = 1
metal.media.number_of_terms = 3
metal.media.epsilon_infty   = 2.36461
metal.media.omega           = 1.106    0.554    0.656
metal.media.alpha           = 0        1        1
metal.media.delta           = 0.004286 0.06393 0.10576
metal.media.epsilon         = 1        0.31504 0.86805
```

What if we have two metals, for example, one as defined above, and the other one by a single Drude term only? Then the input would look as follows:

```

#
# The media group
#
metal.media.number_of_media = 2
metal.media.number_of_terms = 3
#-----
#
#                metal 1                metal 2
#-----
metal.media.epsilon_infty = 2.36461                2.36461
metal.media.omega         = 1.106    0.554    0.656    1.106    0 0
metal.media.alpha         = 0        1        1        0        0 0
metal.media.delta         = 0.004286 0.06393 0.10576 0.004286 0 0
metal.media.epsilon       = 1        0.31504 0.86805 1        0 0
#-----

```

(When we describe media layout, we will refer to metal 1 as “1” and to metal 2 as “2” in the appropriate medium specification arrays.)

Will this work? Killing ω_2 , α_2 , and δ_2 will result in the following update for field \mathbf{S}_2 (cf. Section 2.2, equation (5) on page 6):

$$\mathbf{S}_k \leftarrow 2\mathbf{S}_k - \mathbf{S}_{k\text{old}}. \quad (31)$$

If \mathbf{S}_k and $\mathbf{S}_{k\text{old}}$ are zero for starters, which they are, then \mathbf{S}_k stays zero, and therefore these terms do not alter \mathbf{E} . An observant reader will notice that it is sufficient to kill only ω_k to achieve this effect. An observant reader will also notice that if $\mathbf{S}_k = \mathbf{S}_{k\text{old}}$, then equation (31) turns into $\mathbf{S}_k \leftarrow \mathbf{S}_k$, meaning that \mathbf{S}_k stays constant.

Now, let us turn to the remainder of the *metal* group and see how to define media layout.

SHAPES lets users define layouts by a combination of rectangles (called *boxes* for historical reasons), circles (called *cylinders* for historical reasons), rings, ellipses, and triangles. Multiple occurrences of these figures may be placed within the total field region. The figures may overlap. Additionally, cuts may be made in a layout specified by the imposition of *masks*, which are defined also in terms of mask rectangles, mask circles, mask rings, mask ellipses, and mask triangles.

With each layout object—a rectangle, a circle, a ring, an ellipse, or a triangle—we associate a metal from the list constructed above. It is a good idea *not* to associate different metals with overlapping figures, although the algorithm will handle such a circumstance, letting metals associated with circles *win* over metals associated with rectangles and metals associated with triangles win over metals associated with circles. But this ordering may change in future releases of the program.

The layout figures are defined by the means of the following keywords.

metal.boxes.number (int) Total number of boxes to be declared explicitly.

metal.boxes.x_lo (vector of Real)

metal.boxes.y_lo (vector of Real) The x_{lo} and y_{lo} coordinates (in δl) of the lower-left corner of each box. The coordinates for successive boxes should be listed in the same lines and separated with spaces.

metal.boxes.x_hi (vector of Real)

metal.boxes.y_hi (vector of Real) The x_{hi} and y_{hi} coordinates (in δl) of the upper-right corner of each box. The coordinates for successive boxes should be listed in the same lines and separated with spaces.

metal.boxes.medium (vector of int) The number of the medium associated with a given box. The numbers for successive boxes should be listed in the same line and separated with spaces.

metal.boxes.repeats (vector of int) Number of times each of the boxes specified explicitly should be repeated.

Every individual figure defined on a SHAPES input file can be repeated an arbitrary number of times by translation. All properties of the figure are repeated including the medium. The number of repeats should not be confused with *metal.boxes.number*. The latter should be thought of as the number of columns in the input file definitions rather than the ultimate number of boxes that will be produced by the definitions. Setting *repeats* to zero or omitting them altogether disables repeats.

metal.boxes.vx (vector of Real)

metal.boxes.vy (vector of Real) Translation vectors (v_x, v_y) to be used in repeats, if *repeats* have been requested. One for each box specified explicitly.

For example, consider the layout displayed in figure 8. The outermost box, the one that encloses

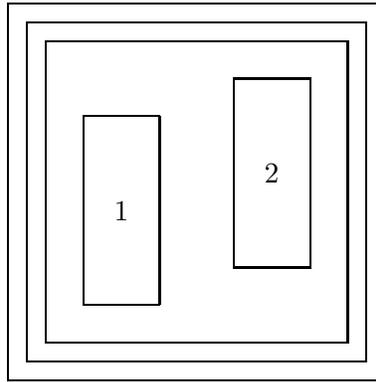


Figure 8: Two metal-filled boxes defined by the specifications provided in the example.

everything, is the whole computational domain defined by the two points $(x_{lo}, y_{lo}) = (0, 0)$ and $(x_{hi}, y_{hi}) = (100, 100)$. Within the domain we have first the PML region given by $(x_{lo}, y_{lo}) = (5, 5)$ and $(x_{hi}, y_{hi}) = (95, 95)$ and then the total field region given by $(x_{lo}, y_{lo}) = (10, 10)$ and $(x_{hi}, y_{hi}) = (90, 90)$. Within the total field region we have two metal boxes. The one on the left is given by $(x_{lo}, y_{lo}) = (20, 20)$ and $(x_{hi}, y_{hi}) = (40, 70)$ and is filled with metal number 1. The one on the right is given by $(x_{lo}, y_{lo}) = (60, 30)$ and $(x_{hi}, y_{hi}) = (80, 80)$ and is filled with metal number 2. Here is what the input file specification of both boxes looks like.

```
metal.boxes.number = 2
#-----
#                box 1  box 2
#-----
metal.boxes.x_lo  = 20    60
metal.boxes.y_lo  = 20    30
metal.boxes.x_hi  = 40    80
```

```

metal.bboxes.y_hi      =   70      80
metal.bboxes.medium    =     1       2
#-----

```

If the boxes were to overlap then the metal of the second box would win over the metal of the first box in the overlap region.

If both boxes were filled with an identical metal, we could use the *repeats* parameter and a translation vector to specify them because they're both of the same shape. In this case the specification would look as follows.

```

metal.bboxes.number    =     1
#-----
#                       box 1
#-----
metal.bboxes.x_lo      =     20
metal.bboxes.y_lo      =     20
metal.bboxes.x_hi      =     40
metal.bboxes.y_hi      =     70
metal.bboxes.medium    =     1
metal.bboxes.repeats   =     1
metal.bboxes.vx         =     40
metal.bboxes.vy         =     10
#-----

```

Let us note that this time *metal.bboxes.number* is 1, not 2, even though we end up with two boxes in the end. This is because we provide explicit specifications for one box only, and then it gets repeated. Also, even though we end up with two boxes, there is only one repeat because one box has already been specified explicitly.

metal.cylinders.number (int) Total number of cylinders (circles) to be declared explicitly.

metal.cylinders.xc (vector of Real)

metal.cylinders.yc (vector of Real) The x_c and y_c coordinates (in δl) of the center of each circle. Entries that correspond to successive circles should be typed in the same lines and separated with spaces.

metal.cylinders.rc (vector of Real) Radius r_c (in δl) of each circle. Entries that correspond to successive circles should be typed in the same line and separated with spaces.

metal.cylinders.medium (vector of int) Number of the medium associated with a given circle. The numbers for successive circles should be listed in the same line and separated with spaces.

metal.cylinders.repeats (vector of int) Number of times each of the circles specified explicitly should be repeated.

metal.cylinders.vx (vector of Real)

metal.cylinders.vy (vector of Real) Translation vectors (v_x, v_y) to be used in the repeats, one for each circle specified explicitly.

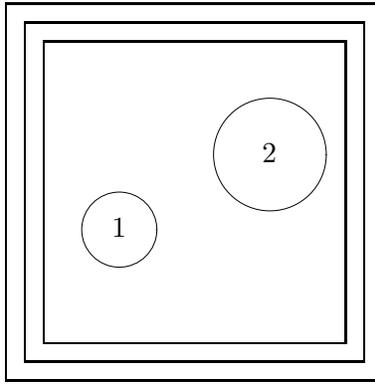


Figure 9: Two metal-filled circles defined by the specifications provided in the example.

For example, consider the layout displayed in Figure 9. Here we have two circles. The one on the left is centered on $(x_c, y_c) = (30, 40)$, its radius r_c is 10, and... is filled with metal number 1. The circle on the right is centered on $(x_c, y_c) = (70, 60)$, its radius r_c is 15, and... is filled with metal number 2. Here is what the SHAPES specification of this layout looks like.

```
metal.cylinders.number = 2
#-----
#           circle 1   circle 2
#-----
metal.cylinders.xc    = 30       70
metal.cylinders.yc    = 40       60
metal.cylinders.rc    = 10       15
metal.cylinders.medium = 1       2
#-----
```

We cannot define this pattern by repeats, not only because the circles are filled with different metals, but also because the circles have different radii.

A single ring could be defined by a combination of a circle and a smaller circular mask, but this would not work for a set of concentric circles. For this reason SHAPES provides explicit keywords for rings.

metal.rings.number (int) Total number of rings to be declared explicitly.

metal.rings.xc (vector of Real)

metal.rings.yc (vector of Real) x and y coordinates of ring centers. Numbers for successive rings should be listed in the same line and separated with spaces.

metal.rings.r_lo (vector of Real)

metal.rings.r_hi (vector of Real) Low and high radii of the rings. Numbers for successive rings should be listed in the same line and separated with spaces.

metal.rings.medium (vector of int) Number of the medium that is associated with a given ring. Entries for successive rings should be listed in the same line and separated with spaces.

metal.rings.repeats (vector of int) Number of times each of the rings specified explicitly should be repeated.

metal.rings.vx (vector of Real)

metal.rings.vy (vector of Real) Translation vectors (v_x, v_y) to be used in the repeats. One for each ring specified explicitly.

Ellipses in SHAPES can be oriented any way, unlike ellipses in T_EX, or in pic, which can be oriented horizontally or vertically only.

metal.ellipses.number (int) Total number of ellipses to be declared explicitly.

metal.ellipses.xa (vector of Real)

metal.ellipses.ya (vector of Real)

metal.ellipses.xb (vector of Real)

metal.ellipses.yb (vector of Real) x and y coordinates of the two foci of the ellipses. The numbers for successive ellipses should be listed in the same lines and separated with spaces.

metal.ellipses.sum (vector of Real) The sum of distances between a point on the circumference of an ellipse and its foci, that is, for a point (x_c, y_c) on the circumference, the sum is

$$\sqrt{(x_c - x_a)^2 + (y_c - y_a)^2} + \sqrt{(x_c - x_b)^2 + (y_c - y_b)^2}.$$

This number is the same for all points of the ellipse and it defines, together with the foci, the ellipse. The orientation of the ellipse is defined by the direction of the vector that links the two foci of the ellipse. The sums for successive ellipses should be listed in the same line and separated with spaces.

metal.ellipses.medium (vector of int) Number of the medium that is associated with a given ellipse. Entries for successive ellipses should be listed in the same line and separated with spaces.

metal.ellipses.repeats (vector of int) Number of times each of the ellipses specified explicitly should be repeated.

metal.ellipses.vx (vector of Real)

metal.ellipses.vy (vector of Real) Translation vectors (v_x, v_y) to be used in repeats. One for each ellipse specified explicitly.

The number referred to as the *sum* above is the length of the major axis of the ellipse as well. Let us call this number l . Let the distance between the foci be d and let the minor axis of the ellipse be s . Then, from the Pythagoras theorem

$$\left(\frac{d}{2}\right)^2 + \left(\frac{s}{2}\right)^2 = \left(\frac{l}{2}\right)^2$$

Hence, the length of the minor axis of the ellipse is given by

$$s = \sqrt{l^2 - d^2}$$

Eccentricity of an ellipse is defined as

$$e = \sqrt{1 - s^2/l^2} = \sqrt{1 - \frac{l^2 - d^2}{l^2}} = \frac{d}{l}$$

and is 0 for a circle ($d = 0$) and 1 for an infinitely thin ellipse ($d = l$). For an ellipse to look more like an egg than a circle, this parameter has to be quite close to 1, for example, 0.8.

Triangles:

metal.triangles.number (int) Total number of triangles to be declared explicitly.

metal.triangles.xa (vector of Real)

metal.triangles.ya (vector of Real)

metal.triangles.xb (vector of Real)

metal.triangles.yb (vector of Real)

metal.triangles.xc (vector of Real)

metal.triangles.yc (vector of Real) The x and y coordinates (in δl) of triangle points A , B and C .

Entries that correspond to successive triangles should be typed in the same line and separated with space.

metal.triangles.medium (vector of int) Number of the medium associated with a given triangle. Entries that correspond to successive triangles should be typed in the same line and separated with spaces.

metal.triangles.repeats (vector of int) Number of times each of the triangles specified explicitly should be repeated.

metal.triangles.vx (vector of Real)

metal.triangles.vy (vector of Real) Translation vectors (v_x, v_y) to be used in repeats. One for each triangle specified explicitly.

For example, consider the layout displayed in Figure 10. Here we have two triangles. The first one is

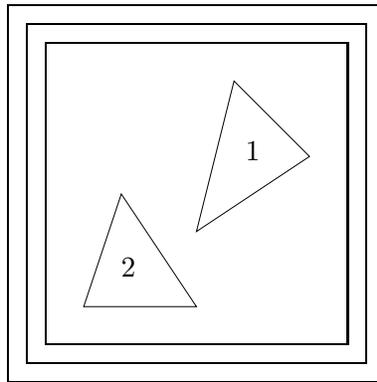


Figure 10: Two metal-filled triangles defined by the specifications provided in the example.

given by $(x_A, y_A) = (20, 20)$, $(x_B, y_B) = (50, 20)$, and $(x_C, y_C) = (30, 50)$ and is filled with metal number 2. The second triangle is given by $(x_A, y_A) = (50, 40)$, $(x_B, y_B) = (80, 60)$, and $(x_C, y_C) = (60, 80)$ and is filled with metal number 1. Their SHAPES specification looks as follows:

```

metal.triangles.number = 2
#-----
#           triangle 1   triangle 2
#-----
metal.triangles.xa    =   20           50
metal.triangles.ya    =   20           40
metal.triangles.xb    =   50           80
metal.triangles.yb    =   20           60
metal.triangles.xc    =   30           60
metal.triangles.yc    =   50           80
metal.triangles.medium =    2           1
#-----

```

These simple semantics, especially the triangle semantics, are surprisingly powerful. It is possible to construct quite elaborate shapes by combining triangles and circles. Every box can be made of two triangles, but it is computationally easier and cheaper to check whether a point belongs to a box.

On top of these we have mask semantics, which can be used in exactly the same way as the above semantics to define various shapes. But with the mask shapes we make *cuts* in the metal layout. A mask object always wins over a metal object.

The mask keywords are as follows:

```

metal.mask.boxes.number (int)

metal.mask.boxes.x_lo (vector of Real)
metal.mask.boxes.y_lo (vector of Real)
metal.mask.boxes.x_hi (vector of Real)
metal.mask.boxes.y_hi (vector of Real)
metal.mask.boxes.repeats (vector of int)
metal.mask.boxes.vx (vector of Real)
metal.mask.boxes.vy (vector of Real)

metal.mask.cylinders.number (int)
metal.mask.cylinders.xc (vector of Real)
metal.mask.cylinders.yc (vector of Real)
metal.mask.cylinders.rc (vector of Real)
metal.mask.cylinders.repeats (vector of in)
metal.mask.cylinders.vx (vector of Real)
metal.mask.cylinders.vy (vector of Real)

metal.mask.rings.number (int)
metal.mask.rings.xc (vector of Real)

```

metal.mask.rings.yc (vector of Real)
metal.mask.rings.r_lo (vector of Real)
metal.mask.rings.r_hi (vector of Real)
metal.mask.rings.repeats (vector of int)
metal.mask.rings.vx (vector of Real)
metal.mask.rings.vy (vector of Real)
metal.mask.ellipses.number (int)
metal.mask.ellipses.xa (vector of Real)
metal.mask.ellipses.ya (vector of Real)
metal.mask.ellipses.xb (vector of Real)
metal.mask.ellipses.yb (vector of Real)
metal.mask.ellipses.sum (vector of Real)
metal.mask.ellipses.repeats (vector of int)
metal.mask.ellipses.vx (vector of Real)
metal.mask.ellipses.vy (vector of Real)
metal.mask.triangles.number (int)
metal.mask.triangles.xa (vector of Real)
metal.mask.triangles.ya (vector of Real)
metal.mask.triangles.xb (vector of Real)
metal.mask.triangles.yb (vector of Real)
metal.mask.triangles.xc (vector of Real)
metal.mask.triangles.yc (vector of Real)
metal.mask.triangles.repeats (vector of int)
metal.mask.triangles.vx (vector of Real)
metal.mask.triangles.vy (vector of Real)

Following is a simple example. Figure 11 shows a rectangle defined by $(x_{lo}, y_{lo}) = (20, 20)$ and $(x_{hi}, y_{hi}) = (80, 80)$ that is filled with metal number 2. There is a circular cut-out in the middle of the rectangle, centered on $(x_c, y_c) = (50, 50)$ with radius $r_c = 15$. Here is the SHAPES specification of this layout

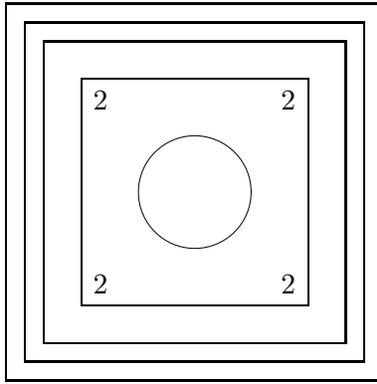


Figure 11: A metal-filled rectangle with a circular cut-out in the middle.

```

metal.bboxes.number      = 1
metal.bboxes.x_lo        = 20
metal.bboxes.y_lo        = 20
metal.bboxes.x_hi        = 80
metal.bboxes.y_hi        = 80
metal.bboxes.medium      = 2
metal.mask.cylinders.number = 1
metal.mask.cylinders.xc   = 50
metal.mask.cylinders.yc   = 50
metal.mask.cylinders.rc   = 15

```

Certain metal-on-metal structures can be hard to construct. For example, a circular metal number 2 patch inside a metal number 1 rectangle can be easily defined, but not the other way round (i.e., a rectangular metal patch inside a circle) because a circle overwrites a rectangle. The right remedy for this shortcoming would be to use a more elaborate *language* for defining layouts, similar to \TeX or pic . We intend to develop and implement such a language for the future 3-dimensional version of the code.

Still, the set of tools provided is powerful and flexible enough that one can do a lot with it. Figure 12 shows how to define a rounded corner within the existing semantics. The circle on the left is a mask, and

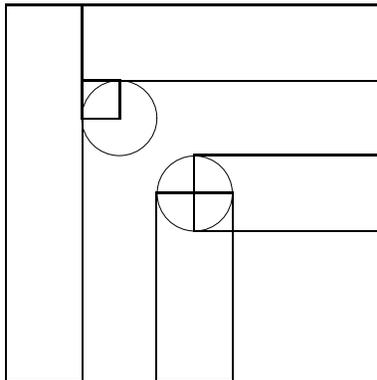


Figure 12: How to define a rounded corner in SHAPES.

it cuts a rounded corner in the little block that is inserted between the vertical and horizontal rectangles

at the left and top sides of the domain. On the other hand, the circle on the right is a full circle that joins the horizontal and vertical rectangles at the inner corner. In this case the curvature radius at both sides of the bend is the same.

The SHAPES mask is a “lift off” mask, to use terminology borrowed from microelectronics. It is defined prior to any deposition of metal, and it “lifts off” whatever metal has been put on top of it.

3.3.7 The *tag* Group

The next two groups, the *tag* group and the *refine* group (see Section 3.3.8) are related to each other and relevant only if construction of a multigrid has been requested.

A multigrid can be generated dynamically, by using the Adaptive Mesh Refinement (AMR) technique, or statically. AMR is not really necessary for the type of problems that SHAPES is designed to simulate, but it is available. A static generation of a multigrid is preferable and results in cheaper time-stepping, because regridding is an expensive procedure. Also possible in SHAPES is a combination of static and dynamic multigriding.

Multigriding works by generating progressively finer meshes in selected locations. The meshes are organized into a vector with each mesh referred to as a *level*. Level 0 is the coarsest mesh that is defined in the *level0* group. Then we have level 1, which must be embedded entirely within level 0, level 2, which must be embedded entirely within level 1, and so on. The levels sit inside each other like Russian dolls. Each level may consist of disconnected patches.

The procedures that build a multigrid work by looking at level 0 grid cells first and tagging them for refinement if need be. The cells get refined, generating a level 1 grid. The procedures are then repeated within the level 1 grid, tagging its cells and refining them so as to generate level 2 grid, and so on until a hierarchy of grids is built. The hierarchy stops when the maximum desired number of levels is reached. This number is passed through the following keyword.

tag.max_number_of_levels (int)

Setting this number to 1 disables the multigrid building procedure and multigrid computations, and this is what the user should do, unless a multigrid is really needed.

If multigriding has been requested by setting *tag.max_number_of_levels* to, say, 3 or 4, then the multigrid generation procedures look at the next three keywords.

tag.on_location (int)

tag.on_diffs (int)

tag.on_values (int)

Each of these is a logical switch. When set to zero, it disables cell tagging *on location* or *on diffs* or *on values*. When set to 1, it enables tagging. When all these switches are set to zero, then no multigrid is built, even if *tag.max_number_of_levels* is greater than 1.

Tagging *on location* means that cells are tagged for refinement depending on where they are. Tagging *on diffs* means that cells are tagged for refinement if the energy density in the cells changes faster than a certain threshold value. Tagging *on values* means that cells are tagged for refinement if the energy density in the cells exceeds a certain threshold value.

When tagging *on diffs* or *on values* is on, then the user must provide vectors of thresholds, that is, a vector of values that will be used as thresholds within subsequent levels. The keywords here are

tag.diff_thresholds (vector of Real)

tag.value_thresholds (vector of Real)

For example, in the following we have requested 4 levels, which will be referred to as level 0, the basic level, level 1, level 2, and level 3.

```
tag.max_number_of_levels = 4
tag.on_diffs              = 1
tag.diff_thresholds      = 0.02 0.05 0.08
tag.on_values             = 1
tag.value_thresholds     = 0.03 0.08 0.12
```

We have also requested that tagging cells for refinement be done by looking at how fast the energy density changes within the cells and how large the value of the energy density actually is. Hence, cells of level 0 will be tagged if the energy density within the cell changes by 0.02 within Δt_0 . Cells of level 1 will be tagged if the energy density within the cell changes by 0.05 within Δt_0 (because the energy density is computed for all levels every level 0 time step, Δt_0 , only). Cells of level 2 will be tagged if the energy density within the cell changes by 0.08 within Δt_0 . Cells within level 3 will not be tagged because we don't want level 4. Hence, there are only three Reals in the vector, not four. Additionally, cells of level 0 will be tagged for refinement if the energy density within the cell exceeds 0.03, cells of level 1 will be tagged if the energy density within the cell exceeds 0.08 and cells of level 2 will be tagged for refinement if the energy density within the cell exceeds 0.12.

If tagging is also activated *on location*, then the locations must be defined by using semantics similar to the metal layout semantics discussed in Section 2.2. The tagging procedures look at the following list of keywords.

tag.boxes.number (int)

tag.boxes.x_lo (vector of Real)

tag.boxes.y_lo (vector of Real)

tag.boxes.x_hi (vector of Real)

tag.boxes.y_hi (vector of Real)

tag.boxes.repeats (vector of int)

tag.boxes.vx (vector of Real)

tag.boxes.vy (vector of Real)

tag.cylinders.number (int)

tag.cylinders.xc (vector of Real)

tag.cylinders.yc (vector of Real)

tag.cylinders.rc (vector of Real)

tag.cylinders.repeats (vector of int)

tag.cylinders.vx (vector of Real)

tag.cylinders.vy (vector of Real)
tag.triangles.number (int)
tag.triangles.xa (vector of Real)
tag.triangles.ya (vector of Real)
tag.triangles.xb (vector of Real)
tag.triangles.yb (vector of Real)
tag.triangles.xc (vector of Real)
tag.triangles.yc (vector of Real)
tag.triangles.repeats (vector of int)
tag.triangles.vx (vector of Real)
tag.triangles.vy (vector of Real)
tag.rings.number (int)
tag.rings.xc (vector of Real)
tag.rings.yc (vector of Real)
tag.rings.r_lo (vector of Real)
tag.rings.r_hi (vector of Real)
tag.rings.repeats (vector of int)
tag.rings.vx (vector of Real)
tag.rings.vy (vector of Real)
tag.ellipses.number (int)
tag.ellipses.xa (vector of Real)
tag.ellipses.ya (vector of Real)
tag.ellipses.xb (vector of Real)
tag.ellipses.yb (vector of Real)
tag.ellipses.sum (vector of Real)
tag.ellipses.repeats (vector of int)
tag.ellipses.vx (vector of Real)
tag.ellipses.vy (vector of Real)
tag.mask_boxes.number (int)

tag.mask.bboxes.x_lo (vector of Real)
tag.mask.bboxes.y_lo (vector of Real)
tag.mask.bboxes.x_hi (vector of Real)
tag.mask.bboxes.y_hi (vector of Real)
tag.mask.bboxes.repeats (vector of int)
tag.mask.bboxes.vx (vector of Real)
tag.mask.bboxes.vy (vector of Real)
tag.mask.cylinders.number (int)
tag.mask.cylinders.xc (vector of Real)
tag.mask.cylinders.yc (vector of Real)
tag.mask.cylinders.rc (vector of Real)
tag.mask.cylinders.repeats (vector of int)
tag.mask.cylinders.vx (vector of Real)
tag.mask.cylinders.vy (vector of Real)
tag.mask.triangles.number (int)
tag.mask.triangles.xa (vector of Real)
tag.mask.triangles.ya (vector of Real)
tag.mask.triangles.xb (vector of Real)
tag.mask.triangles.yb (vector of Real)
tag.mask.triangles.xc (vector of Real)
tag.mask.triangles.yc (vector of Real)
tag.mask.triangles.repeats (vector of int)
tag.mask.triangles.vx (vector of Real)
tag.mask.triangles.vy (vector of Real)
tag.mask.rings.number (int)
tag.mask.rings.xc (vector of Real)
tag.mask.rings.yc (vector of Real)
tag.mask.rings.r_lo (vector of Real)
tag.mask.rings.r_hi (vector of Real)

tag.mask.rings.repeats (vector of int)
tag.mask.rings.vx (vector of Real)
tag.mask.rings.vy (vector of Real)
tag.mask.ellipses.number (int)
tag.mask.ellipses.xa (vector of Real)
tag.mask.ellipses.ya (vector of Real)
tag.mask.ellipses.xb (vector of Real)
tag.mask.ellipses.yb (vector of Real)
tag.mask.ellipses.sum (vector of Real)
tag.mask.ellipses.repeats (vector of int)
tag.mask.ellipses.vx (vector of Real)
tag.mask.ellipses.vy (vector of Real)

Instead of depositing metals, here we deposit subgridding. As was the case with metals, applying a tag *mask* disables the generation of a subgrid in this area. The mask always wins over other shapes.

Observe that if *tag.boxes.number*, *tag.cylinders.number*, *tag.rings.number*, *tag.ellipses.number*, and *tag.triangles.number* are all zero, then no *on location* tagging will be performed, even if *tag.on_location* is 1.

3.3.8 The *refine* Group

Once the cells of a given level have been tagged, they need to be refined, as discussed in Section 2.6.

Four parameters in the *refine* group can be used to provide additional specifications as to how the refinement should be done.

refine.fill_ratio (Real) This parameter tells Chombo how tightly to wrap subgrids around the areas generated by the cell tagging routines. The generated subgrids will be tightest when this parameter is set to 1.0. But this may produce a large number of very small boxes of grid points on top of some larger boxes when the tagged regions are complicated.

The boxes of grid points we are talking about here are the atomic unit of Chombo parallelization (see Section 2.7). Each CPU gets either nothing or a box of grid points to work on or several boxes. If some boxes are very small and other very large, some CPUs may get very little work whereas other CPUs may get overloaded. The fit will be most relaxed (i.e., the grids will be oversized and boxes will be large) when this parameter is set to 0.0. Setting it to, say, 0.5, results in an oversized and relaxed grid. We seldom use anything lower than 0.8.

refine.block_factor (int) This is the smallest size of a box, in the generated grid cells, that Chombo will use when building a refined grid. From our experience - this number should be a power of 2. If set to, say, 3, 5 or 10 it may result in a subgrid build failure. When set to 4, the smallest box that Chombo is going to generate will be 4×4 .

refine.buffer_size (int) This parameter tells Chombo how deep to nest a finer level within a coarser level. The distance between, say, a level 3 border and a level 2 border will be *refine.buffer_size* level 2 cells.

refine.max_size (**int**) This is the maximum size of a box that is going to be generated. When set, say, to 100, the largest boxes that Chombo is going to generate will be 100×100 .

Let us look at the following example.

```
refine.fill_ratio    = 1.0
refine.block_factor  = 2
refine.buffer_size   = 8
refine.max_size      = 50
```

Here we have asked Chombo to generate subgrids as tight as possible around the tagged regions. The smallest boxes in the generated subgrids will be 2×2 and the largest ones 50×50 . Level borders will be separated by margins 8 cells wide.

3.3.9 The *spectral* Group

We have discussed spectral response in Section 2.5, page 16.

Only three keywords in the *spectral* group manage how spectral response is calculated, but these are then enhanced by additional keywords in the *output* group that specify the fields to be analyzed. If no fields are specified in the *output* group, then whatever is specified in the *spectral* group is void.

The keywords are as follows.

spectral.response (**int**) A logical switch. When it is set to 1, the computation of spectral response is activated, but the user still has to specify the fields in the *output* group for this to work. When it is set to 0, the computation of spectral response is disabled, regardless of what is going to be specified in the *output* group.

When *spectral.response* is activated, then the user must also specify a number of frequencies and the frequencies themselves.

spectral.number_of_frequencies (**int**) A number of angular frequencies for which the spectral response is to be evaluated.

spectral.frequencies (**vector of Real**) A vector of *angular* frequencies, that is, $\omega = 2\pi/T$, in natural units, for which the spectral response is to be evaluated.

3.3.10 The *output* Group

The way SHAPES outputs data has been discussed in Section 2.8, page 24.

The following keywords are available to specify the output.

output.gnuplot Generate output for visualization with Gnuplot: 1 enables, 0 disables.

output.hdf5 Generate output in the HDF5 format: 1 enables, 0 disables.

output.Dx (**int**) Output the D_x field: 1 enables, 0 disables. The D_x field is dumped for the centers of the cells, not for the sides, where it resides during the computations. This field is space interpolated between $D_x(x, y - \Delta y/2)$ and $D_x(x, y + \Delta y/2)$ before dumping.

output.Dy (**int**) Output the D_y field: 1 enables, 0 disables. The D_y field is dumped for the centers of the cells, not for the sides, where it resides during the computations. This field is space interpolated between $D_y(x - \Delta x/2, y)$ and $D_y(x + \Delta x/2, y)$ before dumping.

output.Ex (**int**) Output the E_x field: 1 enables, 0 disables. This field is space interpolated before dumping the same way as D_x .

output.Ey (**int**) Output the E_y field: 1 enables, 0 disables. This field is space interpolated before dumping the same way as D_y .

output.E (**int**) Output the $E = \sqrt{E_x^2 + E_y^2}$ field: 1 enables, 0 disables. This field is calculated by using space-interpolated cell-centered fields E_x and E_y .

output.Hz (**int**) Output the H_z field: 1 enables, 0 disables. The H_z field is output *for the same time slice* as the \mathbf{E} field. The data is actually interpolated between $H_z(t - \Delta t/2)$ and $H_z(t + \Delta t/2)$ prior to the dump.

output.Energy (**int**) Output the energy density: 1 enables, 0 disables. The energy density is output *for the same time slice* as the \mathbf{E} field. Time-interpolated H_z data (see above) is used in the computation. It is also output for the centers of the cells, so that space-interpolated \mathbf{E} data is used in the computation.

output.Distrib_Dx (**int**) Output the distribution of metal on the D_x grid sites: 1 enables, 0 disables. Because the distribution of metal does not change with time, it is pointless to dump it for every snapshot. This option is meant to be used for initial runs only. Once the user is happy with the metal distribution and can get the picture, this option should be disabled.

output.Distrib_Dy (**int**) Output the distribution of metal on the D_y grid sites: 1 enables, 0 disables. The same comments apply as for *output.Distrib_Dx*.

output.Ex_ft (**int**) Output the \hat{E}_x field for frequencies specified by the *spectral.frequencies* vector: 1 enables, 0 disables. Cell-centered E_x data is used in this computation.

output.Ey_ft (**int**) Output the \hat{E}_y field for frequencies specified by the *spectral.frequencies* vector: 1 enables, 0 disables. Cell-centered E_y data is used in this computation.

output.E_ft (**int**) Output the $\hat{E} = \hat{F}(\sqrt{E_x^2 + E_y^2})$ field for frequencies specified by the *spectral.frequencies* vector: 1 enables, 0 disables. Cell-centered data is used in this computation.

output.Hz_ft (**int**) Output the \hat{H}_z field for frequencies specified by the *spectral.frequencies* vector: 1 enables, 0 disables. Time-centered H_z data is used in this computation.

output.Energy_ft (**int**) Output the energy spectral response field for frequencies specified by the *spectral.frequencies* vector: 1 enables, 0 disables. Cell- and time-centered data are used in this computation.

output.tags (**int**) Output cells tagged for refinement: 1 enables, 0 disables. This option works with Gnuplot output only.

output.boxes (**int**) Output subgrids: 1 enables, 0 disables. This option works with Gnuplot output only. Subgrids are *always* included in the HDF5 output.

output.some_levels_only (**int**) Output a specified level or range of levels: 1 enables, 0 disables. If enabled, it must be followed by *output.from_level* and *output.to_level* keywords.

output.from_level (**int**) When *output.some_levels_only* is activated, dump levels beginning with this one (inclusive).

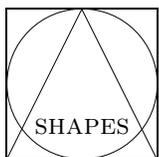
output.to_level (int) When *output.some_levels_only* is activated, dump levels up to this one (inclusive).

For example:

```
output.gnuplot           = 1
output.hdf5              = 0
output.Dx                = 0
output.Distrib_Dx       = 0
output.Dy                = 0
output.Distrib_Dy       = 0
output.Ex                = 1
output.Ex_ft             = 1
output.Ey                = 1
output.Ey_ft             = 1
output.Hz                = 0
output.Hz_ft             = 0
output.Energy            = 0
output.Energy_ft        = 0
output.tags              = 0
output.bboxes            = 1
output.some_levels_only = 1
output.from_level        = 3
output.to_level          = 4
```

Here we have requested Gnuplot style output, but HDF5 output is disabled. We are going to dump data for ***E*** and for its spectral response, but not for ***H*** or energy. Only data for levels 3 and 4 will be dumped. We are also going to dump level 3 and 4 grids (boxes). Note that if the program is run in parallel, no data will be dumped at all, because only HDF5 data can be dumped for parallel runs.

4 Parallel Execution



will run in parallel under MPI on PC farms. We have it running on the ANL Jazz cluster and on the University of Chicago TeraGrid cluster. The binary used for parallel runs is different from the binary used for sequential runs. The parallel binary is wrapped in a script called **pshapes**.

We begin with a simple example that shows how to run parallel SHAPES on both Jazz and the University of Chicago system.

4.1 TeraGrid example

Parallel jobs cannot be run interactively, so here we have to construct a PBS shell script. The script we use on the TeraGrid can be found in the SHAPES source in the **scripts** subdirectory. The file is called **uc_submit.sh**:

```

(tg-login2) $ cat uc_submit.sh
#!/bin/bash
#PBS -m abe
#PBS -M gustav@indiana.edu
#PBS -l walltime=01:00:00
#PBS -l nodes=4:ia64-compute
#PBS -N Example
#PBS -A TG-SEC502004T
#
cd /disks/scratchgpfs1/gustav
[ -d Example ] || mkdir Example
cd Example
cp $HOME/src/Mine/Current/Shapes-Jan19-1811/src/pshapes.input .
NN='wc -l $PBS_NODEFILE | awk ' { print $1 } ''
echo Got $NN nodes on $PBS_NODEFILE:
cat $PBS_NODEFILE
mpirun -np $NN -machinefile $PBS_NODEFILE $HOME/bin/pshapes pshapes.input
(tg-login2) $

```

The first line of the script, the one with the “-m abe” option, tells PBS to send e-mail when the job begins its execution, when it ends and when it aborts (for some reason). The second line provides PBS with the needed e-mail address. Line 3 reserves 1 wall-clock hour for the job’s execution. If the job tries to run longer than 1 hour, PBS will terminate it. Four cluster nodes are requested in the “-l nodes=4:ia64-compute” statement and I ask that they be all of the IA64 variety. The name of the job will be “Example”; this is how the job will show in the PBS listing. Finally there is an account line. The account option is “-A”. The user inserts her own project number here.

Now we have the shell script itself. First we go to our directory on the parallel file system

```
/disks/scratchgpfs1/gustav
```

The construct

```
[ -d Example ] || mkdir Example
```

is equivalent to

If a subdirectory “Example” exists, then fine, otherwise make it.

It is a lazy evaluation OR construct with [being equivalent to the UNIX command `test` and `||` meaning OR.

At this stage the *Example* subdirectory should exist, and we enter it. We copy a new input file `pshapes.input` from the source, then inspect the number of nodes with the `wc` command. This command counts the number of lines on a file, here referred to as `$PBS_NODEFILE`, that is going to be generated by PBS and that will contain the names of the nodes allocated to the run. We print the content of the file with the `cat` command. This is worth doing in case something goes wrong. We can then draw the attention of system administrators to the problem.

Last, we invoke `mpirun` with appropriate options—the number of nodes we want to use and the location of the node file—and point it to the MPI program we want to execute, which here is `pshapes`. The last

argument on the command line is the name of the SHAPES input file, which for the parallel run is called `pshapes.input`.

There are two differences between `pshapes.input` and `shapes.input`. The first difference is that now

```
level0.nbx = 2
```

whereas previously it was 1. This means that SHAPES is going to divide the computational domain into boxes so that two boxes will fit in the x direction and two will also fit in the y direction, because the domain is square. We'll end up with four boxes. This way each of the four MPI processes will get one box to work with.

The second difference is that this time we have disabled GNU style output and enabled HDF5 output:

```
output.gnuplot = 0
output.hdf5     = 1
```

Now we are ready to submit the job:

```
(tg-login2) $ qsub uc_submit.sh
237693.tg-master.uc.teragrid.org
(tg-login2) $
```

and the job is queued. As requested, mail will be sent to the user when the job gets to run.

Once this happens, we'll see that the `Examples` directory fills with HDF5 files:

```
(tg-login2) $ ls Example
fields_001.hdf5  fields_004.hdf5  fields_007.hdf5  pout.0  pout.3
fields_002.hdf5  fields_005.hdf5  fields_008.hdf5  pout.1  pshapes.input
fields_003.hdf5  fields_006.hdf5  fields_009.hdf5  pout.2
(tg-login2) $
```

Also, four additional files have been created in this directory: `pout.0` through `pout.3`. These are diagnostic files that correspond to MPI processes. File `pout.0` contains output similar to that produced by the sequential version of the program. Other files will usually contain less, because some SHAPES routines produce output only for the MPI process of rank 0.

When the program finishes its execution, mail will be sent to the user again, and, additionally, two PBS diagnostics files will be left in the directory from which the job was submitted:

```
(tg-login2) $ ls
Example.e237693  Example.o237693  uc_submit.sh
(tg-login2) $
```

File `Example.e237693` should normally be empty, unless something has gone wrong. File `Example.o237693` contains the output of the PBS script, which here looks as follows:

```

(tg-login2) $ cat Example.o237693
-----
Begin PBS Prologue Fri Jan 20 15:24:46 CST 2006
Job ID:          237693.tg-master.uc.teragrid.org
Username:       gustav
Group:         allocate
Nodes:         tg-c059 tg-c060 tg-c061 tg-c062
End PBS Prologue Fri Jan 20 15:24:48 CST 2006
-----
Got 4 nodes on /var/spool/torque/aux/237693.tg-master.uc.teragrid.org:
tg-c062
tg-c061
tg-c060
tg-c059
-----
Begin PBS Epilogue Fri Jan 20 15:28:34 CST 2006
Job ID:          237693.tg-master.uc.teragrid.org
Username:       gustav
Group:         allocate
Job Name:       Example
Session:        13740
Limits:         nodes=4,walltime=01:00:00
Resources:      cput=00:00:00,mem=40768kb,vmem=69376kb,walltime=00:03:32
Queue:         dque
Account:        TG-SEC502004T
Nodes:         tg-c059 tg-c060 tg-c061 tg-c062

Killing leftovers...

End PBS Epilogue Fri Jan 20 15:28:50 CST 2006
-----
(tg-login2) $

```

When run like this (i.e., without multigridding) SHAPES is highly scalable. It should be possible to run very large configurations on tens of nodes. There is usually no need to go to hundreds of nodes for 2D problems. Of course, the generated HDF5 files may be huge and their postprocessing very difficult.

4.2 Jazz Example

The Jazz example is much the same as the University of Chicago cluster example with some small differences because of the different locations of the parallel file system and SHAPES sources.

Let us look at the Jazz PBS script file, `jazz_submit.sh`, which lives in

```
/soft/apps/packages/photonic-packages/src/Shapes-2.1/doc/examples
```

Here is the file:

```

(jlogin2) $ cat jazz_submit.sh
#!/bin/bash
#PBS -m abe
#PBS -M gustav@indiana.edu
#PBS -l walltime=01:00:00
#PBS -l nodes=4
#PBS -A nanophotonics
#PBS -N Example
### #PBS -q shared
#
cd /pvfs/scratch/meglicki
[ -d Example ] || mkdir Example
cd Example
cp /soft/apps/packages/photonic-packages/src/Shapes-2.1/src/pshapes.input .
NN='wc -l $PBS_NODEFILE | awk ' { print $1 } ''
echo Got $NN nodes on $PBS_NODEFILE:
cat $PBS_NODEFILE
mpirun -np $NN -machinefile $PBS_NODEFILE \
    /soft/apps/packages/photonic-packages/bin/pshapes pshapes.input
(jlogin2) $

```

The various PBS options are the same as in the University of Chicago example except for the account number, which for our research group is `nanophotonics`. There is another option here, which right now is commented out, namely,

```
### #PBS -q shared
```

The `shared` queue is specially configured for testing jobs. But, if we run a job in this queue, we may have to share resources with other users, and may end up waiting a very long time for a job to complete, depending on what else is running on the `shared` nodes. But if the queue is unused—we can check the status easily with the `qstat -q` command—then it is suitable for a small job. This option, like other PBS options, can be used directly on the command line.

The location of the parallel file system on the Jazz cluster is `/pvfs/scratch`. The parallel SHAPES wrapper, called `pshapes`, lives in

```
/soft/apps/packages/photonic-packages/bin
```

Here is how to submit the job:

```

(jlogin2) $ pwd
/soft/apps/packages/photonic-packages/src/Shapes-2.1/doc/examples
(jlogin2) $ ls
jazz_submit.sh  pshapes.input  shapes.input  uc_submit.sh

```

```
(jlogin2) $ qstat -q shared
```

```
server: jmayor5.lcrc.anl.gov
```

Queue	Memory	CPU	Time	Walltime	Node	Run	Que	Lm	State
shared	--	--	--	--	--	0	0	--	E R
						0	0		

```
(jlogin2) $ qsub -q shared jazz_submit.sh
```

```
537037.jmayor5.lcrc.anl.gov
```

```
(jlogin2) $ qstat -u meglicki
```

```
jmayor5.lcrc.anl.gov:
```

Job ID	Username	Queue	Jobname	SessID	NDS	TSK	Req'd Memory	Req'd Time	Elap S	Time
537037.jmayor5.	meglicki	shared	Example	--	4	4	--	01:00	R	--

```
(jlogin2) $
```

... and the job runs. As the program executes on four nodes, HDF5 files are written on PVFS:

```
jazz_submit.sh pshapes.input shapes.input uc_submit.sh
(jlogin2) $ ls /pvfs/scratch/meglicki/Example
fields_001.hdf5 fields_031.hdf5 fields_061.hdf5 fields_091.hdf5
fields_002.hdf5 fields_032.hdf5 fields_062.hdf5 fields_092.hdf5
...
fields_024.hdf5 fields_054.hdf5 fields_084.hdf5 pout.0
fields_025.hdf5 fields_055.hdf5 fields_085.hdf5 pout.1
fields_026.hdf5 fields_056.hdf5 fields_086.hdf5 pout.2
fields_027.hdf5 fields_057.hdf5 fields_087.hdf5 pout.3
fields_028.hdf5 fields_058.hdf5 fields_088.hdf5 pshapes.input
fields_029.hdf5 fields_059.hdf5 fields_089.hdf5
fields_030.hdf5 fields_060.hdf5 fields_090.hdf5
(jlogin2) $
```

Every MPI process logs its actions on its own log file called `pout.?` where the question mark is replaced with the rank of the process.

After the job has completed, PBS log files will be left on the directory from which the job was submitted:

```
(jlogin2) $ pwd
/soft/apps/packages/photonic-packages/src/Shapes-2.1/doc/examples
(jlogin2) $ ls
Example.e537037 jazz_submit.sh shapes.input
Example.o537037 pshapes.input uc_submit.sh
(jlogin2) $
```

The `Example.e537037` file should be empty in this case, unless some errors occurred during execution and were captured by PBS. The `Example.o537037` file should contain the output of the PBS script:

```
(jlogin2) $ cat Example.o537037
Got 4 nodes on /var/spool/PBS/aux/537037.jmajor5.lcrc.anl.gov:
j339
j340
j341
j343
(jlogin2) $
```

In this case the only output is the list of the Jazz nodes that the program has run on.

4.3 Working with ChomboVis on Jazz

Running the parallel SHAPES example job on the Jazz cluster has dumped 300 HDF5 files. There are several ways for us to look at the data on the files. Being HDF5 files, they can be interrogated by using standard HDF5 commands, such as `h5ls` or `h5dump`. But these will not produce images, because the way images are formatted and distributed throughout an HDF5 file is application dependent.

To view the actual images of the fields, we need to use ChomboVis. The best way to run ChomboVis is to run it directly on the Jazz cluster on a PBS allocated node. There is, however, a little complication that has to do with how X11 data generated on the allocated node finds its way to the user's X11 display. Here is a procedure suggested by the Jazz administrators.

1. Connect to the Jazz cluster, and request a PBS node for an interactive job:

```
(jlogin2) $ interactive
qsub: waiting for job 537163.jmajor5.lcrc.anl.gov to start
qsub: job 537163.jmajor5.lcrc.anl.gov ready

(j26) $ echo $DISPLAY

(j26) $
```

We've done this many times before, but here we see that the `DISPLAY` variable hasn't been transferred from the Jazz front end node, and, more important, a socket for the X11 communication hasn't been made. The reason is that the PBS shell is not spawned via a normal `ssh` login that would have done this. But there is a remedy.

2. Create another window on the X11 workstation and connect within it to the Jazz front-end node. Now two processes are running on the Jazz front end nodes.

Having done this, within this other window, `slogin` to the node that's been allocated by PBS:

```
(jlogin2) $ slogin j26
```

Notice to Users

```
...
is feral. Do not feed.
```

```
-----

(j26) $ echo $DISPLAY
localhost:10.0
(j26) $
```

Now we have the `DISPLAY` variable, and the socket has been made, too. This is easy to check by running, say, `xclock`:

```
(j26) $ xclock
```

The clock face should now appear on the X11 display. Let's kill it, and let us invoke ChomboVis. But first we have to define a WWW browser for ChomboVis to display its documentation.

```
(j26) $ export WWW_BROWSER=/soft/apps/packages/photonic-packages/bin/firefox
(j26) $
```

Now we are ready to run ChomboVis itself:

```
(j26) $ /soft/apps/packages/photonic-packages/bin/chombovis
```

After some prolonged hesitation, a ChomboVis window should appear on the user X11 display. It may be a good idea to add

```
/soft/apps/packages/photonic-packages/bin
```

to the command search path and to add `WWW_BROWSER` to the environment. (Also, see the note on page 27 about managing the environment with the `.soft` file.)

Pressing the `help` button in the upper right corner of the ChomboVis window unfolds a small menu. To read the ChomboVis manual on-line we could select `Documentation`.

But here let us simply display our first field image. Pressing on the `File` button in the upper left corner of the ChomboVis window will unfold a menu. We select `Load hdf5`. This should bring up a directory browser window such as that shown in Figure 13. The way to move through the directory tree using the window is much the same on all systems: we click on this and that until we get to where we want to be. And here we want to be where the data is, which is

```
/pvfs/scratch/meglicki/Example
```

Once there, the HDF5 files, `fields_001.hdf5` through `fields_300.hdf5` should be listed in the window. Let us click on `fields_150.hdf5` and press the `Open` button. The ChomboVis window should now resize and we should see the border of the computational domain appear in it.

One can accomplish many a wonderful deed with ChomboVis, but we're going to begin with something quite unspectacular. We press on the `Tools` button and select `Data summary` from the menu. This will bring up a window with the following text:

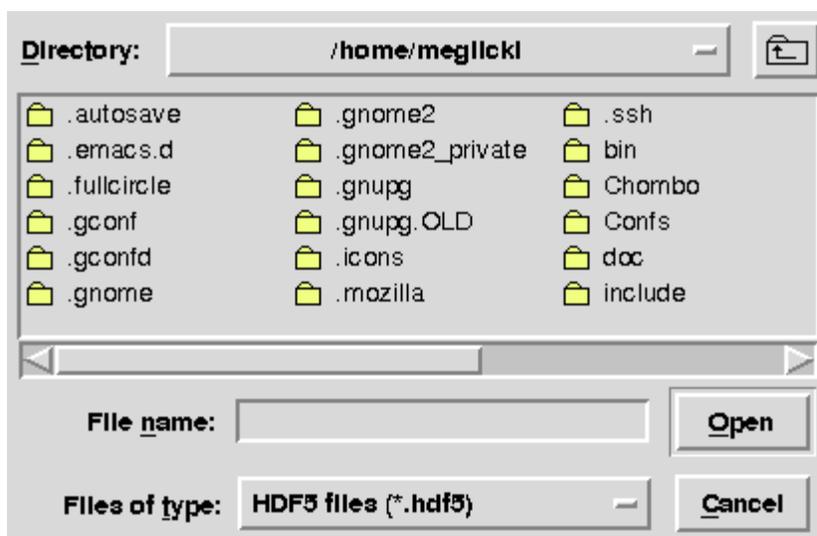


Figure 13: The ChomboVis directory browser window

```

**Box info**
level 0 : 4 boxes, 40000 data points.
All levels: 4 boxes, 40000 data points.

**Domain** : (0.0, 0.0, 0.0), (100.0, 100.0, 0)

**Component ranges** (at levels 0 through 0)
Hz (-1.3948371101104904, 1.8953029259253449)

**dx =(0.5,)
**dt =[0.125]
**time =[300.0]

```

We press the OK button to get rid of it.

To view the actual field, we press the **Visualization** button in the top menu bar of the ChomboVis window and select **Data selection**. This will bring a small window that looks like that in Figure 14. In this case, there is only one field component to choose, H_z , and there is only level 0 data. To view the field, click on Hz in the **Component** field of the **Data selection** window. An image like the one shown in Figure 15 should reward the patient investigator.

We emphasize that it is far better to run ChomboVis directly on the Jazz cluster than to copy the HDF5 files to the user's workstation and then run ChomboVis on the desktop. The copy step is going to get troubling when data files become really large, and it's easy to run out of disk space on the desktop. Running ChomboVis on the desktop is just as difficult, because compilation and installation of ChomboVis are not easy. It is also important to understand that the really heavy part of ChomboVis is communication with the disk and data processing, not the display itself. In both cases, it is better to do these on the PBS allocated Jazz nodes: they can talk directly to PVFS, and they are well equipped in terms of CPUs and memory.

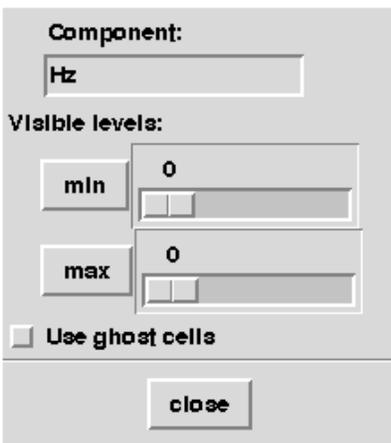


Figure 14: The ChomboVis data selection window

In the future we plan to deploy tools for fully parallel data visualization, display, and processing. Such tools will become more important when we switch to 3D modeling.

The color map in Figure 15, can be altered to emphasize various features of interest. The default color map covers the whole range of H_z values for a given snapshot. To change the range, the user should click on the **Visualization** button and then select **Colormap** in the menu that's going to unfold. A new window will pop up in which there are fields for **Colormap max** and **Colormap min**. These can be altered.

When generating images for an animation (about which more below) the user should adjust the color map so that all images from various snapshots are displayed in the same, shared color map.

Another useful option is isocontours. These can be displayed on top of the image by clicking on the **Visualization** button and selecting **Isocontours** menu entry. This brings up a window with active fields for the number of contours, minimum and maximum, as well as various cosmetic details such as line width and shading.

Another useful option is a generation of image files. Looking at an image in the ChomboVis window is nice, but not much can be done with it unless we can dump it on a file, then process and include it in a document, or combine it with other images to form animations. To do so, we click on the **File** button and then select **Save image** entry. This will bring up another window that asks for the name of the file on which the image is to be saved and the format. ChomboVis can save images in encapsulated Postscript, PPM, BMP, TIFF, various VTK formats, and more. PPM is especially useful because filters exist that let us convert PPM to just about anything else.

We now come to making movies. The obvious way would be to go through, for example, all the 300 images dumped by running SHAPES on the example file provided, generate a PPM image for each file using some shared color map, run `ppmtogif` on each PPM file, and then collate all the GIFs into a GIF animation by using `gifsicle`. This approach will work, but it is a laborious process. Luckily, we can talk to ChomboVis using Python, and some tasks can be automated. Unfortunately, ChomboVis will crash, or at best hang, if we try to make it do too much work, Python or no Python. The problem has something to do with memory leak on the X11 display workstation. One can observe that as ChomboVis goes through its motions, computing and dumping image after image, the size of the X11 server on the display system continues to grow, until the system runs out of memory, whereupon ChomboVis will crash.

An obvious architectural deficiency exists in the image-processing system here. The way it works under ChomboVis is that the latter makes VTK draw an image on the X11 display. Then VTK utilities are invoked to *read* the image back *from* the display and convert it to something else, for example, a PPM



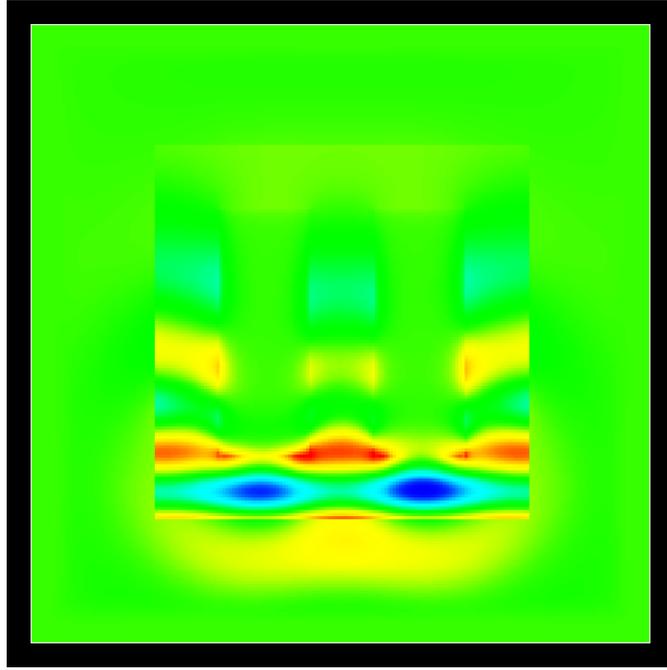


Figure 15: ChomboVis image of the H_z field for $t = 300$.

file. This approach is extremely inefficient. The right way would be to read the data from an HDF5 file, preferably in parallel, calculate an image in memory, preferably in parallel, and dump it on a PPM file, preferably in parallel, all without *ever* having to go to any displays.

Alas, the way tools like ChomboVis and VTK are put together is by scavenging existing functionality here and there rather than doing a clean (and simple) job from scratch.

Be it as it may, we have a couple of shell/Python scripts that can help us automate the task of generating PPM images from HDF5 files. The first such script is `dump_ppms.pysh`. This script takes five command-line arguments: (1) the name of the field for which the animation is to be made, for example, `Ex`, (2) and (3) minimum and maximum values against which the shared color map is to be scaled, for example, `-1.9` and `1.9`—these can be read from a log file dumped by SHAPES during execution; (4) and (5) x-size and y-size of each image in pixels, for example, `400` and `400`. Given this input, `dump_ppms.pysh` (which is a shell script) will generate a Python script for each HDF5 file in its working directory and will invoke ChomboVis on the script. Because of this schizophrenic nature of the original script (a shell script that writes and executes a Python script), the extension on the script itself is `.pysh`.

Because the script invokes ChomboVis, it should be run the same way as ChomboVis itself, that is, we have to get an interactive node allocated to the job by PBS, then we have to slogin to the allocated node via a Jazz front end node from another window to establish the X11 communication links; and then we should change to the data directory and start the script; for example,

```
(jlogin2) $ slogin j183
```

Notice to Users

...

Refreshments will be served on the lawn in front of the archbishop's residence.

```
(j183) $ echo $DISPLAY
localhost:11.0
(j183) $ cd /pvfs/scratch/meglicki/Example
(j183) $ export PATH=/soft/apps/packages/photonic-packages/bin:$PATH
(j183) $ dump_ppms.pysh Hz -1.9 1.9 400 400
```

As the script executes, ChomboVis will write responses to various Python commands on standard output, for example,

```
setpgid(): Operation not permitted
>>> ChomboVis:
>>> >>> >>> >>> Loading fields_093.hdf5
>>> >>> >>> Setting VisibleLevelMin to 0
>>> Setting VisibleLevelMax to 0
>>> >>> Loading Hz
>>> >>> Setting CmappedRangeMax to 1.9
>>> Setting CmappedRangeMin to -1.9
>>> >>> Writing Hz_093.ppm
>>> >>> Done with fields_093.hdf5
>>>
/soft/apps/packages/photonic-packages/bin/chombovis: \
  line 149: kill: (12702) - No such process
-rw-r--r-- 1 meglicki collab 480063 Jan 24 2006 Hz_093.ppm
```

The occasional error messages about `setpgid()` or “No such process” derive from the clumsiness of the tools deployed and can be ignored. When the actual image has been built, just prior to its dumping on the PPM file, it will be briefly flashed on the display. It is theoretically possible to run this job without the flashing, by using MangledMesa's blind buffer, but it doesn't seem to work.

The whole conversion process takes about half a minute per image. So, if we have 300 images to convert, we should reserve about two and a half hours for the job. It is best not to work with the desktop while ChomboVis uses it for drawing and reading images, as this may accidentally corrupt the process, which may then have to be restarted manually.

Once we have the PPM images, the next step is to convert them to PNG images, which then can be converted to GD2 and GIF images and loaded into an animated GIF file with `gifsicle`.

To convert PPM files to PNG files, we need simply run the following:

```
(j183) $ for i in *.ppm
> do
>   pnmtopng $i > `basename $i .ppm`.png
> done
(j183) $
```

and finally

```
(j183) $ pngtomovie.sh
Hz_001.png to ... Hz_001.gd2
Hz_002.png to ... Hz_002.gd2
Hz_003.png to ... Hz_003.gd2
...
Hz_298.gd2 to ... Hz_298.gif
Hz_299.gd2 to ... Hz_299.gif
Hz_300.gd2 to ... Hz_300.gif
-rw-r--r-- 1 meglicki collab 5063168 Jan 24 2006 Hz_movie.gif
(j183) $
```

Now we can view the animation with `gifview`, but only if we have the X11 socket on the node and if the network is fast enough:

```
(j183) $ echo $DISPLAY
localhost:10.0
(j183) $ gifview --animate Hz_movie.gif
```

This animation can be downloaded from our nanophotonics Wiki. Initially not much happens—remember that we’re sending here a parabolic chirp in a Gaussian envelope into two boxes, one filled with a Drude-Lorentz metal and the other filled with a pure Drude only. Both boxes have the same ϵ_∞ , which is different from 1, meaning that various dielectric effects, such as internal reflections, are going to be observable in the simulation, too. The first portion of the animation (long) just shows the tail of the signal envelope, so not much happens. Eventually we get to see the signal getting stronger as it enters the domain. Its propagation through both boxes is initially identical, because the wavelength is long. But as the wavelength gets shorter, the box on the right becomes transparent. Then the wavelength becomes longer again, and only the short-wavelength signal is still propagating within the right box.

To learn more about ChomboVis, SHAPES users should explore it on their own, perhaps using other examples in this guide. This manual is intended to provide the reader with starting points.

5 Working with Multigrid

In this section we demonstrate simple uses of multigrid in order to give the reader some feel for this technique. We begin with a sequential example without media; specifically, we are going to propagate an electromagnetic signal on three levels of resolution. This is an important test because we can see noise problems without complicating the matters additionally with the presence of the media.

5.1 A Simple Example without Media

The SHAPES input file for our example is `shapes-31-nomedia.input`, and it lives in the

```
Shapes-2.1/doc/examples
```

directory in the `photonic-packages` source directory. This file is similar to the `shapes.input` file, which we worked with before, with a small number of important differences.

First, here we have reduced `level0.nx` and `level0.ny` from 200 to 100 and increased `level0.delta_x` and `level0.delta_y` from 0.5 to 1. In other words the level 0 grid is now 100×100 and $\Delta x_0 = \Delta y_0 = 1$.

Second, we have changed `iterate.level0.stride` from 4 to 16. Here is the reason. Since we are going to run this job using a synchronized unistep (`iterate.use_substep` is set to 0), we have to adjust the length of our time step to that required by the finest grid. We are going to have three levels in the system. If the level 0 stride is 16, then the effective level 2 stride will be $16/2/2 = 4$, which is acceptable for level 2. Because we are now going to make 16 level 0 time steps for the wave front to move by one level 0 grid spacing, we are going to increase the total number of time steps to 16,000. We are still going to dump the images every 16×2 time steps, and the image frequency remains unchanged, so that this will give us 500 images.

Because now $\Delta x_0 = \Delta y_0 = 1$ and $\Delta t_0 = 1/16$, the total elapsed time for the model is going to be $16,000 \times 1/16 = 1,000$.

The signal to be injected into the total field region is now of type 3, which is a tanh ramped harmonic wave of length $\lambda = 40$. In time $\Delta t = 1,000$ we could push 25 lambdas through the computational domain. But then we have a signal delay $t_0 = 200$, so there will be only about 20 lambdas in the train, with some additional lambdas of a smaller amplitude as the signal enters the total field region. The ramp coefficient α is 0.02.

We do not change the media definition, but we have set the number of boxes to 0, so that there is no metal within the total field region.

The next change is that we activate `tag.on_location` and specify a single tag box:

```
tag.bboxes.number = 1
tag.bboxes.x_lo   = 40
tag.bboxes.y_lo   = 40
tag.bboxes.x_hi   = 60
tag.bboxes.y_hi   = 60
```

All subgridding will take place within this box. Finally, we change `refine.max_size` to 100.

We now get to run SHAPES on this file:

```
(jlogin2) $ interactive
qsub: waiting for job 537648.jmayor5.lcrc.anl.gov to start
qsub: job 537648.jmayor5.lcrc.anl.gov ready

(j286) $ cd /pvfs/scratch/meglicki
(j286) $ mkdir Multigrid
(j286) $ cd Multigrid
(j286) $ cp /soft/apps/packages/photonic-packages/src\
/Shapes-2.1/doc/examples/shapes-3l-nomedia.input .
(j286) $ shapes shapes-3l-nomedia.input
@Id: shapes.cpp,v 2.0 2006/01/11 15:20:00 gustav Exp @
@Id: shapes.h,v 2.0 2006/01/11 15:24:10 gustav Exp @
@Id: io.cpp,v 2.0 2006/01/11 15:18:01 gustav Exp @
@Id: levels.cpp,v 2.0 2006/01/11 15:18:35 gustav Exp @
```

@Id: update.f,v 2.0 2006/01/11 15:22:21 gustav Exp @
Program developed by Zdzislaw (Gustav) Meglicki, Indiana University

```
print_level: my_level = 0
  levels[0]->domain = Box (0,0) to (99,99) type [(0,0)]
  levels[0]->x0 = 0, levels[0]->y0 = 0
  levels[0]->delta_x = 1, levels[0]->delta_y = 1
  levels[0]->imin = 0, levels[0]->imax = 99
  levels[0]->jmin = 0, levels[0]->jmax = 99
  levels[0]->xmin = 0, levels[0]->xmax = 99
  levels[0]->ymin = 0, levels[0]->ymax = 99
  levels[0]->time_e = 0, levels[0]->time_e_old = -0.0625
  levels[0]->time_h = 0.03125, levels[0]->time_h_old = -0.03125
  levels[0]->delta_t = 0.0625
.....print_level: my_level = 1
  levels[1]->domain = Box (0,0) to (199,199) type [(0,0)]
  levels[1]->x0 = -0.25, levels[1]->y0 = -0.25
  levels[1]->delta_x = 0.5, levels[1]->delta_y = 0.5
  levels[1]->imin = 80, levels[1]->imax = 121
  levels[1]->jmin = 80, levels[1]->jmax = 121
  levels[1]->xmin = 39.75, levels[1]->xmax = 60.25
  levels[1]->ymin = 39.75, levels[1]->ymax = 60.25
  levels[1]->time_e = 1, levels[1]->time_e_old = 0.9375
  levels[1]->time_h = 1.03125, levels[1]->time_h_old = 0.96875
  levels[1]->delta_t = 0.0625
.....print_level: my_level = 2
  levels[2]->domain = Box (0,0) to (399,399) type [(0,0)]
  levels[2]->x0 = -0.375, levels[2]->y0 = -0.375
  levels[2]->delta_x = 0.25, levels[2]->delta_y = 0.25
  levels[2]->imin = 176, levels[2]->imax = 227
  levels[2]->jmin = 176, levels[2]->jmax = 227
  levels[2]->xmin = 43.625, levels[2]->xmax = 56.375
  levels[2]->ymin = 43.625, levels[2]->ymax = 56.375
  levels[2]->time_e = 2, levels[2]->time_e_old = 1.9375
  levels[2]->time_h = 2.03125, levels[2]->time_h_old = 1.96875
  levels[2]->delta_t = 0.0625
...
```

As the program runs, it prints minima and maxima for all three levels.

```
analyze_levels, level = 0, time_e = 68, time_h = 68.0312
  Dx_min = -0.00116738, -0.00119601
  Dx_max = 0.00205831, 0.00205831
  Dy_min = -6.8972e-05, -6.8972e-05
  Dy_max = 6.89347e-05, 6.89347e-05
  Ex_min = -0.00116738, -0.00119601
  Ex_max = 0.00205831, 0.00205831
```

```
Ey_min = -6.8972e-05, -6.8972e-05
Ey_max = 6.89347e-05, 6.89347e-05
Hz_min = -0.00218326, -0.00218326
Hz_max = 0.00114685, 0.00115078
Energy_min = 0, 0
Energy_max = 3.80236e-06, 3.80236e-06
```

```
analyze_levels, level = 1, time_e = 68, time_h = 68.0312
```

```
Dx_min = -0.00109205, -0.00109205
Dx_max = 0.000516083, 0.000516283
Dy_min = -7.06292e-06, -3.37833e-05
Dy_max = 7.07905e-06, 3.37833e-05
Ex_min = -0.00109205, -0.00109205
Ex_max = 0.000516083, 0.000516283
Ey_min = -7.06292e-06, -3.37833e-05
Ey_max = 7.07905e-06, 3.37833e-05
Hz_min = -0.000521339, -0.000526604
Hz_max = 0.00103652, 0.00103652
Energy_min = 2.48711e-11, 0
Energy_max = 1.07445e-06, 1.07445e-06
```

```
analyze_levels, level = 2, time_e = 68, time_h = 68.0312
```

```
Dx_min = -0.000646361, -0.000646361
Dx_max = 0.000514659, 0.000514763
Dy_min = -4.5556e-06, -3.38075e-05
Dy_max = 4.54462e-06, 3.38075e-05
Ex_min = -0.000646361, -0.000646361
Ex_max = 0.000514659, 0.000514763
Ey_min = -4.5556e-06, -3.38075e-05
Ey_max = 4.54462e-06, 3.38075e-05
Hz_min = -0.000521052, -0.000522457
Hz_max = 0.000595156, 0.000595156
Energy_min = 1.33697e-10, 0
Energy_max = 3.53126e-07, 3.53126e-07
```

```
...
```

A look at the data directory from another window on the Jazz front end shows that the working directory is beginning to fill with data files for three levels:

```
(jlogin2) $ ls Hz_?_00[1-4].dat
Hz_0_001.dat  Hz_0_004.dat  Hz_1_003.dat  Hz_2_002.dat
Hz_0_002.dat  Hz_1_001.dat  Hz_1_004.dat  Hz_2_003.dat
Hz_0_003.dat  Hz_1_002.dat  Hz_2_001.dat  Hz_2_004.dat
(jlogin2) $
```

The job runs fairly fast, generating about 30 three-level snapshots per minute. The sizes of the snapshot data files are as follows:

```
(jlogin2) $ ls -l Hz_?_209.dat
-rw-r--r--    1 meglicki collab    304039 Jan 26 16:21 Hz_0_209.dat
-rw-r--r--    1 meglicki collab     58411 Jan 26 16:21 Hz_1_209.dat
-rw-r--r--    1 meglicki collab     86842 Jan 26 16:21 Hz_2_209.dat
(jlogin2) $
```

Let us look at the header of a data file dumped for level 2:

```
# program: shapes, function: write_gnuplot_data
# header:
#   program author: Zdzislaw (Gustav) Meglicki, Indiana University
#   @Id: shapes.cpp,v 2.0 2006/01/11 15:20:00 gustav Exp @
#   @Id: shapes.h,v 2.0 2006/01/11 15:24:10 gustav Exp @
#   @Id: levels.cpp,v 2.0 2006/01/11 15:18:35 gustav Exp @
#   @Id: io.cpp,v 2.0 2006/01/11 15:18:01 gustav Exp @
#   @Id: update.f,v 2.0 2006/01/11 15:22:21 gustav Exp @
#   system kernel: Linux.2.4.29-rc2.#1 SMP Mon Jan 31 10:17:39 CST 2005
#   machine:      i686
#   node:         j286
#   time of dump: Thu Jan 26 16:21:07 2006
# Signal injection group:
#   x_lo:         20.000000
#   y_lo:         20.000000
#   x_hi:         80.000000
#   y_hi:         80.000000
#   mode:         3 (tanh ramped harmonic wave)
#   t0:           200.000000
#   lambda:       40.000000
#   sigma:        0.000000
#   alpha:        0.020000
#   beta:         0.000000
#   vx:           0.000000
#   vy:           1.000000
# Media group:
#   medium 1:
#   epsilon_infty: 2.364610
#   alpha:         0.000000 1.000000 1.000000
#   omega:         0.662850 0.332290 0.393180
#   delta:         0.004290 0.063930 0.105760
#   epsilon:       1.000000 0.315040 0.868050
#   medium 2:
#   epsilon_infty: 2.364610
#   alpha:         0.000000 0.000000 0.000000
#   omega:         0.662850 0.000000 0.000000
#   delta:         0.004290 0.000000 0.000000
```

```

#   epsilon:      1.000000 0.000000 0.000000
# Data group:
#   data for:     Hz
#   level:        2
#   label:        209
#   time_e:       418.000000
#   time_h:       418.031250
#   delta_t:      0.062500
#   delta_t_0:    0.062500
#   xmin:         43.625000
#   xmin_0:       0.000000
#   xmax:         56.375000
#   xmax_0:       99.000000
#   ymin:         43.625000
#   ymin_0:       0.000000
#   ymax:         56.375000
#   ymax_0:       99.000000
#   delta_x:      0.250000
#   delta_x_0:    1.000000
#   delta_y:      0.250000
#   delta_y_0:    1.000000
#   data minimum: -0.999211
#   global minimum: -0.999774
#   data maximum: -0.210539
#   global maximum: 1.000131
# data:
# x:      y:      Hz:
  43.375  43.375  -0.74798
  43.625  43.375  -0.74697
  43.875  43.375  -0.74653
  ...

```

There is a good deal of information here. We find about the exact version of sources used in the compilation of the binary, about the kernel of the node the job has run on, and about which node it was. The time of the dump is here, then all the information about the signal injected into the computational domain. We get not only the signal number but also what this signal actually is. We have two media defined but not distributed. Then we get to the actual data, and we learn that level 2 in this case is confined to a box defined by $(x_{lo}, y_{lo}) = (43, 625, 43, 625)$ and $(x_{hi}, y_{hi}) = (56.375, 56.375)$. We can see that $\Delta x = \Delta y = 0.25$ for this level and that $\Delta t = 0.0625$. Last, we get information about the minimum and the maximum values of the field, H_z , on this file and throughout the whole simulation so far (global minimum and maximum).

The program should complete the execution within less than 20 minutes. Let us look at some of the data generated by SHAPES in this run. From another Jazz window we login on the PBS allocated node:

```
(jlogin2) $ slogin j286
```

Notice to Users

```
...
defamatory material but not child pornography.
```

```
-----
(j286) $ (j286) $ echo $DISPLAY
localhost:10.0
(j286) $ (j286) $ cd /pvfs/scratch/meglicki/Multigrid
(j286) $ gnuplot
```

```
      G N U P L O T
      Version 4.0 patchlevel 0
      last modified Thu Apr 15 14:44:22 CEST 2004
      System: Linux 2.4.29-rc2
```

```
...
Terminal type set to 'x11'
gnuplot> set xrange[35:65]
gnuplot> set yrange[35:65]
gnuplot> splot "Hz_0_450.dat" with lines, \
>             "Hz_1_450.dat" with lines, \
>             "Hz_2_450.dat" with lines
gnuplot>
```

This brings up a plot showing data for all three levels interposed on top of each other. We can rotate the plot by dragging on it with a mouse to get a better view. To dump the figure on a Postscript file for inclusion in this very document, we first change the terminal to `postscript`:

```
gnuplot> set terminal postscript eps colour
Terminal type set to 'postscript'
Options are 'eps noenhanced color colortext \
  dashed dashlength 1.0 linewidth 1.0 defaultplex \
  palfuncparam 2000,0.003 \
  butt "Helvetica" 14'
gnuplot>
```

Then we specify the output file and replot the data:

```
gnuplot> set output "Hz_450.eps"
gnuplot> replot
gnuplot>
```

The resulting picture is shown in Figure 16. It is a good idea to zoom on a corner where the three levels can be seen in closeup.

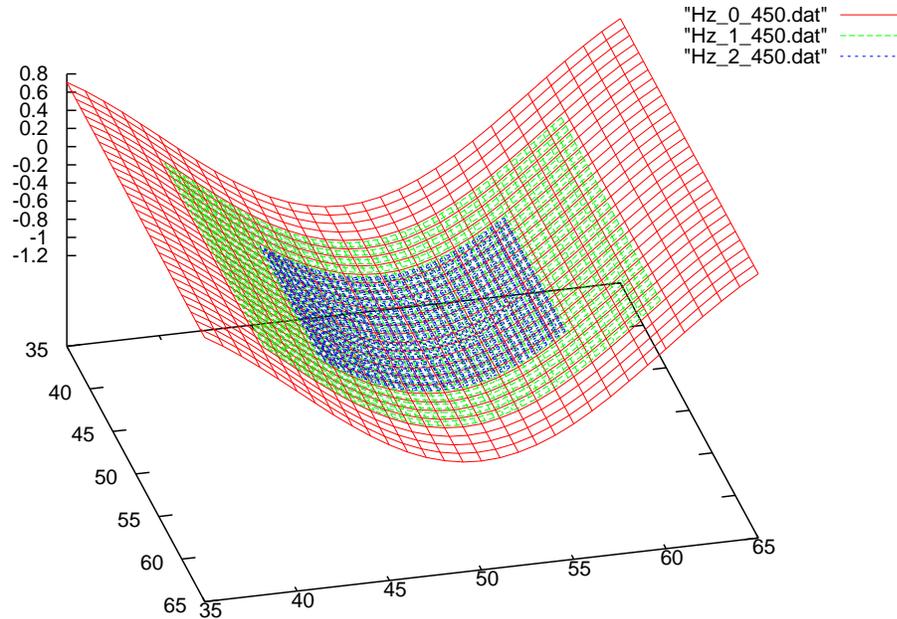


Figure 16: H_z at $t = 900$ for the tanh ramped harmonic wave. All three levels are shown.

```
gnuplot> set terminal x11
Terminal type set to 'x11'
Options are '0'
gnuplot> set view 45,75,1.5,1
gnuplot> replot
gnuplot>
```

Figure 17 shows that the data for all three levels overlaps closely. This can be seen even better by looking at the graph from the side.

```
gnuplot> set xrange[20:80]
gnuplot> set yrange[20:80]
gnuplot> set view 90,90,1,1
gnuplot> replot
gnuplot>
```

Figure 18 shows a profile for the tanh ramped harmonic wave. The thinner the line, the less the noise. Also, if there had been any deviation between the levels, we would have noticed it here. Further, if signal propagation had been incorrect, we would have noticed it; instead we see a perfect, thinly drawn sinusoid.

One can convert the data files to GIF images and collate them to form animations. The following simple script automates the task by invoking other scripts discussed already:

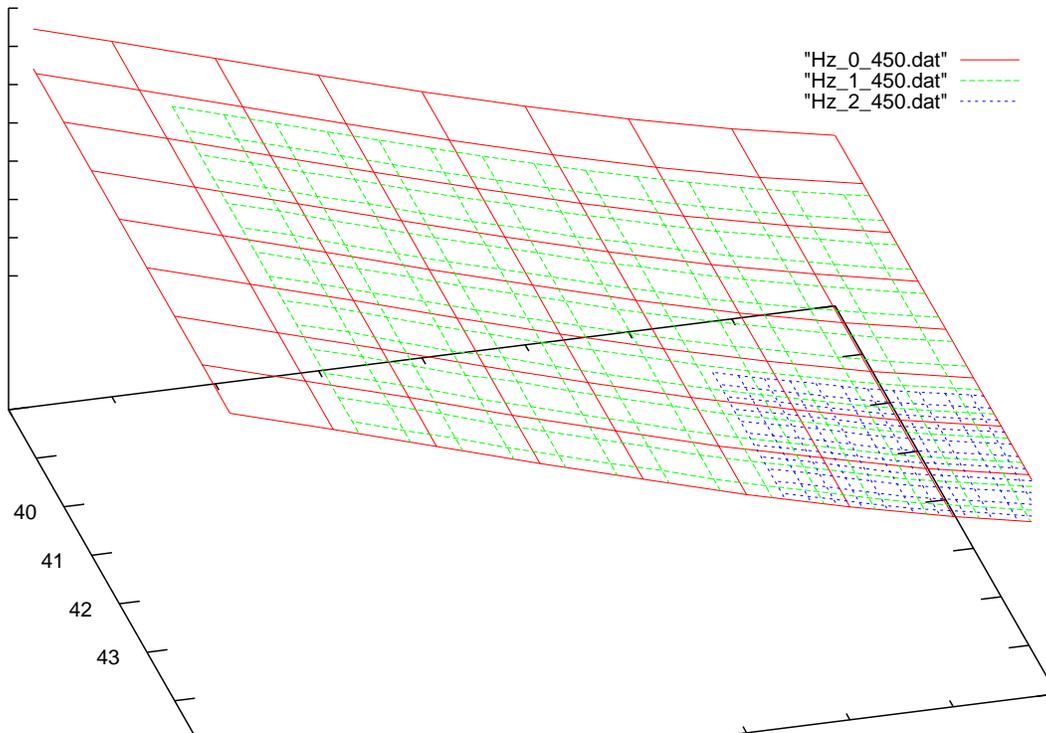


Figure 17: H_z at $t = 900$ for the tanh ramped harmonic wave. All three levels are shown in closeup of a corner.

```
(j286) $ cat movieize.sh
#!/bin/sh
write_gif_files.sh Hz 0 > Hz_0.plt
write_gif_files.sh Hz 1 > Hz_1.plt
write_gif_files.sh Hz 2 > Hz_2.plt
gnuplot Hz_0.plt
gnuplot Hz_1.plt
gnuplot Hz_2.plt
pngtomovie.sh
(j286) $ movieize.sh
...
```

The result of running this script can be seen on our nanophotonics Wiki.

5.2 A Classic Example in Parallel

In this section we present a classic example of a Gaussian wave packet scattering on a metal cylinder. To illustrate the use of the multigrid technique, we scatter the packet on a very narrow cylinder.

The input file used in this simulation is `Cylinder-5.input` which is in the same `Shapes-2.1/doc/examples` directory in the `photonic-packages` area on the Jazz cluster as all the other examples discussed in this guide.

The incident signal is number 5, a harmonic wave in a Gaussian envelope, with λ equal 20 and σ , the half width of the Gaussian envelope, equal 60. We inject the signal with the delay t_0 of 300 and we

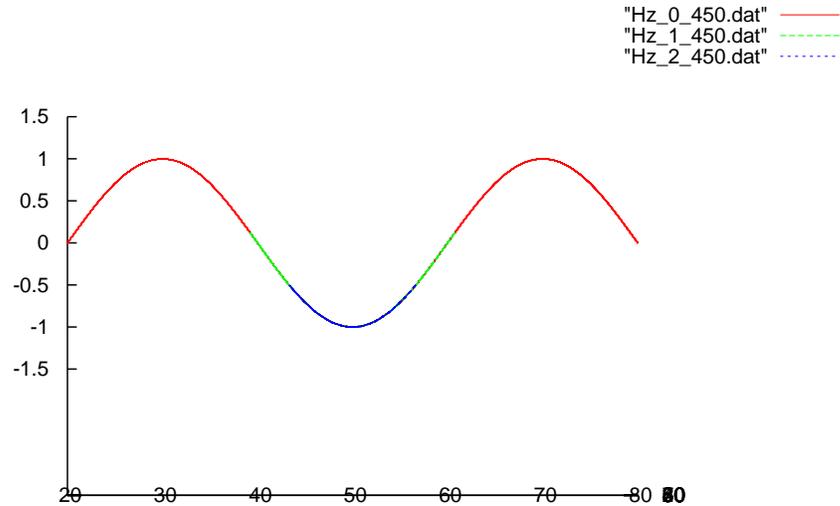


Figure 18: H_z at $t = 900$ for the tanh ramped harmonic wave. All three levels are shown in profile.

propagate the signal for $\Delta t = 650$. These parameters guarantee that the whole total field region is quiescent at the beginning and at the end of the simulation. The signal is shown in Figure 19.

The total field region is visibly smaller than the whole computational domain. We define it by

```

signal.x_lo      = 30
signal.y_lo      = 30
signal.x_hi      = 70
signal.y_hi      = 70

```

This leaves a wide scattered field region, which lets us observe the scattered field.

The cylinder is defined as a *circle* of radius $r_c = 1.25$ centered on $(x_c, y_c) = (50, 50)$. The diameter of the cylinder, 2.5, is sub- λ . If λ was, say, 5000 Å, then the diameter of the cylinder would be 625 Å or 62.5 nm.

We might as well have some other devices within the computational domain, the size of which would be more comparable to λ . If we were to simulate both the devices and scattering on the cylinder, we would have to use very high resolution everywhere or at least sufficiently high resolution in the cylinder region, so as to resolve the cylinder and the field scattered on it.

Setting up the problem requires tinkering with the multigrid layout to ensure that the actual object is well covered. Some of the work may be done by using the sequential version of SHAPES, but HDF5 output should be turned on so that we can look at the actual grids. To view the object against the grids, we turn on `output.Distrib_Dx` and `output.Distrib_Dy` and run the program just long enough for all the levels to be created and dumped on an HDF5 file.

It may also happen that, as we try to come up with an appropriate multigrid, the job hangs—especially



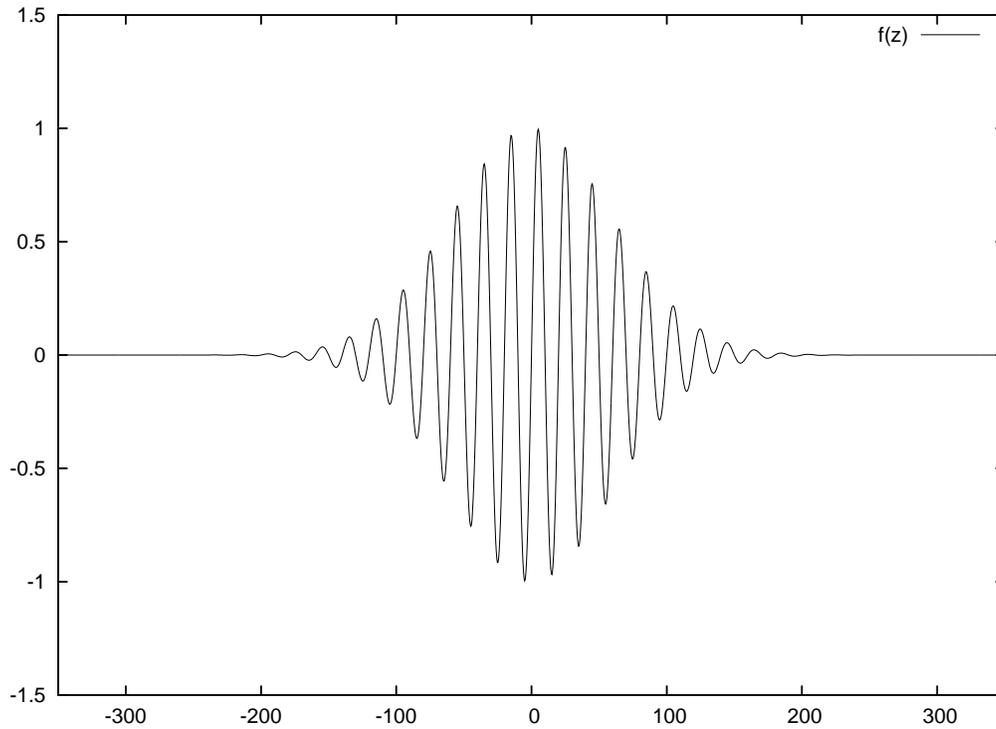


Figure 19: The Gaussian envelope wave packet (signal #5) with $\lambda = 20$ and $\sigma = 60$.

when run in parallel. The remedy is to run the `DEBUG` version of the code, invoked as `pshapes.DEBUG` (or `shapes.DEBUG` in the sequential version). The `DEBUG` code should not hang. Instead it should print various messages that may help diagnose the problem.

The multigrid and metal distribution settings used in the simulation are as follows.

```

metal.cylinders.number      = 1
metal.cylinders.xc          = 50
metal.cylinders.yc          = 50
metal.cylinders.rc          = 1.25
metal.cylinders.medium      = 1
...
tag.max_number_of_levels    = 5
...
tag.bboxes.number           = 1
tag.bboxes.x_lo             = 44
tag.bboxes.y_lo             = 44
tag.bboxes.x_hi             = 56
tag.bboxes.y_hi             = 56
...
refine.fill_ratio           = 1.0
refine.block_factor         = 2
refine.buffer_size          = 16

```

```
refine.max_size      = 50
...
```

Figure 20 illustrates the geometry of the computational domain. The cylinder is represented by the little circle in the center. It is enclosed within the tag box, which is, in turn, enclosed within the total field region box. The scattered field region around it is wide. Eventually we get to the PML boundary and finally the boundary of the computational domain.

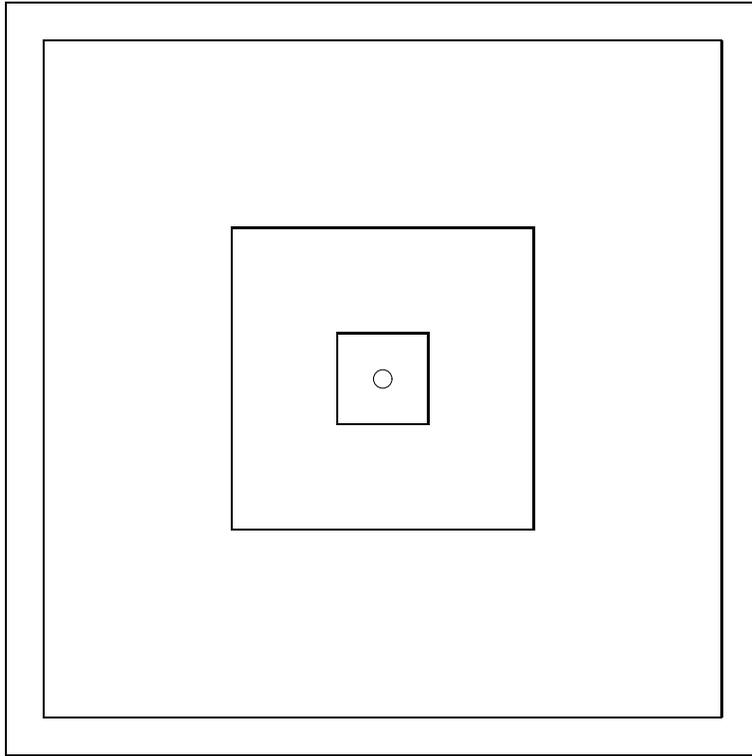


Figure 20: Definition of various regions in the simulation. The innermost box is the tag region, the next box from the center is the total field region. Then we have the PML boundary and finally the boundary of the computational domain. The circle in the center represents the cylinder. All objects are drawn to scale.

The level 0 grid is 400×400 :

```
level0.nx           = 400
level0.ny           = 400
level0.nbx          = 2
level0.x0           = 0
level0.y0           = 0
level0.delta_x      = 0.25
level0.delta_y      = 0.25
```

and the iteration is carried out by using a synchronized unistep with level 0 stride of 64.

```

iterate.level0.stride      =      64
iterate.level0.image_frequency =      20
iterate.level0.number_of_steps = 166400
iterate.use_substep        =         0

```

Observe that the image frequency is 20 not 2. The reason is that $\Delta x = \Delta y = 0.25$ instead of 1, as we had in some of the previous examples, and because we want to generate fewer images this time—130 total. The wave front is therefore going to move by 5 units of length between the snapshots.

Making 166,400 time steps at 64 steps per grid cell and 4 cells per unit of length will advance the system through 650 units of time (remembering that $c = 1$, and $\delta t = \delta x$).

The resulting level 0 grid will be subdivided into four boxes. So will, as it turns out, level 1. We can see this by looking at the Data summary for an HDF5 file dumped by a short test run with ChomboVis:

```

**Box info**
level 0 : 4 boxes, 160000 data points.
level 1 : 4 boxes, 9604 data points.
level 2 : 16 boxes, 17424 data points.
level 3 : 16 boxes, 40000 data points.
level 4 : 64 boxes, 112896 data points.
All levels: 104 boxes, 339924 data points.

**Domain** : (0.0, 0.0, 0.0), (100.0, 100.0, 0)

**Component ranges** (at levels 4 through 4)
Ex (-3.7000376853163741e-313, 8.7373756052750643e-314)
Ey (-1.6072595588452865e-313, 1.6072595588452865e-313)
Hz (-1.6737210128073414e-314, 2.9344187557448877e-313)
Energy (0.0, 0.0)
Distrib_Dx (0.0, 1.0)
Distrib_Dy (0.0, 1.0)

**dx =(0.25, 0.125, 0.0625, 0.03125, 0.015625)
**dt =[0.00390625, 0.001953125, 0.0009765625, 0.00048828125, 0.000244140625]
**time =[1.5, 1.5, 1.5, 1.5, 1.5]

```

We also find that levels 2 and 3 are each covered by 16 boxes and that level 4 is covered by 64 boxes. These numbers suggest that we could run the job on 4 CPUs. Then each CPU would get one level 0 box, one level 1 box, four level 2 boxes, four level 3 boxes and sixteen level 4 boxes, and the division of labor would be equitable.

The information that ChomboVis gives us about the time step for each level, **dt**, is incorrect, because the Chombo function that dumps data makes certain assumptions here, which are not, in our case, valid. At some stage we will have to write our own HDF5 data dumping function, or overload Chombo methods with some SHAPES specific modifications.

The time step for all levels is $\Delta t_0 = 0.00390625 = 0.25/2/2/2/2/4$. In other words, the finest level, which is level 4, is going to make four time steps in order to advance the wave front across one level 4 grid cell.

Figure 21 shows how the cylinder is resolved at various levels, beginning with level 0 on the left through level 4 on the right.

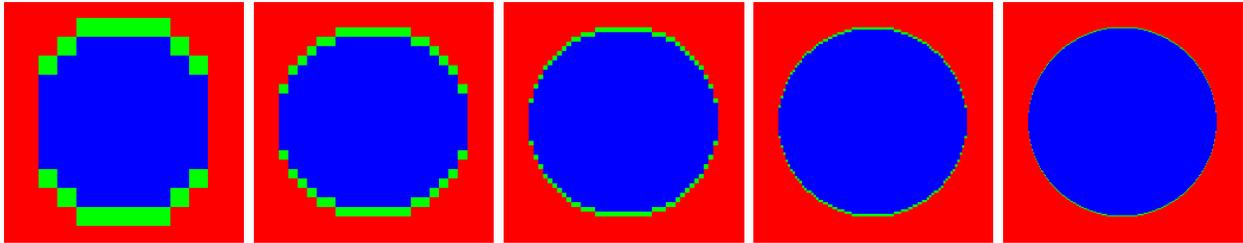


Figure 21: Resolving the cylinder at 5 levels.

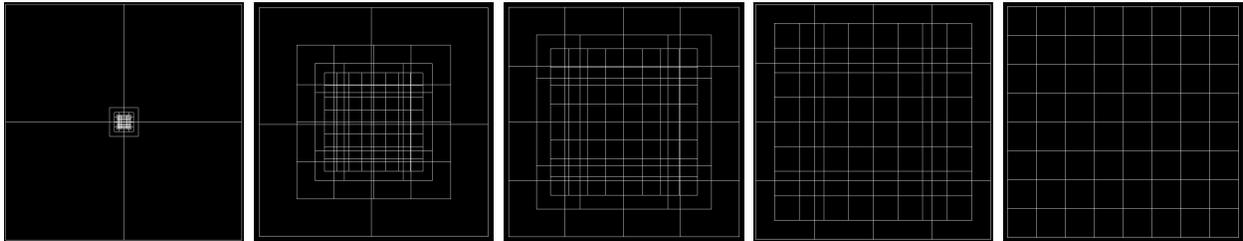


Figure 22: Multigrids for successive levels and how they nest. The leftmost panel shows the four level 0 grid boxes with higher-level grid boxes concentrated in the center. The next panel zooms on the four level 1 grid boxes with higher-level grid boxes nested within it, and so on until we zoom on the level 4 grid boxes in the rightmost panel.

Figure 22 illustrates nesting of grids: higher-level grids nest within the lower-level ones. Here only the boxes into which Chombo divides the grid cells are drawn.

The job file for this run, `Cylinder-5.sh`, looks as follows:

```
(jlogin2) $ cat Cylinder-5.sh
#!/bin/bash
#PBS -m abe
#PBS -M gustav@indiana.edu
#PBS -l walltime=48:00:00
#PBS -l nodes=4
#PBS -A nanophotonics
#PBS -N Cylinder-5
### #PBS -q shared
#
# Editable part of the script
#
MODEL=Cylinder-5
#
# Leave unchanged
#
PHOTONIC_PACKAGES=/soft/apps/packages/photonic-packages
PVFS_SPACE=/pvfs/scratch/meglicki
JOB_DIR=${MODEL}
INPUT_FILE=${MODEL}.input
```

```

JOB_FILE=${MODEL}.sh
#
# Action
#
cd $PVFS_SPACE
[ -d $JOB_DIR ] || mkdir $JOB_DIR
cd $JOB_DIR
cp $PHOTONIC_PACKAGES/src/Shapes-2.1/doc/examples/$INPUT_FILE .
cp $PHOTONIC_PACKAGES/src/Shapes-2.1/doc/examples/$JOB_FILE .
NN='wc -l $PBS_NODEFILE | awk ' { print $1 } ''
echo Got $NN nodes on $PBS_NODEFILE:
cat $PBS_NODEFILE
mpirun -np $NN -machinefile $PBS_NODEFILE \
        $PHOTONIC_PACKAGES/bin/pshapes-2.1 $INPUT_FILE
(jlogin2) $

```

The job file can be found in the examples directory of the SHAPES source (version 2.1). Observe that we copy the job file to the run directory. It is a matter of record keeping—we want to save not only the input to SHAPES but also the PBS file that was used to run the job. Otherwise the script is much the same as before, with the difference that this time we ask for 48 hours of wall-clock time. This turns out to be a little too much, because the job completed in 15 hours and 3 minutes, dumping a snapshot every 7 minutes. But since SHAPES does not provide checkpoint and resubmit functionality at this stage, it is better to request more time than not enough. Because neither Jazz nor the University of Chicago cluster limit job durations, asking for extra time has not been a problem so far. But checkpoint and resubmit will be added to SHAPES eventually.

Each HDF5 data file is slightly more than 28.5 MB. This is not much as per a single snapshot. Still, 130 of these add up to 3.7 GB, so it is better to keep them and to process them on the Jazz PVFS rather than move around.

We request the following field dumps: $E_x, \hat{E}_x, E_y, \hat{E}_y, E = \sqrt{E_x^2 + E_y^2}, \hat{E}, H_z, \hat{H}_z, \mathcal{E} = (E_x^2 + E_y^2 + H_z^2) / 2$ and $\hat{\mathcal{E}}$, where the hatted quantities are Fourier accumulations (spectral responses). We request spectral response at one frequency, $\omega_0 = 2 \times 2\pi / \lambda = 2 \times 2\pi / 20 = 2 \times \pi / 10 = 0.62831853$, remembering that because $c = 1$ we have that $\lambda = T$, where T is the period of vibration of a point through which the wave traverses. Consequently, $2\pi / \lambda = 2\pi / T$ is the angular frequency of the wave. We request spectral response at $2 \times 2\pi / \lambda$ instead of $2\pi / \lambda$ because both the energy density \mathcal{E} and the electric field strength E fluctuate with twice the frequency of the incident signal.

Some of the quantities specified above are computed *on demand* only. Requesting so many of them increases the computation time. Fourier accumulations and E are especially expensive, because sin, cos, atan, and square root are CPU-intensive functions that have to be called for every grid point and at every level 0 time step.

Let us look at some of the snapshots produced by the run. As before, we request an interactive job in a Jazz window.

```

(jlogin2) $ interactive
qsub: waiting for job 537912.jmayor5.lcrc.anl.gov to start
qsub: job 537912.jmayor5.lcrc.anl.gov ready

(j57) $

```



Then we `slogin` to the allocated node in another Jazz window.

```
(jlogin2) $ slogin j57
```

```
-----  
Notice to Users
```

```
...  
horrible deformations.
```

```
-----  
  
(j57) $
```

We change to the run directory and look at the data with ChomboVis.

```
(j57) $ cd /pvfs/scratch/meglicki/Cylinder-5  
(j57) $ chombovis &  
[1] 25347  
(j57) $
```

After a little while the ChomboVis window appears on the display. We press on the `File` button and select `Load hdf5` in the menu that unfolds. Then in the `Open` window, we find `fields_065.hdf5` and double click on it.

The snapshot corresponds to the middle of the Gaussian packet passing through the center of the total field region at $t = 325$, which is where the cylinder is.

The ChomboVis window goes through a bout of hystercics, but calms down after a few seconds, and we see the border of the computational domain in it. We press on the `Visualization` button and select `Data selection`. The popup window should look as before, but this time there are more choices in the `Component` window. We see the following:

```
Ex  
Ey  
E  
Hz  
E amplitude @ 0.628319  
E phase @ 0.628319  
Hz amplitude @ 0.628319  
Hz phase @ 0.628319  
En amplitude @ 0.628319  
En phase @ 0.628319
```

The `En` phase and amplitude images display Fourier accumulations for the energy density.

We also have more levels. To see them, the user should drag the `max` scroll bar all the way to the right. The number above the scroll bar should change to 4. To view an image of the E_x field, the user should click

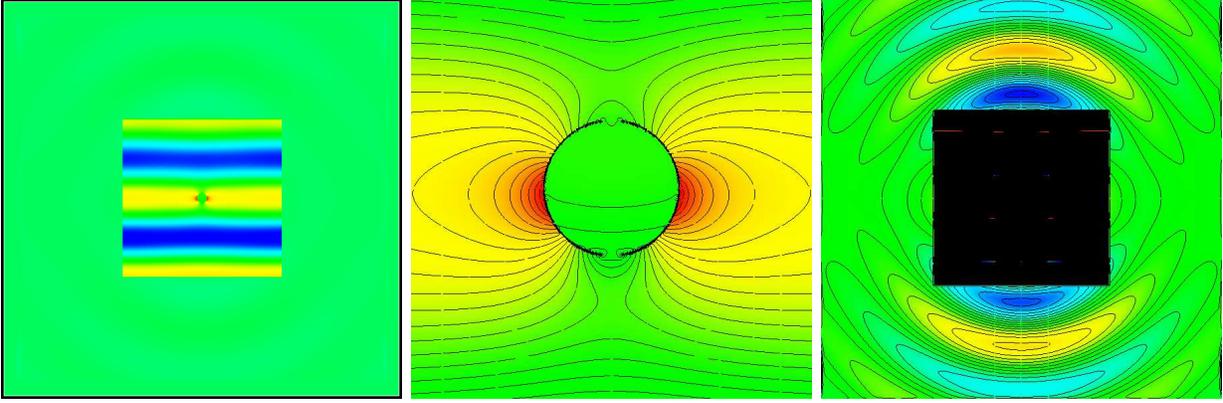


Figure 23: ChomboVis images of the E_x field for the 5 level run for $t = 325$. The leftmost panel shows the whole computational domain with the wave itself visible in the total field region. The center panel shows the magnified image of the E_x field in the center of the computational domain. The rightmost panel shows the field in the scattered field region. To see the field in the scattered field region, the colormap had to be adjusted. This resulted in the blotting out of the total field region.

on E_x . An image should appear similar to the one shown in the leftmost panel of Figure 23. We can zoom on the cylinder by pressing the right button on the mouse and dragging it down across the ChomboVis window. We can also change the colormap by selecting **Colormap** from the **Visualization** menu. The default colormap associates red with the lowest value of E_x , which is -1.49064 , and dark blue with the highest value of E_x , which is 0.973031 . These values often correspond to the spikes on the periphery of the cylinder. To identify the spikes, we can raise the colormap minimum to, say, -1.3 , while requesting that ChomboVis should black the outliers at the same time. This results in two small black blots on both sides of the cylinder. To get rid of the blots we lower the colormap minimum a little, for example, to -1.4 , then still a little more, say, to -1.44 . The blots become isolated dots located on the cylinder circumference. The new maximum of -1.44 is close to the original maximum of -1.49 , which proves that the spikes are small. This is a well known and desirable side-effect of improved resolution. Although stair-case spikes never go away, they become smaller as resolution improves.

We can draw isocontours on the image by selecting **Isocontours** from the **Visualization** menu. The number of isocontours can be changed from the default 10 to another value. We can also reset the minimum and maximum values between which the isocontours are drawn, but these have to remain within the absolute minimum and maximum for the data set. Finally, to see the isocontours, we press the **Make visible** button in the **Isocontours** window. The central panel in Figure 23 shows the magnified image of the E_x field around and inside the cylinder with isocontours. The solution looks smooth. There is no visible noise other than that due to the stair-case spikes.

To observe the scattered field, we zoom out, so as to get the view of the whole computational domain. We lower the colormap thresholds to -0.055 and 0.055 , and blot the total field region by asking for the outliers to be blacked out. We add isocontours to the picture, and the resulting scattered field image is shown in the rightmost panel of Figure 23.

Figures 24 and 25 show similarly produced images of E_y and H_z in the high resolution region and in the scattered field region.

Although this simulation runs to $t = 650$ only, we have run similar 5-level simulations to $t = 1000$ without observing noise. $t = 650$ corresponds to a wave train 32.5 lambdas long.

The purpose of this simulation was to evaluate spectral response of the system at $\omega_0 = 0.62831853$. To look at the spectral response, we select a late snapshot that corresponds to the system in the quiescent

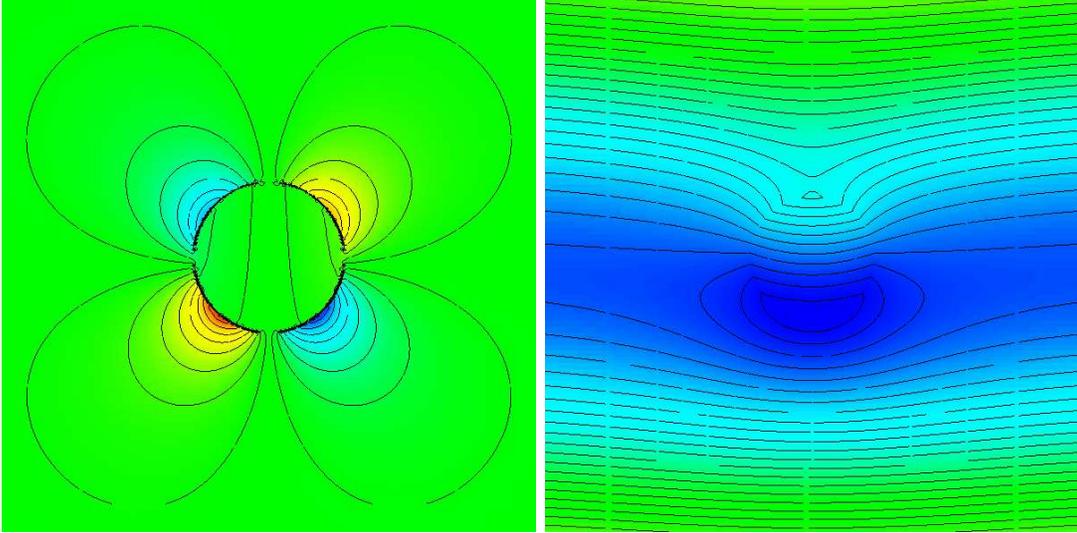


Figure 24: Fields E_y and H_z for the 5-level run for $t = 325$ in the high resolution region.

state, long after the wave packet has passed. We select data file number 130, the last one dumped.

Figure 26 shows the \hat{E} amplitude field at $t = 650$ in the high resolution region, in the total field region, and in the scattered field region. The wiggly topology of the \hat{E} amplitude field in the scattered field region suggests concentric rings underneath, and this is indeed the case as illustrated by Figure 27 that depicts the \hat{E} phase field in the high resolution region and in the scattered field region.

How much do we save by running this example in multigrid, rather than in the target resolution covering the whole region?

The target resolution is very high. The $\Delta x_0 = 0.25$ is refined 4 times down to $\Delta x_4 = \Delta x_0/2/2/2/2 = 0.25/16 = 0.015625$. At this resolution the $100 \delta l \times 100 \delta l$ region would be covered by $6,400 \times 6,400$ grid cells, which is more than 40 million cells. This would be a very large problem, and it would no longer fit in the memory of four nodes. When run on sixteen nodes, the program would take an hour and 10 minutes between the snapshots, instead of just 7 minutes for the 5-level simulation, and the dumped data files would have the size of 3.36 GB. One hundred and thirty of such files would fill more than 436 GB of disk space.

If the problem could somehow be executed on four nodes instead of sixteen, it would take four times longer to run (the single level computation parallelizes well and the scalability is almost linear), that is, 4 hours and 40 minutes per snapshot. To complete the run nearly a month would be needed.

The current program is not highly optimized, but even if we managed to optimize it, say, five times, which would be a lot, the single level configuration would still take days to complete when run on four nodes.

But the size of the dumped data files is the worst problem. ChomboVis could not be used on data sets 3.36 GB large.

Table 2 compares resources used by 5 SHAPES input configurations, all of which model the example discussed in this section. The configurations and the files used to run them can be found in the SHAPES examples directory.

For all configurations the computational domain is $100 \delta l \times 100 \delta l$, the cylinder is located in the center and its diameter is $2.5 \delta l$. For all multilevel configurations, the tag regions are defined in the same way. The *target* resolution for all runs is the same, but in the `Cylinder-5` configuration, it is reached within level 5, as discussed above. In the `Cylinder-4` configuration, we have 4 levels only and the target resolution is

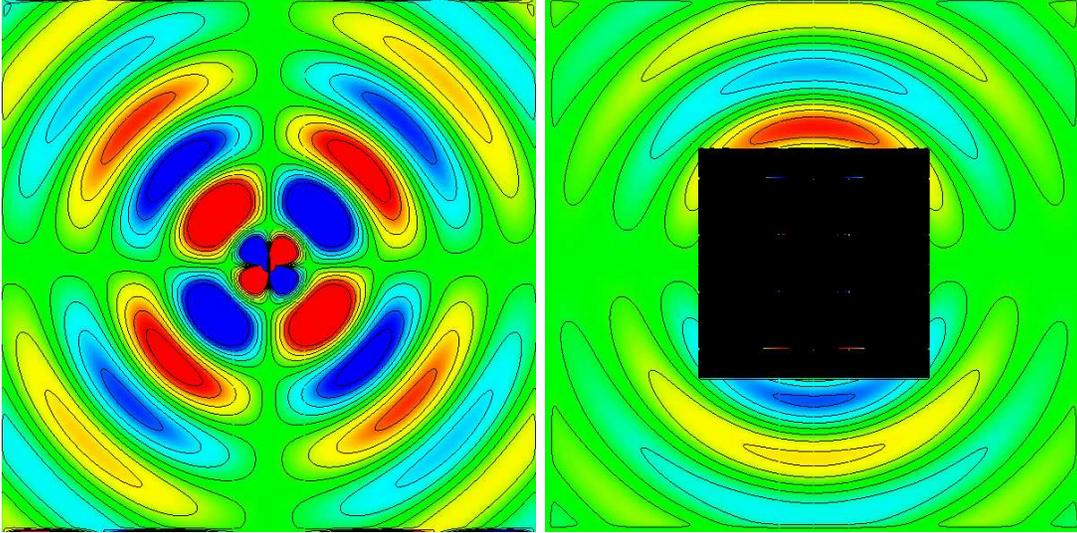


Figure 25: Fields E_y and H_z for the 5-level run for $t = 325$ in the scattered field region.

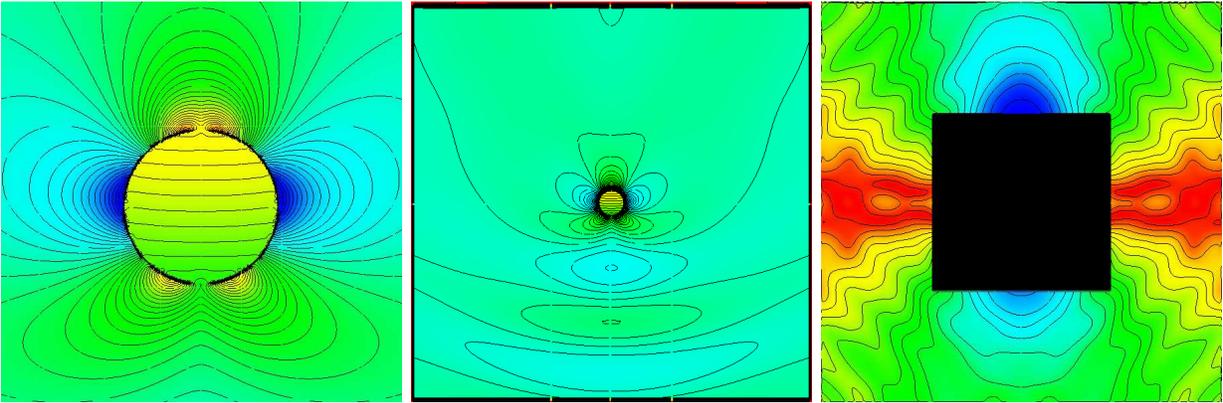


Figure 26: Spectral response \hat{E} amplitude field at $t = 650$ in the high resolution region, on the left, in the total field region, in the middle, and in the scattered field region, on the right.

reached within level 4. To accomplish this, we must enlarge the level 0 grid to 800×800 . Similarly, we have to enlarge the level 0 grid for the 3-level configuration to 1600×1600 , then for the 2-level configuration to 3200×3200 , and finally for the single-level configuration to 6400×6400 .

All runs were carried out on 4 processors. Table 2 shows that the resources are consumed exponentially with the decreasing number of levels, that is,

- the 4 level run takes 2.4 times longer to execute than the 5 level run and the generated data files are 3.3 times larger.
- The 3 level run takes 1.8 times longer than the 4 level run and the generated data files are 2.8 times larger.
- The 2 level run takes 2.5 times longer to execute than the 3 level run and the generated data files are 3.4 times larger.
- The 1 level run takes 3.5 times longer to execute than the 2 level run and the generated data files

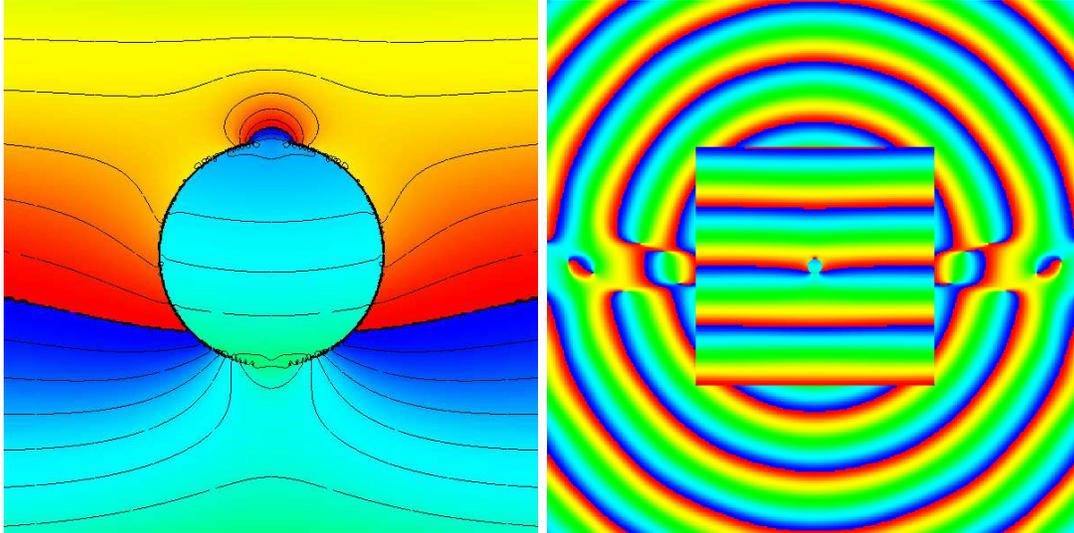


Figure 27: Spectral response \hat{E} phase field at $t = 650$ in the high resolution region, on the left, and in the scattered field region, on the right.

are 3.8 times larger.

Table 2: Resources consumed by cylinder runs with varying number of levels, but with a fixed target resolution, identical physical configuration, and a fixed number of processors. Table entries from left to right: name of configuration, number of levels, size of level 0 grid, time per snapshot and size of snapshot.

configuration	# of levels	$n_x \times n_y$	snapshot	
			time (min)	size (MB)
Cylinder-5	5	400×400	7	28.55
Cylinder-4	4	800×800	17	94.32
Cylinder-3	3	1600×1600	32	269.43
Cylinder-2	2	3200×3200	79	909.15
Cylinder-1	1	6400×6400	280	3437.51

5.3 The Cost of High Resolution

Another way to look at multigrid savings is to emphasize the cost of improving resolution. The cost of improving resolution can be exponential—improving the resolution twice in each dimension results in increasing the number of grid points $2 \times 2 = 4$ times, improving the resolution twice again increases the number of grid points 4 times again, and so on. For n such improvements, the size of the problem grows like 4^n .

Of course, this is not the only way of improving resolution. Resolution may be improved more gradually, not by dividing each cell in half, but, for example, by switching from a 400×400 grid to a 500×500 one for the same domain size. Such gradual improvements do not yield significant results and they cannot be incorporated into a multigrid scheme.

When resolution is improved, exponentially, in a small patch only, the equation changes. Suppose the computational domain is square and the small patch to be refined is square, too, and comprises N_p nodes

in each direction. Suppose, to make the reasoning easier, that it is located in the lower left corner of the computational domain. Then the total number of nodes in each direction is $N = N_p + N_0$, where N_0 is the number of nodes, in each direction, that is not refined. The total number of nodes in the 2D region is

$$(N_p + N_0)^2 = N_p^2 + 2N_p N_0 + N_0^2.$$

Refining the small patch replaces N_p with

$$N_p + 2N_p + 2 \cdot 2N_p + 2 \cdot 2 \cdot 2N_p + \dots = N_p \sum_{k=0}^n 2^k = N_p (2^{n+1} - 1).$$

where n is the number of levels, and where we have made use of the fact that $\sum_{k=0}^n 2^k$ is the sum of a geometric series. Hence, the size of the problem grows with the number of levels n as follows

$$\begin{aligned} & (N_p (2^{n+1} - 1))^2 + 2N_p (2^{n+1} - 1) N_0 + N_0^2 \\ &= N_0^2 \left(1 + 2 \frac{N_p}{N_0} (2^{n+1} - 1) + \left(\frac{N_p}{N_0} \right)^2 (2^{n+1} - 1)^2 \right). \end{aligned}$$

For $N_p (2^{n+1} - 1) \ll N_0$ the growth in the size of the problem with n is negligible, but when $N_p (2^{n+1} - 1)$ approaches N_0 , we're back in the original game, with its onset merely delayed and with the added burden of having to carry out computations for all the intermediate levels.

This observation gives us a criterion regarding a good number of levels worth going for in a given simulation. We are going to look for such n that

$$N_p (2^{n+1} - 1) = N_0$$

From this

$$\begin{aligned} n &= \log_2 \left(\frac{N_0}{N_p} + 1 \right) - 1 \\ &= \log_2 \left(\frac{N_0}{N_p} + 1 \right) - \log_2 2 \\ &= \log_2 \left(\frac{N_0/N_p + 1}{2} \right) \\ &= \frac{1}{\ln 2} \ln \left(\frac{N_0/N_p + 1}{2} \right) \\ &= \frac{1}{\ln 2} \ln \left(\frac{N}{2N_p} \right) \end{aligned}$$

Consider the example discussed in Section 5.2. We had there that $N_p = 12$ and $N = 400$. Using the above formula we obtain

$$n = \frac{1}{\ln 2} \ln \left(\frac{400}{2 \times 12} \right) = 4.05889$$

So, our choice of the maximum level being $n = 4$ was fortuitous.

The above consideration ignores the added cost of communication, both within a node and between nodes, which is very considerable for the multigrid method.

Table 3 illustrates the empirical cost of improving resolution. The configurations in the **Cylinder-Xa** series of runs, where $X = 1, 2, 3, 4, 5$, were constructed by fixing level 0 resolution at 400×400 and

Table 3: The cost of high resolution. The multilevel runs on the left hand side of the table all start with the same level 0 resolution of 400×400 . The single level runs on the right hand side stretch the target resolution that is the same as for the runs on the left hand side over the whole computational domain.

multi-level runs				single level runs			
configuration	# of levels	snapshot		configuration	$n_x \times n_y$	snapshot	
		time (min)	size (MB)			time (min)	size (MB)
Cylinder-1a	1	0.15	13.44	Cylinder-1b	400×400	0.15	13.44
Cylinder-2a	2	0.29	14.25	Cylinder-2b	800×800	0.59	53.72
Cylinder-3a	3	0.74	15.71	Cylinder-3b	1600×1600	4.00	214.85
Cylinder-4a	4	2.00	19.07	Cylinder-4b	3200×3200	34.00	852.38
Cylinder-5a	5	7.00	28.55	Cylinder-5b	6400×6400	280.00	3437.51

then adding levels. The target resolution achieved in these models is obviously different. It is 400×400 for *Cylinder-1a*, 800×800 for *Cylinder-2a*, and so on until we reach 6400×6400 for *Cylinder-5a*. Similarly, the length of the time step shrinks for these models, in order to fit the stability criterion for the finest subgrid. Otherwise the models are the same as the model discussed in Section 5.2. The snapshots are dumped every $\Delta t = 5$, the simulation is carried out until $t = 650$ and 130 snapshots are dumped altogether. The numbers in the third column of table 3 correspond to the wait, in minutes, between successive snapshots. This number grows somewhat faster than exponentially with the number of levels added, but remains acceptably short, even for the 5 level run (7 minutes). Column 4 shows the size of the snapshot in MB. Here the growth is also somewhat faster than exponential, but the size of the data file remains reasonable even for the 5 level run.

The right hand side of Table 3 shows run times and snapshot sizes for single-level configurations, where the target resolution was stretched to cover the whole computational domain. These configurations are called *Cylinder-Xb*, where $X = 1, 2, 3, 4, 5$. We note that the *Cylinder-1a* and *Cylinder-1b* configurations are identical, as are the *Cylinder-5b* configuration and the *Cylinder-1* configuration from Table 2. We observe explosive growth in the snapshot size in column 8. It is here that the cost of high resolution becomes especially painful. The growth in the size of data is also reflected in the execution time, which is shown in column 7. The savings provided by the multigrid method become substantial in the high resolution regime.

The SHAPES input files and PBS scripts for these configurations can be found in the `Shapes-2.1/doc/examples` directory.

5.4 Variations

Having developed and tested an input file for a 5-level single cylinder run, we can modify it to explore other similar systems. We can make the cylinder radius a little larger or a little smaller, but a more interesting change is to replace it with, for example, a ring and investigate what effect this is going to have on scattering. This configuration corresponds to scattering on a hollow cylinder with a sub- λ diameter. A system similar to a nano-tube.

A ring can be defined by combining a cylinder with a mask that cuts out a circle from the interior of the cylinder. But, as we have mentioned before, this construct cannot be used to define concentric rings or a rounded corner in a waveguide. For this reason rings have been added in version 2.1 of SHAPES.

We replace the cylinder in the center of the computational domain with a ring by

1. setting `metal.cylinders.number` to zero, and
2. adding the following definition:

```

metal.rings.number      = 1
metal.rings.xc          = 50
metal.rings.yc          = 50
metal.rings.r_lo       = 1.00
metal.rings.r_hi       = 1.25
metal.rings.medium      = 1

```

File `Ring-5.input` in the `doc/examples` directory in the SHAPES-2.1 source contains this exact change. Files `Ring-5.sh` and `URing-5.sh` are the PBS scripts for submissions on the Jazz cluster and on the University of Chicago cluster respectively.

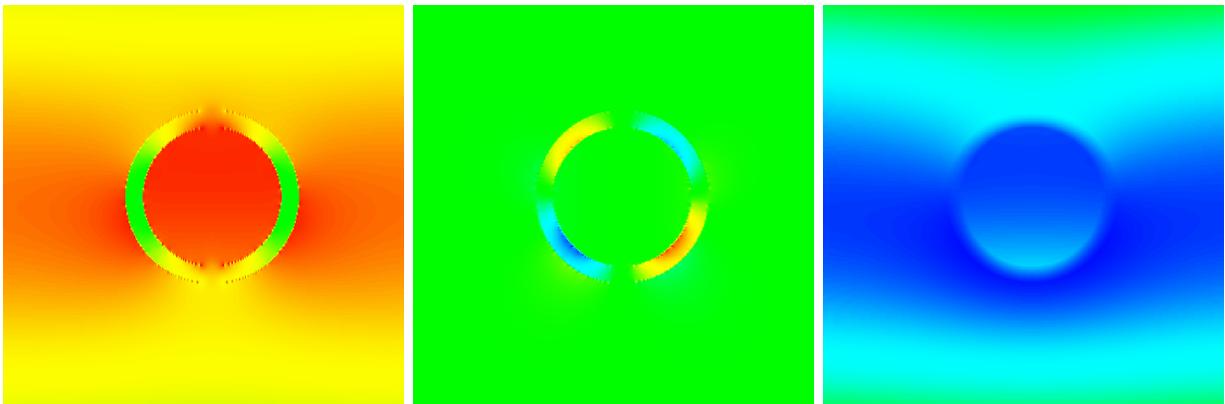


Figure 28: Scattering on a hollow metal cylinder. From left to right: E_x , E_y , and H_z at $t = 325$.

Figure 28 shows the result of the computation. It turns out that such a hollow nano-cylinder scatters light very inefficiently. The incident signal goes right through it, triggering only some interesting activity within the walls of the cylinder, but not outside.

Another interesting variation is to replace the cylinder with an ellipse. For a visible difference the ellipse should have a fairly high eccentricity. File `Ellipse-5.input` shows how to construct this configuration. We have set `metal.cylinders.number` to zero and then added

```

metal.ellipses.number   = 1
metal.ellipses.xa       = 49.3
metal.ellipses.ya       = 49.3
metal.ellipses.xb       = 50.7
metal.ellipses.yb       = 50.7
metal.ellipses.sum      = 2.5
metal.ellipses.medium   = 1

```

The foci of the ellipse are located at $(49.3, 49.3)$ and $(50.7, 50.7)$. The distance between the foci is 1.9798989873, the long axis is 2.5, which yields the eccentricity of $1.9798989873/2.5 = 0.79195959$.

Figure 29 shows fields E_x , E_y , and H_z for $t = 325$ in the vicinity of the ellipse. Figure 30 shows the

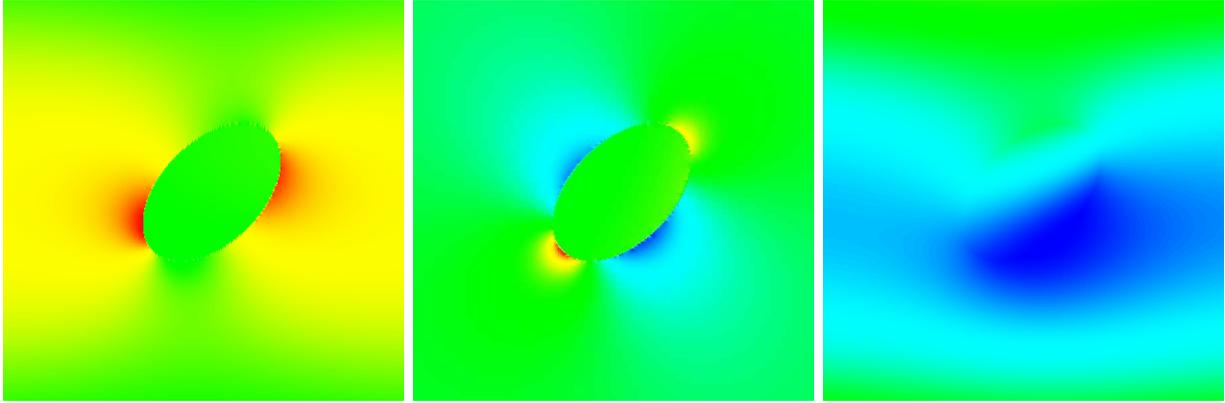


Figure 29: Scattering on an ellipse. From left to right: E_x , E_y , and H_z at $t = 325$.

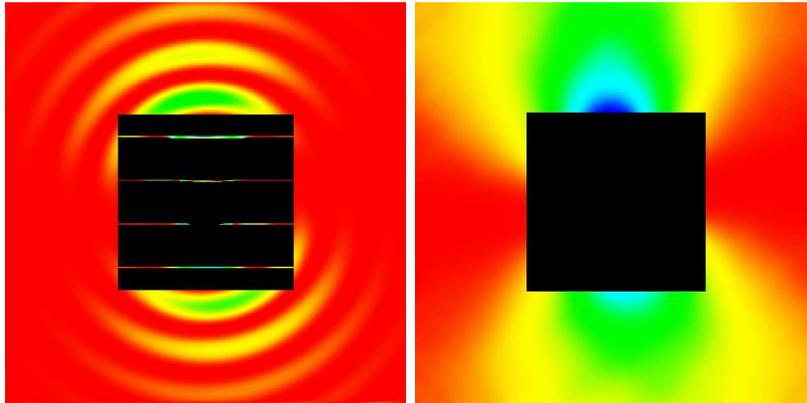


Figure 30: Scattering on an ellipse: scattered field. On the left, the energy density at $t = 325$, on the right Fourier accumulation of the energy density at $t = 650$ and $\omega = 0.62831853$.

scattered fields, the energy density at $t = 325$ and then Fourier accumulation of the energy density at $t = 650$ and $\omega = 0.62831853$. There is only a slight asymmetry in the field.

What would happen if we filled the center of the computational domain with a grid of elliptical cylinders? This configuration is described on file `mEllipse-5.input`. We have defined 25 closely spaced elliptical cylinders of the same material and geometric characteristics as the one discussed above. The media layout definition that utilizes the `repeats` construct looks as follows.

```

metal.ellipses.number      = 5
metal.ellipses.xa          = 43.3 43.3 43.3 43.3 43.3
metal.ellipses.ya          = 43.3 46.3 49.3 52.3 55.3
metal.ellipses.xb          = 44.7 44.7 44.7 44.7 44.7
metal.ellipses.yb          = 44.7 47.7 50.7 53.7 56.7
metal.ellipses.sum         = 2.5 2.5 2.5 2.5 2.5
metal.ellipses.medium      = 1 1 1 1 1
metal.ellipses.repeats     = 4 4 4 4 4
metal.ellipses.vx         = 3 3 3 3 3
metal.ellipses.vy         = 0 0 0 0 0

```

We had to enlarge the tagged region in order to accommodate all the ellipses. The new tag region definition is:

```
tag.bboxes.number      = 1
tag.bboxes.x_lo        = 35
tag.bboxes.y_lo        = 35
tag.bboxes.x_hi        = 65
tag.bboxes.y_hi        = 65
```

We have also changed the partitioning of the level 0 domain in order to distribute the computation over 16 processors.

```
level0.nx              = 400
level0.ny              = 400
level0.nbx             = 4
level0.x0              = 0
level0.y0              = 0
level0.delta_x         = 0.25
level0.delta_y         = 0.25
level0.time            = 0
```

The refinement group has been modified, too.

```
refine.fill_ratio      = 1.0
refine.block_factor    = 2
refine.buffer_size     = 16
refine.max_size        = 100
```

We have made the `max_size` larger. Finally, because the generated data files are large, 150 MB per snapshot, we've increased `image_frequency` to 40, in order to reduce the total amount of data dumped.

When run on the University of Chicago IA64 cluster, the images were produced every 33 minutes, but on the Jazz cluster, the same computation took 47 minutes per snapshot.

Figure 31 shows the energy density field at $t = 330$. This superlattice system has an effective refraction index that is responsible for the strong deformation of the wave front, but there is no visible left-right asymmetry in the scattered field. The reason for this is that the asymmetry of the individual elliptical cylinders averages out. This is further illustrated in figure 32 that shows Fourier accumulation of the energy density at $t = 650$ and $\omega = 0.62831853$.

Throughout this simulation we have not observed high field values. E_x varied between -1.8 and $+1.8$, and E_y and H_z varied between -1 and $+1$. The energy density varied between 0 and 2. The maximum amplitude of the incident signal was 1. Stair-case spikes were visible but small, no more than 1.5%.

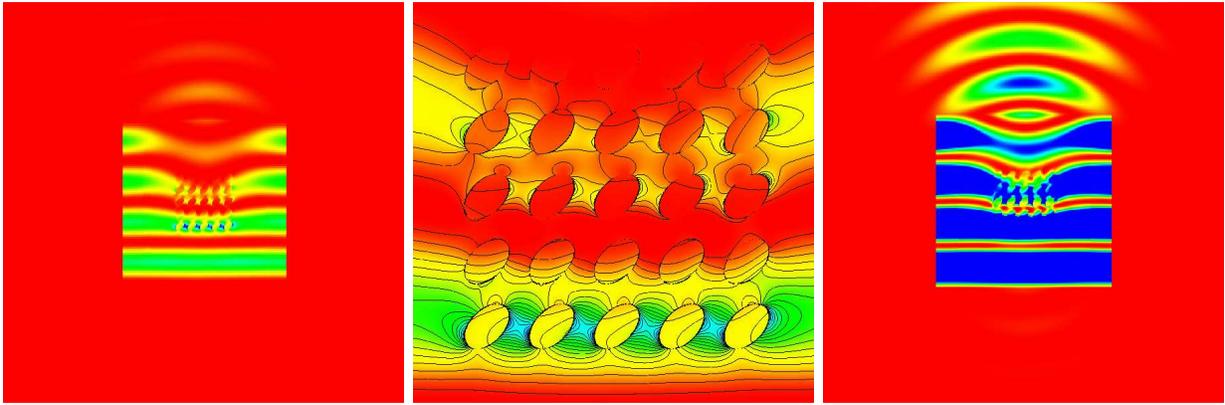


Figure 31: Scattering on a grid of elliptical cylinders. From left to right: the energy density at $t = 330$, a magnified image of the energy density at $t = 330$ in the vicinity of the cylinders, image of the energy density with colour map adjusted to show the scattered field at $t = 330$.

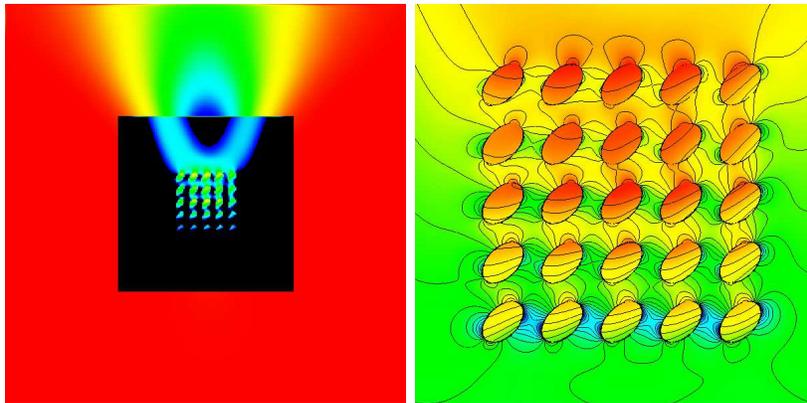


Figure 32: Scattering on a grid of elliptical cylinders. Fourier accumulation of the energy density at $t = 650$ and $\omega = 0.62831853$. A magnified image on the right.

References

- [1] Chombo: see <http://seesar.lbl.gov/anag/chombo/>
- [2] 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
- [3] Z. S. Sacks, D. M. Kingsland, R. Lee, and J. F. Lee, "A perfectly matched anisotropic absorber for use as an absorbing boundary condition," IEEE Transactions on Antennas and Propagation, vol. 43, December 1995, pp. 1460-1463
- [4] Dennis M. Sullivan, "An unsplit step 3-D PML for use with the FDTD method," IEEE Microwave and Guided Wave Letters, vol. 7, July 1997, pp. 184-186.
- [5] Dennis M. Sullivan, "Electromagnetic simulation using the FDTD method," IEEE Press Series on RF and Microwave Technology, IEEE Press, New York, 2000, ISBN 0-7803-4747-1

- [6] Allen Taflove and Susan C. Hagness, Computational Electrodynamics, 2nd ed., Artech House, Boston, 2000.

Index

(E_x, E_y, H_z) mode, 7

SHAPES

debug version, 87

Gnuplot data files, 30, 83

input file

 chat, 38

 example, 28

 format, 38

 groups, 38

 iterate, 41

 level0, 40

 metal, 46

 output, 64

 pml, 43

 refine, 63

 signal, 44

 spectral, 64

 tag, 59

 time step, 42

 unit of length, 40

 watch, 38

layout semantics, 56

man pages, 38

metals

 layout, 50

 mask, 58

output, 29, 64

parallel execution, 66

 on Jazz, 69

 on University of Chicago TeraGrid node, 66

scripts, 27

signal types, 10

tag, 59

verbosity control, 29

wrapper scripts, 28, 35

ABCs, 3, 7

absorbing boundary conditions, 3

ADEs, 3

AMR, 3

auxiliary differential equation, 3

CFL criterion, 7, 18, 42, 99

Chombo, 3, 24

ChomboVis, 3, 72

color map, 94

data summary, 73

data zoom, 94

directory browser, 73

documentation, 73

isocontours, 75, 94

pysh scripts, 76

Python, 75

saving images, 75

using on Jazz, 72

visualization, 74, 93

VTK, 75

dielectrics, 4

Drude model, 4

FDTD, 3

Fortran-77, 3

GIF animations, 26, 34, 37, 75, 78

Gnuplot output, 24, 85

 animation, 26, 86

 example scripts, 27

 file names, 24

 header, 25

 visualization, 25

HDF5 output, 24

 animation, 75

 file names, 24

 hanging, 24

 header, 24

 visualization, 24, 74

human tissue, 4

Jazz, 27

 front end nodes, 28

 GIF animations, 33

 Gnuplot

 PostScript output, 31

 PBS nodes, 27, 34

 photonic-packages, 27

 PVFS, 28

Lawrence Berkeley National Laboratory, 18

leap-frog discretization, 8

Lorentz model, 4, 46

- D* to *E* conversion, 6
 - number of fields required, 7
- ADE, 5
 - discrete form of, 6
 - stability, 7
- dimensionless coefficients, 5
- media equation, 5
- resonances, 4
 - arbitrary number of, 4
 - unit conversion, 5, 46
- Martin, Dan, 18
- Maxwell equations, 3, 47
 - discretized, 12
 - on the total/scattered field boundary, 13
 - elimination of ϵ_0 and μ_0 , 4
 - frequency domain, 7
 - time domain, 8
- media, arbitrary number of, 4
- metals, 4
 - layout, 50
 - material parameters, 46
 - resonance frequency, 49
 - unit conversion, 49
- MPI, 23
 - MPI-IO, 24
- multigrid, 3, 16
 - averaging, 19, 20
 - boundary conditions, 19
 - building levels, 22
 - recursive procedure, 23
 - data flow between levels, 18, 19
 - disconnected patches, 17
 - distribution, 63, 79, 87
 - examples of use, 78, 87
 - flowing, 3
 - for multiscale problems, 87
 - generation, 59, 63
 - interpolation, 19
 - leap-frog, 18, 19, 90
 - recursive procedure, 20
 - levels, 59, 63
 - nesting, 91
 - noise, 22, 85
 - refinement, 17
 - criteria, 16, 22
 - restricted to the total field region, 9
 - savings, 95
 - static, 3
 - synchronized multistep, 18
- nanophotonics Wiki, 28
- parallel execution, 3, 23, 66
 - output, 3
 - HDF5, 68
 - postprocessing, 72
 - PBS script, 67
 - SMP, 23
- PBS, interactive jobs, 28
- perfectly matched layer, 3
- photonic devices, 4
- plasma, 4
- PML, 3, 7, 43
- PNG, conversion to GIF, 26
- PPM, conversion to GIF, 77
- resolution, cost of, 95, 97
- scattering
 - on a cylinder, 86
 - on a grid of elliptical cylinders, 101
 - on a hollow cylinder, 99
 - on an elliptical cylinder, 100
- signal
 - injection, 44
 - types, 10, 45
- spectral response, 16, 94
- speed of light, 3, 4
- stair-case spikes, 94, 102
- TeraGrid
 - Bigben
 - Cray XT3, 27, 34
 - Lustre, 27
 - Cobalt, SGI Altix, 34
 - DAC accounts, 34
 - DataStar, IBM SP, 34
 - University of Chicago node, 27
 - GPFS, 27
 - Itanium 2, 34
 - small file I/O, 37
 - speed comparison with Jazz, 37
 - WAN GPFS, 27
- three-dimensional simulations, 3
- total/scattered field regions, 3, 11
 - boundary corrections, 13, 14, 16

definition, 12
visualization, 94

units, 3, 5, 46

wave train length, 94

