Rapsodia: User Manual

I. Charpentier
J. Utke

Draft compile date March 12, 2014

Manual version: vers. hg:1bef2728f6ea:40
compiled for Rapsodia version: vers. hg:f65927e3ce9b:434

This is hyperref'ed PDF and should be viewed in a PDF reader instead of being printed!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Because this is a joint French-American project where one author is actually German, we thought it proper to use the Spanish/Italian word Rapsodia as an acronym for

__rap__ide __s__urcharge d'__o__pérateur pour la __di__fférentiation __a__utomatique.

The main idea of Rapsodia is to combine operator overloading with code generation. The generator creates a library consisting of active types and operators overloaded for these active types for a given number n of input variables and a given derivative order o. Because n and o are fixed the generator can create a specialized code which yields a performance advantage. This code propagates Taylor coefficients up to degree o in a preset number of directions. The computed Taylor coefficients of the output variables can then be used in an interpolation scheme to compute all entries of the derivtive tensors up to order o.

Not surprisingly, very few differences exist between general-purpose languages such as Fortran and C/C++ in the representation of the elementary operations that are being overloaded (see also Sec. 3.1.1). Therefore, a single generator has been developed to produce a Fortran and a C++ libraries simultaneously.

1.1 Installation

This section covers download, configuration and building (of the non-generated portions) of Rapsodia.

1.1.1 Download the Sources

The Rapsodia **source code can be downloaded** as a tar file from the website

http://www.mcs.anl.gov/rapsodia

or accessed via version control, see below. The source can be unpacked with

```
tar -zxvf Rapsodia_<date> .tgz
```

which will create a `Rapsodia` directory. To avoid having to give system specific variations for each command we also assume that we are on a Linux system in a `bash` environment where GNU make and C/C++/Fortran compilers are installed. The `CC`, `CXX`, and `F90C` shell variables point to the respective compilers. We also assume a Python version 2.4 or newer available through `PATH`. While Python is in principle platform independent, the code in Rapsodia is currently not entirely platform independent.

The **version control** system for the Rapsodia development is *mercurial*[5]. A link to the current development repository and instructions on how to clone the repository are listed in the Rapsodia website.

1.1.2 Configuration

There is a small set of options being set through a configuration step which depend on the compiler versions and external libraries. There are also some checks being executed to ensure proper function. The configuration options are shown by executing `Rapsodia/configure.py -h`.

```
Usage: configure.py [options]
        used to set up make rules and definitions for building Rapsodia generated sources and support libraries

Options:
  -h, --help            show this help message and exit
  -f FORTCOMP, --fortranCompiler=FORTCOMP
                        pick a Fortran compiler (defaults to gfortran) from: [
                        sunf95 | pgf95 | mpixlf90 | xlf | mpif90 | nagfor |
                        g95 | af95 | bgxlf | gfortran | ifort ] - the compiler
                        should be in your PATH
  -c CXXCOMP, --cPlusPlusCompiler=CXXCOMP
                        pick a C++ compiler (defaults to g++) from: [ nvcc |
                        sunCC | xlC | icpc | bgxlC | g++ | pgcpp | mpicxx |
                        mpixlcxx ] - the compiler should be in your PATH
  -m MIXEDLINKER, --mixedLanguageLinker=MIXEDLINKER
                        pick a linker for mixed language applications - the
                        linker should be in your PATH (default none); this
                        configure knows flags for: ('g++', 'gfortran'),
                        ('pgcpp', 'pgf95'), ('g++', 'nagfor'), ('icpc',
                        'ifort')
  -d, --debug           setup for compiling with full debug information
                        (defaults to False)
  --fortranLongLines    permit lines longer than the standard length (default
                        False)
  --fortranFixedFormat  use fixed instead of free format (default False)
  -q, --queue           set up for multithreaded queue(default False)
  --withOpenPA=<path_to_openpa>
                        use the OpenPA library (defaults to False); implies -q
  --noCheck             do not run checks during configuration (defaults to
                        False)
  --noCleanup           if the configuration check fails do not remove the
                        associated files (defaults to False)
  --compilerConfigs     print the flags for all the compilers that this
                        configure script covers (defaults to False)
```

Almost all options are self-explanatory. The queue option `--queue` is explained in Sec. 2.5.2 and it may be combined with the OpenPA library available at `http://trac.mcs.anl.gov/projects/openpa/` whose install directory can be specified using the `--withOpenPA` option. The `configure.py` script creates `Rapsodia/MakeDefs.mk` and `Rapsodia/MakeRules.mk` that contain compiler flags (based on the compilers specified), some environment variables and suffix rules, see also Sec. 2.5.

1.1.3 Build

After the configuration step has completed, the fixed portions of the support library for the interpolation can be compiled within the `Rapsodia` directory by invoking

```
make
```

A sanity check of the tool can be executed via

```
make check
```

1.1.4 Examples

Similar to the core system, a set of examples used in this manual are available via mercurial under the repository name `RapsodiaExamples`. To obtain the examples follow the intructions on `http://www.mcs.anl.gov/rapsodia` (go to *Access*).

The `Makefiles` in the examples assume that in the shell environment `RAPSODIAROOT` is set to the absolute path to the `Rapsodia` directory. Some of the examples are discussed in this manual. All examples can be built and run within the `RapsodiaExamples` directory by invoking

```
make
```

This step, however, will skip the `Extras` subdirectory whose contents is covered in Sec. 2.8.4.

1.2 A One Minute Example

As a starting point we show a small example where we compute `y=sin(x)` at the point `0.3`. The source code for this example can be found in `RapsodiaExamples/CppOneMinute` which can be downloaded as described in Sec. 1.1.4.

The example can be compiled and run by changing to this directory and typing `make`. After convincing ourselfs that the Rapsodia tool runs ok we can now look at the steps that were taken to make this example work. The original code is in `driver0.cpp` which we show in Fig. 1.1. Here we assume we want to compute just the Taylor coefficients (see Sec. 1.3.1) up to order three in two directions. The code changes to the original source, explained in the following text, can be seen in Fig. 1.2. To enable operator overloading we need to change the type of the floating point variables `x` and `y` to a special

```
#include <iostream>
#include <cmath>

int main(void){
  double x,y;
  // the point at which we test
  x=0.3;
  // compute sine
  y=sin(x);
  // print it
  std::cout << "y=" << y << std::endl;
}
```

Figure 1.1: Original source code of the example

type. Rapsodia creates such a type named `RAfloatD` which is the active floating point type where the `D` stands for double precision. We need to include the Rapsodia header file `RAinclude.ipp`. We define constants `d` and `o` for the directions and the order, respectively. We need to initialize the coefficients of the inputs. The actual computation statement `y=sin(x)` remains unchanged. Because the type of `x` and `y` has been changed the overloaded version for `sin` is called automatically. At the end we retrieve and print the Taylor coefficients for `y`. Of course, in a real world example we assume the computation to be much larger than a single line.

Now we need to let Rapsodia generate the library for the two directions and Taylor coefficients up to order three. This is done within `RapsodiaExamples/CppOneMinute` by invoking

```
generate.py -d 2 -o 3 -c RALib
```

One can see this step in `RapsodiaExamples/CppOneMinute/Makefile`. All the header and source files for the generated overloading library can now be found in the `RapsodiaExamples/CppOneMinute/RALib` subdirectory whose name was specified with the `-c` command line switch. Also generated in this subdirectory is a `Makefile`, which is used to compile the generated code and create a static library called `libRapsodia.a`. This library is linked with `driver.o` to create the `driver` binary and when this is run we obtain the output shown in Fig. 1.3.

1.3 Basic Concepts

In the following we want to explain some concepts that are necessary to understand the terms used in the Sec. 2. The readers familiar with automatic differentiation and overloading should skip this section.

1.3.1 Propagating Taylor Coefficients with Automatic Differentiation

The basic principles of automatic differentiation (AD) have been known for several decades [9], but only since the mid 1990s have the tools implementing AD found significant use in optimization, data assimilation, and other applications in need of efficient and accurate derivative information. As a consequence of the wider use of AD, various tools have been developed that address specific application requirements or programming languages. The AD community's website

`www.autodiff.org`

provides a comprehensive overview of the available tools. Only a small subset of these tools is capable of computing higher-order derivatives. Aside from a number of unfinished or no longer maintained projects, the practically relevant choices are AD02 for Fortran [7] and Adol-C for C and C++ [2]. Like Rapsodia, both rely on operator overloading as the vehicle of attaching derivative computations to the elementary operations ϕ provided by the programming language such as the arithmetic operators and intrinsic functions $\sin, e^x$, and so forth.

A program implementing a numerical model

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : I\!R^n \mapsto I\!R^m \tag{1.1}$$

is viewed as a sequence of such elementary operations. As all AD tools do, we perform the computation of the derivative of $\boldsymbol{f}$ by applying the chain rule to the sequence of ϕ comprising $\boldsymbol{f}$.

The commonly taken approach for derivative tensors of order three and above is the forward propagation of Taylor polynomials up to order o in d directions with coefficients $a_j^i, j = 1\ldots o, i = 1\ldots d$ around a common point $a_0 \equiv a_0^i$. The Taylor polynomial is nothing but a Taylor series truncated at order o

$$\phi(a_o + h) = \phi(a_0) + \phi\prime(a_0)\cdot h + \frac{\phi\prime\prime(a_0)}{2!}\cdot h^2 + \ldots + \frac{\phi^{(d)}(a_0)}{o!}\cdot h^o \tag{1.2}$$

```
#include <iostream>
#include <cmath>

#include "RAinclude.ipp"                  // added

int main(void){
  int i,j;                                // added
  const int directions=2;                 // -"-
  const int order=3;                      // -"-
  RAfloatD  x,y;                          // type changed
  // the point at which we test
  x=0.3;
  // initialize the input coefficients    // added
  // in the 2 directions                  // -"-
  for( i=0;i<directions;i++) {            // -"-
    for( j=0;j<order; j++) {              // -"-
      if (j==0)                           // -"-
        x.set(i+1,j+1,0.1*(i+1));         // -"-
      else                                // -"-
        x.set(i+1,j+1,0.0);               // -"-
    }                                     // -"-
  }                                       // -"-
  // compute sine
  y=sin(x);
  // print it
  std::cout << "y="<< y.v << std::endl;   // modified
  // get the output Taylor coefficients   // added
  // for each of the 2 directions         // -"-
  for( i=0;i<directions;i++) {            // -"-
    for( j=0;j<order; j++) {              // -"-
      std::cout << "y["<< i+1             // -"-
                << "," << j+1             // -"-
                << "]="                   // -"-
                << y.get(i+1,j+1)         // -"-
                << std::endl;             // -"-
    }                                     // -"-
  }                                       // -"-
}
```

Figure 1.2: Source code of the example modified for Rapsodia

```
y=0.29552
y[1,1]=0.0955336
y[1,2]=-0.0014776
y[1,3]=-0.000159223
y[2,1]=0.191067
y[2,2]=-0.0059104
y[2,3]=-0.00127378
```

Figure 1.3: Computed output value Taylor coefficients for two directions up to order three

The propagation of coefficients is done on the level of the elementary operations. For each $r = \phi(a, b, \ldots)$ with result r and arguments $a, b, \ldots$[1] we compute the result's Taylor coefficients r^i_j based on the Taylor coefficients of the arguments $a^i_j, b^i_j, \ldots$. For instance, for the addition $r = a + b$ the formula is simply

$$r^i_j = a^i_j + b^i_j \quad .$$

For the multiplication $r = a \cdot b$ it is the convolution

$$r^i_j = \sum_{l=0}^{j} a^i_l \cdot b^i_{j-l} \quad .$$

The complete set of formulas for the elementary operations occurring in C++ and Fortran is given in [4].

In the following we denote with $\boldsymbol{x}_0$ the given point in the domain of $\boldsymbol{f}$. To compute the actual derivative tensors, we employ the approach described in [3], which is based on the propagation of univariate Taylor polynomials in a predefined number of directions and subsequent interpolation of the tensor elements. The details of the approach and the supporting driver routines are described in Sec. 1.3.3. We denote a given set of d directions, also referred to as the seed matrix, with $\boldsymbol{S} \in I\!R^{n\times d}$. The columns of the seed matrix are used to set the respective first Taylor coefficients of the n input variables $\boldsymbol{x}^i_1 = \boldsymbol{s}^i \in I\!R^n$ and the higher order coefficients are nullified $\boldsymbol{x}^i_j = \boldsymbol{O}|_{j=2\ldots o}$. The coefficients of the m outputs $\boldsymbol{y}^i_j$ are the coefficients of the univariate Taylor series $\boldsymbol{f}(\boldsymbol{x}_0 + h\boldsymbol{s}^i)$.

[1] In practice most ϕ are uni- or bivariate.

1.3.2 Active Variables and Overloading

Modern programming languages such as C++ or the more recent versions of Fortran have the capability of overloading operators and intrinsic functions ϕ. It is very conveniently used for automatic differentiation because it allows us to replace the meaning of the programming language's built-in operators and intrinsics with the derivative semantics. The invocation of the overloaded ϕ is tied to the use of program variables of a specific type or set of types. Therefore, aside from initialization and derivative retrieval the only change that intrudes into the original program is a change of type declaration for floating point variables.

The output variables $\boldsymbol{y}$, the input variables $\boldsymbol{x}$, and all intermediate program variables *on any dependency path* from the $\boldsymbol{x}$ to the $\boldsymbol{y}$ are called *active* variables and need to have their type changed to the specific active type(s). This type change triggers the execution of the overloaded ϕ. All other variables are called *passive*. For overloading-based tools in practice often all floating-point variable type declarations are changed to active type. This sometime implies a certain unnecessary overhead because derivative computations are performed for passive variables. See also Sec. 2.

1.3.3 Higher-Order Tensor Drivers

An efficient approach to compute derivative tensors $\mathcal{D}^o$ of order $o \geq 3$ is laid out in [3] and has been previously implemented within Adol-C. Rather than leaving the reimplementation to the user, we provide a Fortran and a C++ implementation as a convenience together with the Rapsodia sources. Because symmetry increases with o in $\mathcal{D}$, we want to compute and represent just the distinct elements. Following [3], we use multi-indices $\boldsymbol{t} \in I\!N_0^n$, where each $t_i, i = 1 \ldots n$ represents the derivative order with respect to input x_i. For instance, for $n = 2, m = 1$, the two Hessian elements $H_{12} = \frac{\partial^2}{\partial x_1 \partial x_2}$ and $H_{21} = \frac{\partial^2}{\partial x_2 \partial x_1}$ are both represented by $\boldsymbol{t} = (1, 1)$. All distinct elements of $\mathcal{D}^o$ are represented by the multi-indices $\boldsymbol{t}$ for which $o \equiv |\boldsymbol{t}| = \sum_{i=1}^{n} t_i$. There are exactly $\binom{n+o-1}{o}$ such multi-indices. We take each multi-index $\boldsymbol{t}^j$ as a direction for which we propagate Taylor polynomials of order o. In other words we have $d \equiv \binom{n+o-1}{o}$ and $\boldsymbol{s}^i \equiv \boldsymbol{t}^i, i = 1 \ldots d$. The resulting Taylor coefficients of the output $\boldsymbol{y}_j^i$ can then be interpolated to retrieve the elements of any $\mathcal{D}^j, j = 1 \ldots o$ again identified by their respective multi-indices $\boldsymbol{t}$ with $|\boldsymbol{t}| \equiv j$. The precomputed interpolation coefficients depend only on o and n.

Chapter 2

Usage

Not surprising, the usage follows the same pattern as all overloading-based AD tools. The pattern can be summarized into the following four steps.

1. identify the set of input variables $\boldsymbol{x}$ and the set of output variables $\boldsymbol{y}$ and a section of the program or top-level subroutine by which the model $\boldsymbol{f}$ is computed;

2. change the type of the inputs and outputs and all variables on dependence paths from the inputs to the outputs to the generated active floating point type. In many cases a global type change for all floating point variables may be appropriate;

3. adjust the code, e.g. statements for I/O and memory allocation;

4. in the driver logic that surrounds the model computation, inject logic to initialize the Taylor coefficients of the inputs and retrieve the Taylor coefficients of the outputs and call the higher order tensor drivers when needed;

5. generate the overloading library and compile and link the source files.

The following Sec. 2.1 through Sec. 2.5 illustrate these steps in detail.

2.1 Identifying the Model Structure

For brevity we refer to the numerical model computation as $\boldsymbol{f}$ with inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. Often the scenario in a real numerical program is a bit more complicated. The inputs $\boldsymbol{x}$ may not be a vector of length n but rather a set of program variables consisting of scalars, vectors and matrices. The same applies to the outputs. For the sake of initialization of the coefficients with directions and retrieval of the result coefficients the programmer has to conceptually enumerate all the scalars, and the individual array elements to n scalar quantities. The Rapsodia interface does not actually require the inputs or outputs to be packed into floating point vectors of length n and m respectively. It does, however, require that the (conceptual) enumeration be kept consistent across all direction initializations and retrieval of tensor elements for the inputs and across retrieval of all result coefficients and passing them to the tensor interpolation routines for the outputs.

Often one will not have separate program variables for the inputs and the outputs but perhaps just some state variable. Here it is simply the location of initialization and result retrieval in the program that implies what the inputs and outputs are. Conceptually one should think of the *value* that a program variable attains at a certain stage in the program, rather than just the program variable.

Finally, we assume in the program a dedicated section for initialization of the inputs and the use of the outputs. Those locations would become the points at which the independents are initialized and the dependents results are retrieved. The comments in the code shown in Fig. 2.1 [1] illustrate the inputs, the outputs and the model section. We have the vector `(a,b)` for both, the inputs and the outputs and assume our enumeration to be

$$(x_1, x_2) = \texttt{(a,b)} = (y_1, y_2) \quad .$$

In the following sections the use of this enumeration will be explained.

[1] See also `RapsodiaExamples/CppStepByStep/driver0.cpp`

```
#include <iostream>
#include <cmath>

int main(void){

  double a,b; // inputs & outputs
  double c,d; // intermediate

  double p=3.14; // a parameter
  // initialize inputs
  a=1.5;
  b=2.5;
  // here comes the program section
  // where the mock up model is computed
  c=sin(a*b);
  d=cos(c+b);
  a=exp(c*p);
  b=sqrt(−d*p);
  // now we are done with the model
  // and use the result, e.g. print it
  std::cout << "a=" << a << std::endl;
  std::cout << "b=" << b << std::endl;
}
```

Figure 2.1: The comments identify the inputs, the outputs and the model computation section.

in Fortran		
real	4 Byte	`RArealS`
real	8 Byte	`RArealD`
complex	4 Byte	`RAcomplexS`
complex	8 Byte	`RAcomplexD`
in C++		
float		`RAfloatS`
double		`RAfloatD`

Table 2.1: Active types generated by default.

2.2 Change Declarations to Active Type

The code generator creates active types that encapsulate different base types and precisions. The types generated by default in Rapsodia are listed in Table 2.1 Based on the basic type and the precision in the original program one has to replace the original floating point type with the appropriate Rapsodia-generated active type. All Rapsodia declarations are provided via a single file that needs to be included.

C++: while forward declarations may suffice for some header files it is easier to just insert the directive

```
#include "RAinclude.ipp"
```

in every file that references any of the Rapsodia active types.

Fortran: every compile unit (e.g. a stand alone subroutine or module) that references the Rapsodia active types needs to

```
include 'RAinclude.i90'
```

Note, that we use the Fortran include syntax but the C preprocessor syntax would work as well.

Returning to our example, we find the variables `a` and `b` to be the inputs and outputs. The definition of active type indicates that all variables on a dependency path are active which in our example are also `c` and `d` but not `p` because there is obviously no path from the inputs to `p`. In AD tools that implement AD via source transformation the set of active variables is determined by a specialized data flow analysis on the source code. For operator overloading one could do a global type change but must observe some exceptions, see Sec. 2.3. In our example we change the `double` declarations at the beginning of `main` to

```
RAfloatD a,b;
RAfloatD d,d;
```

and leave the declaration of `p` untouched.

2.3 Adjustments after the Type Change

Any type change from a built-in floating point data type to one of the Rapsodia active types may encounter the following problems.

Fortran and C++: The two problem scenarios are not particular to AD but rather apply to any type change done inside a given program.

- I/O formats or the data access may have to be adjusted
- black box routines cannot be modified and must be called with the value component.[2]

C++: In many cases rectifying the following issues for AD lead to cleaner and more portable code.

- an active floating point variable may not be part of a `union`, see also Sec. 3.
- memory allocation with `malloc` must use the `sizeof` routine to determine the byte size of the active type and must not hard-code the byte size of the floating point type.
- if the asynchronous queue approach is used (see Sec. 2.5.2), all `malloc` must be replaced with the appropriate `new` calls.

Fortran Calls to user-defined subroutines with constant literals as actual arguments for formal arguments [3] must have an explicit type conversion. For example, a subroutine with a signature defined as

```
subroutine foo(a, b)
  real, intent(in) :: a
  real, intent(out) :: b
```

may in the code be called in another subroutine

```
subroutine bar(...)
  ...
  call foo(2.5,b)
```

but the change of the signature

```
subroutine foo(a, b)
  type(RArealS), intent(in) :: a
  type(RArealS), intent(out) :: b
```

then leads to a compile error for type mismatch if the interface of `foo` is known in `bar`. Otherwise the compiler assumes F77 bindings and that simply leads to undefined behavior including segmentation faults during execution. To remedy the situation one has to explicitly assure matching types in the call by introducing a temporary variable.

```
subroutine bar(...)
  type(RAfloatS) :: temp
  ...
  temp=2.5
  call foo((temp,b)
```

In the case of our example (cf. Fig. 2.1) we encounter the first listed item as a problem. The lines that write the output variables

```
std::cout << "a=" << a << std::endl;
std::cout << "b=" << b << std::endl;
```

have to be adjusted to print only the value component

```
std::cout << "a=" << a.v << std::endl;
std::cout << "b=" << b.v << std::endl;
```

because no output operator has been specified for the active type. While one could specify such an operator it would not be appropriate in all cases. For instance, when data are transmitted via file from one part of the model to another part, then the differentiated version should transmit the original value along with the Taylor coefficients. As long as the output operator remains undefined the compiler has a better chance of alerting the user to the problem and the user can make an *informed* adjustment.

[2] A black box routine may not modify a conceptually active program variable on a path from the inputs to the outputs. If it does, this implies parts of the model computation are not differentiated and consequently the results will be wrong.

[3]The Fortran standard calls them *dummy* arguments.

2.4 Initialization and Result Retrieval

As in the example discussed in Sec. 1.2 we have to initialize the input coefficients and retrieve the output coefficients. This time however we want to obtain the full derivative tensors. The directions for which Taylor coefficients have to be propagated and the interpolation are provided by an instance of `HigherOrderTensor` in both Fortran and C++. Here we continue with our C++ example[4]. We need to include the header file for `HigherOrderTensor`.

```
#include "HigherOrderTensor.hpp"
```

Here we have $n = m = 2$ and assume the order to be $o = 3$. We declare the order, the number inputs/outputs, variables to contain the number of directions, and the instance of `HigherOrderTensor`.

```
unsigned short  n=2,o=3;
HigherOrderTensor hot(n,o);
int dirs=hot.getDirectionCount();
std::cout << "number of directions = " << dirs << std::endl;
```

Here we also write out the number of directions as determined by `getDirectionsCount`. According to Sec. 1.3.3 this will be $d = \binom{n+o-1}{o} = 4$. Then we retrieve the seed matrix $\boldsymbol{S}$ and initialize the input coefficients $\boldsymbol{x}_1^i = \boldsymbol{s}^i \in I\!R^n, i = 1, \ldots, d$ in the program done as follows

```
// get the seed matrix
Matrix<unsigned int> seedMatrix=hot.getSeedMatrix();
// set the input coefficients
for(i=1;i<=dirs;i++) {
  a.set(i,1,seedMatrix[0][i-1]); // first input
  b.set(i,1,seedMatrix[1][i-1]); // second input
}
```

The `Matrix` template class is a very simple matrix container class provided together with the `HigherOrderTensor` class. The original model computation is left untouched and in the previous step we already made the adjustment for the outputs. After that we now retrieve the output coefficients into a coefficient matrix. The tensor interpolation interface is setup to do this separately for each scalar output. Here we do it for `a` as follows.

```
// declare a matrix for the output coefficients
Matrix<double> outputCoefficients(o,dirs);
// transfer the taylor coefficients of a to the matrix
for(i=1;i<=o;i++) {
  for(j=1;j<=dirs;j++) {
    outputCoefficients[i-1][j-1]=a.get(j,i);
  }
}
```

Now we supply the matrix to the interpolation using

```
hot.setTaylorCoefficients(outputCoefficients);
```

And finally we retrieve the compressed tensor and print the tensor elements. The compressed tensor is given as a vector whose element order is exactly the same as the order of multiindices in the seed matrix.

```
std::vector<double> compressedTensor=hot.getCompressedTensor(o);
// print the assoiciated multi-index from the seed-matrix and
// the tensor element value
for(i=1;i<=dirs;i++) {
  std::cout << "a";
  for(j=1;j<=n;j++)
    std::cout << "[" << std::setw(1) << seedMatrix[j-1][i-1] << "]";
  std::cout << " = " << compressedTensor[i-1] << std::endl;
}
```

The output will be shown and explained in the next section. The source code complete with all changes discussed so far is shown in Fig. 2.2.

[4]See `RapsodiaExamples/CppStepByStep/driver.cpp`. The Rapsodia regression tests also contain a Fortran example where the respective Fortran usage can be seen in file `RapsodiaExamples/F90StepByStep/driver.f90`.

```
#include <iostream>
#include <cmath>

#include <iomanip>

#include "RAinclude.ipp"
#include "HigherOrderTensor.hpp"

int main(void){
  int i,j;
  unsigned short  n=2,o=3;
  HigherOrderTensor hot(n,o);
  int dirs=hot.getDirectionCount();
  std::cout << "number of directions = " << dirs << std::endl;

  RAfloatD a,b;   // inputs & outputs
  RAfloatD c,d;   // intermediate

  double  p=3.14; // a parameter
  // initialize inputs
  a=1.5;
  b=2.5;

  // get the seed matrix
  Matrix<unsigned int> seedMatrix=hot.getSeedMatrix();
  // set the input coefficients
  for(i=1;i<=dirs;i++) {
    a.set(i,1,seedMatrix[0][i-1]); // first input
    b.set(i,1,seedMatrix[1][i-1]); // second input
  }

  // here comes the program section
  // where the mock up model is computed
  c=sin(a*b);
  d=cos(c+b);
  a=exp(c*p);
  b=sqrt(-d*p);
  // now we are done with the model
  // and use the result, e.g. print it
  std::cout << "a=" << a.v << std::endl;
  std::cout << "b=" << b.v << std::endl;

  // declare a matrix for the output coefficients
  Matrix<double> outputCoefficients(o,dirs);
  // transfer the taylor coefficients of a to the matrix
  for(i=1;i<=o;i++) {
    for(j=1;j<=dirs;j++) {
      outputCoefficients[i-1][j-1]=a.get(j,i);
    }
  }
  // supply the coefficients to the interpolation utility
  hot.setTaylorCoefficients(outputCoefficients);
  // harvest the compressedTensor,
  // here for the highest degree o=3:
  std::vector<double> compressedTensor=hot.getCompressedTensor(o);
  // print the assoiciated multi-index from the seed-matrix and
  // the tensor element value
  for(i=1;i<=dirs;i++) {
    std::cout << "a";
    for(j=1;j<=n;j++)
      std::cout << "[" << std::setw(1) << seedMatrix[j-1][i-1] << "]";
    std::cout << " = " << compressedTensor[i-1] << std::endl;
  }
}
```

Figure 2.2: The source code of Fig. 2.1 augmented for Rapsodia.

2.5 Generate the Library and Compile

The top-level generator script can be found in `Generator/generator.py`. It requires that two command line parameters be specified, `-o` <*integer*> to set the maximal derivative order (in our example this is 3) and `-d` <*integer*> to set the number of directions (in our example this is 4). As the extension indicates this is a Python script. Often the top-level Python interpreter binary is installed as

```
/usr/bin/python
```

If the Python interpreter cannot be found in the `PATH` one needs to explicitly prepend the path to the command line. Otherwise it can be left off and one just invokes

```
generate.py -d 4 -o 3 -c ./RALib
```

The `-c` flag indicates the name for the subdirectory into which the Rapsodia source files should be generated. The directory is relative to the current working directory. The complete set of command line options of the generator can be displayed in standard Python fashion with the `-h` switch.

```
Usage: generate.py { -d <DIRECTION> -o <ORDER> { -f <FDIR> | -c <CDIR> } [options] } | {-r [options]}

Options:
  -h, --help            show this help message and exit
  -d DIRECTIONS, --directions=DIRECTIONS
                        the number of univariate directions (required)
  -o ORDER, --order=ORDER
                        the maximal order of Taylor coefficients (required)
  -f FDIR, --fdir=FDIR  if specified, generate F90 files into FDIR
  -c CDIR, --cdir=CDIR  if specified, generate C++ files into CDIR
  -s SLICES, --slices=SLICES
                        number of data slices; defaults to 1
  --openmp              if specified along with --slices, adds OpenMP
                        directives to parallelize sliced code
  --openmpChunkSize=OPENMPCHUNKSIZE
                        if specified along with --openmp, sets the chunk size
                        to schedule per iteration (defaults to 1)
  --orphan              if specified along with --openmp, uses orphaned OpenMP
                        directives (EXPERIMENTAL)
  -q, --queue           use a queue and threads to calculate derivatives
                        asynchronously; usable only with -c, conflicts with -f
  -t, --temporariesBug  workaround compiler bugs in sunCC and xlC related to
                        unnamed temporaries in expressions; requires  -q
  --disableInit         use this to improve efficiency but only if the code
                        does not contain array initialization with partial
                        initialization lists; refer to the  manual for
                        details;
  --useOPA              use OpenPA in queue implementation; implies -q
  --inline              generate C++ files using the inline directive
  --interoperable       generate interoperable type declarations using C
                        structs for C++ and iso_c_binding for Fortran
  --fixedFormat         generate Fortran files in fixed format; default is
                        free format
  --cppHeaderExtension=CPPHEADEREXTENSION
                        file extension for C++ header files; default is '.hpp'
  --cppIncludeExtension=CPPINCLUDEEXTENSION
                        file extension for common C++ code snippets included
                        in the generated code; default is '.ipp'
  --cppSourceExtension=CPPSOURCEEXTENSION
                        file extension for C++ source files; default is '.cpp'
  --fortranIncludeExtension=FORTRANINCLUDEEXTENSION
                        file extension for Fortran includes; default is '.i90'
  --fortranSourceExtension=FORTRANSOURCEEXTENSION
                        file extension for Fortran source files; default is
                        '.f90'
  --floatingPointExceptions
                        generate extra code that tests for boundary cases and
                        prevents some floating point exceptions
  --sequenceType        generate Fortran derived types as SEQUENCE types
                        needed for use in common blocks or other sequence
                        types
  --withOpenADconversions
                        generate definitions for conversion routines reference
                        in code that has was transformed for type change with
                        OpenAD
  -r, --reverse         enable reverse mode
  --doubleOnly          double presicion only [Default=False] [Reverse Mode]
  --tl=TL               local tape size       [Default=1000]  [Reverse Mode]
  --ll=LL               initial location size [Default=4000]  [Reverse Mode]
  --bs=BS               stack block size      [Default=4096]  [Reverse Mode]
  --loc                 accumulation on int   [Default=False] [Reverse Mode]
                        [Fortran Only]
```

```
1  ifndef RAPSODIAROOT
2   $(error "environment␣variable␣␣RAPSODIAROOT␣undefined")
3  endif
4  include ${RAPSODIAROOT}/MakeDefs.mk
5  include ${RAPSODIAROOT}/MakeRules.mk
6
7  default: driver
8          ./$^
9
10 GEN_DIR=RALib
11
12 RA_EXTRAS=${RAPSODIAROOT}/hotCpp
13 IPATH+=-I$(GEN_DIR) -I$(RA_EXTRAS)
14
15 OBJS= \
16 $(addprefix $(RA_EXTRAS)/, $(addsuffix .o, $(HOTCPPNAMES))) \
17 driver.o
18
19 driver: $(OBJS) $(GEN_DIR)/libRapsodia.a
20         $(CXX) $(CXXFLAGS) $(IPATH) -o $@ $^
21
22 $(GEN_DIR)/libRapsodia.a : FORCE
23         ${RAPSODIAROOT}/Generator/generate.py -d 4 -o 3 -c $(GEN_DIR)
24         cd $(GEN_DIR) && $(MAKE)
25
26 FORCE:
27
28 clean:
29         rm -rf $(GEN_DIR) *.o driver driver.out
30
31 .PHONY: default clean
32
33 driver.cpp:$(GEN_DIR)/libRapsodia.a
```

Figure 2.3: `Makefile` for our example.

```
number of directions = 4
a=0.166177
b=1.04843
a[0][3] = -15.9285
a[1][2] = -22.3432
a[2][1] = -37.2386
a[3][0] = -73.7429
```

Figure 2.4: Output generated by running our example.

One should specify at least one of the switches `-c` or `-f`, otherwise no source would be generated. An example for the generate and build steps is given in `RapsodiaExamples/CppStepByStep/Makefile` (see also Fig. 2.3). The generate step is on line 23. The generator writes a set of files into the directory specified with the `-c` switch, see lines 10 and 23. Among the generated files are a single file `RAinclude.ipp` (or `RAinclude.i90`, resp.) for comprehensive inclusion of generated definitions in user code, To compile the user code one extends the include path by this directory, see line 13. Among the generated files is also a `Makefile` to compile the generated library itself which is in our example accomplished by the action on line 24, that is, it immediately follows the code generation.

The compile rules and definitions of the other variables is supplied by the files included on lines 4 and 5. These files were created during the configuration of Rapsodia (Sec. 1.1.2) and reflect the choice of compilers and other options made at that time. These files are intended to maintain consistency in settings between the support code, the generated code, and the user code with respect to the compiler and preprocessing flags, e.g. for inlining. Their use is recommended in the user build setup but not strictly required. As their names suggest

`MakeDefs.mk` provides definitions of the commonly used variables for compilers, linkers and flags and also - as a consistency check - tests these variables if they were predefined for consistency; it also sets variables for mixed language compilation and linking, see Sec. 2.5.5 and to indicate what features are enabled.

`MakeRyles.mk` provides common suffix rules.

Because the example uses the higher-order tensor logic (cf. Sec. 2.4) we need to link the respective object files that are provided with `HOTTCPPNAMES` (or `HOTF90NAMES`), see lines 12 and 16.

At the end of Sec. 2.4 we added code to print the interpolated tensor entries. The produced output from the compiled example is shown in Fig. 2.4. Following the explanation in Sec. 1.3.3 we see that there are four distinct tensor elements for

column in $\boldsymbol{S}$	multiindex	entry in $\mathcal{D}^3$	compressed tensor index
1	$(0,3)$	$\frac{\partial^3 \mathtt{a}}{\partial \mathtt{b}^3}$	1
2	$(1,2)$	$\frac{\partial^3 \mathtt{a}}{\partial \mathtt{a} \partial \mathtt{b}^2}$	2
3	$(2,1)$	$\frac{\partial^3 \mathtt{a}}{\partial \mathtt{a}^2 \partial \mathtt{b}}$	3
4	$(3,0)$	$\frac{\partial^3 \mathtt{a}}{\partial \mathtt{a}^3}$	4

Table 2.2: Tensor entries ordered by multiindex (see also Fig. 2.4). While there is only one program variable `a` here its use distinguishes the different *values*, i.e. the input value and output value that `a` can attain.

$n = 2$ and $o = 3$, that is we need to set $d = 4$. The third order tensor of the output `a` with respect to the two inputs `a` and `b` has therefore the entries listed in Table 2.2.

In the example code we retrieve only the coefficients for the first output variable `a`. To retrieve the tensor for the second output variable `b` the only thing to change would be the line that populates the `outputCoefficients` matrix to

```
outputCoefficients[i-1][j-1]=b.get(j,i);
```

Without any new computation, one can retrieve the entries of the lower order tensors. For instance for the Hessian entries one could add

```
// harvest the compressedTensor,
// here for the degree 2
std::vector<double> compressedHessian=hot.getCompressedTensor(2);
// to print it nicely get the multiindices for the Hessian
HigherOrderTensor hessHelp(n,2);
dirs=hessHelp.getDirectionCount();
// get the hessHelp seed matrix
Matrix<unsigned int> hessHelpSeed=hessHelp.getSeedMatrix();
for(i=1;i<=dirs;i++) {
  std::cout << "a";
  for(j=1;j<=n;j++)
    std::cout << "[" << std::setw(1) << hessHelpSeed[j-1][i-1] << "]";
  std::cout << " = " << compressedHessian[i-1] << std::endl;
}
```

A different example can be found in the regression test cases for the function

$$y = \boldsymbol{f}(\boldsymbol{x}) = \prod_{i=1}^{3} \sin(x_i) \quad .$$

The directories `RapsodiaExamples/hotCppR/` and `RapsodiaExamples/hotF90R/` contain the C++ and the Fortran version of this example, respectively.

2.5.1 Slicing for large problems using `--slices`

As the values for `-d` and `-o` increase, the generated code grows exponentially. To reduce this code explosion, Rapsodia provides the `-s/--slices` flag, which, when specified, breaks the generated code into the given number of slices. Thus, rather than having a flat data structure to maintain the derivatives, the structure is broken into n arrays, where n is the number of slices specified by the user. This greatly reduces code bloat and increases both the speed of compilation and even the speed of execution for certain operators.s Consider the following call to the generator.

```
generate.py -d <d> --slices <s> ....
```

Given the number d of directions **the recommendation is that number s of slices be a divisor of** d or else `generate.py` will issue a warning message and internally increase d to be the next largest multiple of s. Our runtime experiments indicate that the overhead for s not being a divisor of d is significant. Slicing is a prerequisite for the parallel execution described in Sec. 2.5.2 and the performance tests are described in [1].

```
ifndef RAPSODIAROOT
 $(error "environment variable  RAPSODIAROOT undefined")
endif
include ${RAPSODIAROOT}/MakeDefs.mk
include ${RAPSODIAROOT}/MakeRules.mk

GEN_DIR=RALib

ifndef RA_USE_QUEUE
default:
	@echo "Rapsodia was not configured for using the queue setup"
else
default: driver
	./$^

RA_EXTRAS=${RAPSODIAROOT}/hotCpp

IPATH+=-I$(GEN_DIR) -I$(RA_EXTRAS)

OBJS= \
$(addprefix $(RA_EXTRAS)/, $(addsuffix .o, $(HOTCPPNAMES))) \
driver.o

driver: $(OBJS)
	$(CXX) $(CXXFLAGS) $(IPATH) -o $@ $(OBJS) -L$(GEN_DIR) $(RA_CXX_LIBS)

$(GEN_DIR)/libRapsodia.a: FORCE
	$(RAPSODIAROOT)/Generator/generate.py -d 126 -o 4 -c $(GEN_DIR) -s 3 -q
	cd $(GEN_DIR) && $(MAKE)

FORCE:
endif

clean:
	rm -rf $(GEN_DIR) *.o driver

.PHONY: default clean

driver.cpp:$(GEN_DIR)/libRapsodia.a
```

Figure 2.5: `Makefile` for queue example.

The above is illustrated in the examples found under `RapsodiaExamples` in `F90StepByStepSlices` and `CppStepByStepSlices`, see Sec. 2.8. The use of the slicing is entirely encapsulated in the generated library. Therefore the source code in `driver.f90` and `driver.cpp` is identical to the respective driver files in `F90StepByStep` and `CppStepByStep` while the only difference in the respective `Makefile`s is the addition of the `-s` option, cf. Fig. 2.3.

2.5.2 Parallelized execution using `--queue`

A parallelization is possible because the propagation of Taylor polynomials in each of the d directions are mutually independent except for the coefficient a_0, see (1.2). Therefore the set of polynomials may be split into slices, see Sec. 2.5.1, and the propagation of each slice can be assigned to a separate thread to make use of current shared memory multicore hardware. To reduce execution dependencies between the threads the propagation can be dispatched asynchronously by placing the necessary information into a queue from which the propagation threads read. The approach is described in detail in [1].

In order to use the queuing functionality, **the library must be configured with `--queue`**, see also Sec. 1.1.2. This flag causes modifications in the `MakeDefs.mk` and `MakeRules.mk` files which the `configure.py` script creates in `Rapsodia/`. Then the generator **must be called with the `--queue` option in conjunction with `--slices`** where the number of slices specified determines the number of propagation threads to be launched. The generated library is adapted to setup the queue and launch the threads such that no changes in the user code are necessary.

The above is illustrated in the example found in `RapsodiaExamples/CppQueue`, see Sec. 2.8. As already mentioned, there is no logic in `CppQueue/driver.cpp` that is specific to using the queue. Reviewing the `CppQueue/Makefile`, see Fig. 2.5, one notices in comparison to Fig. 2.3 the only two essential differences to be the flags `-s 3 -q` (short for `--slices` and `--queue`) at the call to `generate.py` and the `$(RA_CXX_LIBS)` referenced in the link step. The latter is set in `MakeDefs.mk` to refer to the POSIX threading library and optionally to the OpenPA library if so configured, see also Sec. 1.1.2.

2.5.3 Parallelized execution with OpenMP (experimental)

An experimental addition to Rapsodia is the ability to test various AD parallelism techniques implemented with OpenMP. These have not been fully optimized, so the results may be slower than expected. They provide a first look at using parallelism with AD and an easy way to test various approaches.

To enable OpenMP parallelism, use the `-m/--openmp` flag, where the flag parameter represents the chunk size (for most scenarios this can be set to `1`). `-m` must be used along with `-s/--slices`. The options used in association with OpenMP are as follows:

`-m / --orphan` - Causes the generated code to use orphaned OpenMP directives.

`--queue` - Places derivative computations in a queue running in a separate thread to be computed asynchronously from the main program. It can be used in conjunction with `--orphan`.

2.5.4 Language Specific Options

Inevitably there are a few language specific options that one needs to control the code generation.

2.5.4.1 C++

`--inline` **:** inject `inline` directives into the C++ code to guide compiler optimization. This changes the structure of the generated library. The `.cpp` are now included by the respective `.hpp` files as opposed to the non-inlined version where the `.hpp` files are inlcuded the `.cpp` files.

file naming options: The options

`--cppHeaderExtension`

`--cppIncludeExtension`

`--cppSourceExtension`

as their names suggest, determine the extensions of the generated file names.

`--disableInit` **:** the default constructors of the active types behave as if each declared variable was explicitly initialized to `0.0`. This is necessary for the case of an implicit zero initialization of arrays of active variables through a (partial) initializer list. Consider the example declaration

```
double a[3]={2.0};
```

which according to the C++ standard implies that `a[0]` is initialized with `2.0` and `a[1]` and `a[2]` are initialized with `0.0`. If the statement is changed to the active type

```
RAfloatD a[3]={2.0};
```

the semantics has to be preserved. The first element `a[0]` is constructed with the explicitly given value by calling `RAFloatD(2.0)`. However, for the remaining implicitly initialized array elements, rather than calling the `RAFloatD(const double&)` constructor and passing `0.0`, the C++ standard prescribes calling the default constructor. Thus, these array elements remain improperly uninitialized and we have the wrong semantics unless we force the default constructor to always assume initialization to `0.0`. An example can be found in `RapsodiaExamples/CppArrayInit`, see Sec. 2.8.1. If one can ascertain the absence of constructs of the above type one can generate with `--disableInit` and thereby improve the efficiency.

2.5.4.2 Fortran

`--fixedFormat` **:** generated Fortran code adheres to fixed format syntax.

file naming options: The options

`--fortranIncludeExtension`

`--fortranSourceExtension`

as their names suggest, determine the extensions of the generated file names.

`--sequenceType` **:** injects the `sequence` qualifier into the Rapsodia type definitions. This qualifier is needed when the Rapsodia types are used in `common` blocks or within other derived types that themselves are declared as `sequence` type, see also Sec. 2.8.2. Because this can restrict some compiler optimization it is optional.

....

2.5.5 Mixed Language Models

2.6 ERROR Catalogue

error	explanation
RE1	A get or set method was called with a value for the direction parameter that is less than 1 or larger than the value for d supplied to `generate.py`, when the library was generated. Adjust the parameter in your code or regenerate the library.
RE2	A get or set method was called with a value for the order parameter that is less than 1 or larger than the value for o supplied to `generate.py`, when the library was generated. Adjust the parameter in your code or regenerate the library.

2.7 Simple Generator Modifications

While the user can in principle change all aspects of the generator by virtue of modifying the Python source code, we point out the intended approach to quickly achieve some simple modifications.

2.7.1 Changing Prefixes and Names in the Generated Code

Because the Rapsodia library should be usable in the context of any large Fortran or C++ environment, we made the file and type names configurable such that naming conflicts can be easily avoided. All file and type names have a tool prefix that is by default defined in

`Generator/Common/names.py`

as `pN = 'RA'`. In cases of name clashes, this definition can be changed. The same file also contains the definitions for the other portions of precision, type, and structure element names referenced by the generator.

2.7.2 Changing Precision and Type

Fortran and C++ differ in the determination of the precision and the available floating-point types. Table 2.1 lists the default precision and type combinations. The lists of basic types and precision variants are associated with the language specific printer classes. The respective classes for C++ and Fortran are defined in `Generator/Cpp/Printer.py` and `Generator/F90/Printer.py`. Each printer class has a

list of types: `typeList`

and a

dictionary[5] of precisions: `precDict`.

The combination of the two yields all the generated active types. In Fortran the precision dictionary is used to predefine `kind` values generated into `RAprec.f90`. In C++ we just use `typedef` to define the respective variants in the generated file `RAprec.hpp`. The order (lower to higher) in `typeList` and `precDict` is significant because we compare the position to determine the result type that is generated for bivariate, mixed-type elementary operations. For passive arguments we initialize `passiveTypeList` to the `integer` type (and `int` for C++, resp.) to which all combinations of `typeList` and `precDict` are appended in order to cover bivariate elementary operations with a passive argument.

2.8 More Examples

All source code references in this section refer to the `RapsodiaExamples` repository, see Sec. 1.1.4.

`CppOneMinute` - is discussed in Sec. 1.2

`CppQueue` - is discussed in Sec. 2.5.2

`CppStepByStepSlices` / `F90StepByStepSlices` are discussed in Sec. 2.5.1

[5] the Python kind

2.8.1 CppArrayInit - C++ array instatiation with initializer list

The source code for this example can be found in `CppArrayInit`. This example illustrates the need for the explicit setting of the Taylor coefficients in the default constructor. The line in question is
`RAfloatD x[n]=2.0;`
where a value of `2.0` is assigned to the first, and `0.0` to each subsequent array element. The initialization is controlled by the generator flag `--disableInit`. The rationale for it is discussed in Sec. 2.5.4.1.

2.8.2 F90SequenceType - Fortran with `common` blocks and derived types

The source code for this example, see Fig. 2.6 can be found in `F90SequenceType` and demonstrates the use of `--sequenceType`, see also Sec. 2.5.4. The use of the `RARealD` type in the common block `cb` (lines 15 and 26) as well as in the type `t` (line 6)

```
 1  module m
 2    include 'RAinclude.i90'
 3    public
 4    type t
 5       sequence ! makes sequence in RARealD necessary
 6       type(RARealD) :: x,y
 7    end type t
 8  end module m
 9
10  subroutine foo(o)
11    include 'RAinclude.i90'
12    use m
13    implicit none
14    common /cb/ a,b,c
15    type(RARealD) :: a,b,c
16    type(t) :: o
17    o%x=sin(a*b) ! made up for demonstration
18    o%y=cos(a*c) ! made up for demonstration
19  end subroutine
20
21  program driver
22    include 'RAinclude.i90'
23    use m
24    implicit none
25    common /cb/ a,b,c ! makes sequence in RARealD necessary
26    type(RARealD) :: a,b,c
27    type(t) :: output
28    real(kind=RAdKind) :: temp
29    character(*), parameter :: fString='(I6,I10,A,E25.17E3)'
30    integer :: i,j
31    integer :: n=3 ! for a,b,c
32    integer :: o=3, d=2 ! made up for demonstration
33    a = 1.5; b = 2.5; c=3.5
34    do j = 1, d ! set first order coefficients
35       call RAset(a, j, 1, 1.0D0)
36       call RAset(b, j, 1, 2.0D0)
37       call RAset(c, j, 1, 3.0D0)
38    end do
39    call foo(output) ! compute output
40    write (*,'(A)') 'output␣values:'
41    write (*,'(A,E25.17E3)') '␣output%x␣=␣', output%x%v
42    write (*,'(A,E25.17E3)') '␣output%y␣=␣', output%y%v
43    write (*,'(A)') 'Taylor␣Coefficients:'
44    write (*,'(A)') '␣order␣direction␣value'
45    do i = 1, o
46       do j = 1, d
47          call RAget(output%x, j, i, temp)
48          write (*,fString) i,j,'␣output%x␣=␣', temp
49          call RAget(output%y, j, i, temp)
50          write (*,fString) i,j,'␣output%y␣=␣', temp
51       end do
52    end do
53  end program
```

Figure 2.6: Source code of `F90SequenceType/driver.f90`

which itself is declared as `sequence` requires that `RARealD` be defined with the `sequence` qualifier as well. One simply passes the `--sequenceType` to the generator call, see `F90SequenceType/Makefile` to effect this change.

```
time used (sec.microsec) is: 0.006228
for 1000 repeats
```

Figure 2.7: Exemplary first two lines of output of `CppSimpleTiming/driver`

2.8.3 Timing Examples

Of particular interest in the context of Rapsodia is the ratio of the time needed to computate the desired respective derivatives over the time needed to compute the numerical model (1.1) itself. While theoretical estimates in terms of numbers of operations and memory accesses are known [4] the practical results significantly depend on a variety of factors.

- compiler vendor / compiler version / optimization flags
- target programming language
- hardware platform and OS (version)
- the memory footprint of the original code implementing (1.1)
- the choices of o and d
- the percentage of linear/non-linear operations
- the amount of time spent in logic that is not to be differentiated (I/O. memory management etc.)

Because there are so many factors impacting the performance the true timing ratio has to be determined for the given implementation. Here we give a few generic examples to give an impression for the performance to be expected but do not claim to have covered a large portion of the application space. The examples should serve as a starting point for a user's own timing experiments.

2.8.3.1 CppSimpleTiming

The example measures the time taken to compute the 50th derivative of

$$y = e^{x^{\frac{4}{5}}}$$

by running 1000 repeats of the computation. The output shown in Fig. 2.7 was taken from an execution of the test code on a Linux virtual machine running on a Macbook Pro (2.66 GHz Intel Core i7) compiled with g++ -O3. It shows the execution of a single run in about $6\mu s$ to be in line with what one would expect based on the operations count.

2.8.4 Extras

This section covers the examples found under `RapsodiaExamples/Extras`.

2.8.4.1 PTHO99 - Fortran with blas and lapack

This example illustrates a brute force differentiation through a subset of the lapack and blas routines. **While we generally recommend to explicitly code library derivatives instead of differentiating through libaries, it illustrates some aspects of the Rapsodia usage**. – to be completed –

2.8.4.2 Timinigs - for various compilers and hardware

– to be completed –

2.9 Regression Tests

All source code references in this section refer to the `RapsodiaRegression` repository available in the same way as the examples repository, Sec. 1.1.4.

2.9.1 CppHotN1 and F90HotN1 - test multiindex corner case for $n = 1$

The source code for these examples can be found in `CppHotN1` and `F90HotN1` respectively. These test the logic implemented for the multiindices if $n = 1$ in (1.1).

Chapter 3

Generator Design and Extensions

This section provides some details on the design of the code generator, the libraries it produces and instructions on how to implement the code generation for an elementary function. Readers may skip this section unless they intend to fundamentally modify the code generator or extend its functionality. The source code is released under the LGPL. We would encourage any extensions to be contributed to the Rapsodia main line. The process is explained on the Rapsodia website.

The names given in the following sections are relative to the `Generator` subdirectory.

3.1 Design

We selected Python [8] to implement the generator, in part because we were able to use a code generator in PETSc [6] as a starting point, but mainly because it is readily available in most computing environments and the Python programming model appears to be a good fit for the size of this project. The Rapsodia generator consists of three major parts.

1. definitions for classes that form the elements of an abstract syntax tree (AST); see in particular `Generator/Common/ast.py`
2. methods to generate the Taylor coefficient propagation logic expressed as an AST; see the code in `Generator/generate.py` and the code in the files `Generator/Common/genOp*.py`
3. methods for printing the AST-based representation as source code; see `Generator/F90/Printer.py` for Fortran and `Generator/Cpp/Printer.py` for C++

3.1.1 Aspects of the Code Generation

1. Unrolled Loops and Flat Data Structure Hand-written code and general-purpose AD tools will typically use loops and arrays to implement the propagation logic of the Taylor coefficients given as formulas for instance in [4]. Our code generator creates an overloading library in which all loops are unrolled for a fixed order and fixed number of input variables and in which the active type is represented as a flat structure, that is, without the use of arrays. The observed performance improvements can be attributed to widening the scope of such compiler optimizations that are normally limited by the loop and other control flow constructs, and to reducing the conservative aliasing overestimate because of the lack of array accesses. We recognize and exploit the fact that the underlying semantics of the Taylor coefficient formulas can be rewritten as code generating logic. This logic has loops and the notion of indices that are similar to the hand-written Taylor propagation code. However, instead of computing values, the code generator creates expressions and concatenates indices to variable names while iterating through these loops. Thus, it creates the exact same semantics but at a syntactic level that is considerably lower than typical hand-written code and is more amenable to compiler-level optimization. We are aware of the plethora of different ways to express the propagation semantics as generated source code. One might consider a range, perhaps beginning from simple variations of AD02-like source with loops that have compile-time constant loop bounds to source code that uses additional local variables to aid register allocation. While the former falls short of the code generator's potential, the latter would already be somewhat hardware and compiler specific. We believe our approach constitutes a plausible compromise.

2. Complete Set of Type Combinations The generator creates *all* relevant argument combinations for elementary operations, assignments and copy constructors based on a list of precisions and types. Fortran does not provide default (up) conversions of the built-in types to facilitate a match to overloaded methods. Overloaded versions for all type combinations have to be defined explicitly and the generator can easily take over this arduous task.

The result type is determined based on argument types emulating in a limited fashion the language built-in typing rules. In particular, for assignments (and copy constructors) we do not generate assignments that would inadvertently permit a loss of information or precision such as assigning an active variable to a floating-point variable. Generating all the combinations prevents implicit conversions in particular in C++ using copy constructors from passive to active arguments for elementary operations and therefore avoids extraneous computations for variables that are not active.

3. Common Logic for Fortran and C++ While it is easy to see the common logic for implementing the Taylor propagation for the elementary operations some of the other parts have less commonality. For the outer structure of the library we can still achieve some common design, thus eliminating language specific generator portions. Rather than making the definitions of the elementary operations type members, we decided to keep them separate. In particular, for some of the C++ operators one might normally prefer the member declaration over a separate, nonmember declaration, but this approach would not cover to all argument combinations of binary operators, and Fortran does not have the syntactic option. A welcome side effect of separate source files is a less bulky compile step. For a nontrivial order and number of directions, the compiler optimization can take a noticeable amount of time and memory. Splitting up the library source files in this fashion reduces the compiler's resource requirements.

3.1.2 The Abstract Syntax Tree

Many elements found in the AST implementation are what one would expect for languages such as Fortran. Their meaning and use are either obvious from the `class` name or explained in the accompanying `doc` string. We did not attempt to abstract concepts from Fortran and C++ to the greatest possible extent because of the limited scope of the code that is to be generated. More abstraction would unnecessarily complicate the AST generation step. Consequently, the AST is restricted to the relevant language features, and we allow some shortcuts for either C++ or Fortran language-specific features; see also Sec. 3.2. The latter are used sparingly and therefore are not a conceptual concern. Each AST node subclass implements an `accept` method that calls a specialized method on a supplied `visitor` instance. For instance, the `accept` of `Assignment` calls `visitor.visitAssigment`. The implementation of the latter is explained at the end of this section.

3.1.3 Overview of Generated Code

The principal building blocks of the overloading library consist of language specific portions (items 1–4) for the building blocks of the active types on which we operate and the main part, overloading the elementary functions (item 5), as follows.

1. A set of definitions for floating-point precision:
   This is generated from the `precDict` contents by the following methods.
   C++: `Cpp.Printer.CppPrinter.generatePrecision`
   Fortran: `F90.Printer.F90Printer.generatePrecision`
   See also Sec. 2.7.2
2. A set of active types:
   This is generated as an outer product of the `typeList` and the `precDict` contents by `generateTypes` defined in the same language-specific manner as `generatePrecision` above. See also Sec. 2.7.2.
3. Type-specific accessors for the derivative components via integer parameters for direction and order:
   Because the generated code uses a flat data structure it is convenient to have these methods such that the flat structure can be accessed in an array-like fashion. The generator uses the following methods again defined in the respective printer classes.
   C++: `generateTypeGetter` and `generateTypeSetter`
   Fortran: `generateTypeGetterSetter`
4. A set of assignment operators, and additionally for C++ a set of copy constructors:
   C++: `generateCopies` (as part of the active type class)
   Fortran: `generateAsgn` (in a separate module)
   See also Sec. 3.2.2.
5. For each elementary operation we generate
   - declarations for all relevant combinations of active arguments
   - definition stubs
   - definition bodies included in the stubs

   See also Sec. 3.3.
6. generate C++ and Fortran output

Because the language-specific portions of the output logic are contained in the respective printer classes it is reasonable to also place language specific generator logic (items 1-4) into these classes.

3.1.4 Printing the AST as Fortran or C++

Once the Taylor propagation logic has been generated as an AST, we can print the contents as Fortran or C++ code by visiting all nodes in the AST in depth-first fashion. As explained in Sec. 3.1.3 there is a printer class implementation for each language. Both inherit from `Common.depthFirstVisitor.DepthFirstVisitor`. This visitor base class provides default implementations for all the specialized `visit`<*node_name*> methods called in the `accept` implementations of the AST *node_name*. The default `visit` implementation does nothing other than to proceed to the current node's children. The subclasses `F90Printer` and `CppPrinter` override the default implementations as necessary to produce an output stream. For instance, `F90Printer` overrides `visitAssignment` such that it first prints the left-hand side by calling `accept` on the first child node, then prints the `identifier` member `'='`, and then prints the right-hand side via a call to `accept` on the second child node.

We want to point out a tradeoff in the AST design. Taken the assignment, for example, it is of course possible to declare specifically named members such as `mRHS` and `mLHS` for the left- and right-hand sides instead of the more anonymous first and second child in the base class list `Common.ast.Node.children`. On the other hand, such a specialization also requires a specialized default implementation for `DepthFirstVisitor.visitAssignment`, and one loses the convenience of the generic tree traversal done with `visitChildren`. Our choices represent a compromise we find to be balanced. The subclasses `F90Output` and `CppOutput` redirect their respective output streams into physical files.

3.2 Language-Specific Concerns

Numerous differences exist between C++ and Fortran. In the following we explain those that are relevant for the Rapsodia generator.

3.2.1 Module vs. Header and Source

A Fortran module file contains both declarations and subroutine definitions, while in typical, noninlined C++ the declarations are in some header file and the definitions in a separate source file. To allow for both concepts, the AST has an `ObjectSource` class that encapsulates declarations and definitions. For an `ObjectSource` instance, the Fortran printer produces a single file with the module source code, while the C++ printer produces a header and a separate source file. References to such declarations are done in Fortran by a `use` statement and in C++ by an `#include`, both of which are represented in the AST by an instance of the `Common.ast.ObjectReference` class.

3.2.2 User-Defined Type Assignments and Copy Constructors

In Fortran, assignments to a user-defined type can be defined in a module *different* from the module containing the type definition itself. In C++, such assignments have to be *members* of the defining class. Consequently, the logic to generate this code is language-specific and attached to the respective printer class. While the generator covers all argument combinations for the elementary operations, there remains a need for defining copy constructors for calling user-defined methods that have active arguments. Similarly to the example in Sec. 2.3 we can consider

```
void foo(const double& a, double& b);
```

whose signature may be changed to

```
void foo(const RAfloatD& a, RAfloat& b);
```

but when called with a constant literal

```
RAfloatD s;
foo(2.5, s);
```

there has to be a copy constructor to convert the constant literal into an `RAfloatD` instance. Like the assignment operators the copy constructors in C++ have to be members of the type defining class.

As indicated in Sec. 2.3 for Fortran this situation is currently not solvable via the library (because the standard does not provide a feature resembling a C++ copy constructor).

Fortran	C++
`a**b`	`pow(a,b)`
`int`	N/A
`nint`	N/A
`real`	N/A
`\=`	`!=`
`sign`	N/A
`sum`	N/A
`matmul`	N/A
`maxloc`	N/A
`maxval`	N/A
`abs(f)`	`fabs(f)`

Table 3.1: Partial list of differences in elementary operations

3.2.3 Overloading

While overloading in C++ allows declarations with different argument types and the same method name, Fortran module interfaces contain module procedure declarations with different names which resembles a manual name mangling. To cover both concepts we generate for each specific operator/intrinsic overloading two subtrees. One subtree is used for the module `contains` section (or the source file for C++). The second subtree is identical to the first except it does not contain the actual implementation body. It is used for declarations in module interfaces (or header files for C++) with a signature specific appendix (for Fortran) or all arguments with types (for C++). The declaration and definition subtrees are collected in two groups that are children of a common `ObjectSource` node. The printer class distinguishes the declaration from the definition context and produces the proper output.

3.2.4 Elementary Operations

With a few exceptions the uses of elementary operations ϕ are identical in Fortran and C++. One such exception is $\phi \equiv a^b$, in Fortran expressed as operator `a**b` and in C++ as an intrinsic function `pow(a,b)`. The respective printer classes ensure the proper representation. Another set of differences relates to the combinations of permitted argument types. While the generator produces all combinations of arguments, the printer classes filter out all invalid cases. In the same spirit as the limited generalization of the AST explained in Sec. 3.1, we view this pragmatic solution as a good compromise, given the limited scope of the generator. Table 3.1 gives a list with some differences regarding the elementary operations.

3.3 Implementing an Elementary Operation

To illustrate the steps for implementing an elementary operation, we use the multiplication operator for `a*b`. We start with the logic for the propagation of the Taylor coefficients. For a given activity pattern of the arguments this logic is identical across all variations of the actual active type. As for most elementary operations we generate the common logic into a separate file that is then included into the respective method stubs that implement the overloaded operation. The multiplication operator can have two active arguments, covered by `Common.genOpMult.genOpMultAABody`, or only one active argument. The latter is distinguished by either the left or the right operand being active; see a `Common.genOpMult.genOpMultAPBody`, where we simply multiply the value and all Taylor coefficients by the respective passive argument. The only distinction in the two method bodies (see the generated `RAmultAP`[1] and `RAmultPA` files) are the names of the active vs. the passive variable. The case for two active arguments requires a convolution on the Taylor coefficients for each direction i:

$$\sum_{j=0}^{o} a_j^i * b_{o-j}^i$$

. Because convolutions occur with a few variations in a number of elementary operations, we extracted the generator logic into `Common.util.generateConvolution`. The operator implementation body is written to file with the base name `RAmultAA` and the appropriate language-specific extension. We now need to generate the method definition stubs for all appropriate argument type combinations and the include statement for the logic generated above. The same logic can be used to also

[1] The generated file names have language-specific extensions left off here.

generate the accompanying method declarations by simply leaving off the parts for the method body, which completes the generation steps of the product operator.

Because the stub declaration and definition are common to all the bivariate operators, we extracted the logic into `Common.util.generateBinaryIntrinsic` and call this in `Common.genOpMult.genMult`. The `Common.util.Util` class also provides common elements for the generation of unary operators; see, for example, `Common.utile.Util.generateUnaryOpAll` used in the source file `Common/genOpSinCos.py`. Referring to the existing implementation as an example, one can extend the library for additional elementary operations.

3.4 The implementation of `HigherOrderTensor`

The source code for computing the multi-indices/directions $\boldsymbol{s}^i$ and the interpolation routines are in `Rapsodia/hotF90` and `Rapsodia/hotCpp`. Because of syntax differences, the interfaces differ slightly between C++ and Fortran. As with the `set` and `get` methods for the active types we did not abandon the more concise style of calling a member function in the C++ implementation just because Fortran does not provide it. All functionality is tied to a `HigherOrderTensor` object, which needs to be initialized with n and o. The direction count d as well as the matrix $\boldsymbol{S} = [\boldsymbol{s}^i] \in \mathbb{N}_0^{n\times d}$ can be obtained by calling `getDirectionCount` and `getSeedMatrix`, respectively. Assuming the Rapsodia library has been generated for the given o and d, one can then compute and set the output Taylor coefficients for a single output y_i as a $\mathbb{R}^{o\times d}$ matrix by calling `setTaylorCoefficients`. For that particular output variable one retrieves the entries of any $\mathcal{D}^j, j \in [1\ldots o]$ by calling `getCompressedTensor`. This returns a vector of length $\binom{n+j-1}{j}$ whose lth entry corresponds to the lth multi-index $\boldsymbol{s}^l$ returned by `getSeedMatrix` computed for order j. In particular for order o, `getCompressedTensor` returns a vector of length d whose lth entry corresponds to the lth direction/multi-index we propagated through our model. In the current implementation the pair of calls to `setTaylorCoefficients` and `getCompressedTensor` have to be repeated for each of the m outputs.

3.5 Contributing to the Rapsodia Source Code and this Manual

The development version of the Rapsodia sources and this manual are kept under mercurial revision control. Please refer to the Rapsodia website for details.

Acknowledgments

Utke was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

Contributions to the tool were made by Alexis Malozemoff (slicing and queue logic), Darius Buntinas (hookup to the OpenPA library), and Mu Wang (reverse mode).

Bibliography

[1] Darius Buntinas, Alexis J. Malozemoff, and Jean Utke. Multithreaded derivative computation with generated libraries. *Journal of Computational Science*, 1(2):89 – 97, 2010.

[2] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[3] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of Computation*, 69:1117–1130, 2000.

[4] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[5] Mercurial website. http://www.selenic.com/mercurial.

[6] Portable, Extensible Toolkit for Scientific Computation (PETSc). http://www.mcs.anl.gov/petsc.

[7] John D. Pryce and John K. Reid. ADO1, a Fortran 90 code for automatic differentiation. Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England, 1998.

[8] Python Programming Language. http://www.python.org/.

[9] R. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.