Forms

Gustav Meglicki (code and weave)

February 16, 2009

Contents

1 Copyright and Authors

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

2 Introduction

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

The program uses Fortran stubs for basic FDTD computations [18, 19], Chombo C++ wrapper for general logic and multigrid operations [3], and Guile, which is a dialect of Scheme [10], for media layout specifications, for media properties specifications, for injected signal specifications, for output specifications, and for virtual measurements.

The standard Chombo library is additionally enhanced by new templates and classes that provide basic utilities for face and edge mounted data in parallel computing context. These borrow from functions of the **staggeredChombo** package mercifully contributed to the project by Dan Martin of the Berkeley National Lab. Bless him! The **staggeredChombo** classes implement divergence-free averaging from fine to coarse grids, curl-refluxing corrections on the coarse side of the fine/coarse grid boundaries, and piecewise linear interpolation filling of cells on the fine side of the fine/coarse grid boundaries for face-mounted data. See [1, 14, 15] for more details.

The program also builds on experiences gained and ideas developed in an earlier two dimensional AMR FDTD program called SHAPES, which was described in [17].

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

◀ 

2.1 Building the Code

The code is built by invoking two Makefiles, which are listed in Section 14, page 380.

The first Makefile, called `srcMakefile`, extracts the program source from this Web document, then prettifies it by stripping Web comments and blank lines with `sed` and by indenting it with `astyle`. Finally, it moves most of the source so generated to the `src` subdirectory leaving only `Forms.cpp` and `Forms.H` in the main code directory, which is as Chombo wants it nowadays. The original Web files, and any auxiliary files generated on the way, are removed. The procedure leaves a clean C++/C/Fortran source behind. Another thing that the target `all` of `srcMakefile` does is to format this document and leave it on `doc/Forms.pdf`.

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

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

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

```
$ make −f srcMakefile
```

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

The generated source files should be as follows:

`Forms.cpp` The C++ code mainlines. Discussed in Sections 4, page 131, and 5, page 132.

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

`src/Auxiliary.cpp` This file contains code for various auxiliary functions, as defined in Section 10, page 220.

`src/Conversion.c` This file contains plain C wrappers for calling Scheme functions that convert the $\boldsymbol{D}$ fields to $\boldsymbol{E}$ fields. The wrappers are invoked from Fortran, which is why they have to be done in plain C. In future $\boldsymbol{B} \rightarrow \boldsymbol{H}$ conversions may be added as well. Discussed in Section 7.3.8, page 175.

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

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

`src/CurlBox.cpp` The C++ implementation of the **CurlBox** class. Discussed in Section 12.4, page 323.

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

`src/EdgeFabImplem.H` Implementation of the **EdgeFab** template. Discussed in Section 12.2, page 300.

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

`src/FaceFabImplem.H` Implementation of the **FaceFab** template. Discussed in Section 12.3, page 316.

`src/IO.cpp` C++ code for dumping three-dimensional HDF5 field images. In future, generation and reading of restart files will be added too. Discussed in Section 9, page 211.

`src/Initialize.cpp` This file contains code that generates and fills Chombo levels. At present, the code uses one level only, but functions defined here know how to build more. Discussed in Section 6, page 140.

`src/Injection.c` This file contains C wrappers for Scheme functions, *ex_lambda*, *ey_lambda*, and *ez_lambda*, to be defined by the user. The C wrappers are called by Fortran functions defined on `inject_b.f` and `inject_d.f`, which are, in turn, called from `Iteration.cpp`. The wrappers are discussed in Section 7.2.1, page 165.

`src/Injection.h` This file contains C language headers required by `src/Injection.c`. It is discussed in Section 7.2.1, page 167.

`src/Iteration.cpp` This file contains Chombo wrappers that invoke the computational guts of the program: Fortran routines that advance the $\boldsymbol{D}$ and $\boldsymbol{B}$ fields, routines that inject and extract the incident field into and from the total field region, and routines that convert $\boldsymbol{D}$ to $\boldsymbol{E}$ and $\boldsymbol{B}$ to $\boldsymbol{H}$. Its content is discussed in Section 7, page 158.

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

`src/SCMParmParse.cpp` This file contains the implementation of the **SCMParmParse** C++/Chombo class. The class is reminiscent of the Chombo **ParmParse** class, but it works against the Scheme space and provides additional Scheme related methods. It can be compiled in a stand-alone mode that does not depend on Chombo. It is discussed in Section 12.1, page 252.

`src/Scheme.cpp` This file contains C++ definitions of native Scheme functions, such as *draw_box*, *draw_ball*, *draw_ellipsoid*, and other that may yet be defined. These are really wrappers that may call other, more specific C++ functions, for example, to operate on Chombo objects, and these are also defined here. A function that initializes Scheme at the beginning, *initialize_scheme*( ) is defined on this file, as well. These are all discussed in Section 8, page 187.

`src/b_to_h.f` This Fortran file contains definitions of functions that convert $\boldsymbol{B}$ to $\boldsymbol{H}$. The conversion within the UPML region is included. Listed in Section 13.4, page 350. It is here that user provided Scheme code is invoked through plain C wrappers.

`src/d_to_e.f` This Fortran file contains definitions of functions that convert $\boldsymbol{D}$ to $\boldsymbol{E}$. The conversion within the UPML region is included. Listed in Section 13.3, page 339. It is here that user provided Scheme code is invoked through plain C wrappers.

`src/draw.f` Fortran subroutines in this file draw media on Chombo fields. They are listed in Section 13.5, page 355.

`src/inject_b.f` Fortran subroutines in this file carry out the $\boldsymbol{B}$ field injection/extraction on the total field boundary. They call user provided Scheme code through plain C wrappers defined on `Injection.c`. They are listed in Section 13.7, page 368.

`src/inject_d.f` Fortran subroutines in this file carry out the $\boldsymbol{D}$ field injection/extraction on the total field boundary. They call user provided Scheme code through plain C wrappers defined on `Injection.c`. They are listed in Section 13.6, page 363.

`src/output.f` These are auxiliary Fortran subroutines used in computing cell-centered fields for output on the HDF5 files. Additional computations that evaluate energy and Poynting vector are defined here, as well. They are listed in Section 13.8, page 373.

`src/update_b.f` These subroutines implement leap-frog time step for the $\boldsymbol{B}$ field with or without UPMLs. They are listed in Section 13.2, page 335.

`src/update_d.f` These subroutines implement leap-frog time step for the $\boldsymbol{D}$ field with or without UPMLs. They are listed in Section 13.1, page 331.

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

```
$ make Forms
```

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

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

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

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

```
$ CC=icc make Forms
```

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

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

... but not 1.8.0 with Chombo-2.0. The newly released HDF5-1.8.0 is not fully compatible with the earlier verions, which results in compilation errors.

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

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

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

Guile version must be 1.8.x or later. Earlier versions of Guile do not have some of the functions that FORMS uses.

The Chombo file `mk/Make.rules` must be modified to add rules for depending and building plain C programs, because the wrappers that provide a bridge between Fortran and Scheme are written in plain C. A version of `Make.rules` that works with Chombo-2.0 is provided in the `Patches` subdirectory of the FORMS directory tree. ◀

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

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

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

```
$ make run
```

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

2.1.1 Parallel Build and Runs

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

1. Enabling "Parallel HDF5" on HDF5 `configure`.
2. Pointing HDF5 to MPI C compiler, `CC=mpicc`, both on `configure` and `make`.

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

utility	version	notes
		Required for building the source
GNU RCS	5.7	
GNU `make`	3.81	
`ctangle`	3.5.4	
`cweave`	3.5.4	
`sed`	4.1.5	
`astyle`	1.21	
LaTeX	3.141592-1.21a-2.2	
`cweb.cls`	3.6	*LaTeX class, patch it for LaTeX2e*
`dvipdfm`	0.13.2c	
		Required for building the binary
`g++`	3.4.4	`icpc 8.1` *is fine*
`f77`	3.4.4	`ifort 8.1` *is fine*
HDF5	1.6.6	*but not 1.8.0 with Chombo 2.0*
Chombo	2.0-Aug2007	
Guile	1.8.2	*must be 1.8.x at least*
mk/Make.rules	patched	*provided with* Forms

3. HDF5 C++ interface has to be disabled for parallel compilation.

4. Defining `MPI = TRUE` on the Chombo's `mk/Make.defs.local` file.

5. Defining `CXX = mpiCC` on the Chombo's `mk/Make.defs.local` file (preferably full path resolution should be specified).

6. Defining `HDFMPIINCFLAGS` and `HDFMPILIBFLAGS` on the Chombo's `mk/Make.defs.local` file and pointing them towards the parallel HDF5 location, for example,

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

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

```
CHOMBO_HOME =
/home/meglicki/Chombo-2.0-Oct2007/lib3d.Linux.mpiCC.ifort.OPT.MPI
```

Normally, parallel HDF5 should be installed on an MPI system by administrators already, so it's a matter of finding where it is. As has been remarked above, HDF5-1.8.0 will not work with Chombo-2.0 at present, hence a 1.6.x version of HDF5 must be used instead.

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

```
Forms3d.Linux.mpiCC.ifort.OPT.MPI.ex
```

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

Running Parallel Forms on Jazz

The batch execution and configuration depends on the batch system in use. PBS is one of the most commonly used, and installed on the ANL Jazz cluster. It is also important that data is written by Forms on a parallel

file system and not on the NFS. The latter is extremely inefficient and may flood the normally limited NFS mounted volume with production data. It must *not* be done.

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

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

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

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

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

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

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

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

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

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

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

The command

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

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

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

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

and the HDF5 libraries live in

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

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

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

The script is submitted with the PBS command

```
$ qsub jazz_submit.sh
```

Its status can be viewed with

```
$ qstat -u meglicki
```

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

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

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

The above invocations are *not enough* to ensure that the job really runs in parallel. For this to happen the computational domain must be additionally partitioned on the FORMS input file. This is discussed in more detail in Section 2.5, page 48.

2.2 The Input File

FORMS expects to read a single file that contains Scheme code on entry. This is discussed in more detail in Section 5.1, page 134. Specifically, if no file name is provided on the command line, the program aborts.

If a file name has been provided, it is handled by the **SCMParmParse** class constructor, which is discussed in Section 12.1.2, page 255, module ⟨ **SCMParmParse** implementation: constructors and destructors 199 ⟩. The constructor inspects the file name. If it is OK, that is, not null, and the file pointed to by the name exists and is readable, it is loaded into the Scheme by a call to *scm_c_primitive_load*.

And this is all.

Whatever is on the file is evaluated at this stage.

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

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

2.2.1 The `grid` group

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

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

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

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

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

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

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

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

The grid constant in FORMS is the same in all three directions.

2.2.2 The `pml` group

The next group of parameters describes the UPML region at the computational domain's boundary. The UPML formalism used by the program is discussed in Section 3.4, page 99, and its implementation is discussed in Section 7.1.1, page 159, Section 7.4.1, page 177, Section 7.3.2, page 168, and Section 7.6.2, page 184. Additionally, Fortran listings of relevant UPML operations are provided in Section 13.1, page 331, Section 13.2, page 335, Section 13.3, page 339, and Section 13.4, page 350.

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

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

Here `lo` and `hi` stand for low and high corners of the *non*-UPML region, that is, of the interior of the computational domain. Everything outside this box is UPML. The `sigma_max` parameter is $\sigma_{\max}$ from equation (169), page 99. Finally, the `print_arrays` parameter tells FORMS to list arrays C_{0x} through C_{4z} as defined by equations (227–241), page 103, when it is set to 1 and not print them when set to 0.

2.2.3 The `signal` group

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

The mathematics of the procedure is discussed in Section 3.3, page 88. Its implementation is discussed in Section 7.2, page 163, and Section 7.5, page 181. The related Fortran functions are listed in Section 13.6, page 363, and Section 13.7, page 368.

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

It's complicated and in the ideal world we wouldn't do it like this. But this is not the ideal world. Chombo was designed for Fortran specifically, not for Scheme in the anticipation that users would write their modules pluggable into the Chombo shell in Fortran. But this would require the code to be recompiled and relinked prior to execution.

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

```
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
(define forms.signal.direction '( 1.0 1.0 1.0))
;;
(define wavelength 30.0)
(define alpha 0.2)
(define delay 20.0)
(define pi (* 2.0 (asin 1.0)))
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (* 0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
    (* -0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.normalized 1)
```

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

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

$$E_y(\zeta) = -E_x(\zeta), \quad (2)$$

$$E_z(\zeta) = 0, \quad (3)$$

where $\zeta = \boldsymbol{n}\cdot\boldsymbol{r} - t$, $\boldsymbol{n}$ is a normalized signal direction vector, and $\boldsymbol{r} = (x, y, z)$. See Section 3.3, page 88, for the discussion. In the above we have also specified that

$$\lambda = 30.0, \quad (4)$$

$$\alpha = 0.2, \quad (5)$$

$$\delta = 20.0, \quad (6)$$

$$\pi = 2\arcsin 1. \quad (7)$$

The last equation sets π to the machine accuracy.

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

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

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

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

The formula provided will be invoked at every point of the total/scattered field boundary, as summarized by equations (138–161), page 96, at every time step. The user may request that garbage collection be triggered after every time step by setting `garbage_collect` to 1. This is not strictly necessary, because Scheme invokes garbage collection when needed anyway.

Because variable lookups are expensive in Scheme, it is more efficient to replace calls to `wavelength`, `alpha`, and `pi`, in the above formula with the actual numbers. It is also possible to define the functions in C and precompile them, and then load the binaries into Scheme. We discuss this in Section 2.4, page 30. So what's presented here is at once the most inefficient, but also the most readable way of doing it.

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

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

```
# Gnuplot file for drawing the waveform.
# Draw the form on the display.
#
set samples 192
plot [z = -128:64] \
    f(z) = 0.5 * (1.0 - tanh(a * (z + d))) * sin(2 * pi * (z + d) / w), \
    w = 30, a = 0.2, d = 20, pi = 3.14, f(z)
#
# Now redraw on an Encapsulated Postscript file.
#
set term postscript eps
set output "signal.eps"
replot
```

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

```
\begin{center}
  \includegraphics*[width=0.7\textwidth]{signal.eps}
```

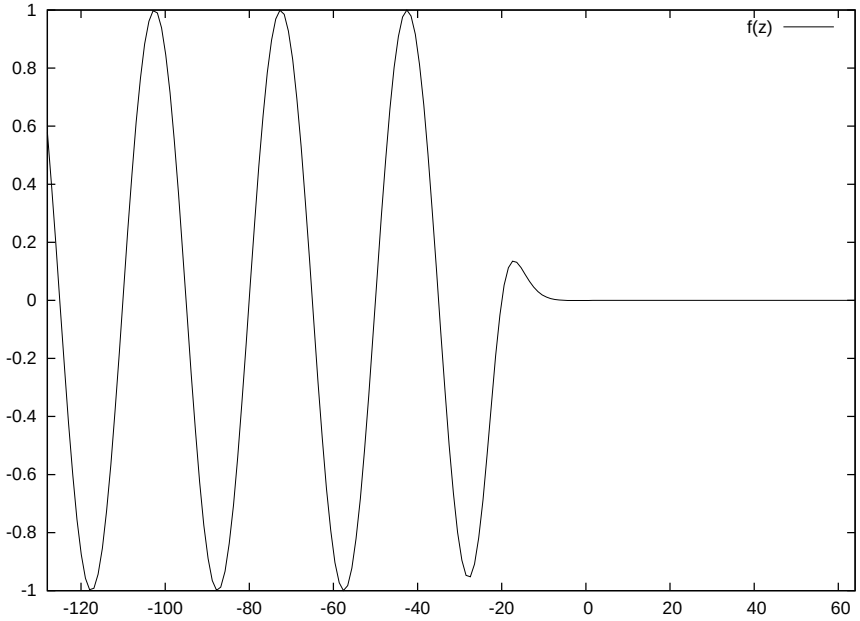


Figure 1: Gnuplot graph of the incident signal defined by equation (1).

```
  \caption{Gnuplot graph of the incident signal defined by
    equation~(\ref{e:signal}).}
  \label{f:waveform}
\end{center}
```

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

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

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

2.2.4 The `media` group

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

FORMS defines two reserved symbols, `forms.media.model.e_lambda` and `forms.media.distribution.lambda`, that must be specified further by the user. Additionally, FORMS provides auxiliary low level functions, `draw-box`, `draw-ball`, and `draw-ellipsoid`, that "draw" the actual media distribution on the Chombo data structures. Furthermore FORMS defines the following symbols that can be used in calculations here and that are going to be set to appropriate values during program execution:

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

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

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

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

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

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

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

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

The job of `forms.media.model.e_lambda` is to tell FORMS what E_i is for $i =$ `forms.direction` and other parameters listed above.

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

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

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

The above states that where the medium type is

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

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

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

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

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

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

The $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion is discussed in Section 7.3, page 168.

Basically various C++/Chombo wrappers here assess the context (level 0 versus a higher level, with or without auxiliary fields) and call an appropriate Fortran routine, all of which are listed in Section 13.3, page 339.

Because for level 0 there is also a special $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion within the UPML region, the region is assigned medium number -2 by default and the Fortran routine invokes the UPML conversion there automatically. The user cannot overwrite this action. Vacuum is assigned medium number 0, and within this medium the routines assign $\boldsymbol{E} = \boldsymbol{D}$. This action is automatic too and cannot be modified.

A user's script is invoked for media numbered from +1 onwards. This is done by calling a plain C wrapper from within Fortran, which is listed in Section 7.3.8, page 175.

The $\boldsymbol{B}$ field is similarly converted to the $\boldsymbol{H}$ field, with the exception that, at this stage, there is no provision for the users to submit their own conversion scripts (this can be easily altered). So, the only conversion that happens is the one within the UPML region and in vacuum, where $\boldsymbol{B} = \boldsymbol{H}$. The $\boldsymbol{B}$-to-$\boldsymbol{H}$ conversion is discussed in Section 7.6, page 184, and the corresponding Fortran routines are listed in Section 13.4, page 350.

Here is an example of the media distribution script:

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

The expression draws a box on the Chombo data structures with its low corner at $(60, 60, 60)$ and its high corner at $(68, 68, 68)$ and assigns medium #1 to all cells within the box. Then it draws a ball of radius 5.0, centered on $(40, 40, 40)$ and assigns medium #1 to all cells within it, too. Finally, it draws an ellipsoid with its foci at $(70, 70, 70)$ and $(80, 80, 80)$, the major axis of which is 19.0 units long, and assigns medium #2 to all cells within it.

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

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

```
(define dd 3)
(define dq 0)
(define sum1 12)
(define sum2 8)
(define delta 25)
(define i0 38)
(define j0 38)
(define k0 38)
(define imax 213)
(define jmax 213)
(define kmax 213)
;;
(use-modules (ice-9 format))
;;
(define forms.media.distribution.lambda
  (lambda ()
    (let ((count 0))
      (do ((i i0 (+ i delta)))
          ((> i imax))
        (do ((j j0 (+ j delta)))
            ((> j jmax))
          (do ((k k0 (+ k delta)))
```

```
        ((> k kmax))
      (let* ((f1x i)
             (f1y j)
             (f1z k)
             (f2x (+ i dd))
             (f2y (+ j dd))
             (f2z (+ k dd))
             (f3x (+ f1x dq))
             (f3y (+ f1y dq))
             (f3z (+ f1z dq))
             (f4x (- f2x dq))
             (f4y (- f2y dq))
             (f4z (- f2z dq)))
        (draw-ellipsoid (f64vector f1x f1y f1z) (f64vector f2x f2y f2z) sum1 1)
        (draw-ellipsoid (f64vector f3x f3y f3z) (f64vector f4x f4y f4z) sum2 2)
        (set! count (+ count 1))))))
  (format #t "~a grains positioned" count)
  (newline))))
```

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

Further down, in Section 2.4, page 30, we discuss coding Scheme function directly in C for improved performance. But there is no need to do this for the distribution lambda, because this code is executed once only. On the other hand, the `e_lambda` code, which converts $\boldsymbol{D}$ to $\boldsymbol{E}$ is executed at every non-vacuum point of the grid at every iteration. This code must be optimal for the best performance and it is here that it pays to express the function in C.

The drawing procedures that are provided for the use in the distribution lambda are rather complex and time consuming, because the drawing has to be done for every one of the six basic face and edge mounted fields separately. This is discussed in Section 6.2, page 152, where the user provided lambda is invoked for every level of the multigrid. The actual media drawing, Scheme callable C++ functions are discussed in Section 8, page 187, and the specific Fortran routines invoked listed in Section 13.5, page 355.

There are two more parameters in this group. Scheme garbage collection after every iteration can be forced by setting

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

Garbage collection is not invoked manually (Scheme does it anyway) when this parameter is left at 0. The number of auxiliary fields required for the conversion is specified by

```
(define forms.media.number_of_auxiliary_fields 3)
```

If set as above, 3 additional scalar fields will be provided bound into a uniform vector called `S`. These can be used to store, for example, values of $\boldsymbol{E}$ even older than $\boldsymbol{E}_{\text{old}}$, as some AEDs may require, or other values such as induced currents, and the like. Users must ensure within their scripts that these are correctly updated when used.

2.2.5 The `iterate` group

The next group describes iterations. Here is an example:

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

Table 2: Predefined cell-centered fields.

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

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

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

2.2.6 The `output` group

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

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

The output procedure, which is discussed in Section 9, page 211, works as follows. The user specifies fields to be precomputed, in such order as is required for the operation, by defining an ordered list of strings, for example

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

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

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

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

In the future, users will be able to define their own string symbols within the `precompute` list, and supplement them with lambda scripts that will tell FORMS how to evaluate the requested fields by using the previously listed (and evaluated) ones. Fortran listings of predefined field operations are provided in Section 13.8, page 373.

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

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

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

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

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

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

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

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

If specified as above, the file names will be

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

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

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

2.2.7 The `post` group

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

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

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

The main utility that users may invoke here is a Scheme function `interpolate`, which is discussed in Section 2.7.5, page 79, Section 3.5.4, page 121, and then in Section 8.6, page 203, module ⟨ Interpolation 136 ⟩. It provides interpolated access to all major Chombo fields. Additionally, elementary parallel programming utilities are provided as well, in case some quantities should be accumulated across the process pool or Scheme output written on process specific files. These are discussed in Section sss:scheme-parallel, page 119, and then in Section 8.5, page 201, module ⟨ MPI Functions 135 ⟩.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.8 The `watch` group

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

Here is an example:

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

```
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
(define forms.watch.fill_PML_arrays 1)
```

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

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

2.3 Postprocessing

HDF5 organized data dumped by FORMS can be viewed and analyzed with CHOMBOVIS. CHOMBOVIS can slice 3-dimensional data sets in all principal directions, it can produce isosurfaces and contour plots on the slices. It can even do volume rendering, if slowly.

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

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

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

The `<TAB>` symbols in the listing stand for tabs, which are essential in Python, in which indentation is not just a matter of decoration.

If the original HDF5 files are as listed in Section 2.2.6, page 20, then the PPM files will be

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

To combine the PPM images produced into a GIF anuimation, we need to convert them to GIF files first and then merge into a single animated GIF file. The following shell script shows how to do this:

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

The script invokes CHOMBOVIS on the Python script listed above, then converts each PPM image to a GIF image and finally assembles GIF file sinto a single moving GIF file.

The first line of the script sources CHOMBOVIS environment. The program normally takes a great deal of labor and tinkering to get going and may require special environment settings to run.

2.4 Defining Scheme Functions in C

FORMS extends Scheme by defining its own Scheme callable C++ functions that access Chombo fields. These are discussed in Section 8, page 187.

Once defined, they are loaded by function *initialize_scheme*( ), which is discussed in Section 8.7, page 207, whereupon they are available to users.

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

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

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

The *draw_box* function itself has the following interface,

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

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

What's inside *draw_box* is discussed in Section 8.1, page 188. It is too complicated for most users, because the function has to *draw* a box on Chombo fields, either media or tags, and this is a convoluted process.

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

This section explains how to do this and discusses examples.

The following is an example[2] listing of a simple wrapper that invokes the `libm.a` defined Bessel function J_0.

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

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

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

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

```
linux: libguile−bessel.so

cygwin: libguile−bessel.dll

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

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

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

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

Typing

```
$ make cygwin
```

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

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

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

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

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

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

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

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

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

2.4.1 Two Dielectrics

Our first example is a C implementation of the double dielectric model described by the Scheme code presented on page 17. The code is only a little more complicated, but it should run much faster.

```
#include <libguile.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

#define EPS_1 4.0
#define EPS_2 2.0

SCM dielectrics()
{

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

```
  int medium;
  double E, D;

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

      first = FALSE;
    }

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

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

  return scm_from_double(E);

}

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

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

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

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

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

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

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

```
guile> (load−extension "libguile−dielectric.dll" "init_e_lambda")
guile> (define forms.medium 1)
guile> (define forms.D 0.5)
guile> (forms.media.model.e_lambda)
```

```
0.125
guile> (* 0.5 (/ 1.0 4.0))
0.125
guile> (set! forms.medium 2)
guile> (forms.media.model.e_lambda)
0.25
guile> (* 0.5 (/ 1.0 2.0))
0.25
guile>
```

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

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

Function *does_scm_symbol_exist*, which is discussed in Section 8.4, page 200, can be used to check if `epsilon_1` and `epsilon_2` have been defined. If not some evasive action can be taken, for example, substituting `EPS_1` and `EPS_2` as defaults.

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

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

2.4.2 A Drude Metal

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

$$\epsilon(\omega) = \epsilon_\infty + \frac{\sigma}{\mathrm{i}\omega\epsilon_0} + \frac{\chi}{1+\mathrm{i}\omega\tau}, \tag{9}$$

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

Substituting

$$\chi = -\omega_0^2\tau^2 \quad \text{and} \tag{10}$$
$$\sigma = \omega_0^2\tau\epsilon_0 \tag{11}$$

yields the Drude expression,

$$\begin{aligned}\epsilon(\omega) &= \epsilon_\infty + \frac{\omega_0^2\tau}{\mathrm{i}\omega} - \frac{\omega_0^2\tau^2}{1+\mathrm{i}\omega\tau} = \epsilon_\infty + \frac{\omega_0^2}{-\omega^2+\mathrm{i}\omega/\tau} \\ &= \epsilon_\infty + \frac{\omega_0^2}{\omega(\mathrm{i}/\tau-\omega)}.\end{aligned} \tag{12}$$

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

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

$$\boldsymbol{D}(\omega) = \epsilon(\omega)\boldsymbol{E}(\omega) = \left(\epsilon_\infty + \frac{\omega_0^2}{\omega(\mathrm{i}/\tau-\omega)}\right)\boldsymbol{E}(\omega) = \epsilon_\infty\boldsymbol{E}(\omega) + \boldsymbol{S}(\omega), \tag{13}$$

where

$$\boldsymbol{S}(\omega) = \frac{\omega_0^2}{\omega(\mathrm{i}/\tau-\omega)}\boldsymbol{E}(\omega), \tag{14}$$

which implies that

$$\mathrm{i}\omega\nu\boldsymbol{S} - \omega^2\boldsymbol{S} = \omega_0^2\boldsymbol{E}. \tag{15}$$

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

$$\frac{\partial}{\partial t}\boldsymbol{S} = \mathrm{i}\omega\boldsymbol{S} \quad \text{and} \tag{16}$$

$$\frac{\partial^2}{\partial t^2}\boldsymbol{S} = -\omega^2\boldsymbol{S}, \tag{17}$$

which lets us rewrite equation (15) as

$$\nu\frac{\partial}{\partial t}\boldsymbol{S} + \frac{\partial^2}{\partial t^2}\boldsymbol{S} = \omega_0^2\boldsymbol{E}. \tag{18}$$

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

$$\frac{\partial^2}{\partial t^2}\boldsymbol{S} \approx \frac{\boldsymbol{S}^{(n+1)} - 2\boldsymbol{S}^{(n)} + \boldsymbol{S}^{(n-1)}}{\Delta t^2} \quad \text{and} \tag{19}$$

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

Substituting these in equation (18) yields

$$\nu\frac{\boldsymbol{S}^{(n+1)} - \boldsymbol{S}^{(n-1)}}{2\Delta t} + \frac{\boldsymbol{S}^{(n+1)} - 2\boldsymbol{S}^{(n)} + \boldsymbol{S}^{(n-1)}}{\Delta t^2} = \omega_0^2\boldsymbol{E}^{(n)}. \tag{21}$$

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

$$\boldsymbol{S}^{(n+1)}\left(2 + \nu\Delta t\right) - 4\boldsymbol{S}^{(n)} + \boldsymbol{S}^{(n-1)}\left(2 - \nu\Delta t\right) = 2\omega_0^2\Delta t^2\boldsymbol{E}^{(n)}. \tag{22}$$

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

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

Returning to equation (13) and discretizing it in the time domain we find that

$$\boldsymbol{D}^{(n)} = \epsilon_\infty\boldsymbol{E}^{(n)} + \boldsymbol{S}^{(n)}, \tag{24}$$

hence

$$\boldsymbol{E}^{(n)} = \frac{1}{\epsilon_\infty}\left(\boldsymbol{D}^{(n)} - \boldsymbol{S}^{(n)}\right), \tag{25}$$

which together with equation (23) provides us with detailed prescription for how to find $\boldsymbol{E}^{(n)}$ given

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

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

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

The $\boldsymbol{S}$ field's implementation within the program is defined, alongside with $\boldsymbol{E}$ and $\boldsymbol{D}$, in Section 11.6, page 249, module ⟨ Structure level 191 ⟩. We can see there that S is a **LevelData⟨CurlBox⟩**. Each single S slot, therefore, defines one full $\boldsymbol{S} = (S_x, S_y, S_z)$. For $\boldsymbol{S}$ and $\boldsymbol{S}_{\text{old}}$ we need to reserve two slots.

Space reservation for $\boldsymbol{S}$ is discussed in Section 6.2, page 152, module ⟨ Function *fill_levels* 74 ⟩. It is a long module, and the relevant entries to how $\boldsymbol{S}$ space is allocated are:

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

and then

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

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

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

on the input file in the first place.

To see what happens next we need to turn to Section 7.3.2, page 168, module ⟨ Function *d_to_e_0* 96 ⟩. There we find that the top level function that does the $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion at level 0, checks if auxiliary fields have been requested and if so, it invokes the *d_to_e_0_n* function to do the job. This function is discussed in Section 7.3.4, page 170, module ⟨ Function *d_to_e_0_n* 98 ⟩. All it does is to prepare things for calling the Fortran subroutine *d_to_e_0_n_* on each box of its current level. This, however, includes defining which are going to be $(\boldsymbol{E}, \boldsymbol{D})$ and which are going to be $(\boldsymbol{E}_{\rm old}, \boldsymbol{D}_{\rm old})$ in this current conversion call, since these swap around on each consecutive iteration.

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

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

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

            if (return_status != EXIT_SUCCESS)
            {
```

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

The Fortran subroutine *d_to_e_0_n* is listed in Section 13.3, page 339. The important part of the listing is

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

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

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

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

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

Now let us have a look at *scm_d_to_e_n_*. This is a plain C-wrapper that is discussed in Section 7.3.8, page 175, module ⟨ `Conversion.c_std` 102 ⟩. This module creates a separate file. The function interface looks as follows:

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

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

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

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

are program globals, which are defined in Section 8.7, page 207, module ⟨ Initialize Scheme 137 ⟩. In case of the $\boldsymbol{S}$ field we need the variable reference, because it is an array. This is generated inside *initialize_scheme*( ) by

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

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

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

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

Because $\boldsymbol{S}$ is an array, we have to fuss more with it. Here the transfer is

```
S_ptr = S;

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

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

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

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

```
S_ptr = S;

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

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

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

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

We begin by rewriting equations (23) and (25) in terms of $\boldsymbol{S}$, $\boldsymbol{S}_{\rm old}$, $\boldsymbol{E}_{\rm old}$ and $\boldsymbol{D}$, and adding $\boldsymbol{S}$ update steps at the same time:

$$\boldsymbol{S}_{\rm tmp} \leftarrow \frac{4}{2+\nu\Delta t}\boldsymbol{S} - \frac{2-\nu\Delta t}{2+\nu\Delta t}\boldsymbol{S}_{\rm old} + \frac{2\omega_0^2\Delta t^2}{2+\nu\Delta t}\boldsymbol{E}_{\rm old} \tag{26}$$

$$\boldsymbol{E} \leftarrow \frac{1}{\epsilon_\infty}\left(\boldsymbol{D} - \boldsymbol{S}_{\rm tmp}\right) \tag{27}$$

$$\boldsymbol{S}_{\rm old} \leftarrow \boldsymbol{S} \tag{28}$$

$$\boldsymbol{S} \leftarrow \boldsymbol{S}_{\rm tmp} \tag{29}$$

And this leads to the following C-function:

```
#include <libguile.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

SCM drude()
{
  /*

  Provide definitions for two media:

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

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

  */

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

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

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

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

      dt = scm_to_double(scm_variable_ref(dt_var));

      /* User provided variables for this module */
```

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

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

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

      first = FALSE;
    }

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

  switch (medium)
    {
    case 1:

      E = one_by_eps_1 * D;
      break;

    case 2:

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

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

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

      break;
    }

  return scm_from_double(E);

}

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

We compile this function, as before, with

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

and then load it into Scheme with

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

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

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

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

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

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

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

```
guile> (set! forms.E_old (test.e_lambda))
guile> forms.S
#f64(1.02779864763336 0.760330578512397)
guile> forms.E_old
-0.0138993238166791
```

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

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

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

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

But before we go there, we can make one more improvement. Signal injection is carried out by three lambdas. These we can replace by C-functions, as well. The following listing shows an example file. The injected signal is the same as described in Section 2.2.3, page 14.

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

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

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

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

  if (first)
    {
```

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

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

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

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

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

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

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

The file is compiled as above, that is

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

and then loaded into Scheme with

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

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

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

```
;; Forms input file
;;
```

```
;; $Id: TestBallC.scm,v 1.3 2008/04/03 18:53:29 gustav Exp $
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Grid group
;;
;; Level 0 parameters
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Signal group
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 1.0 1.0 1.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define wavelength 30.0)
(define alpha 0.2)
(define delay 20.0)
;;
(load-extension "Examples/libguile-inject" "init_signal_lambdas")
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Media group
;;
(define forms.media.garbage_collect 0)
;;
;; The media distribution lambda
;;
(define forms.media.distribution.lambda
  (lambda ()
    (draw-ball (f64vector 64 64 64) 10 1)
    (draw-ball (f64vector 64 64 64) 6 2)))
;;
```

```
;; The media model lambda
;;
(define epsilon_1 2.7)
(define epsilon_infty 2.36461)
(define nu 0.004286)
(define omega_0 1.106)
;;
(define forms.media.number_of_auxiliary_fields 2)
;;
(load-extension "Examples/libguile-drude" "init_e_lambda")
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
(define forms.iterate.number_of_steps 1028)
(define forms.iterate.stride 4)
(define forms.iterate.image_frequency 16)
(define forms.iterate.t0 0.0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Output group
;;
(define forms.output.precompute '("Ex" "Ey" "Ez" "Bx" "By" "Bz" "En"))
(define forms.output.write '("En"))
(define forms.output.filename.root "En")
(define forms.output.filename.number_of_digits 3)
(define forms.output.report_outliers 1)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Watch group. This is just for debugging.
;;
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
(define forms.watch.fill_PML_arrays 1)
;;
;; end
;;
```

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

after they have been evaluted.

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

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

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

```
    (* -0.5 (- 1.0 (tanh (* alpha (+ zeta delay))))
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Media group
;;
(define forms.media.garbage_collect 0)
;;
;; The media distribution lambda
;;
(define forms.media.distribution.lambda
  (lambda ()
    (draw-ball (f64vector 64 64 64) 10 1)
    (draw-ball (f64vector 64 64 64) 6 2)))
;;
;; The media model lambda
;;
(define epsilon_1 2.7)
(define epsilon_infty 2.36461)
(define nu 0.004286)
(define omega_0 1.106)
;;
(define forms.media.number_of_auxiliary_fields 2)
;;
(define forms.media.model.e_lambda
  (lambda ()
    (case forms.medium
      ((1) (let ((one_by_epsilon_1 (/ 1.0 epsilon_1)))
             (* one_by_epsilon_1 forms.D)))
      ((2) (let* ((one_by_epsilon_infty (/ 1.0 epsilon_infty))
                  (factor (* nu forms.dt))
                  (denom (+ 2.0 factor))
                  (C1 (/ 4.0 denom))
                  (C2 (/ (- 2.0 factor) denom))
                  (C3 (/ (* 2.0 omega_0 omega_0 forms.dt forms.dt) denom))
                  (S (f64vector-ref forms.S 0))
                  (S_old (f64vector-ref forms.S 1))
                  (S_tmp (+ (* C1 S) (* -1.0 C2 S_old) (* C3 forms.E_old)))
                  (E (* one_by_epsilon_infty (- forms.D S_tmp))))
             (f64vector-set! forms.S 1 S)
             (f64vector-set! forms.S 0 S_tmp)
             E)))))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
(define forms.iterate.number_of_steps 1028)
(define forms.iterate.stride 4)
(define forms.iterate.image_frequency 16)
(define forms.iterate.t0 0.0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

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

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

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

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

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

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

2.5 Input for Parallel Execution

This section discusses computational domain partitioning, mentioned in Section 2.1.1, page 10.

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

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

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

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

All this is accomplished by the following five Chombo statements that are discussed in more detail in Section 6.1.3, page 145, module ⟨ *build_levels*: build level 0 70 ⟩:

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

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

For a single level run data between boxes is exchanged in two places only. We exchange the $\boldsymbol{H}$ field before the $\boldsymbol{D}$ update (see Section 7.1.1, page 159):

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

and then we exchange the $\boldsymbol{E}$ field before the $\boldsymbol{B}$ update (see Section 7.4.1, page 177):

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

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

What would happen if we were to define

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

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

2.6 Tests

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

2.6.1 Plane Wave Propagation

File: `Examples/PlaneWaveB.scm`

It is worth pointing out that in a sense the code self-tests. This is because the injected signal is defined analytically. Hence, the discrepancy between the injected signal at various points along the scattered/total field boundary, and the numerically propagated signal *within* the total field region, provides us with the measure of the code's accuracy. We have already discussed this in Section 2.2.7, page 21, and illustrated therein.

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

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

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

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

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

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

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

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

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

2.6.2 Fluxes

File: `Examples/WaveFlux.scm`

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

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

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

```
  (lambda (center radius)
    (let* ((local-flux (compute-local-flux center radius))
           (status (gather fluxes local-flux root-rank))
           (flux (if (= my-rank root-rank)
                     (do ((count 0 (1+ count))
                          (sum 0.0))
                         ((>= count pool-size) sum)
                       (set! sum (+ sum (vector-ref fluxes count))))
                     0)))
      (broadcast flux root-rank))))
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format my-log "post.iteration: t = ~a, flux = ~a~%"
            forms.time_e (compute-flux center radius))
    (force-output my-log)))
;;
(define forms.post.all.lambda
  (lambda ()
    (format my-log "post.all: finished~%")
    (close-port my-log)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

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

The `forms.post.iteration.lambda` computes the flux of $\boldsymbol{E}$ through a sphere of radius 25 centered on $(60, 60, 60)$. The detailed mathematics for this computation is discussed in Section 3.5.2, page 108, and a listing similar to the one above is provided there, too. The main difference here is that we calculate the flux of $\boldsymbol{E}$, not the flux of $\boldsymbol{n}$, and we do so over the whole sphere, not a hemisphere. We make a modest attempt to optimize the computation by evaluating certain quantities, such as $\phi_{\min}$, $\phi_{\max}$, $\Delta\phi$, $\theta_{\min}$, $\theta_{\max}$, and $\Delta\theta$, up front, and storing the results on globals.

Function `forms.post.iteration.lambda` is merely a wrapper, which writes the result of the computation on the log file, invoking the computation at the same time. The computation itself is deferred to function `compute-flux`, which is itself a wrapper, taking care of collecting (by calling `gather`) locally evaluated fluxes from the participating MPI processes, and adding them up. The result is then broadcast back to participating processes, but this is not strictly necessary. The `root-rank` process can be made the only one writing the answer on the log file. The FORMS function `interpolate` is designed to return zero, if a particular data item is not found within any of the grid boxes a given process looks after. This is specially done to facilitate and speed up the evaluation of fluxes—a script that calls `interpolate` does not have to check function returns for misses.

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

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

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

```
post.iteration: t = 127.625, flux = 0.00129535512972012
post.iteration: t = 127.75, flux = 0.00168322203256821
post.iteration: t = 127.875, flux = 0.00161249794277296
post.iteration: t = 128.0, flux = 0.00107854314621392
post.all: finished
```

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

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

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

2.6.3 Virtual Measurements

File: `Examples/PowerFlux.scm`

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

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

```
;;
;; Define the detectors
;;
(define center_1 #(40.0 64.0 40.0))
(define center_2 #(88.0 64.0 40.0))
(define normal_1 #(-1 0 1))
(define normal_2 #(1 0 1))
(define radius 10.0)
(define dx (/ (* 2.0 radius) 40.0))
(define dy dx)
;;
;; Normalize the detectors and define their
;; target surfaces
;;
(define n1 (vector-normalize normal_1))
(define n2 (vector-normalize normal_2))
(define ex1 (first-perpendicular n1))
(define ey1 (cross-product ex1 n1))
(define ex2 (first-perpendicular n2))
(define ey2 (cross-product ex2 n2))
;;
```

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

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

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

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

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

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

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

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

radius The radius R of the disk.

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

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

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

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

$$\boldsymbol{r} = \boldsymbol{r}_0 + x'\boldsymbol{e}_{x'} + y'\boldsymbol{e}_{y'}, \tag{30}$$

for x' and y' running between $\pm(R - \Delta x'/2)$ and $\pm(R - \Delta y'/2)$ respectively. The computation is triggered by (x', y') being within the radius of the disk. For such locations we find $\boldsymbol{E}$ and $\boldsymbol{H}$—function `interpolate` not only interpolates $\boldsymbol{H}$ in space, but in time, too, so both $\boldsymbol{E}$ and $\boldsymbol{H}$ are returned for the same time slice and space point. Next, the function evalutes $\boldsymbol{P} = \boldsymbol{E} \times \boldsymbol{H}$, and then takes its dot product with the normal, $\boldsymbol{P} \cdot \boldsymbol{n}$. Finally, it adds the result to the already accumulated flux with the $\mathrm{d}x\,\mathrm{d}y$ weight, which encodes the surface integral, as given by equation (287) on page 108.

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

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

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

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

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

```
Guile log:
   process number: 0
   pool size: 1
Computing flux of E x H through two photomultiplier tubes
   tube 1: radius = 10.0, center = #(40.0 64.0 40.0), normal = #(-0.707106781186547 0.0 0.707106781186547)
           ex = #(0.0 1.0 0), ey = #(0.707106781186547 0.0 0.707106781186547)
   tube 2: radius = 10.0, center = #(88.0 64.0 40.0), normal = #(0.707106781186547 0.0 0.707106781186547)
           ex = #(0.0 -1.0 0), ey = #(-0.707106781186547 0.0 0.707106781186547)
t = 0.25
```

```
    current flux on detector 1 = 0.0, total absorbed = 0.0
    current flux on detector 2 = 0.0, total absorbed = 0.0
t = 0.5
    current flux on detector 1 = 0.0, total absorbed = 0.0
    current flux on detector 2 = 0.0, total absorbed = 0.0
...
t = 127.5
    current flux on detector 1 = 108.585969016262, total absorbed = 11319.7390109556
    current flux on detector 2 = 108.593816242442, total absorbed = 11320.7787778787
t = 127.75
    current flux on detector 1 = 105.898295724077, total absorbed = 11346.2135848866
    current flux on detector 2 = 105.908488324884, total absorbed = 11347.2558999599
t = 128.0
    current flux on detector 1 = 103.265772252444, total absorbed = 11372.0300279497
    current flux on detector 2 = 103.277403693938, total absorbed = 11373.0752508834
post.all: finished
```

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

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

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

2.6.4 Fourier Analysis

File: `Examples/FourierFlux.scm`

For the same set up as discussed in the previous section, that is, two angled cylindrical detectors, we can ask about power received through filters at specific frequencies, where filters act not on power itself, but on $\boldsymbol{E}$ and $\boldsymbol{H}$ separately, filtering $\boldsymbol{E}(\omega)$ and $\boldsymbol{H}(\omega)$ only. This case is discussed in Section 3.5.1, page 106, where we demonstrate that the average energy flux over the period of vibration is given by (cf. equation (274))

$$\frac{1}{2}\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right). \tag{31}$$

To evaluate this quantity, we have to calculate Fourier transform of the incident signal for $\boldsymbol{E}(t)$ and $\boldsymbol{H}(t)$ at every point of the detector's surface and for every frequency of interest for $t \in [t_{\text{begin}}, t_{\text{end}}]$, and only after this computation completes, can we evaluate (31).

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

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

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

$$\hat{\boldsymbol{E}}(\boldsymbol{r},\omega) \quad \leftarrow \quad \hat{\boldsymbol{E}}(\boldsymbol{r},\omega) + \boldsymbol{E}(\boldsymbol{r},t)e^{-\mathrm{i}\omega t}\,\mathrm{d}t \quad \text{and} \tag{32}$$

$$\hat{\boldsymbol{H}}(\boldsymbol{r},\omega) \quad \leftarrow \quad \hat{\boldsymbol{H}}(\boldsymbol{r},\omega) + \boldsymbol{H}(\boldsymbol{r},t)e^{-\mathrm{i}\omega t}\,\mathrm{d}t \tag{33}$$

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

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

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

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

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

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

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

The flux of (31) through the detector's surface is computed only after the iterations have completed, by function `forms.post.all.lambda`, which in this case is somewhat more elaborate. Before we proceed to discuss the function, we first have a look at how the local flux of (31) is computed on the node. This is done by function `compute-local-flux`, which is listed below.

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

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

1. evaluate (31),

2. take the dot product of the resulting vector with the normal for the given detector,

3. multiply the resulting number by $\mathrm{d}^2 S$,

4. and add it to whatever has been already accumulated in the k-th slot of the corresponding `flux-vector`, which has been prepared and initialized to zero beforehand.

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

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

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

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

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

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

```
;;
(define pi (* 2.0 (asin 1.0)))
```

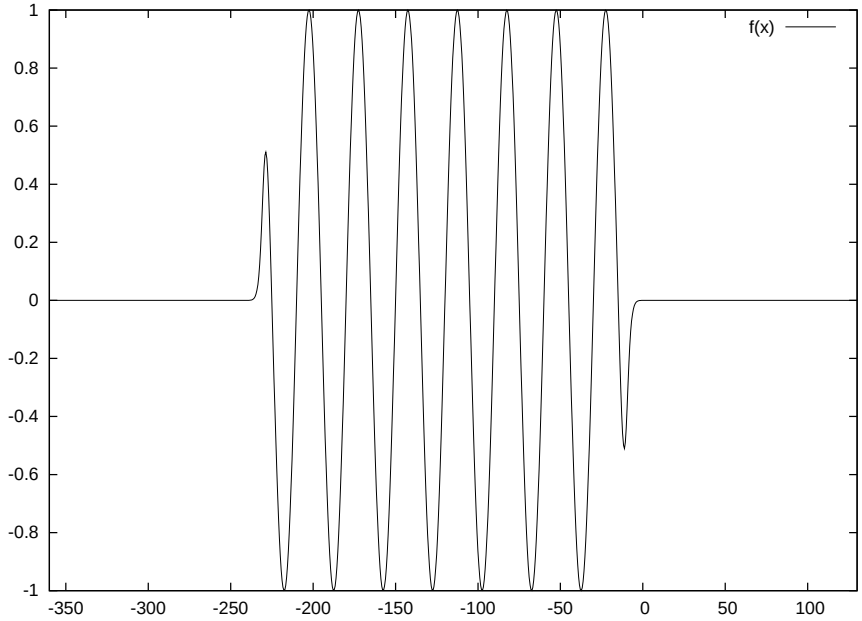


Figure 2: Incident signal used in the Fourier transform test.

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

This signal is shown in Figure 2. The following shows a sample of output for the `root-rank` process:

```
Guile log:
    process number: 0
    pool size: 8
```

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

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

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

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

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

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

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

2.7 The Library

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

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

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

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

or

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

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

2.7.1 Constants

Synopsis:

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

This file defines mathematical, physics and fitted constants of use in nano-photonics. The constants are as shown in Tables 5 and 6. The way the constants appear in specific formulas is as follows:

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

$$\epsilon(\omega) = 1 - \frac{\omega_p^2}{\omega(\omega + \mathrm{i}\gamma)}. \tag{34}$$

- For the Drude model with ϵ_∞ (Gray and Kupka [9]):

$$\epsilon(\omega) = \epsilon_\infty - \frac{\omega_p^2}{\omega(\omega + \mathrm{i}\gamma)}. \tag{35}$$

- For the two pole Lorentz/Drude model (Lee and Gray [12]):

$$\epsilon(\omega) = \epsilon_\infty - \frac{\omega_D^2}{\omega(\omega + \mathrm{i}\gamma_D)} - \frac{\omega_1^2 g_1 \Delta\epsilon}{\omega^2 - \omega_1^2 + 2\mathrm{i}\gamma_1\omega} - \frac{\omega_2^2 g_2 \Delta\epsilon}{\omega^2 - \omega_2^2 + 2\mathrm{i}\gamma_2\omega}, \tag{36}$$

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

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

$$\omega_p = 11.585 \times 1.60210 \times 10^{-19}\,\mathrm{J} \times 2 \times \pi/(6.6256 \times 10^{-34}\,\mathrm{Js}) \approx 1.76 \times 10^{16}\,\mathrm{rad/s}. \tag{37}$$

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

Table 5: Constants defined on `constants.scm`

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

Table 6: Constants defined on `constants.scm`, continued

symbol	value	comment
A two-pole Lorentz/Drude model with ϵ_∞, from Lee and Gray [12], based on Johnson and Christy [11] *All parameters apply to silver for* $\lambda \in [250\,\mathrm{nm}, 1000\,\mathrm{nm}]$		
epsilon_LG_Ag_D_250_1000	$\epsilon_\infty = 2.3646$	
epsilon_LG_Ag_1_250_1000	$\epsilon_1 = g_1 \Delta\epsilon = 0.2663 \times 1.1831$	
epsilon_LG_Ag_2_250_1000	$\epsilon_2 = g_2 \Delta\epsilon = 0.7337 \times 1.1831$	$g_1 + g_2 = 1$
omega_LG_Ag_D_250_1000	$\omega_D = 8.73770\,\mathrm{eV}/\hbar$	
omega_LG_Ag_1_250_1000	$\omega_1 = 4.38020\,\mathrm{eV}/\hbar$	
omega_LG_Ag_2_250_1000	$\omega_2 = 5.18300\,\mathrm{eV}/\hbar$	
gamma_LG_Ag_D_250_1000	$\gamma_D = 0.07489\,\mathrm{eV}/\hbar$	
gamma_LG_Ag_1_250_1000	$\gamma_1 = 0.28000\,\mathrm{eV}/\hbar$	
gamma_LG_Ag_2_250_1000	$\gamma_2 = 0.54820\,\mathrm{eV}/\hbar$	
A two-pole Lorentz/Drude model with ϵ_∞, from McMahon, et al. [16], also based on [11] *All parameters apply to gold for* $\lambda \in [250\,\mathrm{nm}, 1000\,\mathrm{nm}]$		
epsilon_JMM_Au_D_250_1000	$\epsilon_\infty = 5.39833498$	
epsilon_JMM_Au_1_250_1000	$\epsilon_1 = g_1 \Delta\epsilon = 0.267871678 \times 2.541747093$	
epsilon_JMM_Au_2_250_1000	$\epsilon_2 = g_2 \Delta\epsilon = 0.73212832 \times 2.541747093$	$g_1 + g_2 = 1$
omega_JMM_Au_D_250_1000	$\omega_D = 9.2006880\,\mathrm{eV}/\hbar$	
omega_JMM_Au_1_250_1000	$\omega_1 = 2.8131406\,\mathrm{eV}/\hbar$	
omega_JMM_Au_2_250_1000	$\omega_2 = 3.4394149\,\mathrm{eV}/\hbar$	
gamma_JMM_Au_D_250_1000	$\gamma_D = 0.068017714\,\mathrm{eV}/\hbar$	
gamma_JMM_Au_1_250_1000	$\gamma_1 = 0.286542200\,\mathrm{eV}/\hbar$	
gamma_JMM_Au_1_250_1000	$\gamma_2 = 0.434928310\,\mathrm{eV}/\hbar$	

2.7.2 Vectors

Synopsis:

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

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

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

The library overloads the basic arithmetic functions `+`, `-`, `*`, and `/`, to enable addition and subtraction of two vectors, as well as their multiplication and division by a number. The number in question *may* be complex. The two methods `real-part` and `imag-part` have been extended to return real or imaginary part of a complex valued three-dimensional vector. The `=` operator has been overloaded to return true if its two vector arguments are equal. Here is an example of how these operations work:

```
guile> (define v (make <3D−vector> #:x 1+2i #:y 1−2i #:z 2+1i))
guile> v
<1.0+2.0i, 1.0−2.0i, 2.0+1.0i>
guile> (real−part v)
<1.0, 1.0, 2.0>
guile> (imag−part v)
<2.0, −2.0, 1.0>
guile> (∗ 2 v)
<2.0+4.0i, 2.0−4.0i, 4.0+2.0i>
guile> (/ v 2)
<0.5+1.0i, 0.5−1.0i, 1.0+0.5i>
guile>
```

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

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

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

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

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

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

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

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

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

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

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

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

$$(*\,\boldsymbol{u}\,\boldsymbol{v}) = \boldsymbol{u}\cdot\boldsymbol{v} = \sum_{i\in\{x,y,z\}} u_i v_i^*. \tag{38}$$

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

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

```
4.0−24.0i
guile> (+ (∗ (x−component u) (complex−conjugate (x−component v)))
... (∗ (y−component u) (complex−conjugate (y−component v)))
... (∗ (z−component u) (complex−conjugate (z−component v))))
4.0+24.0i
guile>
```

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

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

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

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

The cross-product of two vectors is defined by

$$(\times\,\boldsymbol{u}\,\boldsymbol{v}) = \boldsymbol{u}\times\boldsymbol{v} = \sum_i \left(\sum_{j,k}\epsilon_{ijk}u_j v_k^*\right)\boldsymbol{e}_i \tag{39}$$

As above, we apply complex conjugate to the second vector in the product. This is so that we can use the formula to render expressions such as $\hat{\boldsymbol{E}}(f)\times\hat{\boldsymbol{H}}^*(f)$ that appear in the Parseval's theorem and other similar expressions, cf. equations (266), (268), and (274) in Section 3.5.1, page 106.

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

```
guile> (define ex (make <3D−vector> #:x 1 #:y 0 #:z 0))
guile> (define ey (make <3D−vector> #:x 0 #:y 1 #:z 0))
guile> (x ex ey)
<0, 0.0, 1.0>
guile> (x ey ex)
<0.0, 0, −1.0>
guile>
```

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

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

This somewhat contrived operation works both for real and for complex-valued vectors. Here is a demonstration:

```
guile> a
<1, 0, 0>
guile> b
<0, 1, 0>
guile> c
```

```
<0, 0, 1>
guile> (perpendicular−to a)
<0.0, 1.0, 0.0>
guile> (perpendicular−to b)
<−1.0, 0.0, 0.0>
guile> (perpendicular−to c)
<1.0, 0.0, 0.0>
guile> (load−from−path "constants")
guile> (define w (make <3D−vector>
... #:x (exp (* 0+i (/ pi 4)))
... #:y (exp (* 0+i (* 3 (/ pi 4))))
... #:z (exp (* 0+i (* 5 (/ pi 4))))))
guile> w
<0.707106781186548+0.707106781186547i,
−0.707106781186547+0.707106781186548i,
−0.707106781186548−0.707106781186547i>
guile> (perpendicular−to w)
<0.5+0.5i, 0.5−0.5i, 0.0>
guile> (* w (perpendicular−to w))
0.0
guile> (norm (perpendicular−to w))
1.0
guile>
```

2.7.3 Signals

Synopsis:

```
guile> (load−from−path "chirps")
guile> (load−from−path "windows")
```

There are two files that provide functions for defining signals. The first one, `"chirps"`, defines frequency modulated signals of constant amplitude. These can then be multiplied by envelopes taken from `"windows"`, to produce finite pulses of various forms.

Both files carry extensive comments that explain the signals defined, providing their analytical form in TEX, example synopsis in Scheme, and Gnuplot commands to graph them. They also provide functions for numerical evaluation of the signals. The functions write their output on `"test.out"`. It can be then fed into Gnuplot for display. This is different from Gnuplot displaying a function according to an analytical formula provided.

Additionally, all functions are source-documented. Invoking `procedure-documentation` on a function prints its synopsis.

The signals and the windows are defined to unfold from right to left, because this is how they are injected into the total field region of FORMS.

Chirps

test-chirp

synopsis (test-chirp chirp width ω_0 α)

documentation

```
guile> (procedure−documentation test−chirp)
"Synopsis: (test−chirp chirp width omega_0 alpha)
Test the chirp function for x in [−6/5 * width, width/5]."
```

linear-chirp

formula $f(x) = -\sin\left(\left(\omega_0 - \alpha x\right) x\right)$

synopsis (linear-chirp x ω_0 α)

documentation

```
guile> (procedure−documentation linear−chirp)
"Synopsis: (linear−chirp x omega_0 alpha)
Return linear chirp of the form −sin((omega_0 − alpha ∗ x) ∗ x)."
```

test

```
guile> (define omega_0 (/ (∗ 2 pi) 1800.0))
guile> (test−chirp linear−chirp 60 omega_0 0.02)
```

exponential-chirp-1

formula $f(x) = \sin\left(\omega_0 \left(e^{-\alpha x} - 1\right)\right)$

synopsis (exponential-chirp-1 x ω_0 α)

documentation

```
guile> (procedure−documentation exponential−chirp−1)
"Synopsis: (exponential−chirp−1 x omega_0 alpha)
Return exponential chirp of the form sin(omega_0 ∗ (exp (−alpha∗x) − 1))."
```

test

```
guile> (define omega_0 (/ (∗ 2 pi) 20.0))
guile> (test−chirp exponential−chirp−1 60 omega_0 0.1)
```

exponential-chirp-2

formula $f(x) = -\sin\left(\omega_0 e^{-\alpha x} x\right)$

synopsis (exponential-chirp-2 x ω_0 α)

documentation

```
guile> (procedure−documentation exponential−chirp−2)
"Synopsis: (exponential−chirp−2 x omega_0 alpha)
Return exponential chirp of the form −sin(omega_0 ∗ exp (−alpha ∗ x) ∗ x)."
```

test

```
guile> (define omega_0 (/ (∗ 2 pi) 20.0))
guile> (test−chirp exponential−chirp−2 500 omega_0 −0.002)
```

Procedure `test-chirp` writes two columns of numbers on file `"test.out"`. To display them in Gnuplot, one just has to issue the command:

```
gnuplot> plot "test.out" with lines
```

Windows

test-window

synopsis (test-window window width #:optional σ)

documentation

```
guile> (procedure−documentation test−window)
"Synopsis: (test−window window width #:optional sigma)
Test the window function for x in [−6/5 ∗ width, width/5]."
```

double-tanh No compact support.

formula $w(x, \text{width}, \sigma) = \frac{1}{4}\left(1 - \tanh(\sigma x)\right)\left(1 + \tanh(\sigma(x + \text{width}))\right)$

synopsis (double-tanh x width σ)

documentation

```
guile> (procedure−documentation double−tanh)
"Synopsis: (double−tanh x width sigma)
Return a window of the form
0.25 ∗ (1 − tanh (sigma ∗ x)) ∗ (1 + tanh (sigma ∗ (x + width))).
The window does not have a compact support."
```

test

```
guile> (test−window double−tanh 100 0.2)
```

gauss No compact support.

formula $w(x, \text{width}, \sigma) = \exp\left(-\frac{1}{2}\left(\frac{x+\text{width}/2}{\sigma\cdot\text{width}/2}\right)^2\right)$

synopsis (gauss x width σ)

documentation

```
guile> (procedure−documentation gauss)
"Synopsis: (gauss x width sigma)
Return a window of the form
exp (−0.5 ∗ ((x + width / 2) / (sigma ∗ width / 2))∗∗2).
The window does not have a compact support."
```

test

```
guile> (test−window gauss 100 0.3)
```

blackman-harris Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.35875 - 0.48829\cos\frac{2\pi x}{\text{width}} + 0.14128\cos\frac{4\pi x}{\text{width}} - 0.01168\cos\frac{6\pi x}{\text{width}}$

synopsis (blackman-harris x width)

documentation

```
guile> (procedure−documentation blackman−harris)
"Synopsis: (blackman−harris x width)
Return a window of the form
0.35875 − 0.48829 ∗ cos (2 ∗ pi ∗ x / width)
+ 0.14128 ∗ cos (4 ∗ pi ∗ x / width)
− 0.01168 ∗ cos (6 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window blackman−harris 100)
```

blackman-nuttall Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.3635819 - 0.4891775 \cos \frac{2\pi x}{\text{width}} + 0.1365995 \cos \frac{4\pi x}{\text{width}} - 0.0106411 \cos \frac{6\pi x}{\text{width}}$

synopsis (blackman-nuttall x width)

documentation

```
guile> (procedure−documentation blackman−nuttall)
"Synopsis: (blackman−nuttall x width)
Return a window of the form
0.3635819 − 0.4891775 ∗ cos (2 ∗ pi ∗ x / width)
+ 0.1365995 ∗ cos (4 ∗ pi ∗ x / width)
− 0.0106411 ∗ cos (6 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window blackman−nuttall 100)
```

nuttall Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.355768 - 0.487396 \cos \frac{2\pi x}{\text{width}} + 0.144232 \cos \frac{4\pi x}{\text{width}} - 0.012604 \cos \frac{6\pi x}{\text{width}}$

synopsis (nuttall x width)

documentation

```
guile> (procedure−documentation nuttall)
"Synopsis: (nuttall x width)
Return a window of the form
0.355768 − 0.487396 ∗ cos (2 ∗ pi ∗ x / width)
+ 0.144232 ∗ cos (4 ∗ pi ∗ x / width)
− 0.012604 ∗ cos (6 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window nuttall 100)
```

blackman Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{1-\alpha}{2} - \frac{1}{2} \cos \frac{2\pi x}{\text{width}} + \frac{\alpha}{2} \cos \frac{4\pi x}{\text{width}}$, where $\alpha = 0.16$.

synopsis (blackman x width)

documentation

```
guile> (procedure−documentation blackman)
"Synopsis: (blackman x width)
Return a window of the form
(1 − alpha) / 2 − 0.5 ∗ cos (2 ∗ pi ∗ x / width)
+ alpha / 2 ∗ cos (4 ∗ pi ∗ x / width),
where alpha = 0.16, between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window blackman 100)
```

lanczos Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{\sin\left(\pi\left(\frac{2x}{\text{width}}+1\right)\right)}{\left(\pi\left(\frac{2x}{\text{width}}+1\right)\right)}$

synopsis (lanczos x width)

documentation

```
guile> (procedure−documentation lanczos)
"Synopsis: (lanczos x width)
Return a window of the form
sin (pi ∗ (2 ∗ x / width + 1)) / (pi ∗ (2 ∗ x / width + 1))
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window lanczos 100)
```

hamming Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.53836 - 0.46164 \cos\left(\frac{2\pi x}{\text{width}}\right)$

synopsis (hamming x *width*)

documentation

```
guile> (procedure−documentation hamming)
"Synopsis: (hamming x width)
Return a window of the form
0.53836 − 0.46164 ∗ cos (2 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window hamming 100)
```

hann Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi x}{\text{width}}\right)\right)$

synopsis (hann x width)

documentation

```
guile> (procedure−documentation hann)
"Synopsis: (hann x width)
Return a window of the form
0.5 ∗ (1 − cos (2 ∗ pi ∗ x / width))
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window hann 100)
```

bartlett Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = \frac{2}{\text{width}}\left(\frac{\text{width}}{2} - \left|x + \frac{\text{width}}{2}\right|\right)$

synopsis (bartlett x width)

documentation

```
guile> (procedure−documentation bartlett)
"Synopsis: (bartlett x width)
Return a window of the form
2 / width ∗ (width / 2 − abs (x + width / 2))
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window bartlett 100)
```

bartlett-hann Compact support window defined on $[-\text{width}, 0]$.

formula $w(x, \text{width}) = 0.62 - 0.48 \left| \frac{x}{\text{width}} + \frac{1}{2} \right| - 0.38 \cos\left(\frac{2\pi x}{\text{width}}\right)$

synopsis (bartlett-hann x width)

documentation

```
guile> (procedure−documentation bartlett−hann)
"Synopsis: (bartlett−hann x width)
Return a window of the form
0.62 − 0.48 ∗ abs (x / width + 0.5) − 0.38 ∗ cos ( 2 ∗ pi ∗ x / width)
between x = −width and x = 0 and zero otherwise."
```

test

```
guile> (test−window bartlett−hann 100)
```

The `test-window` procedure invoked on a window function generates a two column output on `"test.out"`, which can be then displayed with gnuplot by issuing the command

```
gnuplot> plot "test.out" with lines
```

Signals

Synopsis:

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

This file loads `"windows"` and `"chirps"`, and then it defines procedure `test-chirp-in-window`, which combines chirps and windows, evaluating the resulting signal at 2,000 points within the window. The signal is printed out on `"test.out"` in two columns, ready to be displayed with gnuplot.

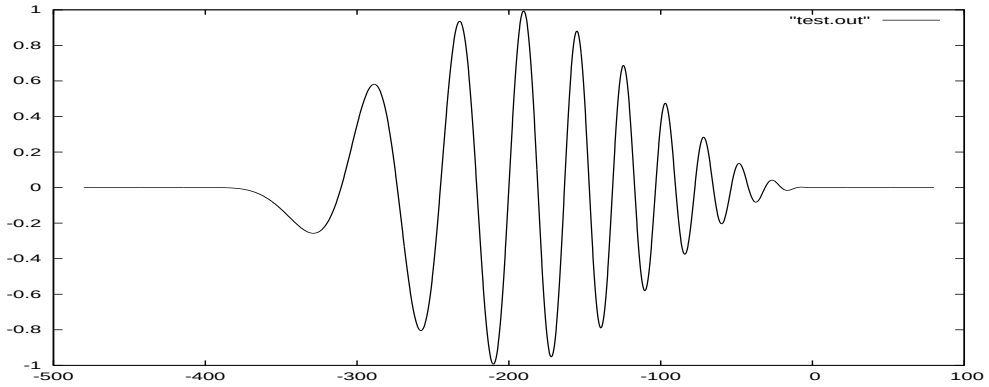
test-chirp-in-window

documentation

```
guile> (procedure−documentation test−chirp−in−window)
"Synopsis: (test−chirp−in−window chirp omega_0 alpha window width #:optional
sigma)
Test the chirp in the window for x in [−6/5 ∗ width, width/5]."
```

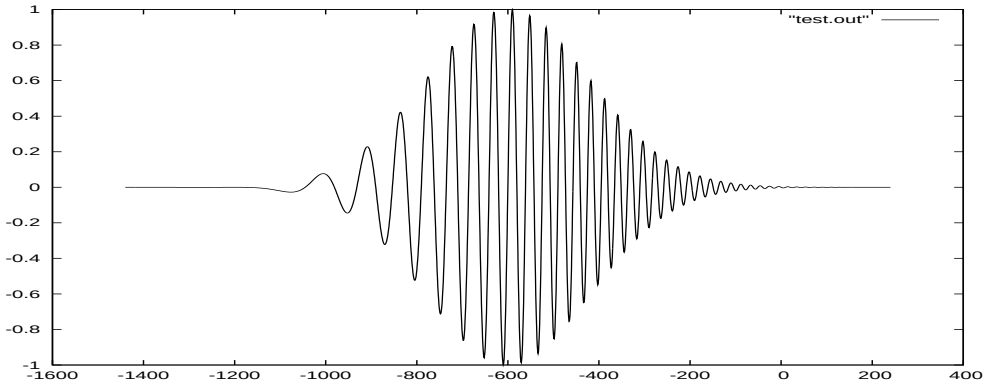
examples

1. A stretching linear chirp in Hann window:



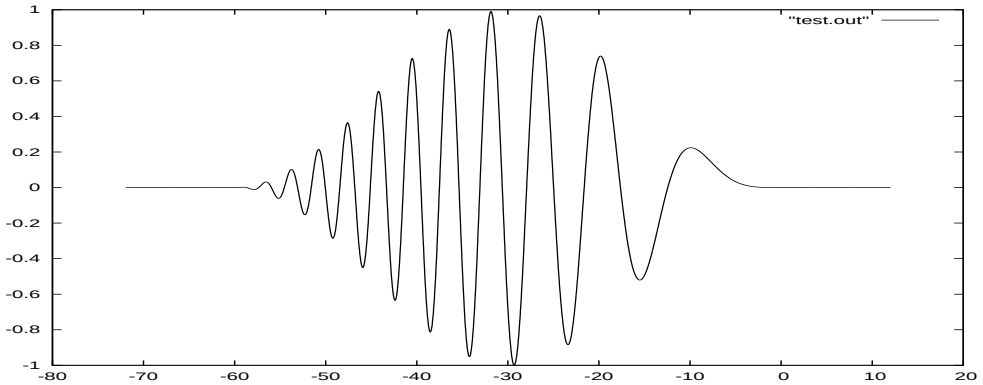
```
guile> (define omega_0 (/ (* 2 pi) 20.0))
guile> (define width 400.0)
guile> (test-chirp-in-window linear-chirp omega_0 (- (/ omega_0 (* 2.0 width)))
hann width)
```

2. A stretching linear chirp in Gauss window:



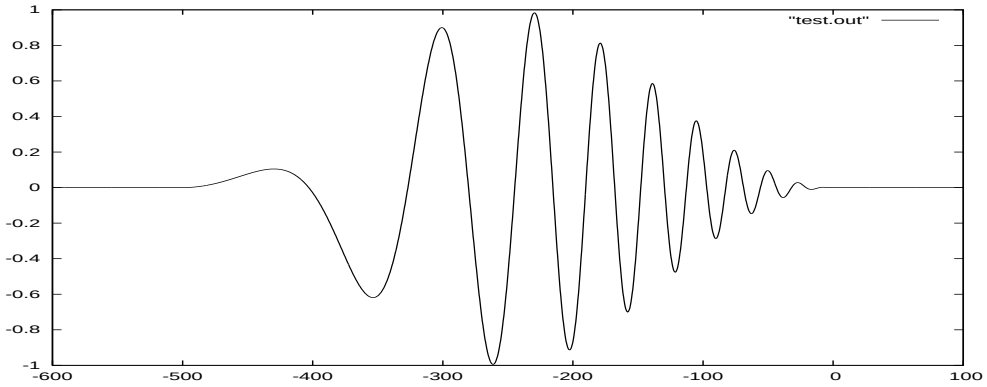

```
guile> (define omega_0 (/ (* 2 pi) 20.0))
guile> (define width 1200.0)
guile> (test-chirp-in-window linear-chirp omega_0 (- (/ omega_0 (* 2.0 width)))
gauss width 0.3)
```

3. A shrinking linear chirp in Hann window:



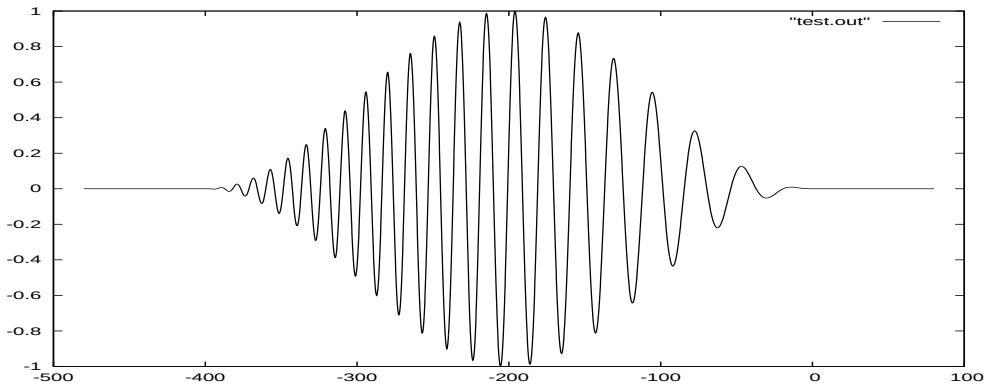

```
guile> (define omega_0 (/ (* 2 pi) 1800.0))
guile> (define width 60.0)
guile> (test-chirp-in-window linear-chirp omega_0 0.02 hann width)
```

4. A stretching exponential chirp in Hann window:



```
guile> (define omega_0 (/ (* 2 pi) 20.0))
guile> (define width 500.0)
guile> (test-chirp-in-window exponential-chirp-2 omega_0 -0.002 hann width)
```

5. A shrinking exponential chirp in Hann window:




```
guile> (define omega_0 (/ (* 2 pi) 40.0))
guile> (define width 400.0)
guile> (test-chirp-in-window exponential-chirp-2 omega_0 0.002 hann width)
```

2.7.4 Media

Synopsis:

```
guile> (load-from-path "media")
```

This file contains precooked auxiliary differential equation (ADE) solvers for various media types. The solvers are not "stand-alone". They are designed to slot into the FORMS input file, and utilize global objects defined and filled by FORMS, such as `forms.D`, `forms.media.number_of_auxiliary_fields`, `forms.S`, `forms.dt`, and `forms.E_old`. In effect, the solvers deliver

$$\boldsymbol{E}(t_n) \leftarrow \mathcal{E}\left(\boldsymbol{D}(t_n), \boldsymbol{S}(t_n), \boldsymbol{S}(t_{n-1}), \ldots, \boldsymbol{E}(t_{n-1}), t_n, \mathrm{d}t\right) \tag{40}$$

As a side effect, the solvers also update the auxiliary $\boldsymbol{S}$ field.

The media model implemented at present is isotropic. Minor changes to the code itself will be needed to add a tensor material model. This will be implemented in future releases ◀

Users should understand that the commonly used Courant-Friedrichs-Lewy time step criterion as applied to Maxwell equations does not cover stability of the ADE. It may happen that the Maxwell equations solver time step is too long for the ADE used. ◀

The solvers provided at present are

dielectric Frequency independent scalar dielectric characterized by ϵ.

formula $\epsilon(\omega) = \epsilon = \text{const}$

ADE $\boldsymbol{E}(t_n) = \epsilon^{-1}\boldsymbol{D}(t_n)$

synopsis (set! E (dielectric ϵ^{-1}))

documentation

guile> (**procedure−documentation** dielectric)
"Synopsis:

(dielectric one_by_epsilon)

Return the updated value for E that corresponds to this one_by_epsilon
and forms.D."

drude Drude metal characterized by ω_p, γ and ϵ_∞.

formula

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

ADE

$$\begin{aligned}
\boldsymbol{S}(t_n) &= \frac{1}{2+\gamma\Delta t}\left(4\boldsymbol{S}(t_{n-1}) - (2-\gamma\Delta t)\,\boldsymbol{S}(t_{n-2}) - 2\omega_p^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_\infty}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}(t_n)\right)\\
\boldsymbol{S}(t_{n-2}) &\leftarrow \boldsymbol{S}(t_{n-1})\\
\boldsymbol{S}(t_{n-1}) &\leftarrow \boldsymbol{S}(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 2)
(set! E (drude ϵ_∞^{-1} ω_p^2 γ))

documentation

guile> (**procedure−documentation** drude)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 2)
(drude one_by_epsilon_D omega_p_square gamma)

Return the updated value for E that corresponds to these three
parameters, forms.D, forms.E_old−−this contains the version of E that
is one time step behind D, forms.S[0], and forms.S[1]. Update forms.S
at the same time. The function does not check if there is enough space
in forms.S."

lorentz_1 A one-pole Lorentz medium characterized by ω_p, γ, ω_t and ϵ_L.

formula

$$\epsilon(\omega) = \epsilon_L - \frac{\omega_p^2}{\omega^2 + \mathrm{i}\gamma\omega - \omega_t^2}$$

ADE

$$\begin{aligned}
\boldsymbol{S}(t_n) &= \frac{1}{2+\gamma\Delta t}\left(\left(4-2\omega_t^2\Delta t^2\right)\boldsymbol{S}(t_{n-1}) - (2-\gamma\Delta t)\,\boldsymbol{S}(t_{n-2}) - 2\omega_p^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_L}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}(t_n)\right)\\
\boldsymbol{S}(t_{n-2}) &\leftarrow \boldsymbol{S}(t_{n-1})\\
\boldsymbol{S}(t_{n-1}) &\leftarrow \boldsymbol{S}(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 2)
(set! E (lorentz_1 ϵ_L^{-1} ω_p^2 ω_t^2 γ))

documentation

guile> (**procedure−documentation** lorentz_1)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 2)
(lorentz_1 one_by_epsilon_L omega_p_square omega_t_square gamma)

Return the updated value for E that corresponds to these four parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, forms.S[0], and forms.S[1]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

lorentz_2 A two pole Lorentz medium characterized by ϵ_L, then for the first pole ω_{p1}, ω_{t1}, and γ_1, and for the second pole ω_{p2}, ω_{t2}, and γ_2.

formula

$$\epsilon(\omega) = \epsilon_L - \frac{\omega_{p1}^2}{\omega^2 + \mathrm{i}\gamma_1\omega - \omega_{t1}^2} - \frac{\omega_{p2}^2}{\omega^2 + \mathrm{i}\gamma_2\omega - \omega_{t2}^2}$$

ADE

$$\begin{aligned}
\boldsymbol{S}_1(t_n) &= \frac{1}{2+\gamma_1\Delta t}\left(\left(4-2\omega_{t1}^2\Delta t^2\right)\boldsymbol{S}_1(t_{n-1}) - (2-\gamma_1\Delta t)\,\boldsymbol{S}_1(t_{n-2}) - 2\omega_{p1}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_2(t_n) &= \frac{1}{2+\gamma_2\Delta t}\left(\left(4-2\omega_{t2}^2\Delta t^2\right)\boldsymbol{S}_2(t_{n-1}) - (2-\gamma_2\Delta t)\,\boldsymbol{S}_2(t_{n-2}) - 2\omega_{p2}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_L}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}_1(t_n) + \boldsymbol{S}_2(t_n)\right)\\
\boldsymbol{S}_1(t_{n-2}) &\leftarrow \boldsymbol{S}_1(t_{n-1})\\
\boldsymbol{S}_1(t_{n-1}) &\leftarrow \boldsymbol{S}_1(t_n)\\
\boldsymbol{S}_2(t_{n-2}) &\leftarrow \boldsymbol{S}_2(t_{n-1})\\
\boldsymbol{S}_2(t_{n-1}) &\leftarrow \boldsymbol{S}_2(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 4)
(set! E (lorentz_2 ϵ_L^{-1} ω_{p1}^2 ω_{t1}^2 γ_1 ω_{p2}^2 ω_{t2}^2 γ_2))

documentation

guile> (**procedure−documentation** lorentz_2)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 4)
(lorentz_2 one_by_epsilon_L omega_p_square_1 omega_t_square_1 gamma_1
omega_p_square_2 omega_t_square_2 gamma_2)

Return the updated value for E that corresponds to these seven parameters, forms.D, forms.E_old−−this contains the version of E that is one time step behind D, and forms.S[n], for n in [0..3]. Update forms.S at the same time. The function does not check if there is enough space in forms.S."

drude_lorentz_1 A mixed Drude-one-pole-Lorentz medium characterized by the following Drude parameters, ω_D and γ_D; then by the following Lorentz parameters, ω_{pL}, ω_{tL} and γ_L; and by ϵ_∞ here referred to as ϵ_{DL}.

formula

$$\epsilon(\omega) = \epsilon_{DL} - \frac{\omega_D^2}{\omega^2 + \mathrm{i}\gamma_D\omega} - \frac{\omega_{pL}^2}{\omega^2 + \mathrm{i}\gamma_L\omega - \omega_{tL}^2}$$

ADE

$$\begin{aligned}
\boldsymbol{S}_D(t_n) &= \frac{1}{2+\gamma_D\Delta t}\left(4\boldsymbol{S}_D(t_{n-1}) - (2-\gamma_D\Delta t)\boldsymbol{S}_D(t_{n-2}) - 2\omega_D^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_L(t_n) &= \frac{1}{2+\gamma_L\Delta t}\left((4-2\omega_{tL}^2\Delta t^2)\boldsymbol{S}_L(t_{n-1}) - (2-\gamma_L\Delta t)\boldsymbol{S}_L(t_{n-2}) - 2\omega_{pL}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_{DL}}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}_D(t_n) + \boldsymbol{S}_L(t_n)\right)\\
\boldsymbol{S}_D(t_{n-2}) &\leftarrow \boldsymbol{S}_D(t_{n-1})\\
\boldsymbol{S}_D(t_{n-1}) &\leftarrow \boldsymbol{S}_D(t_n)\\
\boldsymbol{S}_L(t_{n-2}) &\leftarrow \boldsymbol{S}_L(t_{n-1})\\
\boldsymbol{S}_L(t_{n-1}) &\leftarrow \boldsymbol{S}_L(t_n)
\end{aligned}$$

synopsis

(define forms.media.number_of_auxiliary_fields 4)
(set! E (drude_lorentz_1 ϵ_{DL}^{-1} ω_D^2 γ_D ω_{pL}^2 ω_{tL}^2 γ_L))

documentation

guile> (**procedure−documentation** drude_lorentz_1)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 4)
(drude_lorentz_1 one_by_epsilon_DL
omega_p_square_D gamma_D
omega_p_square_L omega_t_square_L gamma_L)

Return the updated value for E that corresponds to these six
parameters, forms.D, forms.E_old−−this contains the version of E that
is one time step behind D, and forms.S[n], for n in [0..3]. Update
forms.S at the same time. The function does not check if there is
enough space in forms.S."

drude_lorentz_2 A mixed Drude-two-pole-Lorentz medium characterized by the following Drude parameters, ω_D and γ_D; then by the following Lorentz parameters, for the first pole ω_{p1}, ω_{t1} and γ_1, and for the second pole ω_{p2}, ω_{t2} and γ_2; and by ϵ_∞ here referred to as ϵ_{DL}.

formula

$$\epsilon(\omega) = \epsilon_{DL} - \frac{\omega_D^2}{\omega^2 + \mathrm{i}\gamma_D\omega} - \frac{\omega_{p1}^2}{\omega^2 + \mathrm{i}\gamma_1\omega - \omega_{t1}^2} - \frac{\omega_{p2}^2}{\omega^2 + \mathrm{i}\gamma_2\omega - \omega_{t2}^2}$$

ADE

$$\begin{aligned}
\boldsymbol{S}_D(t_n) &= \frac{1}{2+\gamma_D\Delta t}\left(4\boldsymbol{S}_D(t_{n-1}) - (2-\gamma_D\Delta t)\boldsymbol{S}_D(t_{n-2}) - 2\omega_D^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{S}_1(t_n) &= \frac{1}{2+\gamma_1\Delta t}\left((4-2\omega_{t1}^2\Delta t^2)\boldsymbol{S}_1(t_{n-1}) - (2-\gamma_1\Delta t)\boldsymbol{S}_1(t_{n-2}) - 2\omega_{p1}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)
\end{aligned}$$

$$
\begin{aligned}
\boldsymbol{S}_2(t_n) &= \frac{1}{2+\gamma_2\Delta t}\left((4-2\omega_{t2}^2\Delta t^2)\boldsymbol{S}_2(t_{n-1}) - (2-\gamma_2\Delta t)\boldsymbol{S}_2(t_{n-2}) - 2\omega_{p2}^2\Delta t^2\boldsymbol{E}(t_{n-1})\right)\\
\boldsymbol{E}(t_n) &= \frac{1}{\epsilon_{DL}}\left(\boldsymbol{D}(t_n) + \boldsymbol{S}_D(t_n) + \boldsymbol{S}_1(t_n) + \boldsymbol{S}_2(t_n)\right)\\
\boldsymbol{S}_D(t_{n-2}) &\leftarrow \boldsymbol{S}_D(t_{n-1})\\
\boldsymbol{S}_D(t_{n-1}) &\leftarrow \boldsymbol{S}_D(t_n)\\
\boldsymbol{S}_1(t_{n-2}) &\leftarrow \boldsymbol{S}_1(t_{n-1})\\
\boldsymbol{S}_1(t_{n-1}) &\leftarrow \boldsymbol{S}_1(t_n)\\
\boldsymbol{S}_2(t_{n-2}) &\leftarrow \boldsymbol{S}_2(t_{n-1})\\
\boldsymbol{S}_2(t_{n-1}) &\leftarrow \boldsymbol{S}_2(t_n)
\end{aligned}
$$

synopsis

(define forms.media.number_of_auxiliary_fields 6)
(set! E (drude_lorentz_2 ϵ_{DL}^{-1} ω_D^2 γ_D ω_{p1}^2 ω_{t1}^2 γ_1 ω_{p2}^2 ω_{t2}^2 γ_2))

documentation

guile> (**procedure−documentation** drude_lorentz_2)
"Synopsis:

(define forms.media.number_of_auxiliary_fields 6)
(drude_lorentz_1 one_by_epsilon_DL
omega_p_square_D gamma_D
omega_p_square_1 omega_t_square_1 gamma_1
omega_p_square_2 omega_t_square_2 gamma_2)

Return the updated value for E that corresponds to these nine
parameters, forms.D, forms.E_old−−this contains the version of E that
is one time step behind D, and forms.S[n], for n in [0..5]. Update
forms.S at the same time. The function does not check if there is
enough space in forms.S."

2.7.5 Virtual Measurements

Synopsis:

guile> (**load−from−path** "measurements")

The virtual measurements library utilizes the built-in Scheme function `interpolate`—see Sections 2.2.7 above and 3.5.4, page 121, also Section 8.6, page 203, module ⟨ Interpolation 136 ⟩—and then wraps higher level utilities around it.
Synopsis:

guile> (**interpolate** name x y z)

where `name` is one of `"Ex"`, `"Ey"`, `"Ez"`, `"Dx"`, `"Dy"`, `"Dz"`, `"Hx"`, `"Hy"`, `"Hz"`, `"Bx"`, `"By"`, `"Bz"`, and `x`, `y`, and `z` are the x, y and z coordinates of a point onto which the fields are to be interpolated. The function returns a linear, volume weighted interpolation of a named field at (x, y, z) from the eight field mounting points that define the cell to which (x, y, z) belongs. The function knows how the fields are mounted (face versus edge), and takes this into account.

The interpolation takes place at the time point in the integration at which it is invoked. This point always coincides with $\boldsymbol{E}$ and $\boldsymbol{D}$, but not with $\boldsymbol{H}$ and $\boldsymbol{B}$. Consequently, the $\boldsymbol{H}$ and $\boldsymbol{B}$ fields are additionally interpolated in time.

Surface

The library file `measurements.scm` defines class `<surface>`, that provides general means for defining a *single* surface coordinate patch. If a given surface requires multiple coverings, multiple instances of class `<surface>` must be created. The slots of the class are as follows:

equation This is a lambda of two real arguments that returns a `3D-vector`. It corresponds to an equation that describes a surface, cf. Section 3.5.2, page 108, where we describe a surface in terms of three equations:

$$x = x_S(u, v), \tag{41}$$
$$y = y_S(u, v), \tag{42}$$
$$z = z_S(u, v). \tag{43}$$

The lambda represents the equations—all three.

accessor `eqn`
init-keyword `#:eqn`
default (**lambda** (u v) (**make** <3D−vector> #:x u #:y v))

The surface specified by the lambda is then covered with a discrete grid of points beginning with (u_0, v_0) and ending at (u_f, v_f) with increments of $(\Delta u, \Delta v)$, by providing the following:

initial-values This is a two element vector of reals that corresponds to u_0 and v_0 of the coordinate patch.

accessor `initial-values`
init-keyword `#:initial-values`
default (**make−vector** 2 0)

final-values This is a two element vector of reals that corresponds to u_f and v_f of the coordinate patch.

accessor `final-values`
init-keyword `#:final-values`
default (**make−vector** 2 10)

increments This is a two element vector of reals that corresponds to Δu and Δv of the coordinate patch.

accessor `increments`
init-keyword `#:increments`
default (**make−vector** 2 1)

restriction A predicate that tells if a given pair (u, v) belongs to the domain of the surface.

accessor `restriction`
init-keyword `#:restriction`
default (**lambda** (u v) #t)

plaquettes A list of plaquettes covering the surface as generated by the `iterator` method, see Section 2.7.5 below.

accessor `plaquettes`
init-keyword `#:plaquettes`
default '()

Inherent in this definition is that the surface boundary is, topologically, a rectangle that can be mapped onto $[u_{\mathrm{lo}}, u_{\mathrm{hi}}] \times [v_{\mathrm{lo}}, v_{\mathrm{hi}}]$. Hemispheres and circles can be mapped this way using angular coordinates, at a cost of having to deal with coordinate system singularities. Alternatively, we can use the `restriction` to clip some of the (u, v) pairs off. This way, we can define, for example, a disk without having to resort to angular coordinates and thus avoiding the singularity. To do so, we map a disk of radius r onto Cartesian (x, y) and impose the `restriction` of $x^2 + y^2 \leq r^2$.

Surface Iterator

The `iterator` is a method that takes an instance of class `<surface>` as its argument and initializes its list of plaquettes. Each plaquette is described in terms of a pair, the first element of which is a point roughly in the centre of the plaquette, $\boldsymbol{r}(u_c, v_c)$, and the second element of which is $\boldsymbol{n}\,\mathrm{d}^2S$ of the plaquette, that is, a vector normal to the plaquette, of length equal to the surface area of the plaquette.
Synopsis:

```
guile> (define s (make <surface>))
guile> (iterator s)
```

Arbitrary iterative operations over the surface, such as computation of a flux, can be then implemented by employing a `do` and crawling over the list with a `car/cdr` combination (This is the fastest method to process a list, sic! To access list elements by index may easily take ten times longer.). Each pair returned that represents a plaquette can then be split, also with `car/cdr` into $\boldsymbol{r}(u_c, v_c)$ and $\boldsymbol{n}\,\mathrm{d}^2S$.

The library provides convenience functions that make a hemisphere, a rectangular plane, or a disk. The functions invoke the `iterator` internally so that the plaquette slot of the returned object is fully initialized.

If the object itself is modified, for example, by altering its inital or final values, or its increments, the `plaquettes` slot has to be recomputed by invoking the `iterator` explicitly.

Surface Area

Method `surface-area`, defined on `measurements.scm`, provides a simple example of `<surface>` operations. It looks as follows:

```
(define-method (surface-area (surface <surface>))
  (do ((plaquettes (plaquettes surface) (cdr plaquettes))
       (area 0 (+ area (norm (cdr (car plaquettes))))))
      ((null? plaquettes) area)))
```

In the following example, we create a hemisphere of radius 1, then evaluate its surface area a couple of times, while increasing the density of the covering mesh.

```
guile> (set! %load-path (cons "/home/gustav/src/Forms/lib" %load-path))
guile> (load-from-path "measurements")
guile> (define hemisphere (make-hemisphere 1))
guile> (surface-area hemisphere)
6.24525810396011
guile> (* 2 pi)
6.28318530717959
```

The error in the computed result is 0.6%. We can do better by halving the increments:

```
guile> (slot-set! hemisphere 'increments (vector (/ degree 2) (/ degree 2)))
guile> (iterator hemisphere)
guile> (surface-area hemisphere)
6.31050118963148
```

This time the error is 0.4%. Halving the increments again yields:

```
guile> (slot-set! hemisphere 'increments (vector (/ degree 4) (/ degree 4)))
guile> (iterator hemisphere)
guile> (surface-area hemisphere)
6.30124099476907
```

This time the error has been reduced to 0.3%.

The hemisphere in the example is oriented upwards. It is easy to re-orient the one created already otherwise, by editing its slots:

```
guile> (slot-set! hemisphere 'initial-values (vector (- half_pi) 0))
guile> (slot-set! hemisphere 'final-values (vector half_pi pi))
guile> (slot-set! hemisphere 'increments (vector degree degree))
guile> (iterator hemisphere)
guile> (surface-area hemisphere)
6.28278657190559
guile>
```

This time, the calculated surface area is markedly more accurate, too. The error is 0.006% only. The reason for this is that there is less contribution now from the vicinity of the coordinate system singularity at the north pole, although we still approach both the north and the south poles. It is best to stay away from such singularities altogether, if possible.

The following illustrates a simple use of the restriction predicate to define a disk without a coordinate system singularity:

```
guile> (define disk (make <surface>
... #:equation (lambda (x y) (make <3D-vector> #:x x #:y y #:z 0))
... #:initial-values (vector -1 -1)
... #:final-values (vector 1 1)
... #:increments (vector 0.01 0.01)
... #:restriction (lambda (x y) (<= (+ (* x x) (* y y)) 1.0))))
guile> (describe disk)
#<<surface> 7f864560> is an instance of class <surface>
Slots are:
equation = #<procedure #f (x y)>
initial-values = #(-1 -1)
final-values = #(1 1)
increments = #(0.01 0.01)
restriction = #<procedure #f (x y)>
plaquettes = ()
```

It is not a good idea to invoke `describe` on a fully defined `<surface>`, because the procedure would list all plaquettes. But in this case, `make` has initialized `plaquettes` to `'()`, so we're OK.

Now we initialize the plaquettes and then compute the surface area:

```
guile> (iterator disk)
guile> (surface-area disk)
3.14280000000221
```

The relative error in this computation is about 0.04%.

In the following we list utility methods that generate certain predefined surfaces.

3 Numerics

Here is a complete summary of numerics, and explicit listing of all formulas used throughout the code.

Mathematics comes first! There is no point, absolutely no point, laying your fingers on the keyboard until you've got the mathematics worked out in all required detail. All the clever coding in the world cannot replace this most basic of all steps.

So, here we are...

3.1 Maxwell Equations

We start from the following form of equations as listed in Feynman [6], Chapter 32, pages 32-4/5.

$$\begin{aligned}
\boldsymbol{\nabla} \cdot \boldsymbol{D} &= \rho_{\text{free}}, && (44)\\
\boldsymbol{\nabla} \cdot \boldsymbol{B} &= 0, && (45)\\
\boldsymbol{\nabla} \times \boldsymbol{H} &= \boldsymbol{j}_{\text{free}} + \partial_t \boldsymbol{D}, && (46)\\
\boldsymbol{\nabla} \times \boldsymbol{E} &= -\partial_t \boldsymbol{B}, && (47)\\
\boldsymbol{D} &= \epsilon \boldsymbol{E}, && (48)\\
\boldsymbol{B} &= \mu \boldsymbol{H}. && (49)
\end{aligned}$$

The $\boldsymbol{D}$ and $\boldsymbol{H}$ are, as Feynman puts it, "hidden ways of *not* paying attention to what is going on inside the material."

In this code we have broadened equations (48) and (49) to cover *any* dependence of $\boldsymbol{E}$ on $\boldsymbol{D}$ and $\boldsymbol{H}$ on $\boldsymbol{B}$, including tensor dependencies, field accumulations, nonlinear dependencies, dependence on location, time, etc.,

$$\begin{aligned}
\boldsymbol{E} &= \boldsymbol{E}\left(\boldsymbol{D}, \boldsymbol{r}, t, \int_0^t f(\boldsymbol{E}, \boldsymbol{r}, t'), \mathrm{d}t', \ldots\right), && (50)\\
\boldsymbol{H} &= \boldsymbol{H}\left(\boldsymbol{B}, \boldsymbol{r}, t, \int_0^t f(\boldsymbol{H}, \boldsymbol{r}, t'), \mathrm{d}t', \ldots\right). && (51)
\end{aligned}$$

The user defines these by writing a Scheme script. This is discussed in more details in Section 7.3, page 168. Apart from an almost unlimited range of material properties, the user may define moving media, i.e., media whose distribution is a function of time, as well.

On the other hand, FORMS does not allow for the presence of free charges or currents, because it is a strictly scattering code, hence

$$\begin{aligned}
\rho_{\text{free}} &= 0, \quad \text{and} && (52)\\
\boldsymbol{j}_{\text{free}} &= 0, && (53)
\end{aligned}$$

and the resulting two equations

$$\begin{aligned}
\boldsymbol{\nabla} \cdot \boldsymbol{D} &= 0, \quad \text{and} && (54)\\
\boldsymbol{\nabla} \cdot \boldsymbol{B} &= 0, && (55)
\end{aligned}$$

are satisfied automatically by the solution method (FDTD), with the notable exception of multigrid edges, where special action must be taken to ensure the preservation of divergence-free field configurations. The same applies to field averaging procedures that transfer data from finer to coarser grids.

Although there is neither ϵ_0 nor μ_0 used explicitly in equations (44) through (47), we nevertheless eliminate these two pesky constants by the following substitutions

$$\begin{aligned}
\tilde{\boldsymbol{E}} &= \sqrt{\frac{\epsilon_0}{\mu_0}}\boldsymbol{E}, && (56)\\
\tilde{\boldsymbol{D}} &= \sqrt{\frac{1}{\epsilon_0 \mu_0}}\boldsymbol{D}, && (57)\\
\tilde{\boldsymbol{B}} &= \frac{1}{\mu_0}\boldsymbol{B}, && (58)\\
\tilde{\boldsymbol{H}} &= \boldsymbol{H}, && (59)\\
c &= \sqrt{\frac{1}{\epsilon_0 \mu_0}}. && (60)
\end{aligned}$$

The resulting charge and current free equations for $\tilde{\boldsymbol{E}}$, $\tilde{\boldsymbol{D}}$, $\tilde{\boldsymbol{B}}$, and $\tilde{\boldsymbol{H}}$ look as follows:

$$\boldsymbol{\nabla} \cdot \tilde{\boldsymbol{D}} = 0, \quad (61)$$

$$\boldsymbol{\nabla}\cdot\tilde{\boldsymbol{B}} = 0, \tag{62}$$
$$\boldsymbol{\nabla}\times\tilde{\boldsymbol{H}} = \frac{\partial}{\partial ct}\tilde{\boldsymbol{D}}, \tag{63}$$
$$\boldsymbol{\nabla}\times\tilde{\boldsymbol{E}} = -\frac{\partial}{\partial ct}\tilde{\boldsymbol{B}}. \tag{64}$$

We are free to choose our unit of time so that $c = 1$, whereupon, dropping the tildas, we get

$$\boldsymbol{\nabla}\times\boldsymbol{H} = \partial_t\boldsymbol{D}, \tag{65}$$
$$\boldsymbol{\nabla}\times\boldsymbol{E} = -\partial_t\boldsymbol{B}, \tag{66}$$
$$\boldsymbol{E} = \boldsymbol{E}\left(\boldsymbol{D},\boldsymbol{r},t,\int_0^t f(\boldsymbol{E},\boldsymbol{r},t'),\mathrm{d}t',\ldots\right), \tag{67}$$
$$\boldsymbol{H} = \boldsymbol{H}\left(\boldsymbol{B},\boldsymbol{r},t,\int_0^t f(\boldsymbol{H},\boldsymbol{r},t'),\mathrm{d}t',\ldots\right). \tag{68}$$

Whereas these equations look exactly as (46) and (47), we find that in vacuum

$$\boldsymbol{E} = \boldsymbol{D}, \quad \text{and} \tag{69}$$
$$\boldsymbol{H} = \boldsymbol{B}. \tag{70}$$

On the other hand, because the basic equations are unchanged by this transformation, and because users are free to define $\boldsymbol{E}(\boldsymbol{D})$ and $\boldsymbol{H}(\boldsymbol{B})$ as they wish, they can keep ϵ_0 and μ_0 around.

It is useful to rewrite equations (65) and (66) in an explicit component notation, suitable for direct conversion into computer code:

$$\partial_t D_x = \partial_y H_z - \partial_z H_y, \tag{71}$$
$$\partial_t D_y = \partial_z H_x - \partial_x H_z, \tag{72}$$
$$\partial_t D_z = \partial_x H_y - \partial_y H_x, \tag{73}$$
$$\boldsymbol{D} \rightarrow \boldsymbol{E}, \tag{74}$$
$$\partial_t B_x = \partial_z E_y - \partial_y E_z, \tag{75}$$
$$\partial_t B_y = \partial_x E_z - \partial_z E_x, \tag{76}$$
$$\partial_t B_z = \partial_y E_x - \partial_x E_y, \tag{77}$$
$$\boldsymbol{B} \rightarrow \boldsymbol{H}. \tag{78}$$

3.2 Discretization

We distribute fields $\boldsymbol{E}$ and $\boldsymbol{D}$ on cell edges, and field $\boldsymbol{H}$ on cell faces, as shown in Figure 3, page 89, and as is described in Section 12.2. See also Figure 5, page 300.

The specific locations of the fields within an (i, j, k) cell, where i, j, and k correspond to the cell center, are as follows:

$$\text{Location}\left(E_x(i,j,k)\right) = \boldsymbol{r}_0 + \left(i, j-\frac{1}{2}, k-\frac{1}{2}\right)\Delta_g, \tag{79}$$

$$\text{Location}\left(E_y(i,j,k)\right) = \boldsymbol{r}_0 + \left(i-\frac{1}{2}, j, k-\frac{1}{2}\right)\Delta_g, \tag{80}$$

$$\text{Location}\left(E_z(i,j,k)\right) = \boldsymbol{r}_0 + \left(i-\frac{1}{2}, j-\frac{1}{2}, k\right)\Delta_g, \tag{81}$$

$$\text{Location}\left(H_x(i,j,k)\right) = \boldsymbol{r}_0 + \left(i-\frac{1}{2}, j, k\right)\Delta_g, \tag{82}$$

$$\text{Location}\left(H_y(i,j,k)\right) = \boldsymbol{r}_0 + \left(i, j-\frac{1}{2}, k\right)\Delta_g, \tag{83}$$

$$\text{Location}\left(H_z(i,j,k)\right) = \boldsymbol{r}_0 + \left(i, j, k-\frac{1}{2}\right)\Delta_g, \tag{84}$$

where $\boldsymbol{r}_0$ is the location of the $(0,0,0)$ cell center and Δ_g is the grid constant—here assumed to be the same in all three directions.

FDTD is basically a leapfrog in time and space. Assuming $\boldsymbol{E}$ and $\boldsymbol{D}$ are defined at the n^{th} time step $t = t_E = t_0 + n\Delta t$ and $\boldsymbol{H}$ is defined at $t_H = t_E + \Delta t/2$, which we are going to mark as $(n+1/2)^{\text{th}}$ time step, we can rewrite equations (71) through (77) as follows:

$$D_x^{n+1}(i,j,k) = D_x^n(i,j,k) + \frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right), \tag{85}$$

$$D_y^{n+1}(i,j,k) = D_y^n(i,j,k) + \frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1) - H_z^{n+1/2}(i,j,k) + H_z^{n+1/2}(i-1,j,k)\right), \tag{86}$$

$$D_z^{n+1}(i,j,k) = D_z^n(i,j,k) + \frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k) - H_x^{n+1/2}(i,j,k) + H_x^{n+1/2}(i,j-1,k)\right), \tag{87}$$

$$\boldsymbol{D}^{n+1} \to \boldsymbol{E}^{n+1}, \tag{88}$$

$$B_x^{n+3/2}(i,j,k) = B_x^{n+1/2}(i,j,k) + \frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j,k+1) - E_y^{n+1}(i,j,k) - E_z^{n+1}(i,j+1,k) + E_z^{n+1}(i,j,k)\right), \tag{89}$$

$$B_y^{n+3/2}(i,j,k) = B_y^{n+1/2}(i,j,k) + \frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i+1,j,k) - E_z^{n+1}(i,j,k) - E_x^{n+1}(i,j,k+1) + E_x^{n+1}(i,j,k)\right), \tag{90}$$

$$B_z^{n+3/2}(i,j,k) = B_z^{n+1/2}(i,j,k) + \frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i,j+1,k) - E_x^{n+1}(i,j,k) - E_y^{n+1}(i+1,j,k) + E_y^{n+1}(i,j,k)\right), \tag{91}$$

$$\boldsymbol{B}^{n+3/2} \to \boldsymbol{H}^{n+3/2}. \tag{92}$$

Certain helpful regularities are easily seen here. For example, when evaluating $\boldsymbol{\nabla} \times \boldsymbol{H}$, we reach *down* towards $i-1$, $j-1$ and $k-1$, and the down-reached component is the second and the fourth in the formula. On the other hand, when evaluating $\boldsymbol{\nabla} \times \boldsymbol{E}$ we reach *up* towards $i+1$, $j+1$ and $k+1$, and the up-reached component is the first and the third in the formula.

The way we have arrived at these specific equations was, first, to look at the corresponding differential equation, and then to look at the exact positions of the differentiated fields with respect to the ones being advanced. For example, H_y is attached at $(i, j-1/2, k)$, but the E_z field is attached at $(i-1/2, j-1/2, k)$. The $\partial_x E_z$ derivative has to fall exactly on $(i, j-1/2, k)$ in this case, and it will do so, if we subtract $E_z(i-1/2, j-1/2, k)$ from $E_z(i+1/2, j-1/2, k)$.

3.3 Signal Injection

Signal injection is one of these tricky and immensely confusing parts of FDTD, but it is this that makes it fly. Tremendously useful, together with PML boundaries, which are discussed in the next section. The explicit restating of FDTD equations, (85) through (91) will help us in deriving specifications for the signal injection and extraction algorithm.

We assume the following general signal property (in vacuum, hence $\boldsymbol{E} = \boldsymbol{D}$ and $\boldsymbol{H} = \boldsymbol{B}$)

$$\begin{aligned} \boldsymbol{E}(\boldsymbol{r}, t) &= \boldsymbol{E}(\boldsymbol{n} \cdot \boldsymbol{r} - t), & (93) \\ \boldsymbol{H}(\boldsymbol{r}, t) &= \boldsymbol{H}(\boldsymbol{n} \cdot \boldsymbol{r} - t), & (94) \end{aligned}$$

where $\boldsymbol{n}$ is a normal vector in the direction of signal propagation. Let us call

$$\zeta = \boldsymbol{n} \cdot \boldsymbol{r} - t \tag{95}$$

then

$$\begin{aligned} \boldsymbol{E} &= \boldsymbol{E}(\zeta), & (96) \\ \boldsymbol{H} &= \boldsymbol{H}(\zeta). & (97) \end{aligned}$$

It is easy to see that

$$\partial_t \boldsymbol{E} = \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} \partial_t \zeta = -\frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta}, \tag{98}$$

and similarly

$$\partial_t \boldsymbol{H} = -\frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta}. \tag{99}$$

We also have that

$$\boldsymbol{\nabla} \cdot \boldsymbol{E} = \sum_i \partial_i E_i = \sum_i \frac{\mathrm{d}E_i}{\mathrm{d}\zeta} \partial_i \zeta = \sum_i \frac{\mathrm{d}E_i}{\mathrm{d}\zeta} n_i = \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} \cdot \boldsymbol{n}, \tag{100}$$

and similarly

$$\boldsymbol{\nabla} \cdot \boldsymbol{H} = \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta} \cdot \boldsymbol{n}. \tag{101}$$

Since in vacuum $\boldsymbol{\nabla} \cdot \boldsymbol{E} = \boldsymbol{\nabla} \cdot \boldsymbol{H} = 0$, we get

$$\frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} \cdot \boldsymbol{n} = \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta} \cdot \boldsymbol{n} = 0. \tag{102}$$

This can be satisfied easily by making

$$\boldsymbol{E}(\zeta) \perp \boldsymbol{n} \quad \text{and} \quad \boldsymbol{H}(\zeta) \perp \boldsymbol{n}. \tag{103}$$

Curls are similarly evaluated. For example,

$$\boldsymbol{\nabla} \times \boldsymbol{E} = \sum_{ijk} \epsilon_{ijk} \partial_i E_j \boldsymbol{e}_k = \sum_{ijk} \epsilon_{ijk} \frac{\mathrm{d}E_j}{\mathrm{d}\zeta} \partial_i \zeta \boldsymbol{e}_k = \sum_{ijk} \epsilon_{ijk} \frac{\mathrm{d}E_j}{\mathrm{d}\zeta} n_i \boldsymbol{e}_k = \boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta}. \tag{104}$$

Similarly,

$$\boldsymbol{\nabla} \times \boldsymbol{H} = \boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta}. \tag{105}$$

The two Maxwell equations (65) and (66) become

$$\begin{aligned} \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta} &= -\boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta}, & (106) \\ \frac{\mathrm{d}\boldsymbol{H}}{\mathrm{d}\zeta} &= \boldsymbol{n} \times \frac{\mathrm{d}\boldsymbol{E}}{\mathrm{d}\zeta}. & (107) \end{aligned}$$

These two equations are automatically satisfied when $\boldsymbol{n} \cdot \boldsymbol{n} = 1$ and when equations (102) are satisfied as well.

A simple solution of these is

$$\boldsymbol{H}(\zeta) = \boldsymbol{n} \times \boldsymbol{E}(\zeta) \tag{108}$$

Fields $\boldsymbol{H}$ and $\boldsymbol{E}$ are always in-phase. Feynman [6] discusses such a general solution in section 20.1, equations (20.25).

The FORMS user specifies the signal by providing vector $\boldsymbol{n}$ and three arbitrary functions $E_x(\zeta)$, $E_y(\zeta)$ and $E_z(\zeta)$. FORMS will normalize $\boldsymbol{n}$ and will apply

$$\boldsymbol{E}_\perp(\zeta) = \boldsymbol{E}(\zeta) - (\boldsymbol{E}(\zeta) \cdot \boldsymbol{n})\boldsymbol{n} \tag{109}$$

as well as $\boldsymbol{H}(\zeta) = \boldsymbol{n} \times \boldsymbol{E}_\perp(\zeta)$ to the total field region boundary at each time step.

Here is how.

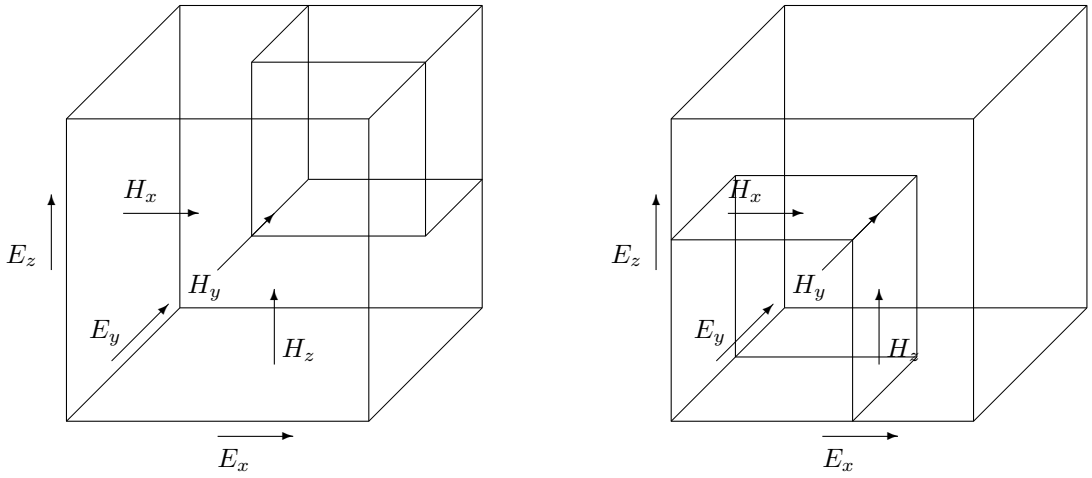


Figure 3: Location of fields in the defining corners of the injection box. The figure on the left shows the low corner, and the figure on the right shows the high corner. The smaller box within the box shows the intersection of the total field box with the corner cell.

Let us suppose, we inject the signal into a box defined by its two corners, $(i_{\rm lo}, j_{\rm lo}, k_{\rm lo})$ and $(i_{\rm hi}, j_{\rm hi}, k_{\rm hi})$. The boundary of the total field region is defined so that the cell centers, also specified by the $(i_{\rm lo}, j_{\rm lo}, k_{\rm lo})$ and $(i_{\rm hi}, j_{\rm hi}, k_{\rm hi})$ **IntVect**s *belong* to the total field box, as shown in Figure 3.

Data on the six faces of the box, as well as data on its edges and in the corners, require special handling. We are going to consider them for each face separately.[3]

west $i = i_{\rm lo}$

[3]The methodology of arriving at the equations is simple, if one thinks about it methodically. In every equation that straddles the face, all terms but one are on one side, and just one term is on the other. The trick is to bring the term that is on the wrong side onto the majority side, by either adding, if it is outside the total signal box, or subtracting, if it is inside the total signal box, the incident signal.

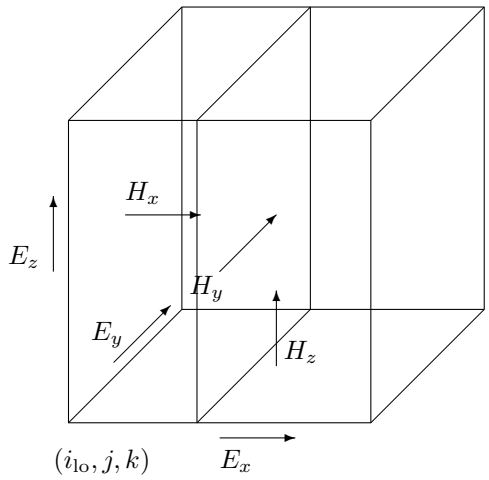


diagram $(i_{\rm lo}, j, k)$

inside $E_x(i_{\rm lo}, j, k)$, $D_x(i_{\rm lo}, j, k)$, $H_y(i_{\rm lo}, j, k)$, $B_y(i_{\rm lo}, j, k)$, $H_z(i_{\rm lo}, j, k)$, $B_z(i_{\rm lo}, j, k)$.

outside $E_y(i_{\rm lo}, j, k)$, $D_y(i_{\rm lo}, j, k)$, $E_z(i_{\rm lo}, j, k)$, $D_z(i_{\rm lo}, j, k)$ $H_x(i_{\rm lo}, j, k)$, $B_x(i_{\rm lo}, j, k)$

equations

1. The $\partial_t D_x$ equation is unaffected, because it involves H_z and H_y, which are both on the same side (but not for $k = k_{\rm lo}$ and not for $j = j_{\rm lo}$, we will have to make sure that these are handled properly, perhaps by formulas for other sides).
2. The $\partial_t D_y$ equation is affected, because one of the two H_zs it needs is inside and the other one is outside. On the other hand, both H_xs it uses are outside. So, here we have

$$\begin{aligned} D_y^{n+1}(i_{\rm lo}, j, k) &= D_y^n(i_{\rm lo}, j, k) \\ &+ \frac{\Delta t}{\Delta_g} \left(H_x^{n+1/2}(i_{\rm lo}, j, k) - H_x^{n+1/2}(i_{\rm lo}, j, k-1) - \underbrace{H_z^{n+1/2}(i_{\rm lo}, j, k)}_{\rm inside} + H_z^{n+1/2}(i_{\rm lo}-1, j, k) \right) \\ &+ \frac{\Delta t}{\Delta_g} \left\{ H_z^{n+1/2} \right\}_{\rm incident} (i_{\rm lo}, j, k). \end{aligned} \tag{110}$$

   However, we do not have to implement this correction for $k = k_{\rm lo}$, because here H_z is also outside. On the other hand, for $j = j_{\rm lo}$ and $k > k_{\rm lo}$, the correction must be implemented, because then H_z is inside. At the high end, the corrections are valid for both $k = k_{\rm hi}$ and $j = j_{\rm hi}$. In summary, the summation should run

$$\forall_{j \in [j_{\rm lo}, j_{\rm hi}], k \in [k_{\rm lo}+1, k_{\rm hi}]} \tag{111}$$

3. The $\partial_t D_z$ equation is also affected, because one of the H_ys, it uses, is inside.

$$\begin{aligned} D_z^{n+1}(i_{\rm lo}, j, k) &= D_z^n(i_{\rm lo}, j, k) \\ &+ \frac{\Delta t}{\Delta_g} \left(\underbrace{H_y^{n+1/2}(i_{\rm lo}, j, k)}_{\rm inside} - H_y^{n+1/2}(i_{\rm lo}-1, j, k) - H_x^{n+1/2}(i_{\rm lo}, j, k) + H_x^{n+1/2}(i_{\rm lo}, j-1, k) \right) \\ &- \frac{\Delta t}{\Delta_g} \{ H_y^{n+1/2} \}_{\rm incident} (i_{\rm lo}, j, k). \end{aligned} \tag{112}$$

   This correction does not apply to the $j = j_{\rm lo}$ case, because here H_y is outside. So, the ranges are

$$\forall_{j \in [j_{\rm lo}+1, j_{\rm hi}], k \in [k_{\rm lo}, k_{\rm hi}]} \tag{113}$$

4. The $\partial_t B_x$ equation is unaffected, because all it needs, E_y and E_z, lives on the same side as B_x.

5. The $\partial_t B_y$ equation is affected, because one of the E_zs it uses, the left one, is on the outside, whereas B_y is inside. So we have to correct this as follows.

$$B_y^{n+3/2}(i_{\rm lo},j,k) = B_y^{n+1/2}(i_{\rm lo},j,k) + \frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i_{\rm lo}+1,j,k) - \underbrace{E_z^{n+1}(i_{\rm lo},j,k)}_{\rm outside} - E_x^{n+1}(i_{\rm lo},j,k+1) + E_x^{n+1}(i_{\rm lo},j,k)\right) - \frac{\Delta t}{\Delta_g}\{E_z^{n+1}\}_{\rm incident}(i_{\rm lo},j,k). \quad (114)$$

This correction does not apply at $j = j_{\rm lo}$, because in this case all fields are outside. At $k = k_{\rm lo}$ there is also a correction to E_x, but this should be handled by the bottom face case (to be revisited at the end). In summary, this correction applies

$$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]} \quad (115)$$

6. The $\partial_t B_z$ equation is affected because one of the E_ys is needs is outside.

$$B_z^{n+3/2}(i_{\rm lo},j,k) = B_z^{n+1/2}(i_{\rm lo},j,k) + \frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i_{\rm lo},j+1,k) - E_x^{n+1}(i_{\rm lo},j,k) - E_y^{n+1}(i_{\rm lo}+1,j,k) + \underbrace{E_y^{n+1}(i_{\rm lo},j,k)}_{\rm outside}\right) + \frac{\Delta t}{\Delta_g}\{E_y^{n+1}\}_{\rm incident}(i_{\rm lo},j,k). \quad (116)$$

This correction does not apply at $k = k_{\rm lo}$ and at $j = j_{\rm lo}$ there should be another correction added to E_x, but this should be handled by the front face case. In summary, the correction applies

$$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]} \quad (117)$$

east $i = i_{\rm hi}$

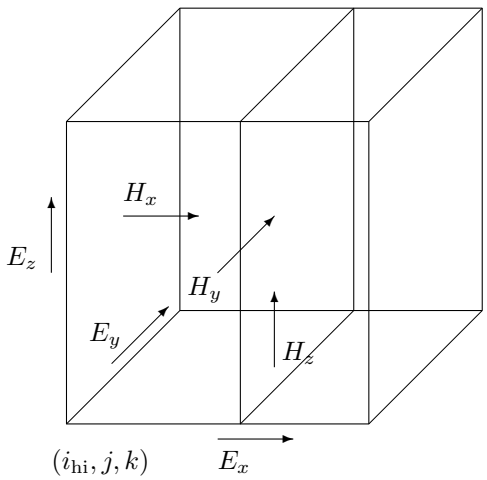


diagram $(i_{\rm hi},j,k)$

inside $E_x(i_{\rm hi},j,k)$, $E_y(i_{\rm hi},j,k)$, $E_z(i_{\rm hi},j,k)$, $H_x(i_{\rm hi},j,k)$, $H_y(i_{\rm hi},j,k)$, $H_z(i_{\rm hi},j,k)$.

outside $E_x(i_{\rm hi}+1,j,k)$, $E_y(i_{\rm hi}+1,j,k)$, $E_z(i_{\rm hi}+1,j,k)$, $H_x(i_{\rm hi}+1,j,k)$, $H_y(i_{\rm hi}+1,j,k)$, $H_z(i_{\rm hi}+1,j,k)$.

equations

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$D_y^{n+1}(i_{\rm hi}+1,j,k) = D_y^n(i_{\rm hi}+1,j,k)$

$$+\frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i_{\rm hi}+1,j,k)-H_x^{n+1/2}(i_{\rm hi}+1,j,k-1)-H_z^{n+1/2}(i_{\rm hi}+1,j,k)+\underbrace{H_z^{n+1/2}(i_{\rm hi},j,k)}_{\rm inside}\right)$$

$$-\frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i_{\rm hi},j,k), \tag{118}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i_{\rm hi}+1,j,k)=D_z^n(i_{\rm hi}+1,j,k)$$

$$+\frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i_{\rm hi}+1,j,k)-\underbrace{H_y^{n+1/2}(i_{\rm hi},j,k)}_{\rm inside}-H_x^{n+1/2}(i_{\rm hi}+1,j,k)+H_x^{n+1/2}(i_{\rm hi}+1,j-1,k)\right)$$

$$+\frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i_{\rm hi},j,k), \tag{119}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_y^{n+3/2}(i_{\rm hi},j,k)=B_y^{n+1/2}(i_{\rm hi},j,k)$$

$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{E_z^{n+1}(i_{\rm hi}+1,j,k)}_{\rm outside}-E_z^{n+1}(i_{\rm hi},j,k)-E_x^{n+1}(i_{\rm hi},j,k+1)+E_x^{n+1}(i_{\rm hi},j,k)\right)$$

$$+\frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i_{\rm hi}+1,j,k), \tag{120}$$

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i_{\rm hi},j,k)=B_z^{n+1/2}(i_{\rm hi},j,k)$$

$$+\frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i_{\rm hi},j+1,k)-E_x^{n+1}(i_{\rm hi},j,k)-\underbrace{E_y^{n+1}(i_{\rm hi}+1,j,k)}_{\rm outside}+E_y^{n+1}(i_{\rm hi},j,k)\right)$$

$$-\frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i_{\rm hi}+1,j,k). \tag{121}$$

front $j=j_{\rm lo}$

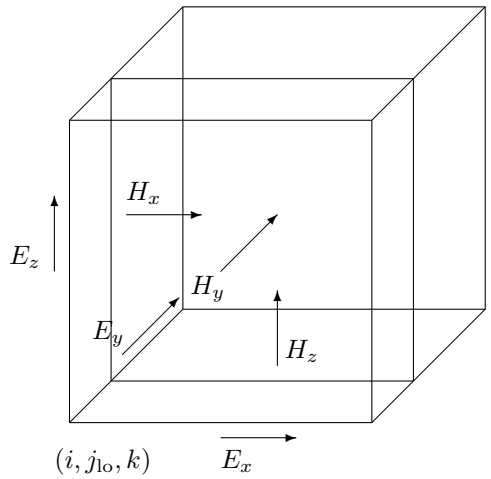


diagram $(i,j_{\rm lo},k)$

inside $E_y(i,j_{\rm lo},k)$, $D_y(i,j_{\rm lo},k)$, $H_x(i,j_{\rm lo},k)$, $B_x(i,j_{\rm lo},k)$, $H_z(i,j_{\rm lo},k)$, $B_z(i,j_{\rm lo},k)$.

outside $E_x(i,j_{\rm lo},k)$, $D_x(i,j_{\rm lo},k)$, $E_z(i,j_{\rm lo},k)$, $D_z(i,j_{\rm lo},k)$, $H_y(i,j_{\rm lo},k)$, $B_y(i,j_{\rm lo},k)$.

equations

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$D_x^{n+1}(i,j_{\rm lo},k) = D_x^n(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{H_z^{n+1/2}(i,j_{\rm lo},k)}_{\rm inside} - H_z^{n+1/2}(i,j_{\rm lo}-1,k) - H_y^{n+1/2}(i,j_{\rm lo},k) + H_y^{n+1/2}(i,j_{\rm lo},k-1)\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{122}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i,j_{\rm lo},k) = D_z^n(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i,j_{\rm lo},k) - H_y^{n+1/2}(i-1,j_{\rm lo},k) - \underbrace{H_x^{n+1/2}(i,j_{\rm lo},k)}_{\rm inside} + H_x^{n+1/2}(i,j_{\rm lo}-1,k)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{123}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_x^{n+3/2}(i,j_{\rm lo},k) = B_x^{n+1/2}(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j_{\rm lo},k+1) - E_y^{n+1}(i,j_{\rm lo},k) - E_z^{n+1}(i,j_{\rm lo}+1,k) + \underbrace{E_z^{n+1}(i,j_{\rm lo},k)}_{\rm outside}\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i,j_{\rm lo},k), \tag{124}$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i,j_{\rm lo},k) = B_z^{n+1/2}(i,j_{\rm lo},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(E_x^{n+1}(i,j_{\rm lo}+1,k) - \underbrace{E_x^{n+1}(i,j_{\rm lo},k)}_{\rm outside} - E_y^{n+1}(i+1,j_{\rm lo},k) + E_y^{n+1}(i,j_{\rm lo},k)\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j_{\rm lo},k) \tag{125}$$

back $j = j_{\rm hi}$

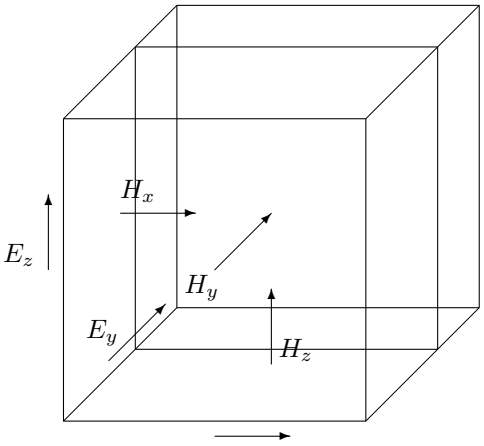


diagram $(i, j_{\rm hi}, k)$

inside $E_x(i,j_{\rm hi},k)$, $E_y(i,j_{\rm hi},k)$, $E_y(i,j_{\rm hi},k)$, $H_x(i,j_{\rm hi},k)$, $H_y(i,j_{\rm hi},k)$, $H_y(i,j_{\rm hi},k)$.

outside $E_x(i,j_{\rm hi}+1,k)$, $E_y(i,j_{\rm hi}+1,k)$, $E_y(i,j_{\rm hi}+1,k)$, $H_x(i,j_{\rm hi}+1,k)$, $H_y(i,j_{\rm hi}+1,k)$, $H_y(i,j_{\rm hi}+1,k)$.

equations

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$D_x^{n+1}(i,j_{\rm hi}+1,k) = D_x^n(i,j_{\rm hi}+1,k)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j_{\rm hi}+1,k) - \underbrace{H_z^{n+1/2}(i,j_{\rm hi},k)}_{\rm inside} - H_y^{n+1/2}(i,j_{\rm hi}+1,k) + H_y^{n+1/2}(i,j_{\rm hi}+1,k-1)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i,j_{\rm hi},k), \tag{126}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$D_z^{n+1}(i,j_{\rm hi}+1,k) = D_z^n(i,j_{\rm hi}+1,k)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_y^{n+1/2}(i,j_{\rm hi}+1,k) - H_y^{n+1/2}(i-1,j_{\rm hi}+1,k) - H_x^{n+1/2}(i,j_{\rm hi}+1,k) + \underbrace{H_x^{n+1/2}(i,j_{\rm hi},k)}_{\rm inside}\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j_{\rm hi},k), \tag{127}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$

$$B_x^{n+3/2}(i,j_{\rm hi},k) = B_x^{n+1/2}(i,j_{\rm hi},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j_{\rm hi},k+1) - E_y^{n+1}(i,j_{\rm hi},k) - \underbrace{E_z^{n+1}(i,j_{\rm hi}+1,k)}_{\rm outside} + E_z^{n+1}(i,j_{\rm hi},k)\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i,j_{\rm hi}+1,k), \tag{128}$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$B_z^{n+3/2}(i,j_{\rm hi},k) = B_z^{n+1/2}(i,j_{\rm hi},k)$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{E_x^{n+1}(i,j_{\rm hi}+1,k)}_{\rm outside} - E_x^{n+1}(i,j_{\rm hi},k) - E_y^{n+1}(i+1,j_{\rm hi},k) + E_y^{n+1}(i,j_{\rm hi},k)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j_{\rm hi}+1,k) \tag{129}$$

bottom $k = k_{\rm lo}$

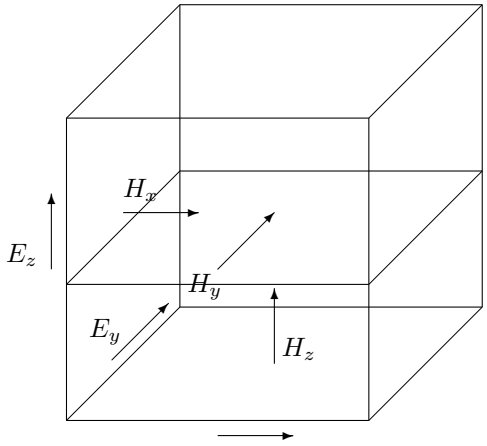


diagram $(i,j,k_{\rm lo})$

inside $E_z(i,j,k_{\rm lo})$, $D_z(i,j,k_{\rm lo})$, $H_x(i,j,k_{\rm lo})$, $B_x(i,j,k_{\rm lo})$, $H_y(i,j,k_{\rm lo})$, $B_y(i,j,k_{\rm lo})$.

outside $E_x(i,j,k_{\rm lo})$, $D_x(i,j,k_{\rm lo})$, $E_y(i,j,k_{\rm lo})$, $D_y(i,j,k_{\rm lo})$, $H_z(i,j,k_{\rm lo})$, $B_z(i,j,k_{\rm lo})$.

equations

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$

$$D_x^{n+1}(i,j,k_{\rm lo}) = D_x^n(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j,k_{\rm lo} - H_z^{n+1/2}(i,j-1,k_{\rm lo}) - \underbrace{H_y^{n+1/2}(i,j,k_{\rm lo})}_{\rm inside} + H_y^{n+1/2}(i,j,k_{\rm lo}-1)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{130}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$

$$D_y^{n+1}(i,j,k_{\rm lo}) = D_y^n(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{H_x^{n+1/2}(i,j,k_{\rm lo})}_{\rm inside} - H_x^{n+1/2}(i,j,k_{\rm lo}-1) - H_z^{n+1/2}(i,j,k_{\rm lo}) + H_z^{n+1/2}(i-1,j,k_{\rm lo})\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{131}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$

$$B_x^{n+3/2}(i,j,k_{\rm lo}) = B_x^{n+1/2}(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(E_y^{n+1}(i,j,k_{\rm lo}+1) - \underbrace{E_y^{n+1}(i,j,k_{\rm lo})}_{\rm outside} - E_z^{n+1}(i,j+1,k_{\rm lo}) + E_z^{n+1}(i,j,k_{\rm lo})\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{132}$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$

$$B_y^{n+3/2}(i,j,k_{\rm lo}) = B_y^{n+1/2}(i,j,k_{\rm lo})$$
$$+\frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i+1,j,k_{\rm lo}) - E_z^{n+1}(i,j,k_{\rm lo}) - E_x^{n+1}(i,j,k_{\rm lo}+1) + \underbrace{E_x^{n+1}(i,j,k_{\rm lo})}_{\rm outside}\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j,k_{\rm lo}). \tag{133}$$

top $k = k_{\rm hi}$

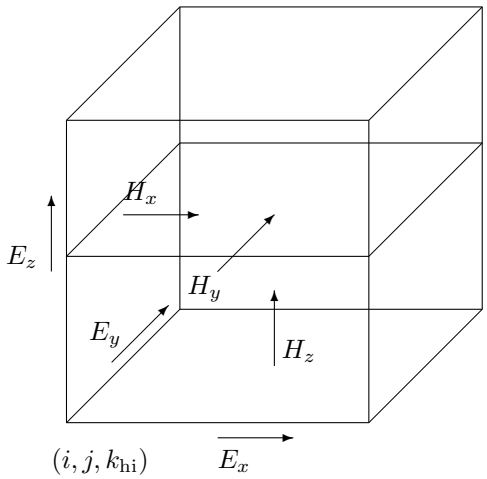


diagram $(i,j,k_{\rm hi})$

inside $E_x(i,j,k_{\rm hi})$, $E_y(i,j,k_{\rm hi})$, $E_z(i,j,k_{\rm hi})$, $H_x(i,j,k_{\rm hi})$, $H_y(i,j,k_{\rm hi})$, $H_z(i,j,k_{\rm hi})$.

outside $E_x(i,j,k_{\rm hi}+1)$, $E_y(i,j,k_{\rm hi}+1)$, $E_z(i,j,k_{\rm hi}+1)$, $H_x(i,j,k_{\rm hi}+1)$, $H_y(i,j,k_{\rm hi}+1)$, $H_z(i,j,k_{\rm hi}+1)$.

equations

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$

$$D_x^{n+1}(i,j,k_{\rm hi}+1) = D_x^n(i,j,k_{\rm hi}+1)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_z^{n+1/2}(i,j,k_{\rm hi}+1) - H_z^{n+1/2}(i,j-1,k_{\rm hi}+1) - H_y^{n+1/2}(i,j,k_{\rm hi}+1) + \underbrace{H_y^{n+1/2}(i,j,k_{\rm hi})}_{\rm inside}\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm hi}) \tag{134}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$
$$D_y^{n+1}(i,j,k_{\rm hi}+1) = D_y^n(i,j,k_{\rm hi}+1)$$
$$+\frac{\Delta t}{\Delta_g}\left(H_x^{n+1/2}(i,j,k_{\rm hi}+1) - \underbrace{H_x^{n+1/2}(i,j,k_{\rm hi})}_{\rm inside} - H_z^{n+1/2}(i,j,k_{\rm hi}+1) + H_z^{n+1/2}(i-1,j,k_{\rm hi}+1)\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm hi}) \tag{135}$$

$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$
$$B_x^{n+3/2}(i,j,k_{\rm hi}) = B_x^{n+1/2}(i,j,k_{\rm hi})$$
$$+\frac{\Delta t}{\Delta_g}\left(\underbrace{E_y^{n+1}(i,j,k_{\rm hi}+1)}_{\rm outside} - E_y^{n+1}(i,j,k_{\rm hi}) - E_z^{n+1}(i,j+1,k_{\rm hi}) + E_z^{n+1}(i,j,k_{\rm hi})\right)$$
$$+\frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i,j,k_{\rm hi}+1) \tag{136}$$

$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$
$$B_y^{n+3/2}(i,j,k_{\rm hi}) = B_y^{n+1/2}(i,j,k_{\rm hi})$$
$$+\frac{\Delta t}{\Delta_g}\left(E_z^{n+1}(i+1,j,k_{\rm hi}) - E_z^{n+1}(i,j,k_{\rm hi}) - \underbrace{E_x^{n+1}(i,j,k_{\rm hi}+1)}_{\rm outside} + E_x^{n+1}(i,j,k_{\rm hi})\right)$$
$$-\frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j,k_{\rm hi}+1) \tag{137}$$

summary Signal injection equations can be thought of as an additional correction that is put on top of normal curl computation. This leads to the following final injection formulæ:

west

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$
$$D_y^{n+1}(i_{\rm lo},j,k) \leftarrow D_y^{n+1}(i_{\rm lo},j,k) + \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\rm incident}(i_{\rm lo},j,k), \tag{138}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$
$$D_z^{n+1}(i_{\rm lo},j,k) \leftarrow D_z^{n+1}(i_{\rm lo},j,k) - \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i_{\rm lo},j,k), \tag{139}$$

$\forall_{j\in[j_{\rm lo}+1,j_{\rm hi}],k\in[k_{\rm lo},k_{\rm hi}]}$
$$B_y^{n+3/2}(i_{\rm lo},j,k) \leftarrow B_y^{n+3/2}(i_{\rm lo},j,k) - \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\rm incident}(i_{\rm lo},j,k), \tag{140}$$

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$
$$B_z^{n+3/2}(i_{\rm lo},j,k) \leftarrow B_z^{n+3/2}(i_{\rm lo},j,k) + \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i_{\rm lo},j,k). \tag{141}$$

east

$\forall_{j\in[j_{\rm lo},j_{\rm hi}],k\in[k_{\rm lo}+1,k_{\rm hi}]}$

$$D_y^{n+1}(i_{\mathrm{hi}}+1,j,k) \leftarrow D_y^{n+1}(i_{\mathrm{hi}}+1,j,k) - \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\mathrm{incident}}(i_{\mathrm{hi}},j,k), \tag{142}$$

$\forall_{j\in[j_{\mathrm{lo}}+1,j_{\mathrm{hi}}],k\in[k_{\mathrm{lo}},k_{\mathrm{hi}}]}$

$$D_z^{n+1}(i_{\mathrm{hi}}+1,j,k) \leftarrow D_z^{n+1}(i_{\mathrm{hi}}+1,j,k) + \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\mathrm{incident}}(i_{\mathrm{hi}},j,k), \tag{143}$$

$\forall_{j\in[j_{\mathrm{lo}}+1,j_{\mathrm{hi}}],k\in[k_{\mathrm{lo}},k_{\mathrm{hi}}]}$

$$B_y^{n+3/2}(i_{\mathrm{hi}},j,k) \leftarrow B_y^{n+3/2}(i_{\mathrm{hi}},j,k) + \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\mathrm{incident}}(i_{\mathrm{hi}}+1,j,k), \tag{144}$$

$\forall_{j\in[j_{\mathrm{lo}},j_{\mathrm{hi}}],k\in[k_{\mathrm{lo}}+1,k_{\mathrm{hi}}]}$

$$B_z^{n+3/2}(i_{\mathrm{hi}},j,k) \leftarrow B_z^{n+3/2}(i_{\mathrm{hi}},j,k) - \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\mathrm{incident}}(i_{\mathrm{hi}}+1,j,k). \tag{145}$$

front

$\forall_{i\in[i_{\mathrm{lo}},i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}}+1,k_{\mathrm{hi}}]}$

$$D_x^{n+1}(i,j_{\mathrm{lo}},k) \leftarrow D_x^{n+1}(i,j_{\mathrm{lo}},k) - \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\mathrm{incident}}(i,j_{\mathrm{lo}},k), \tag{146}$$

$\forall_{i\in[i_{\mathrm{lo}}+1,i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}},k_{\mathrm{hi}}]}$

$$D_z^{n+1}(i,j_{\mathrm{lo}},k) \leftarrow D_z^{n+1}(i,j_{\mathrm{lo}},k) + \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\mathrm{incident}}(i,j_{\mathrm{lo}},k), \tag{147}$$

$\forall_{i\in[i_{\mathrm{lo}}+1,i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}},k_{\mathrm{hi}}]}$

$$B_x^{n+3/2}(i,j_{\mathrm{lo}},k) \leftarrow B_x^{n+3/2}(i,j_{\mathrm{lo}},k) + \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\mathrm{incident}}(i,j_{\mathrm{lo}},k), \tag{148}$$

$\forall_{i\in[i_{\mathrm{lo}},i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}}+1,k_{\mathrm{hi}}]}$

$$B_z^{n+3/2}(i,j_{\mathrm{lo}},k) \leftarrow B_z^{n+3/2}(i,j_{\mathrm{lo}},k) - \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\mathrm{incident}}(i,j_{\mathrm{lo}},k) \tag{149}$$

back

$\forall_{i\in[i_{\mathrm{lo}},i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}}+1,k_{\mathrm{hi}}]}$

$$D_x^{n+1}(i,j_{\mathrm{hi}}+1,k) \leftarrow D_x^{n+1}(i,j_{\mathrm{hi}}+1,k) + \frac{\Delta t}{\Delta_g}\left\{H_z^{n+1/2}\right\}_{\mathrm{incident}}(i,j_{\mathrm{hi}},k), \tag{150}$$

$\forall_{i\in[i_{\mathrm{lo}}+1,i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}},k_{\mathrm{hi}}]}$

$$D_z^{n+1}(i,j_{\mathrm{hi}}+1,k) \leftarrow D_z^{n+1}(i,j_{\mathrm{hi}}+1,k) - \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\mathrm{incident}}(i,j_{\mathrm{hi}},k), \tag{151}$$

$\forall_{i\in[i_{\mathrm{lo}}+1,i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}},k_{\mathrm{hi}}]}$

$$B_x^{n+3/2}(i,j_{\mathrm{hi}},k) \leftarrow B_x^{n+3/2}(i,j_{\mathrm{hi}},k) - \frac{\Delta t}{\Delta_g}\left\{E_z^{n+1}\right\}_{\mathrm{incident}}(i,j_{\mathrm{hi}}+1,k), \tag{152}$$

$\forall_{i\in[i_{\mathrm{lo}},i_{\mathrm{hi}}],k\in[k_{\mathrm{lo}}+1,k_{\mathrm{hi}}]}$

$$B_z^{n+3/2}(i,j_{\mathrm{hi}},k) \leftarrow B_z^{n+3/2}(i,j_{\mathrm{hi}},k) + \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\mathrm{incident}}(i,j_{\mathrm{hi}}+1,k). \tag{153}$$

bottom

$\forall_{i\in[i_{\mathrm{lo}},i_{\mathrm{hi}}],j\in[j_{\mathrm{lo}}+1,j_{\mathrm{hi}}]}$

$$D_x^{n+1}(i,j,k_{\mathrm{lo}}) \leftarrow D_x^{n+1}(i,j,k_{\mathrm{lo}}) + \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\mathrm{incident}}(i,j,k_{\mathrm{lo}}), \tag{154}$$

$\forall_{i\in[i_{\mathrm{lo}}+1,i_{\mathrm{hi}}],j\in[j_{\mathrm{lo}},j_{\mathrm{hi}}]}$

$$D_y^{n+1}(i,j,k_{\mathrm{lo}}) \leftarrow D_y^{n+1}(i,j,k_{\mathrm{lo}}) - \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\mathrm{incident}}(i,j,k_{\mathrm{lo}}), \tag{155}$$

$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$$
$$B_x^{n+3/2}(i,j,k_{\rm lo}) \leftarrow B_x^{n+3/2}(i,j,k_{\rm lo}) - \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i,j,k_{\rm lo}), \tag{156}$$
$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$
$$B_y^{n+3/2}(i,j,k_{\rm lo}) \leftarrow B_y^{n+3/2}(i,j,k_{\rm lo}) + \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j,k_{\rm lo}) \tag{157}$$

top

$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$
$$D_x^{n+1}(i,j,k_{\rm hi}+1) \leftarrow D_x^{n+1}(i,j,k_{\rm hi}+1) - \frac{\Delta t}{\Delta_g}\left\{H_y^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm hi}), \tag{158}$$
$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$$
$$D_y^{n+1}(i,j,k_{\rm hi}+1) \leftarrow D_y^{n+1}(i,j,k_{\rm hi}+1) + \frac{\Delta t}{\Delta_g}\left\{H_x^{n+1/2}\right\}_{\rm incident}(i,j,k_{\rm hi}), \tag{159}$$
$$\forall_{i\in[i_{\rm lo}+1,i_{\rm hi}],j\in[j_{\rm lo},j_{\rm hi}]}$$
$$B_x^{n+3/2}(i,j,k_{\rm hi}) \leftarrow B_x^{n+3/2}(i,j,k_{\rm hi}) + \frac{\Delta t}{\Delta_g}\left\{E_y^{n+1}\right\}_{\rm incident}(i,j,k_{\rm hi}+1), \tag{160}$$
$$\forall_{i\in[i_{\rm lo},i_{\rm hi}],j\in[j_{\rm lo}+1,j_{\rm hi}]}$$
$$B_y^{n+3/2}(i,j,k_{\rm hi}) \leftarrow B_y^{n+3/2}(i,j,k_{\rm hi}) - \frac{\Delta t}{\Delta_g}\left\{E_x^{n+1}\right\}_{\rm incident}(i,j,k_{\rm hi}+1) \tag{161}$$

3.3.1 Relationship to Formulæ in Taflove and Hagness

One can compare the above formulæ to those in [19], section 5.8.1, pages 216–221.

First, we observe that the signs of the corrections are in agreement. But, should they be?

Second, we have to contend with different notation regarding indexes. Whereas ours are computational, theirs are logical, organized so that (1) they match ours for the electric field, and (2) $+1/2$ and $-1/2$ shifts are used for the magnetic field, with the $+1/2$ shifts placing the field in the same cell as our electric field. Or, to put it in other words, their field placements are reversed with respect to ours—electric fields are face centered and magnetic fields are edge centered.

Finally, we have to make a correction for the positioning of the total field region.

Whereas ours is defined in terms of cell centers, with cell centers incorporated *into* the total field box, theirs is defined in terms of cell corners (in our specific field mounting system). This affects the location of the correction field, as well as ranges of the complementary indexes. Briefly speaking, for the electric fields, they correct internal (with respect to the total field box) electric fields with external incident magnetic fields. We correct external electric fields with internal magnetic fields. For the magnetic fields it's the other way round. They correct external magnetic fields with internal electric fields and we correct internal magnetic fields with external electric fields.

Because of this one might expect opposite signs of their corrections compared to ours. But their procedure swaps the outside/inside marker too, so this is why the correction signs remain the same.

But what stays reversed, because of different box positioning, are index ranges. For example, their D_y correction on the west face runs for $j \in [j_{\rm lo}, j_{\rm hi}-1], k \in [k_{\rm lo}, k_{\rm hi}]$, which corresponds to the available placements for D_y in the total field box that stretches, in our notation, from $(i_{\rm lo}-1/2, j_{\rm lo}-1/2, k_{\rm lo}-1/2)$ to $(i_{\rm hi}+1/2, j_{\rm hi}+1/2, k_{\rm hi}+1/2)$. Index ranges for our correction to D_y on the same face, in turn, can be understood in terms of the available placements for H_z within the total field box that stretches from $(i_{\rm lo}, j_{\rm lo}, k_{\rm lo})$ to $(i_{\rm hi}, j_{\rm hi}, k_{\rm hi})$. Such a box placement will always miss $H_z(i,j,k_{\rm lo})$, but will capture $H_z(i,j,k_{\rm hi})$.

3.4 UPML Boundary

An Anisotropic Perfectly Matched Layer absorbing medium for truncation of FDTD lattices was first proposed by Stephen Gedney in 1996 [7, 19]. The technique has since been referred to as UPML, and this is what we're going to use in the code.

The method derives from Gedney's observation that a wave incident on a plane perpendicular to $\boldsymbol{e}_z$, and whose material properties are defined by

$$\hat{\boldsymbol{B}} = \boldsymbol{\mu} \cdot \hat{\boldsymbol{H}}, \tag{162}$$

$$\hat{\boldsymbol{D}} = \boldsymbol{\epsilon} \cdot \hat{\boldsymbol{E}}, \tag{163}$$

where the hats mark phasor vectors, and $\boldsymbol{\mu}$ and $\boldsymbol{\epsilon}$ are two identical tensors given by

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & \frac{1}{a} \end{pmatrix}, \tag{164}$$

where a is an arbitrary, possibly complex, number, that this wave does not reflect from the interface, regardless of its frequency and direction. So, if we make the medium of the plane absorbing, for example, by setting

$$a = 1 + \frac{\sigma}{\mathrm{i}\omega}, \tag{165}$$

and the plane sufficiently thick, we may produce a perfect absorbing truncation for an FDTD lattice.

Equation (164) will work on the top and bottom boundaries. On the front/back boundaries the $1/a$ term goes in the middle and on the west/east boundaries it goes into the upper left corner of the matrix.

In summary,

west/east boundary

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} \frac{1}{1+\frac{\sigma}{\mathrm{i}\omega}} & 0 & 0 \\ 0 & 1+\frac{\sigma}{\mathrm{i}\omega} & 0 \\ 0 & 0 & 1+\frac{\sigma}{\mathrm{i}\omega} \end{pmatrix} \tag{166}$$

front/back boundary

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} 1+\frac{\sigma}{\mathrm{i}\omega} & 0 & 0 \\ 0 & \frac{1}{1+\frac{\sigma}{\mathrm{i}\omega}} & 0 \\ 0 & 0 & 1+\frac{\sigma}{\mathrm{i}\omega} \end{pmatrix} \tag{167}$$

bottom/top boundary

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} 1+\frac{\sigma}{\mathrm{i}\omega} & 0 & 0 \\ 0 & 1+\frac{\sigma}{\mathrm{i}\omega} & 0 \\ 0 & 0 & \frac{1}{1+\frac{\sigma}{\mathrm{i}\omega}} \end{pmatrix} \tag{168}$$

The UPML layer is thus specified by two parameters only: the width of the layer and its σ, which should be chosen so that all incident signal becomes attenuated before it can reach the back side of the UPML boundary, where all fields are fixed at zero values. Gedney recommends to vary σ with depth into the UPML layer as follows

$$\sigma(z) = \frac{\sigma_{\max} \left| z - z_0 \right|^m}{d^m}, \tag{169}$$

where z_0 is the position of the UPML boundary, d is the thickness of the UPML layer, and $m = 4$ seems to give the best result following extensive numerical tests. $\sigma_{\max}$, the maximum value of $\sigma(z)$ within the UPML layer, is another phenomenological parameter, found to produce best results when set to

$$\sigma_{\max} \approx \frac{m+1}{150\pi\Delta_g} \tag{170}$$

where Δ_g is the grid constant *in meters*.

The reason for this fuss is that the behaviour of the incident signal on a *discrete* UPML medium is not exactly the same as on a continuous one, so some tweaking is required to make it work best. Taflove and Hagness discuss various other forms of $\sigma(z)$ in [19], Chapter 7.

Edges and corners of UPML region require special treatment. This is accomplished by expressing tensor $\boldsymbol{\mu} = \boldsymbol{\epsilon}$ as a product of (166), (167) and (168), which evaluates to:

$$\boldsymbol{\mu} = \boldsymbol{\epsilon} = \begin{pmatrix} s_y s_z / s_x & 0 & 0 \\ 0 & s_z s_x / s_y & 0 \\ 0 & 0 & s_x s_y / s_z \end{pmatrix}, \tag{171}$$

where

$$s_x = 1 + \frac{\sigma_x(x)}{\mathrm{i}\omega}, \tag{172}$$

$$s_y = 1 + \frac{\sigma_y(y)}{\mathrm{i}\omega}, \tag{173}$$

$$s_z = 1 + \frac{\sigma_z(z)}{\mathrm{i}\omega}, \tag{174}$$

where $\sigma_x(x)$, $\sigma_y(y)$ and $\sigma_z(z)$ are zero without[4] their *respective*, i.e., perpendicular to $\boldsymbol{e}_x$, $\boldsymbol{e}_y$, and $\boldsymbol{e}_z$, UPML regions, and given, for example, by (169) within.

The resulting Maxwell equations in the frequency domain look as follows

$$\partial_y \hat{H}_z - \partial_z \hat{H}_y = \mathrm{i}\omega \frac{s_y s_z}{s_x} \hat{E}_x, \tag{175}$$

$$\partial_z \hat{H}_x - \partial_x \hat{H}_z = \mathrm{i}\omega \frac{s_z s_x}{s_y} \hat{E}_y, \tag{176}$$

$$\partial_x \hat{H}_y - \partial_y \hat{H}_x = \mathrm{i}\omega \frac{s_x s_y}{s_z} \hat{E}_z, \tag{177}$$

$$\partial_z \hat{E}_y - \partial_y \hat{E}_z = \mathrm{i}\omega \frac{s_y s_z}{s_x} \hat{H}_x, \tag{178}$$

$$\partial_x \hat{E}_z - \partial_z \hat{E}_x = \mathrm{i}\omega \frac{s_z s_x}{s_y} \hat{H}_y, \tag{179}$$

$$\partial_y \hat{E}_x - \partial_x \hat{E}_y = \mathrm{i}\omega \frac{s_x s_y}{s_z} \hat{H}_z. \tag{180}$$

Their conversion to the time domain is non trivial, because the $1/(1 + \sigma/\mathrm{i}\omega)$ term results in a nonlinear dependence on ω. It is dispersive and characterized by a negative conductivity along the $\boldsymbol{e}_z$ direction.

Gedney (see also [19], Section 7.7.1) proposes to deal with the problem by redefining $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{B}}$ in the UPML medium as follows

$$\hat{D}_x = \frac{s_z}{s_x} \hat{E}_x, \tag{181}$$

$$\hat{D}_y = \frac{s_x}{s_y} \hat{E}_y, \tag{182}$$

$$\hat{D}_z = \frac{s_y}{s_z} \hat{E}_z, \tag{183}$$

$$\hat{B}_x = \frac{s_z}{s_x} \hat{H}_x, \tag{184}$$

$$\hat{B}_y = \frac{s_x}{s_y} \hat{H}_y, \tag{185}$$

$$\hat{B}_z = \frac{s_y}{s_z} \hat{H}_z. \tag{186}$$

This simple substitution changes equations (175) to (180) into the following simpler group of differential equations that are combined with material equations.

$$\partial_y \hat{H}_z - \partial_z \hat{H}_y = \mathrm{i}\omega s_y \hat{D}_x, \tag{187}$$

[4]The word "without" is used here in its Shakespearean sense, meaning "outside".

$$\begin{aligned}
\partial_z \hat{H}_x - \partial_x \hat{H}_z &= \mathrm{i}\omega s_z \hat{D}_y, && (188)\\
\partial_x \hat{H}_y - \partial_y \hat{H}_x &= \mathrm{i}\omega s_x \hat{D}_z, && (189)\\
\partial_z \hat{E}_y - \partial_y \hat{E}_z &= \mathrm{i}\omega s_y \hat{B}_x, && (190)\\
\partial_x \hat{E}_z - \partial_z \hat{E}_x &= \mathrm{i}\omega s_z \hat{B}_y, && (191)\\
\partial_y \hat{E}_x - \partial_x \hat{E}_y &= \mathrm{i}\omega s_x \hat{B}_z, && (192)
\end{aligned}$$

to be followed by

$$\begin{aligned}
s_x \hat{D}_x &= s_z \hat{E}_x, && (193)\\
s_y \hat{D}_y &= s_x \hat{E}_y, && (194)\\
s_z \hat{D}_z &= s_y \hat{E}_z, && (195)\\
s_x \hat{B}_x &= s_z \hat{H}_x, && (196)\\
s_y \hat{B}_y &= s_x \hat{H}_y, && (197)\\
s_z \hat{B}_z &= s_y \hat{H}_z. && (198)
\end{aligned}$$

The term

$$\mathrm{i}\omega s_k = \mathrm{i}\omega \left(1 + \frac{\sigma_k}{\mathrm{i}\omega}\right) = \mathrm{i}\omega + \sigma_k \tag{199}$$

converts into

$$\partial_t + \sigma_k, \tag{200}$$

which yields

$$\begin{aligned}
\partial_y H_z - \partial_z H_y &= \partial_t D_x + \sigma_y D_x, && (201)\\
\partial_z H_x - \partial_x H_z &= \partial_t D_y + \sigma_z D_y, && (202)\\
\partial_x H_y - \partial_y H_x &= \partial_t D_z + \sigma_x D_z, && (203)\\
\partial_z E_y - \partial_y E_z &= \partial_t B_x + \sigma_y B_x, && (204)\\
\partial_x E_z - \partial_z E_x &= \partial_t B_y + \sigma_z B_y, && (205)\\
\partial_y E_x - \partial_x E_y &= \partial_t B_z + \sigma_x B_z. && (206)
\end{aligned}$$

These look like Maxwell equations with currents, including a magnetic monopole current even. But the currents vanish everywhere with the exception of edges and corners.

Now, let us turn to the accompanying material equations. Let us consider the first one:

$$\left(1 + \frac{\sigma_x}{\mathrm{i}\omega}\right) \hat{D}_x = \left(1 + \frac{\sigma_z}{\mathrm{i}\omega}\right) \hat{E}_x \tag{207}$$

We multiply both sides by $\mathrm{i}\omega$, which yields

$$\left(\mathrm{i}\omega + \sigma_x\right) \hat{D}_x = \left(\mathrm{i}\omega + \sigma_z\right) \hat{E}_x. \tag{208}$$

Again, we convert $\mathrm{i}\omega$ to ∂_t and obtain the following set of equations

$$\begin{aligned}
\partial_t D_x + \sigma_x D_x &= \partial_t E_x + \sigma_z E_x, && (209)\\
\partial_t D_y + \sigma_y D_y &= \partial_t E_y + \sigma_x E_y, && (210)\\
\partial_t D_z + \sigma_z D_z &= \partial_t E_z + \sigma_y E_z, && (211)\\
\partial_t B_x + \sigma_x B_x &= \partial_t H_x + \sigma_z H_x, && (212)\\
\partial_t B_y + \sigma_y B_y &= \partial_t H_y + \sigma_x H_y, && (213)\\
\partial_t B_z + \sigma_z B_z &= \partial_t H_z + \sigma_y H_z. && (214)
\end{aligned}$$

Equations (201) to (206) and (209) to (214) constitute a full set of *differential* equations that describe propagation of electromagnetic fields in the UPML medium.

Now we have to discretize them. We take equations (85) through (91) as our starting point. But the current term, $\sigma_k D_j$ should be evaluated at the same point in time as the $\boldsymbol{\nabla} \times \boldsymbol{H}$ term on the other side. This we do by averaging $\boldsymbol{D}$ between $\boldsymbol{D}^{n+1}$ and $\boldsymbol{D}^n$. So, for example, we get

$$\begin{aligned}
&\frac{D_x^{n+1}(i,j,k) - D_x^n(i,j,k)}{\Delta t} + \sigma_y \frac{D_x^{n+1}(i,j,k) + D_x^n(i,j,k)}{2} \\
&= D_x^{n+1}(i,j,k)\left(\frac{1}{\Delta t} + \frac{\sigma_y}{2}\right) - D_x^n(i,j,k)\left(\frac{1}{\Delta t} - \frac{\sigma_y}{2}\right) \\
&= D_x^{n+1}(i,j,k)\frac{2+\sigma_y\Delta t}{2\Delta t} - D_x^n(i,j,k)\frac{2-\sigma_y\Delta t}{2\Delta t} \\
&= \frac{1}{\Delta_g}\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right).
\end{aligned}$$

This trick yields the following update equations for all components of $\boldsymbol{D}$ and $\boldsymbol{B}$ fields

$$\begin{aligned}
D_x^{n+1}(i,j,k) &= \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t} D_x^n(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_y\Delta t)}\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right), \quad (215) \\
D_y^{n+1}(i,j,k) &= \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t} D_y^n(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_z\Delta t)}\left(H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1) - H_z^{n+1/2}(i,j,k) + H_z^{n+1/2}(i-1,j,k)\right), \quad (216) \\
D_z^{n+1}(i,j,k) &= \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t} D_z^n(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_x\Delta t)}\left(H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k) - H_x^{n+1/2}(i,j,k) + H_x^{n+1/2}(i,j-1,k)\right), \quad (217)
\end{aligned}$$

⟨⟨ Insert equations (221) through (223) here. ⟩⟩

$$\begin{aligned}
B_x^{n+3/2}(i,j,k) &= \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t} B_x^{n+1/2}(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_y\Delta t)}\left(E_y^{n+1}(i,j,k+1) - E_y^{n+1}(i,j,k) - E_z^{n+1}(i,j+1,k) + E_z^{n+1}(i,j,k)\right), \quad (218) \\
B_y^{n+3/2}(i,j,k) &= \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t} B_y^{n+1/2}(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_z\Delta t)}\left(E_z^{n+1}(i+1,j,k) - E_z^{n+1}(i,j,k) - E_x^{n+1}(i,j,k+1) + E_x^{n+1}(i,j,k)\right), \quad (219) \\
B_z^{n+3/2}(i,j,k) &= \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t} B_z^{n+1/2}(i,j,k) \\
&+ \frac{2\Delta t}{\Delta_g\,(2+\sigma_x\Delta t)}\left(E_x^{n+1}(i,j+1,k) - E_x^{n+1}(i,j,k) - E_y^{n+1}(i+1,j,k) + E_y^{n+1}(i,j,k)\right), \quad (220)
\end{aligned}$$

⟨⟨ Insert equations (224) through (226) here. ⟩⟩

It is easy to see that these equations reduce to (85) through (91) for $\sigma_i = 0$.

Equations (215) through (220) must be solved simultaneously with material equations that result from discretization of (209) through (214). Here we also have current terms such as $\sigma_x D_x$ and $\sigma_z E_x$, and these we evaluate at half-time by averaging them between n and $n+1$. For example, equation (215) discretizes as follows

$$\begin{aligned}
&\frac{D_x^{n+1}(i,j,k) - D_x^n(i,j,k)}{\Delta t} + \sigma_x \frac{D_x^{n+1}(i,j,k) + D_x^n(i,j,k)}{2} \\
&= \frac{E_x^{n+1}(i,j,k) - E_x^n(i,j,k)}{\Delta t} + \sigma_z \frac{E_x^{n+1}(i,j,k) + E_x^n(i,j,k)}{2},
\end{aligned}$$

or

$$\frac{2+\sigma_x\Delta t}{2\Delta t} D_x^{n+1}(i,j,k) - \frac{2-\sigma_x\Delta t}{2\Delta t} D_x^n(i,j,k)$$

$$= \frac{2+\sigma_z\Delta t}{2\Delta t}E_x^{n+1}(i,j,k) - \frac{2-\sigma_z\Delta t}{2\Delta t}E_x^n(i,j,k),$$

which solves to the following equations—this time for all components of $\boldsymbol{E}$ and $\boldsymbol{H}$.

$$E_x^{n+1}(i,j,k) = \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t}E_x^n(i,j,k) + \frac{2+\sigma_x\Delta t}{2+\sigma_z\Delta t}D_x^{n+1}(i,j,k) - \frac{2-\sigma_x\Delta t}{2+\sigma_z\Delta t}D_x^n(i,j,k), \quad (221)$$

$$E_y^{n+1}(i,j,k) = \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t}E_y^n(i,j,k) + \frac{2+\sigma_y\Delta t}{2+\sigma_x\Delta t}D_y^{n+1}(i,j,k) - \frac{2-\sigma_y\Delta t}{2+\sigma_x\Delta t}D_y^n(i,j,k), \quad (222)$$

$$E_z^{n+1}(i,j,k) = \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t}E_z^n(i,j,k) + \frac{2+\sigma_z\Delta t}{2+\sigma_y\Delta t}D_z^{n+1}(i,j,k) - \frac{2-\sigma_z\Delta t}{2+\sigma_y\Delta t}D_z^n(i,j,k), \quad (223)$$

$$H_x^{n+3/2}(i,j,k) = \frac{2-\sigma_z\Delta t}{2+\sigma_z\Delta t}H_x^{n+1/2}(i,j,k) + \frac{2+\sigma_x\Delta t}{2+\sigma_z\Delta t}B_x^{n+3/2}(i,j,k) - \frac{2-\sigma_x\Delta t}{2+\sigma_z\Delta t}B_x^{n+1/2}(i,j,k), \quad (224)$$

$$H_y^{n+3/2}(i,j,k) = \frac{2-\sigma_x\Delta t}{2+\sigma_x\Delta t}H_y^{n+1/2}(i,j,k) + \frac{2+\sigma_y\Delta t}{2+\sigma_x\Delta t}B_y^{n+3/2}(i,j,k) - \frac{2-\sigma_y\Delta t}{2+\sigma_x\Delta t}B_y^{n+1/2}(i,j,k), \quad (225)$$

$$H_z^{n+3/2}(i,j,k) = \frac{2-\sigma_y\Delta t}{2+\sigma_y\Delta t}H_z^{n+1/2}(i,j,k) + \frac{2+\sigma_z\Delta t}{2+\sigma_y\Delta t}B_z^{n+3/2}(i,j,k) - \frac{2-\sigma_z\Delta t}{2+\sigma_y\Delta t}B_z^{n+1/2}(i,j,k). \quad (226)$$

This method forces us to keep $\boldsymbol{D}_{\text{old}}$ and $\boldsymbol{B}_{\text{old}}$ around, as well as the latest $\boldsymbol{D}$ and $\boldsymbol{B}$. For $\sigma_i = 0$, equation (221) through (226) reduce to

$$\begin{aligned} \boldsymbol{E}^{n+1} - \boldsymbol{E}^n &= \boldsymbol{D}^{n+1} - \boldsymbol{D}^n, \\ \boldsymbol{H}^{n+3/2} - \boldsymbol{H}^{n+1/2} &= \boldsymbol{B}^{n+3/2} - \boldsymbol{B}^{n+1/2}, \end{aligned}$$

which are satisfied automatically wherever $\boldsymbol{E} = \boldsymbol{D}$ and $\boldsymbol{H} = \boldsymbol{B}$, which is the case in vacuum.

As we have remarked above, the field update equations (215) through (220) reduce to plain Maxwell for $\sigma_k = 0$. This can be checked with **if** statements, but this would be very costly, because **if**s are expensive. It is cheaper computationally to introduce the following one-dimensional arrays:

$$C_{3x}(i) = 2 + \sigma_x(i)\Delta t, \quad (227)$$

$$C_{4x}(i) = 2 - \sigma_x(i)\Delta t, \quad (228)$$

$$C_{0x}(i) = \frac{1}{C_{3x}(i)} = \frac{1}{2+\sigma_x(i)\Delta t}, \quad (229)$$

$$C_{1x}(i) = C_{4x}(i)C_{0x}(i) = \frac{2-\sigma_x(i)\Delta t}{2+\sigma_x(i)\Delta t}, \quad (230)$$

$$C_{2x}(i) = 2\frac{\Delta t}{\Delta_g}C_{0x}(i) = \frac{2\Delta t}{\Delta_g(2+\sigma_x(i)\Delta t)}, \quad (231)$$

$$C_{3y}(j) = 2 + \sigma_y(j)\Delta t, \quad (232)$$

$$C_{4y}(j) = 2 - \sigma_y(j)\Delta t, \quad (233)$$

$$C_{0y}(j) = \frac{1}{C_{3y}(j)} = \frac{1}{2+\sigma_y(j)\Delta t}, \quad (234)$$

$$C_{1y}(j) = C_{4y}(j)C_{0y}(j) = \frac{2-\sigma_y(j)\Delta t}{2+\sigma_y(j)\Delta t}, \quad (235)$$

$$C_{2y}(j) = 2\frac{\Delta t}{\Delta_g}C_{0y}(j) = \frac{2\Delta t}{\Delta_g(2+\sigma_y(j)\Delta t)}, \quad (236)$$

$$C_{3z}(k) = 2 + \sigma_z(k)\Delta t, \quad (237)$$

$$C_{4z}(k) = 2 - \sigma_z(k)\Delta t, \quad (238)$$

$$C_{0z}(k) = \frac{1}{C_{3z}(k)} = \frac{1}{2+\sigma_z(k)\Delta t}, \quad (239)$$

$$C_{1z}(k) = C_{4z}(k)C_{0z}(k) = \frac{2-\sigma_z(k)\Delta t}{2+\sigma_z(k)\Delta t}, \quad (240)$$

$$C_{2z}(k) = 2\frac{\Delta t}{\Delta_g}C_{0z}(k) = \frac{2\Delta t}{\Delta_g(2+\sigma_z(k)\Delta t)} \quad (241)$$

In principle we may consider calculating them separately for $\boldsymbol{D}$ and for $\boldsymbol{B}$ updates, because the $\boldsymbol{D}$ and $\boldsymbol{B}$ fields are mounted differently. But, since UPMLs are not physical for starters and all we want from them is to quench any incident signal, we don't have to be so uptight about it, and we can evaluate the sigmas simply for the center of each cell.

With the help of these, equations (215) through (220) become

$$\begin{aligned}
&D_x^{n+1}(i,j,k) = C_{1y}(j)D_x^n(i,j,k) \\
&\quad +C_{2y}(j)\left(H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k) - H_y^{n+1/2}(i,j,k) + H_y^{n+1/2}(i,j,k-1)\right), \qquad (242)\\
&D_y^{n+1}(i,j,k) = C_{1z}(k)D_y^n(i,j,k) \\
&\quad +C_{2z}(k)\left(H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1) - H_z^{n+1/2}(i,j,k) + H_z^{n+1/2}(i-1,j,k)\right), \qquad (243)\\
&D_z^{n+1}(i,j,k) = C_{1x}(i)D_z^n(i,j,k) \\
&\quad +C_{2x}(i)\left(H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k) - H_x^{n+1/2}(i,j,k) + H_x^{n+1/2}(i,j-1,k)\right), \qquad (244)\\
&\langle\langle \quad \text{Insert equations (248) through (250) here.} \quad \rangle\rangle \\
&B_x^{n+3/2}(i,j,k) = C_{1y}(j)B_x^{n+1/2}(i,j,k) \\
&\quad +C_{2y}(j)\left(E_y^{n+1}(i,j,k+1) - E_y^{n+1}(i,j,k) - E_z^{n+1}(i,j+1,k) + E_z^{n+1}(i,j,k)\right), \qquad (245)\\
&B_y^{n+3/2}(i,j,k) = C_{1z}(k)B_y^{n+1/2}(i,j,k) \\
&\quad +C_{2z}(k)\left(E_z^{n+1}(i+1,j,k) - E_z^{n+1}(i,j,k) - E_x^{n+1}(i,j,k+1) + E_x^{n+1}(i,j,k)\right), \qquad (246)\\
&B_z^{n+3/2}(i,j,k) = C_{1x}(i)B_z^{n+1/2}(i,j,k) \\
&\quad +C_{2x}(i)\left(E_x^{n+1}(i,j+1,k) - E_x^{n+1}(i,j,k) - E_y^{n+1}(i+1,j,k) + E_y^{n+1}(i,j,k)\right), \qquad (247)\\
&\langle\langle \quad \text{Insert equations (251) through (253) here.} \quad \rangle\rangle
\end{aligned}$$

And equations (221) through (226) become

$$\begin{aligned}
E_x^{n+1}(i,j,k) &= C_{1z}(k)E_x^n(i,j,k) + C_{3x}(i)C_{0z}(k)D_x^{n+1}(i,j,k) - C_{4x}(i)C_{0z}(k)D_x^n(i,j,k), &(248)\\
E_y^{n+1}(i,j,k) &= C_{1x}(i)E_y^n(i,j,k) + C_{3y}(j)C_{0x}(i)D_y^{n+1}(i,j,k) - C_{4y}(j)C_{0x}(i)D_y^n(i,j,k), &(249)\\
E_z^{n+1}(i,j,k) &= C_{1y}(j)E_z^n(i,j,k) + C_{3z}(k)C_{0y}(j)D_z^{n+1}(i,j,k) - C_{4z}(k)C_{0y}(j)D_z^n(i,j,k), &(250)\\
H_x^{n+3/2}(i,j,k) &= C_{1z}(k)H_x^{n+1/2}(i,j,k) + C_{3x}(i)C_{0z}(k)B_x^{n+3/2}(i,j,k) - C_{4x}(i)C_{0z}(k)B_x^{n+1/2}(i,j,k), &(251)\\
H_y^{n+3/2}(i,j,k) &= C_{1x}(i)H_y^{n+1/2}(i,j,k) + C_{3y}(j)C_{0x}(i)B_y^{n+3/2}(i,j,k) - C_{4y}(j)C_{0x}(i)B_y^{n+1/2}(i,j,k), &(252)\\
H_z^{n+3/2}(i,j,k) &= C_{1y}(j)H_z^{n+1/2}(i,j,k) + C_{3z}(k)C_{0y}(j)B_z^{n+3/2}(i,j,k) - C_{4z}(k)C_{0y}(j)B_z^{n+1/2}(i,j,k). &(253)
\end{aligned}$$

Compared to the size of 3-dimensional data arrays of FORMS, the one dimensional arrays $C_{0x}(i)$ through $C_{4z}(k)$ are tiny. They can therefore be evaluated and stored beforehand within each MPI process separately and used for local updates of electromagnetic fields within level 0.

For higher levels, we just use plain Maxwell equations, because higher level grids are confined to the total field region and do not touch UPML boundaries.

3.5 Fourier Transforms and Fluxes

The Fourier transform of a time-dependent function $x(t)$, given by

$$\hat{x}(f) = \int_{-\infty}^{\infty} x(t) e^{-\mathrm{i}2\pi f t}\,\mathrm{d}t, \tag{254}$$

decomposes the $x(t)$ Hilbert space vector in the frequency basis, so that the reverse holds as well:

$$x(t) = \int_{-\infty}^{\infty} \hat{x}(f) e^{\mathrm{i}2\pi f t}\,\mathrm{d}f. \tag{255}$$

Here f is the plain Hertz frequency (not an ω) and t is time. When expressed in terms of $\omega = 2\pi f$ the formulas become:

$$\hat{x}(\omega) = \int_{-\infty}^{\infty} x(t) e^{-\mathrm{i}\omega t}\,\mathrm{d}t, \tag{256}$$

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{x}(\omega) e^{\mathrm{i}\omega t}\,\mathrm{d}\omega. \tag{257}$$

Every field computed by FORMS, $\boldsymbol{E}$, $\boldsymbol{D}$, $\boldsymbol{H}$, $\boldsymbol{B}$ can be decomposed into its constituent harmonics. For example,

$$\hat{\boldsymbol{E}}(\boldsymbol{r}, f) = \int_{-\infty}^{\infty} \boldsymbol{E}(\boldsymbol{r}, t) e^{-\mathrm{i}2\pi f t}\,\mathrm{d}t \approx \sum_{n=0}^{n_{\max}} \boldsymbol{E}^{(n)}(i,j,k) e^{-\mathrm{i}2\pi f(t_0+n\Delta t)}\Delta t, \tag{258}$$

where

$$\boldsymbol{r} = \boldsymbol{r}_0 + i\Delta x \boldsymbol{e}_x + j\Delta y \boldsymbol{e}_y + k\Delta z \boldsymbol{e}_z, \tag{259}$$

and where

$$\boldsymbol{E}(\boldsymbol{r}, t) = 0 \quad \text{for} \quad t < 0 \quad \text{and} \quad t > t_0 + n_{\max}\Delta t \tag{260}$$

within the computational domain covered by the range of (i, j, k).

In particular we may be interested in the flux of $\boldsymbol{E}(\boldsymbol{r}, t)$ or $\hat{\boldsymbol{E}}(\boldsymbol{r}, f)$ through some two-dimensional surface S, for example

$$\int_S \hat{\boldsymbol{E}}(\boldsymbol{r}, f)\cdot\boldsymbol{n}\,\mathrm{d}^2S = \int_S \left(\int_{-\infty}^{\infty} \boldsymbol{E}(\boldsymbol{r}, t) e^{-\mathrm{i}2\pi f t}\,\mathrm{d}t\right)\cdot\boldsymbol{n}\,\mathrm{d}^2S = \int_{-\infty}^{\infty}\left(\int_S \boldsymbol{E}(\boldsymbol{r}, t)\cdot\boldsymbol{n}\,\mathrm{d}^2S\right) e^{-\mathrm{i}2\pi f t}\,\mathrm{d}t. \tag{261}$$

This yields a simple prescription for such quantities. At every time step we would evaluate the total flux of $\boldsymbol{E}(\boldsymbol{r}, t)$ through S, which is just a single number, and then we would accumulate it for the requested frequencies f_k for $k \in \{1, \ldots, k_{\max}\}$ in an array of $k_{\max}$ entries. When the program completes, the array could be written on a file or on standard output.

In our case, computation of Fourier fields across the whole three-dimensional domain, although it can be done in principle, would be prohibitively expensive in terms of memory. We would have to replicate each of the fields to be processed for every frequency of interest. Assuming, say, 100 such frequencies and our interest in Fourier transforms of $\boldsymbol{E}$ and $\boldsymbol{H}$ throughout the whole computational domain, we would have to allocate space for 600 additional three-dimensional fields. This, clearly, can't be done. Normally, a three-dimensional computation, without the Fourier component, tends to fill the whole available memory of the node.

If only a handful of frequencies is of interest, then the incident signal can be conditioned so as to vibrate at each of the frequencies for a certain amount of time and required characteristics collected this way—this is a memory-cheap way of doing it, because no additional data stores are needed.

Sending a whistle in a Gaussian envelope is a somewhat similar procedure. In this case though Fourier transforms of fields may have to be collected for the specific frequencies of interest, because the whistle frequency changes continuously. Such a procedure may be included in the $\boldsymbol{D}$-to-$\boldsymbol{E}$ routine and its results saved on an auxiliary $\boldsymbol{S}$ fields—new $\boldsymbol{S}$ fields must be requested for each ω, and for each physical field (like $\boldsymbol{E}$ or $\boldsymbol{H}$) to be processed this way.

But if spectral response of the investigated system is required for frequency continuum throughout the whole domain, then such data can only be harvested at the post-processing stage by re-analyzing the already evolved

system for every frequency within the range—or reaching for a spectral code, which would be more appropriate for the task.

On the other hand, if users are interested in spectral dependence of some flux quantities only, then this can be computed easily and at no significant cost, neither to memory nor to CPUs, by using equation (261) above. For this reason we will restrict ourselves in what follows to this simpler situation.

3.5.1 Energy Flux

The flux of Poynting vector across a surface is a special case that requires more consideration.

Given that $\boldsymbol{P} = \boldsymbol{E} \times \boldsymbol{H}$ we can rewrite the expression for $\hat{\boldsymbol{P}}$ as follows:

$$\begin{aligned}\hat{\boldsymbol{P}}(\boldsymbol{r}, f) &= \int_{-\infty}^{\infty} \sum_{ijk} \epsilon_{ijk} E_j(\boldsymbol{r}, t) H_k(\boldsymbol{r}, t) \boldsymbol{e}_i e^{-\mathrm{i}2\pi f t}\, \mathrm{d}t \\ &= \sum_{ijk} \epsilon_{ijk} \boldsymbol{e}_i \int_{-\infty}^{\infty} E_j(\boldsymbol{r}, t) H_k(\boldsymbol{r}, t) e^{-\mathrm{i}2\pi f t}\, \mathrm{d}t \\ &= \sum_{ijk} \epsilon_{ijk} \boldsymbol{e}_i \widehat{E_j H_k}(\boldsymbol{r}, f). \end{aligned} \tag{262}$$

Alas, this is not the same as

$$\sum_{ijk} \epsilon_{ijk} \boldsymbol{e}_i \hat{E}_j(\boldsymbol{r}, f) \hat{H}_k(\boldsymbol{r}, f), \tag{263}$$

because for any two functions E and H

$$\widehat{EH} \neq \hat{E}\hat{H} \tag{264}$$

in general.

How does then $\boldsymbol{E} \times \boldsymbol{H}$ relate to $\widehat{\boldsymbol{E} \times \boldsymbol{H}}$ and to $\hat{\boldsymbol{E}} \times \hat{\boldsymbol{H}}$ and what may be the physical meaning of the latter two?

From the convolution theorem we know that

$$\widehat{E_j H_k}(\boldsymbol{r}, f) = \left(\hat{E}_j \star \hat{H}_k\right)(\boldsymbol{r}, f) = \int_{-\infty}^{\infty} \hat{E}_j\left(\boldsymbol{r}, f\right) \hat{H}_k\left(\boldsymbol{r}, \boldsymbol{f - f'}\right) \mathrm{d}f'. \tag{265}$$

This means that *every* frequency term of H_k contributes to $\widehat{E_j H_k}(\boldsymbol{r}, f)$. The power spectrum mixes all frequencies of its contributing $\boldsymbol{E}$ and $\boldsymbol{H}$.

On the other hand, Parseval's theorem, states that for any two functions E and H,

$$\int_{-\infty}^{\infty} E(t) \bar{H}(t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \hat{E}(f) \bar{\hat{H}}(f)\, \mathrm{d}f. \tag{266}$$

In other words, the scalar product of two Hilbert space vectors, $E(t)$ and $H(t)$, is invariant with respect to the change of basis.

Because in the time domain $H_k(\boldsymbol{r}, t)$ is purely real, $\bar{H}_k(\boldsymbol{r}, t) = H_k(\boldsymbol{r}, t)$ and

$$\int_{-\infty}^{\infty} E_j(\boldsymbol{r}, t) H_k(\boldsymbol{r}, t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \hat{E}_j(f) \bar{\hat{H}}_k(f)\, \mathrm{d}f. \tag{267}$$

Multiplying both sides by $\epsilon_{ijk}\boldsymbol{e}_i$ and adding for all values of i, j, and k yields the cross product,

$$\int_{-\infty}^{\infty} \boldsymbol{P}(\boldsymbol{r}, t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \boldsymbol{E}(\boldsymbol{r}, t) \times \boldsymbol{H}(\boldsymbol{r}, t)\, \mathrm{d}t = \int_{-\infty}^{\infty} \hat{\boldsymbol{E}}(\boldsymbol{r}, f) \times \bar{\hat{\boldsymbol{H}}}(\boldsymbol{r}, f)\, \mathrm{d}f. \tag{268}$$

We can therefore employ the $\hat{\boldsymbol{E}}$ and $\hat{\boldsymbol{H}}$ fields to evaluate the total energy flux across the surface over the whole time of the simulation, although, in our case, this presents no advantage, because we already have $\boldsymbol{E}$ and $\boldsymbol{H}$, so we can evaluate this quantity directly.

But it turns out that $\hat{\boldsymbol{E}} \times \bar{\hat{\boldsymbol{H}}}$ has a local meaning, as well, that can be arrived at as follows. Let

$$\begin{aligned} \mathcal{E}(\omega) &= \hat{\boldsymbol{E}}(\omega)e^{-\mathrm{i}\omega t} = (\boldsymbol{E}_c(\omega) + \mathrm{i}\boldsymbol{E}_s(\omega))\, e^{-\mathrm{i}\omega t}, \quad \text{and} \\ \mathcal{H}(\omega) &= \hat{\boldsymbol{H}}(\omega)e^{-\mathrm{i}\omega t} = (\boldsymbol{H}_c(\omega) + \mathrm{i}\boldsymbol{H}_s(\omega))\, e^{-\mathrm{i}\omega t}, \end{aligned} \tag{269}$$

where $\mathcal{E}$ and $\mathcal{H}$ are complex phasors. Let us expand the above for $\mathcal{E}(\omega)$:

$$\begin{aligned} \mathcal{E}(\omega) &= (\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s)\,(\cos\omega t - \mathrm{i}\sin\omega t) \\ &= (\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t) + \mathrm{i}\,(\boldsymbol{E}_s \cos\omega t - \boldsymbol{E}_c \sin\omega t)\,. \end{aligned} \tag{270}$$

First, this explains our notation. The c subscript, which here marks the real component of $\hat{\boldsymbol{E}}(\omega)$ also denotes the cosine component of the real $\boldsymbol{E}(t)$, and the s subscript, which here marks the imaginary component of $\hat{\boldsymbol{E}}(\omega)$ also denotes the sine component of the real $\boldsymbol{E}(t)$. Next, $\Re\mathcal{E}(\omega) = \boldsymbol{E}_\omega(t)$, the real component of $\boldsymbol{E}(t)$ that vibrates with the frequency of ω.

Taking $\mathcal{E}(\omega) \times \bar{\mathcal{H}}(\omega)$ kills the $\exp(-\mathrm{i}\omega)$ factor, so this is the same as $\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)$, and what remains is

$$\begin{aligned} \hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega) &= (\boldsymbol{E}_c + \mathrm{i}\boldsymbol{E}_s) \times (\boldsymbol{H}_c - \mathrm{i}\boldsymbol{H}_s) \\ &= (\boldsymbol{E}_c \times \boldsymbol{H}_c + \boldsymbol{E}_s \times \boldsymbol{H}_s) + \mathrm{i}\,(\boldsymbol{E}_s \times \boldsymbol{H}_c - \boldsymbol{E}_c \times \boldsymbol{H}_s) \end{aligned} \tag{271}$$

Thus,

$$\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right) = \boldsymbol{E}_c \times \boldsymbol{H}_c + \boldsymbol{E}_s \times \boldsymbol{H}_s. \tag{272}$$

How does this relate to $\boldsymbol{E}_\omega \times \boldsymbol{H}_\omega$? This is not a part of $\boldsymbol{P}$ that is due to ω components of $\boldsymbol{E}$ and $\boldsymbol{H}$, because $\boldsymbol{P}$ mixes all frequencies, as we have observed above. This expression describes what $\boldsymbol{P}(\omega)$ *would be* if the system was illuminated by monochromatic light of this frequency. In this case $\boldsymbol{E}_\omega$ and $\boldsymbol{H}_\omega$ would be the only frequency components involved.

Let us observe that

$$\begin{aligned} \boldsymbol{E}_\omega \times \boldsymbol{H}_\omega &= (\boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t) \times (\boldsymbol{H}_c \cos\omega t + \boldsymbol{H}_s \sin\omega t) \\ &= \boldsymbol{E}_c \times \boldsymbol{H}_c \cos^2\omega t + \boldsymbol{E}_s \times \boldsymbol{H}_s \sin^2\omega t + (\boldsymbol{E}_c \times \boldsymbol{H}_s + \boldsymbol{E}_s \times \boldsymbol{H}_c)\cos\omega t \sin\omega t. \end{aligned} \tag{273}$$

Integrating this over time between $\omega t = 0$ and $\omega t = 2\pi$ kills the $\cos\omega t \sin\omega t$ component, and converts both the $\sin^2\omega t$ and $\cos^2\omega t$ components to $T/2$, where $T = 2\pi/\omega$ is the period of vibration. In summary

$$\frac{1}{T}\int_0^T \boldsymbol{E}_\omega \times \boldsymbol{H}_\omega \,\mathrm{d}t = \frac{1}{2}\left(\boldsymbol{E}_c \times \boldsymbol{H}_c + \boldsymbol{E}_s \times \boldsymbol{H}_s\right) = \frac{1}{2}\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right). \tag{274}$$

And so we find that the integral on the right hand side of equation (268) can be understood as *twice* the sum of "monochromatic" contributions given by equation (274)—and with the $\Re$ operator dropped. Why twice? This is because the monochromatic contributions alone do not account for the frequency mixing terms, which add to the total energy. Parity properties of $\hat{\boldsymbol{E}}$ and $\hat{\boldsymbol{H}}$ ensure that the integral in (268) is real.

Evaluation of the flux of $\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right)/2$ across a surface is going to be costly, because we will have to accumulate Fourier transforms $\hat{\boldsymbol{E}}(\omega)$ and $\hat{\boldsymbol{H}}(\omega)$ at every plaquette of the surface and for every ω of interest, and store them as the computation unfolds. Only after we have processed the whole signal, can we evaluate $\Re\left(\hat{\boldsymbol{E}}(\omega) \times \bar{\hat{\boldsymbol{H}}}(\omega)\right)/2$ for each plaquette, and for each frequency of interest, using the accumulated quantities, and sum the local fluxes over the whole surface.

So, the question arises, if a procedure can be found that can accumulate the total flux for a given frequency in a more economic way and without using so much memory. The simplest way is to inject monochromatic light into the system and evaluate the flux of $\boldsymbol{P}$ following equation (261). If there are no non-linear materials in the system, there should be no frequency-shifting and the obtained result, averaged over a vibration period (or more, for better accuracy) should be exactly as given by equation (274).

3.5.2 Evaluation of Fluxes

The quantity we want to evaluate is

$$\int_S \boldsymbol{P}(\boldsymbol{r},t)\cdot\boldsymbol{n}\,\mathrm{d}^2S \tag{275}$$

for an arbitrary two-dimensional surface S given by parametric equations

$$\begin{aligned} x &= x_S(u,v), &(276)\\ y &= y_S(u,v), &(277)\\ z &= z_S(u,v), &(278)\end{aligned}$$

where $u\in[u_{\min},u_{\max}]$ and $v\in[v_{\min},v_{\max}]$ are the parameters. For example,

hemisphere

$$\begin{aligned} x(\phi,\theta) &= R\cos\phi\sin\theta, &(279)\\ y(\phi,\theta) &= R\sin\phi\sin\theta, &(280)\\ z(\phi,\theta) &= R\cos\theta, &(281)\end{aligned}$$

where the parameters are ϕ and θ (instead of u and v), and $\phi\in[-\pi/2,+\pi/2]$ whereas $\theta\in[0,\pi]$.

plane

$$\begin{aligned} x(u,v) &= x_0, &(282)\\ y(u,v) &= u, &(283)\\ z(u,v) &= v, &(284)\end{aligned}$$

where $u\in[y_{\min},y_{\max}]$ and $v\in[z_{\min},z_{\max}]$.

The first step in evaluating the flux integral (275) is to figure out the way to calculate $\boldsymbol{n}\,\mathrm{d}^2S$ in terms of u and v. On changing u or v by $\mathrm{d}u$ or $\mathrm{d}v$ respectively, the corresponding change in x, y and z is expressed in terms of two infinitesimal vectors,

$$\begin{pmatrix}\partial x(u,v)/\partial u\\ \partial y(u,v)/\partial u\\ \partial z(u,v)/\partial u\end{pmatrix}\mathrm{d}u \quad\text{and}\quad \begin{pmatrix}\partial x(u,v)/\partial v\\ \partial y(u,v)/\partial v\\ \partial z(u,v)/\partial v\end{pmatrix}\mathrm{d}v. \tag{285}$$

The surface area of a parallelogram subtended by the two vectors is equal to the length of their cross product, whereas the latter's direction is perpendicular to the parallelogram. Hence

$$\boldsymbol{n}\,\mathrm{d}^2S=\begin{pmatrix}\partial_u x(u,v)\\ \partial_u y(u,v)\\ \partial_u z(u,v)\end{pmatrix}\mathrm{d}u\times\begin{pmatrix}\partial_v x(u,v)\\ \partial_v y(u,v)\\ \partial_v z(u,v)\end{pmatrix}\mathrm{d}v=\begin{pmatrix}\partial_u y(u,v)\partial_v z(u,v)-\partial_u z(u,v)\partial_v y(u,v)\\ \partial_u z(u,v)\partial_v x(u,v)-\partial_u x(u,v)\partial_v z(u,v)\\ \partial_u x(u,v)\partial_v y(u,v)-\partial_u y(u,v)\partial_v x(u,v)\end{pmatrix}\mathrm{d}u\,\mathrm{d}v, \tag{286}$$

where $\partial_u=\partial/\partial u$ and $\partial_v=\partial/\partial v$.

The flux then is

$$\begin{aligned}&\int_S \boldsymbol{P}(\boldsymbol{r},t)\cdot\boldsymbol{n}\,\mathrm{d}^2S\\ &=\int_{v_{\min}}^{v_{\max}}\int_{u_{\min}}^{u_{\max}}\begin{pmatrix}P_x(x(u,v),y(u,v),z(u,v),t)\\ P_y(x(u,v),y(u,v),z(u,v),t)\\ P_z(x(u,v),y(u,v),z(u,v),t)\end{pmatrix}\cdot\begin{pmatrix}\partial_u y(u,v)\partial_v z(u,v)-\partial_u z(u,v)\partial_v y(u,v)\\ \partial_u z(u,v)\partial_v x(u,v)-\partial_u x(u,v)\partial_v z(u,v)\\ \partial_u x(u,v)\partial_v y(u,v)-\partial_u y(u,v)\partial_v x(u,v)\end{pmatrix}\mathrm{d}u\,\mathrm{d}v.\end{aligned} \tag{287}$$

We can check the formula by evaluating, for example,

$$\int_S \boldsymbol{n}\cdot\boldsymbol{n}\,\mathrm{d}^2S=\mathcal{A}(S), \tag{288}$$

which is also a flux—of a kind—and which should return the area of surface S. We'll do it for the hemisphere and for the plane, as defined by equations (276–278) and (279–281) above.

hemisphere The parameterizing variables here are ϕ and θ. The two infinitesimal vectors that correspond to infinitesimal changes in ϕ and θ are

$$\begin{pmatrix} -R\sin\phi\sin\theta \\ R\cos\phi\sin\theta \\ 0 \end{pmatrix} \mathrm{d}\phi \quad \text{and} \quad \begin{pmatrix} R\cos\phi\cos\theta \\ R\sin\phi\cos\theta \\ -R\sin\theta \end{pmatrix} \mathrm{d}\theta. \tag{289}$$

Their cross product is

$$\begin{pmatrix} -R^2\cos\phi\sin^2\theta \\ -R^2\sin\phi\sin^2\theta \\ -R^2\sin\theta\cos\theta \end{pmatrix} \mathrm{d}\phi\,\mathrm{d}\theta. \tag{290}$$

Let us observe that, because of how $\mathrm{d}\phi$ and $\mathrm{d}\theta$ point, the latter points downward, their cross product points inward, into the ball, not outward.

The length of the vector (and therefore also the parallelogram's area) is

$$R^2\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta = R\sin\theta\,\mathrm{d}\phi\,R\,\mathrm{d}\theta. \tag{291}$$

Vector $\boldsymbol{n}$ pointing in the same direction as vector (290) is

$$\frac{1}{R^2\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta} \begin{pmatrix} -R^2\cos\phi\sin^2\theta \\ -R^2\sin\phi\sin^2\theta \\ -R^2\sin\theta\cos\theta \end{pmatrix} \mathrm{d}\phi\,\mathrm{d}\theta = \begin{pmatrix} -\cos\phi\sin\theta \\ -\sin\phi\sin\theta \\ -\cos\theta \end{pmatrix}. \tag{292}$$

Its dot product with vector (290) is

$$\begin{pmatrix} -\cos\phi\sin\theta \\ -\sin\phi\sin\theta \\ -\cos\theta \end{pmatrix} \cdot \begin{pmatrix} -R^2\cos\phi\sin^2\theta \\ -R^2\sin\phi\sin^2\theta \\ -R^2\sin\theta\cos\theta \end{pmatrix} \mathrm{d}\phi\,\mathrm{d}\theta = R^2\sin\theta\,\mathrm{d}\phi\,\mathrm{d}\theta, \tag{293}$$

and the flux integral (for $\phi \in [-\pi/2, \pi/2]$ and $\theta \in [0, \pi]$) is

$$\int_0^{\pi}\int_{-\pi/2}^{\pi/2} R\,\mathrm{d}\phi\,R\sin\theta\mathrm{d}\theta = \int_1^{-1} R\pi R(-\mathrm{d}\cos\theta) = 2\pi R^2. \tag{294}$$

plane The two infinitesimal vectors that correspond to infinitesimal changes in u and v are

$$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \mathrm{d}u \quad \text{and} \quad \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \mathrm{d}v. \tag{295}$$

Their cross product is

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \mathrm{d}u\,\mathrm{d}v. \tag{296}$$

The length of the vector (and therefore also the parallelogram's area) is

$$\mathrm{d}u\,\mathrm{d}v. \tag{297}$$

Vector $\boldsymbol{n}$ pointing in the same direction as vector (296) is

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{298}$$

Its dot product with vector (296) is

$$\mathrm{d}u\,\mathrm{d}v. \tag{299}$$

and the flux integral (for $u \in [y_{\min}, y_{\max}]$ and $v \in [z_{\min}, z_{\max}]$ is

$$\int_{z_{\min}}^{z_{\max}}\int_{y_{\min}}^{y_{\max}} \mathrm{d}u\,\mathrm{d}v = (y_{\max} - y_{\min})\cdot(z_{\max} - z_{\min}). \tag{300}$$

The main philosophy of FORMS is to let the user provide required model specifications functionally, in Scheme or in C, while the code attends to the basic machinery of (possibly multigrid) FDTD, that includes signal injection (but not signal definition) and scattered field absorption on the boundary, within its internal computational engine. The user specifies media types and their distribution, as well as the injected signal, by providing lambdas.

Similarly any auxiliary computation on the FDTD generated data is to be specified by the user. Our only task here is to figure out what it is that FORMS has to provide to make this possible.

To this end we are now going to express the computation of the flux and the following computation of the flux Fourier transform for an arbitrary number of frequencies in Scheme.

The following listing illustrates the flux computation through a hemispherical surface.

```
;; $Id: Flux_1.scm,v 1.1 2008/04/30 17:43:37 gustav Exp $
;;
;; A simple shot at flux computation through a hemisphere.
;; Functions P_x_fun, P_y_fun and P_z_fun define a sample
;; vector field.
;;

(define P_x_fun
  (lambda (x y z)
    (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
      (/ x length))))

(define P_y_fun
  (lambda (x y z)
    (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
      (/ y length))))

(define P_z_fun
  (lambda (x y z)
    (let ((length (sqrt (+ (* x x) (* y y) (* z z)))))
      (/ z length))))

(define compflux
  (lambda (r_0 n_phi n_theta)
    (let* ((r 50)
           (x_0 (f64vector-ref r_0 0))
           (y_0 (f64vector-ref r_0 1))
           (z_0 (f64vector-ref r_0 2))
           (pi (* 2 (asin 1)))
           (phi_min (* -0.5 pi))
           (phi_max (* 0.5 pi))
           (theta_min 0)
           (theta_max pi)
           (d_phi (/ (- phi_max phi_min) n_phi))
           (d_theta (/ (- theta_max theta_min) n_theta))
           (flux 0))
      (do ((phi (+ phi_min (/ d_phi 2)) (+ phi d_phi)))
          ((> phi phi_max))
        (do ((theta (+ theta_min (/ d_theta 2)) (+ theta d_theta)))
            ((> theta theta_max))
          (let* ((sin_theta (sin theta))
                 (n_x (* (cos phi) sin_theta))
                 (n_y (* (sin phi) sin_theta))
                 (n_z (* (cos theta)))
                 (x (+ x_0 (* r n_x)))
                 (y (+ y_0 (* r n_y)))
                 (z (+ z_0 (* r n_z)))
                 (d2s (* r r sin_theta d_phi d_theta))
```

```
                (P_x (P_x_fun x y z))
                (P_y (P_y_fun x y z))
                (P_z (P_z_fun x y z))
                (P_dot_n (+ (* P_x n_x) (* P_y n_y) (* P_z n_z))))
            (set! flux (+ flux (* d2s P_dot_n))))))
    flux)))
```

The function that computes the flux is `compflux`. It takes three arguments, the center of the hemisphere $\boldsymbol{r}_0$, which is an `f64vector` of three doubles, number of segments in the ϕ direction n_ϕ, and number of segments in the θ direction n_θ. Within its main `let*` clause, angle ϕ varies from $-\pi/2$ to $\pi/2$ and θ varies from 0 to π. Within the double `do` loop of the function we position ourselves *in the center* of each $\mathbf{d}\phi \wedge \mathbf{d}\theta$ plaquette—not in the corner—and evaluate $\boldsymbol{n}$ and d^2S for this location. The radius of the hemisphere is 50. Then we find its x, y and z coordinates, and call three functions that give us $\boldsymbol{P}(x, y, z)$ (time is not used in the calculation explicitly). We take the dot product of $\boldsymbol{P}$ with $\boldsymbol{n}$, multiply by the corresponding d^2S and accumulate the result on `flux`.

We can test the function as follows. Functions `P_x_fun`, `P_y_fun` and `P_z_fun` define the $\boldsymbol{n}$ field for sphere centered at $\boldsymbol{r}_0 = \mathbf{0}$. So, the result should be the surface area of the hemisphere, that is, $2\pi R^2$. And this is indeed what we get, as the following illustrates:

```
guile> (load "Flux_1.scm")
guile> (compflux (f64vector 0 0 0) 180 180)
15708.1626413537
guile> (define pi (* 2 (asin 1)))
guile> (* 2 pi 50 50)
15707.963267949
guile> (/ (- 15708.1626413537 15707.963267949) 15707.963267949)
1.26925051516877e-5
guile>
```

We find that the function evaluates the flux with 1×10^{-5} accuracy for 180 divisions (one degree each) in both directions. If we were to divide the hemisphere into smaller plaquettes, for example, a $1/8^{\text{th}}$ of a degree in each direction, the resulting accuracy would improve to 2×10^{-7}:

```
guile> (compflux (f64vector 0 0 0) 1440 1440)
15707.9663831536
guile> (/ (- 15707.9663831536 15707.963267949) 15707.963267949)
1.9832008441403e-7
guile>
```

This is a computation that users can express themselves on the FORMS input file assuming that

1. the FORMS code invokes it after every iteration;
2. the FORMS code provides functions for field evaluation at a given position and time for all fields it handles, including the user defined auxiliary fields $\boldsymbol{S}$.

The above will work for the sequential version of the code. For the parallel version, we need to add functions for data reduction on the Scheme level and define the behaviour of field evaluation functions in case the data is not available on the node. We will return to this issue later.

The flux computation presented here is not expensive. For example, to evaluate the above flux on $720 \times 720 = 518,400$ plaquettes takes...

```
guile> (current-time) (compflux (f64vector 0 0 0) 720 720) (current-time)
1209589621
15707.975728672
1209589632
guile> (- 1209589632 1209589621)
11
guile>
```

eleven seconds of unoptimized and interpreted Scheme code. But normally, we would not do it on such a large number of plaquettes, because we wouldn't have this many cells in our three-dimensional system to deliver the required resolution anyway. A $512 \times 512 \times 512$ cell job would normally run on 64 nodes, handling a cube of $128 \times 128 \times 128 = 2,097,152$ cells per node. A smaller job could work with a $256 \times 256 \times 256$ grid on eight nodes. So, a 128×128 evaluation of the flux is more likely and this takes less than a second. If the computation becomes more involved, a C routine can be substituted in place of the interpreted Scheme code. Also, we would not normally compute the flux through the whole hemisphere. A smaller spherical cap, or a small portion of a flat plane, are more likely to be used in this context, because the purpose of such evaluations is to simulate measurements.

The flux Fourier transform can be evaluated easily by making use of the Scheme's complex number arithmetic. And so, having evaluated the flux itself, we would multiply it by $\exp(-\mathrm{i}2\pi f t)\,\mathrm{d}t$ and add to the sum of such terms evaluated so far. The corresponding Scheme code may look as follows:

```
;; Forms input file
;;
;; $Id: FourierFlux.scm,v 1.4 2008/06/08 19:40:30 gustav Exp $
;;
;; Inject a plane wave signal into the total field region. Test the
;; solution by evaluating the flux of E x H across two obliquely
;; oriented circular detectors at each time step and accumulating
;; the total power absorbed.
;;
;; All functions are openly defined in Scheme.
;;
;; This version of the input file is for parallel environment: Scheme opens
;; separate log files, numbered by (proc-id).
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Auxiliary modules
;;
(use-modules (ice-9 format))
;;
;; The Grid group
;;
;; Level 0 parameters
;;
(define forms.grid.level0.cells '(128 128 128))
(define forms.grid.level0.origin '(.0 .0 .0))
(define forms.grid.level0.delta 1.0)
(define forms.grid.max_box_size 128)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The PML group
;;
(define forms.pml.lo '( 10.0 10.0 10.0))
(define forms.pml.hi '(117.0 117.0 117.0))
(define forms.pml.sigma_max 3.5)
(define forms.pml.print_arrays 0)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Signal group
;;
(define forms.signal.garbage_collect 0)
(define forms.signal.lo '( 18.0 18.0 18.0))
(define forms.signal.hi '(109.0 109.0 109.0))
;;
```

```
;; The signal propagates in the diagonal direction
;;
(define forms.signal.direction '( 0.0 0.0 1.0))
;;
;; No E field perpendicularization is needed on evaluation.
;;
(define forms.signal.normalized 1)
;;
;; My constants
;;
(define pi (* 2.0 (asin 1.0)))
;;
(define wavelength 30.0)
(define half-width 110.0)
(define delay 120.0)
(define slope 0.4)
(define t_begin 0.0)
(define t_end 360.0)
;;
(define window
  (lambda (zeta)
    (* 0.25
       (+ 1.0 (tanh (* slope (+ zeta half-width delay))))
       (- 1.0 (tanh (* slope (+ zeta (- half-width) delay)))))))
;;
(define forms.signal.ex_lambda
  (lambda (zeta)
    (* (window zeta)
       (sin (/ (* 2.0 pi (+ zeta delay)) wavelength)))))
;;
(define forms.signal.ey_lambda
  (lambda (zeta)
    0.0))
;;
(define forms.signal.ez_lambda
  (lambda (zeta)
    0.0))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Media group
;;
;; We do not distribute any media for this run, so
;; forms.media.distribution.lambda is undefined. This will
;; trigger a warning, but the program will run.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Iterate group
;;
;; Here we define stride only. The other three parameters are
;; calculated from our requests for t_begin and t_end in the
;; signal group.
;;
(define forms.iterate.stride 4)
;;
(define number_of_steps (* (/ (- t_end t_begin) forms.grid.level0.delta)
                           forms.iterate.stride))
;;
```

```
(define forms.iterate.number_of_steps number_of_steps)
(define forms.iterate.image_frequency number_of_steps)
(define forms.iterate.t0 t_begin)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Postprocessing group
;;
;; Elementary vector algebra
;;
(define vector-norm
  (lambda (n)
    (do ((length (vector-length n))
         (sum 0)
         (count 0 (1+ count)))
        ((>= count length) (sqrt sum))
      (set! sum (+ sum (expt (vector-ref n count) 2))))))
;;
(define vector-normalize
  (lambda (n)
    (let ((length (vector-length n)))
      (do ((norm (vector-norm n))
           (v (make-vector length 0))
           (count 0 (1+ count)))
          ((>= count length) v)
        (vector-set! v count (/ (vector-ref n count) norm))))))
;;
(define vector-normalized?
  (lambda (n)
    (= (vector-norm n) 1)))
;;
(define first-perpendicular
  (lambda (n)
    (if (= (vector-length n) 3)
        (if (vector-normalized? n)
            (let ((nx (vector-ref n 0))
                  (ny (vector-ref n 1))
                  (nz (vector-ref n 2)))
              (if (and (= nx 0) (= ny 0))
                  (vector 0 nz 0)
                  (let ((denom (sqrt (+ (* nx nx) (* ny ny)))))
                    (vector (/ ny denom) (/ (- nx) denom) 0))))
            (first-perpendicular (vector-normalize n)))
        #f)))
;;
(define cross-product
  (lambda (u v)
    (if (and (= (vector-length u) 3) (= (vector-length v) 3))
        (let ((ux (vector-ref u 0))
              (uy (vector-ref u 1))
              (uz (vector-ref u 2))
              (vx (vector-ref v 0))
              (vy (vector-ref v 1))
              (vz (vector-ref v 2)))
          (vector
           (- (* uy vz) (* uz vy))
           (- (* uz vx) (* ux vz))
           (- (* ux vy) (* uy vx))))
        #f)))
```

```
;;
(define dot-product
  (lambda (u v)
    (let ((length-u (vector-length u))
          (length-v (vector-length v)))
      (if (= length-u length-v)
          (do ((sum 0)
               (count 0 (1+ count)))
              ((>= count length-u) sum)
            (set! sum (+ sum (* (vector-ref u count) (vector-ref v count)))))
          #f))))
;;
(define vector-add
  (lambda (u v)
    (if (and (= (vector-length u) 3) (= (vector-length v) 3))
        (let ((ux (vector-ref u 0))
              (uy (vector-ref u 1))
              (uz (vector-ref u 2))
              (vx (vector-ref v 0))
              (vy (vector-ref v 1))
              (vz (vector-ref v 2)))
          (vector
           (+ ux vx)
           (+ uy vy)
           (+ uz vz)))
        #f)))
;;
(define vector-mult
  (lambda (a u)
    (if (and (complex? a) (vector? u) (= (vector-length u) 3))
        (let ((ux (vector-ref u 0))
              (uy (vector-ref u 1))
              (uz (vector-ref u 2)))
          (vector
           (* a ux)
           (* a uy)
           (* a uz)))
        #f)))
;;
(define vector-div
  (lambda (u a)
    (if (and (complex? a) (not (= a 0)))
        (vector-mult (/ 1.0 a) u)
        #f)))
;;
(define complex-conjugate
  (lambda (z)
    (make-rectangular (real-part z) (- (imag-part z)))))
;;
(define vector-complex-conjugate
  (lambda (v)
    (vector
     (complex-conjugate (vector-ref v 0))
     (complex-conjugate (vector-ref v 1))
     (complex-conjugate (vector-ref v 2)))))
;;
(define vector-real-part
  (lambda (v)
    (vector
```

```
      (real-part (vector-ref v 0))
      (real-part (vector-ref v 1))
      (real-part (vector-ref v 2)))))
;;
(define vector-imag-part
  (lambda (v)
    (vector
      (imag-part (vector-ref v 0))
      (imag-part (vector-ref v 1))
      (imag-part (vector-ref v 2)))))
;;
;; Scan your parallel environment
;;
(define my-rank (proc-id))
(define root-rank (unique-proc))
(define pool-size (num-proc))
(define my-log (open-file (format #f "scm_out.~a" my-rank) "a"))
(format my-log "Guile log:~%")
(format my-log " process number: ~a~%" my-rank)
(format my-log " pool size: ~a~%" pool-size)
;;
;; Define the detectors
;;
(define center_1 #(40.0 64.0 40.0))
(define center_2 #(88.0 64.0 40.0))
(define normal_1 #(-1 0 1))
(define normal_2 #(1 0 1))
(define radius 10.0)
(define n_points 40)
(define dx (/ (* 2.0 radius) n_points))
(define dy dx)
(define d2s (* dx dy))
(define x-lo (- (- radius (/ dx 2))))
(define y-lo x-lo)
(define x-hi (- radius (/ dx 2)))
(define y-hi x-hi)
(define radius-square (* radius radius))
;;
;; Normalize the detectors and define their
;; target surfaces
;;
(define n1 (vector-normalize normal_1))
(define n2 (vector-normalize normal_2))
(define ex1 (first-perpendicular n1))
(define ey1 (cross-product ex1 n1))
(define ex2 (first-perpendicular n2))
(define ey2 (cross-product ex2 n2))
;; Prepare data structures for Fourier analysis
;;
(define omega_1 (/ (* 2 pi) wavelength))
(define omega_0 (/ omega_1 2.0))
(define n_frequencies 10)
(define d_omega (/ (- omega_1 omega_0) (- n_frequencies 1)))
;;
(define frequencies (make-vector n_frequencies 0.0))
(define flux-vector-1 (make-vector n_frequencies 0.0))
(define flux-vector-2 (make-vector n_frequencies 0.0))
;;
(do ((count 0 (1+ count)))
```

```
    ((>= count n_frequencies))
  (vector-set! frequencies count (+ omega_0 (* count d_omega))))
;;
(define E-array-1 (make-array #(0 0 0) n_points n_points n_frequencies))
(define H-array-1 (make-array #(0 0 0) n_points n_points n_frequencies))
(define E-array-2 (make-array #(0 0 0) n_points n_points n_frequencies))
(define H-array-2 (make-array #(0 0 0) n_points n_points n_frequencies))
;;
;; Write on the log what you're going to do
;;
(format my-log "Computing flux of E(omega) x H(omega) through two photomultiplier tubes~%")
(format my-log " tube 1: radius = ~a, center = ~a, normal = ~a~%" radius center_1 n1)
(format my-log " ex = ~a, ey = ~a~%" ex1 ey1)
(format my-log " tube 2: radius = ~a, center = ~a, normal = ~a~%" radius center_2 n2)
(format my-log " ex = ~a, ey = ~a~%" ex2 ey2)
(format my-log " frequencies = ~a~%" frequencies)
(format my-log "Expected number of iterations: ~a~%" number_of_steps)
(force-output my-log)
;;
;; Define the function that accumulates the Fourier transform.
;; Similar to compute-local-flux.
;;
(define accumulate-fourier-transform
  (lambda (center n ex ey E-array H-array)
    (do ((x x-lo (+ x dx))
         (i 0 (1+ i)))
        ((> x x-hi))
      (do ((y y-lo (+ y dy))
           (j 0 (1+ j)))
          ((> y y-hi))
        (if (<= (+ (* x x) (* y y)) radius-square)
            (let* ((position (vector-add
                                center (vector-add (vector-mult x ex)
                                                   (vector-mult y ey))))
                   (x0 (vector-ref position 0))
                   (y0 (vector-ref position 1))
                   (z0 (vector-ref position 2))
                   (E (vector
                       (interpolate "Ex" x0 y0 z0)
                       (interpolate "Ey" x0 y0 z0)
                       (interpolate "Ez" x0 y0 z0)))
                   (H (vector
                       (interpolate "Hx" x0 y0 z0)
                       (interpolate "Hy" x0 y0 z0)
                       (interpolate "Hz" x0 y0 z0))))
              (do ((k 0 (1+ k)))
                  ((>= k n_frequencies))
                (let ((factor (* (exp (* 0-i (vector-ref frequencies k)) forms.time_e)) forms.dt)))
                  (array-set! E-array
                              (vector-add (array-ref E-array i j k)
                                          (vector-mult factor E))
                              i j k)
                  (array-set! H-array
                              (vector-add (array-ref H-array i j k)
                                          (vector-mult factor H))
                              i j k)))))))))
;;
;; Define the function that computes local flux
;;
```

```
(define compute-local-flux
  (lambda (n E-array H-array flux-vector)
    (do ((k 0 (1+ k)))
        ((>= k n_frequencies))
      (do ((x x-lo (+ x dx))
           (i 0 (1+ i)))
          ((> x x-hi))
        (do ((y y-lo (+ y dy))
             (j 0 (1+ j)))
            ((> y y-hi))
          (if (<= (+ (* x x) (* y y)) radius-square)
              (vector-set!
               flux-vector k
               (+ (vector-ref flux-vector k)
                  (* d2s 0.5
                     (dot-product
                      n
                      (vector-real-part
                       (cross-product (array-ref E-array i j k)
                                      (vector-complex-conjugate (array-ref H-array i j k))))))))))))
;;
;; Now the post-iteration function
;;
(define forms.post.iteration.lambda
  (lambda ()
    (format my-log "t = ~a, accumulating Fourier transforms ... " forms.time_e)
    (accumulate-fourier-transform center_1 n1 ex1 ey1 E-array-1 H-array-1)
    (accumulate-fourier-transform center_2 n2 ex2 ey2 E-array-2 H-array-2)
    (format my-log "done.~%")
    (force-output my-log)))
;;
(define forms.post.all.lambda
  (lambda ()
    (format my-log "Computing local fluxes ... ")
    (compute-local-flux n1 E-array-1 H-array-1 flux-vector-1)
    (compute-local-flux n2 E-array-2 H-array-2 flux-vector-2)
    (format my-log "done~%")
    (force-output my-log)
    (format my-log "Collecting local fluxes from MPI processes ... ")
    (let* ((buffer-1 (make-vector pool-size 0))
           (buffer-2 (make-vector pool-size 0)))
      (do ((k 0 (1+ k)))
          ((>= k n_frequencies))
        (gather buffer-1 (vector-ref flux-vector-1 k) root-rank)
        (gather buffer-2 (vector-ref flux-vector-2 k) root-rank)
        (if (= my-rank root-rank)
            (begin
              (vector-set! flux-vector-1 k
                           (do ((count 0 (1+ count))
                                (sum 0))
                               ((>= count pool-size) sum)
                             (set! sum (+ sum (vector-ref buffer-1 count)))))
              (vector-set! flux-vector-2 k
                           (do ((count 0 (1+ count))
                                (sum 0))
                               ((>= count pool-size) sum)
                             (set! sum (+ sum (vector-ref buffer-2 count)))))))))
    (format my-log "done.~%")
    (force-output my-log)
```

```
    (format my-log " frequencies = ~a~%" frequencies)
    (format my-log " flux-vector-1 = ~a~%" flux-vector-1)
    (format my-log " flux-vector-2 = ~a~%" flux-vector-2)
    (close-port my-log)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Output group
;;
;; There is nothing here: we don't want any images dumped.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The Watch group. This is just for debugging.
;;
(define forms.watch.main 1)
(define forms.watch.build_levels 1)
(define forms.watch.fill_levels 1)
(define forms.watch.draw_box 1)
(define forms.watch.draw_ball 1)
(define forms.watch.draw_ellipsoid 1)
(define forms.watch.effective_grid_bounds 1)
(define forms.watch.make_tag_set 1)
(define forms.watch.initialize_scheme 1)
(define forms.watch.advance_d 0)
(define forms.watch.inject_d 0)
(define forms.watch.d_to_e 0)
(define forms.watch.advance_b 0)
(define forms.watch.inject_b 0)
(define forms.watch.b_to_h 0)
(define forms.watch.mark_regions 1)
(define forms.watch.mark_TFR_faces 1)
(define forms.watch.dump_data 1)
(define forms.watch.fill_PML_arrays 1)
(define forms.watch.scheme 0)
;;
;; end
;;
```

The user defines `fourier_flux` prior to all other actions. The function `post_iteration` is then invoked by FORMS after every iteration and whatever actions are specified by the user are performed. Here we add to `fourier_flux` whatever `compflux` returns, multiplied by the exponential factor and Δt. The complex number accumulated in the variable bound to the `fourier_flux` atom is the sought after Fourier transform of the flux. It can be processed further, to extract real and imaginary parts, or the magnitude and the angle, or just written on standard output or a file.

> In summary, apart from the need to let users evaluate values of various fields at any point, at a given time, we need to provide users with the current time and time increment, as well.

Because the (x, y, z) point specified by the user may fall at any location within the computational cell, the returned value must be interpolated from the available grid values. So, the next section talks about such interpolations.

3.5.3 Parallel Utilities for Scheme

But before we get there, a few words about computing fluxes within a parallel job. In this case, each process, and therefore each Scheme instance (because there are as many Schemes running now as there are MPI processes), controls a subset of the computational domain. The flux surface is likely to span several such subsets.

There are two ways to handle the computation in this case.

The first one is to nominate a single node for carrying out the computation. The node would then collect required data from other nodes, as needed, and would evaluate the flux. The second one is to make every node evaluate its own portion of the flux, if the flux surface cuts through its computational sub-domain. At the end of the computation, the contributions evaluated by each process would be reduced to produce the total flux.

The second way is better, because it reduces communication between nodes, which is always going to be horrendously expensive, also, because it evaluates the flux in parallel. The strategy for the field value return function here would be to return zero, if a given point is not within the sub-domain of the local process, and an interpolated value otherwise.

To support these and other parallel operations on the Scheme level, FORMS Scheme-wraps basic inter-process communication primitives of Chombo, and makes them available to the user. The wrapper code is provided in module ⟨ MPI Functions 135 ⟩, Section 8.5, page 201, and function loads into Scheme appear in function *initialize_scheme*( ), defined in module ⟨ Initialize Scheme 137 ⟩, Section 8.7, page 8.7. Quoting from the latter:

```
scm_c_define_gsubr("num−proc", 0, 0, 0, (SCM(∗)())SCMnumProc);
scm_c_define_gsubr("proc−id", 0, 0, 0, (SCM(∗)())SCMprocID);
scm_c_define_gsubr("barrier", 0, 0, 0, (SCM(∗)())SCMbarrier);
scm_c_define_gsubr("unique−proc", 0, 0, 0, (SCM(∗)())SCMuniqueProc);
scm_c_define_gsubr("gather", 3, 0, 0, (SCM(∗)())SCMgather);
scm_c_define_gsubr("broadcast", 2, 0, 0, (SCM(∗)())SCMbroadcast);
```

The functions' synopses are the same as those of their Chombo equivalents, around which they wrap: *numProc*, *procID*, *barrier*, *uniqueProc*, *gather*, and *broadcast*.

The following code illustrates how they can be used in a Scheme script

```
(let∗ ((pool−size (num−proc))
       (my−rank (proc−id))
       (my−log (open−file (format #f "scm_out.~a" my−rank) "a"))
       (root−rank (unique−proc))
       (data (make−vector pool−size 0))
       (status (gather data (1+ my−rank) root−rank))
       (sum (if (= my−rank root−rank)
                (do ((count 0)
                     (my−sum 0))
                    ((>= count pool−size) my−sum)
                  (set! my−sum (+ my−sum (vector−ref data count)))
                  (set! count (1+ count)))
                0))
       (rcvd (broadcast sum root−rank)))
  (format my−log "pool size = ~a~%" pool−size)
  (format my−log "my rank = ~a~%" my−rank)
  (format my−log "root rank = ~a~%" root−rank)
  (format my−log "received = ~a~%" rcvd)
  (close−port my−log)
```

Definitions of `pool-size`, `my-rank` and `root-rank` are trivial and require no additional explanations. Functions `num-proc`, `proc-id` and `unique-proc` return the number of processes in the pool, the rank number of *this* process and the rank number of a root process to be used by the two collective operations supported: broadcast and gather.

We use the rank of *this* MPI process to create a log file for it, here referred to by `my-log`, so that any output generated by *this* Scheme instance can be saved on it. Otherwise it would be either lost or scrambled, because Scheme does not write on the Chombo's `pout.<proc>` files—unless specifically configured to do so: for example, instead of

```
(format #f "scm_out.~a" my−rank)
```

we could use

```
(format #f "pout.~a" my−rank)
```

The `gather` function takes three arguments: a vector of size greater than the pool size, into which the root process is to receive the data, an item to be sent, which must be either an integer or a real (and the vector entries must be of the same type), and the rank of the root process. The vector must be made and initialized before it is used by `gather`.

Finally, function `broadcast` takes two arguments: an item to be broadcast, which must be either a real or an integer, and the rank of the root process.

The semantics of `broadcast` and `gather` are the same as those of the Chombo (and eventually MPI) functions that are invoked by the wrappers. For `gather`, the root process is the only one that ends up with meaningful data in the vector, with data sent by process of rank n in the n-th position. For `broadcast` all processes end up with the same data returned by the function.[5]

When this job is run on four CPUs, and the `watch.scheme` parameter is set to 1, we may see the following on, say `pout.2`:

```
SCMnumProc: procs in pool = 4
SCMprocID: my rank = 2
SCMuniqueProc: unique proc rank = 0
SCMgather: sending 3
SCMbroadcast: received: 10
```

For the root process, the Chombo log would say a little more:

```
SCMnumProc: procs in pool = 4
SCMprocID: my rank = 0
SCMuniqueProc: unique proc rank = 0
SCMgather: sending 1
SCMgather: received 1 2 3 4
SCMbroadcast: sending: 10
SCMbroadcast: received: 10
```

And the Scheme log on, say, `scm_out.1`, looks as follows:

```
pool size = 4
my rank   = 1
root rank = 0
received  = 10
```

3.5.4 Grid Field Interpolation on an Arbitrary Point within the Cell

Mixed Tri/Bi/Uni-Linear Interpolation

A simple three-dimensional interpolation from the eight corner points of a cube is of the form

$$f(x,y,z) = c_{xyz}xyz + c_{yz}yz + c_{zx}zx + c_{xy}xy + c_x x + c_y y + c_z z + c_0. \tag{301}$$

The eight equations for the eight cube corners are sufficient to determine the eight coefficients uniquely. For any fixed pair, (y_0, z_0), (z_0, x_0), or (x_0, y_0), the remaining dependence on x, y, or z respectively is linear.

For a given grid cell, the cube corners can be specified by using the `lo/hi` notation, with, say, x_1 meaning the high value of x within the cube and x_0 meaning the low value. This lets us write the following eight equations:

$$f_7 = f_{111} = f(x_1,y_1,z_1) = c_{xyz}x_1y_1z_1 + c_{yz}y_1z_1 + c_{zx}z_1x_1 + c_{xy}x_1y_1 + c_x x_1 + c_y y_1 + c_z z_1 + c_0, \tag{302}$$

$$f_6 = f_{110} = f(x_1,y_1,z_0) = c_{xyz}x_1y_1z_0 + c_{yz}y_1z_0 + c_{zx}z_0x_1 + c_{xy}x_1y_1 + c_x x_1 + c_y y_1 + c_z z_0 + c_0, \tag{303}$$

$$f_5 = f_{101} = f(x_1,y_0,z_1) = c_{xyz}x_1y_0z_1 + c_{yz}y_0z_1 + c_{zx}z_1x_1 + c_{xy}x_1y_0 + c_x x_1 + c_y y_0 + c_z z_1 + c_0, \tag{304}$$

$$f_4 = f_{100} = f(x_1,y_0,z_0) = c_{xyz}x_1y_0z_0 + c_{yz}y_0z_0 + c_{zx}z_0x_1 + c_{xy}x_1y_0 + c_x x_1 + c_y y_0 + c_z z_0 + c_0, \tag{305}$$

[5] We must remember here that Scheme passes its arguments by value, hence, the value of the `item` argument cannot change. The user may do so explicitly by evaluating `(set! item (gather item root-rank))`. On the other hand, if the argument is an array, its value is its address, and so the function *can* change argument array entries by evaluating `vector-set!` forms within its body.

$$
\begin{aligned}
f_3 &= f_{011} = f(x_0, y_1, z_1) = c_{xyz}x_0y_1z_1 + c_{yz}y_1z_1 + c_{zx}z_1x_0 + c_{xy}x_0y_1 + c_x x_0 + c_y y_1 + c_z z_1 + c_0, && (306)\\
f_2 &= f_{010} = f(x_0, y_1, z_0) = c_{xyz}x_0y_1z_0 + c_{yz}y_1z_0 + c_{zx}z_0x_0 + c_{xy}x_0y_1 + c_x x_0 + c_y y_1 + c_z z_0 + c_0, && (307)\\
f_1 &= f_{001} = f(x_0, y_0, z_1) = c_{xyz}x_0y_0z_1 + c_{yz}y_0z_1 + c_{zx}z_1x_0 + c_{xy}x_0y_0 + c_x x_0 + c_y y_0 + c_z z_1 + c_0, && (308)\\
f_0 &= f_{000} = f(x_0, y_0, z_0) = c_{xyz}x_0y_0z_0 + c_{yz}y_0z_0 + c_{zx}z_0x_0 + c_{xy}x_0y_0 + c_x x_0 + c_y y_0 + c_z z_0 + c_0, && (309)
\end{aligned}
$$

where we have reduced binary forms such as 011 to their decimal equivalents, in this case 3. The same in the matrix form,

$$
\begin{pmatrix}
x_1y_1z_1 & y_1z_1 & z_1x_1 & x_1y_1 & x_1 & y_1 & z_1 & 1\\
x_1y_1z_0 & y_1z_0 & z_0x_1 & x_1y_1 & x_1 & y_1 & z_0 & 1\\
x_1y_0z_1 & y_0z_1 & z_1x_1 & x_1y_0 & x_1 & y_0 & z_1 & 1\\
x_1y_0z_0 & y_0z_0 & z_0x_1 & x_1y_0 & x_1 & y_0 & z_0 & 1\\
x_0y_1z_1 & y_1z_1 & z_1x_0 & x_0y_1 & x_0 & y_1 & z_1 & 1\\
x_0y_1z_0 & y_1z_0 & z_0x_0 & x_0y_1 & x_0 & y_1 & z_0 & 1\\
x_0y_0z_1 & y_0z_1 & z_1x_0 & x_0y_0 & x_0 & y_0 & z_1 & 1\\
x_0y_0z_0 & y_0z_0 & z_0x_0 & x_0y_0 & x_0 & y_0 & z_0 & 1
\end{pmatrix}
\begin{pmatrix}
c_{xyz}\\ c_{yz}\\ c_{zx}\\ c_{xy}\\ c_x\\ c_y\\ c_z\\ c_0
\end{pmatrix}
=
\begin{pmatrix}
f_7\\ f_6\\ f_5\\ f_4\\ f_3\\ f_2\\ f_1\\ f_0
\end{pmatrix}. \qquad (310)
$$

The structure of the matrix in this equation is such that its determinant is guaranteed to be different from zero. Furthermore, if

$$
\begin{aligned}
x_0 = y_0 = z_0 &= 0 \quad \text{and} && (311)\\
x_1 = y_1 = z_1 &= \Delta, && (312)
\end{aligned}
$$

then we can simplify it further, namely

$$
\begin{pmatrix}
\Delta^3 & \Delta^2 & \Delta^2 & \Delta^2 & \Delta & \Delta & \Delta & 1\\
0 & 0 & 0 & \Delta^2 & \Delta & \Delta & 0 & 1\\
0 & 0 & \Delta^2 & 0 & \Delta & 0 & \Delta & 1\\
0 & 0 & 0 & 0 & \Delta & 0 & 0 & 1\\
0 & \Delta^2 & 0 & 0 & 0 & \Delta & \Delta & 1\\
0 & 0 & 0 & 0 & 0 & \Delta & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & \Delta & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
c_{xyz}\\ c_{yz}\\ c_{zx}\\ c_{xy}\\ c_x\\ c_y\\ c_z\\ c_0
\end{pmatrix}
=
\begin{pmatrix}
f_7\\ f_6\\ f_5\\ f_4\\ f_3\\ f_2\\ f_1\\ f_0
\end{pmatrix}. \qquad (313)
$$

In this form the equations are easy to solve. Proceeding back to front (mostly) we obtain

$$
\begin{aligned}
c_0 &= f_0 && (314)\\
c_z &= (f_1 - f_0)/\Delta, && (315)\\
c_y &= (f_2 - f_0)/\Delta, && (316)\\
c_x &= (f_4 - f_0)/\Delta, && (317)\\
c_{yz} &= (f_3 - f_2 - f_1 + f_0)/\Delta^2, && (318)\\
c_{zx} &= (f_5 - f_4 - f_1 + f_0)/\Delta^2, && (319)\\
c_{xy} &= (f_6 - f_4 - f_2 + f_0)/\Delta^2, && (320)\\
c_{xyz} &= (f_7 + f_1 + f_2 - f_3 + f_4 - f_5 - f_6 - f_0)/\Delta^3. && (321)
\end{aligned}
$$

We can subject this solution to various tests. The first test is to evaluate it for $x = 0$ and then for $x = \Delta$. Without much ado, we see that for $x = 0$ equation (301) becomes

$$
\begin{aligned}
f(0, y, z) &= c_{yz}yz + c_y y + c_z z + c_0\\
&= (f_3 - f_2 - f_1 + f_0)\, yz/\Delta^2\\
&\quad + (f_2 - f_0)\, y/\Delta\\
&\quad + (f_1 - f_0)\, z/\Delta\\
&\quad + f_0. && (322)
\end{aligned}
$$

Hence $f(0, y, z)$ depends on $f_3 = f_{011}$, $f_2 = f_{010}$, $f_1 = f_{001}$ and $f_0 = f_{000}$ only. These are the values of f at the corners of the $x = 0$ cube face. Let us now see if equation (301) combined with solution (314–321) similarly

predicts that $f(\Delta, y, z)$ should depend on the values of f at the corners of the $x = \Delta$ cube face only. This time we find that

$$\begin{aligned} f(\Delta, y, z) &= c_{xyz}\Delta yz + c_{yz}yz + c_{zx}\Delta z + c_{xy}\Delta y + c_x\Delta + c_y y + c_z z + c_0 \\ &= (f_7 + f_1 + f_2 - f_3 + f_4 - f_5 - f_6 - f_0)yz/\Delta^2 \\ &\quad + (f_3 - f_2 - f_1 + f_0)yz/\Delta^2 \\ &\quad + (f_5 - f_4 - f_1 + f_0)z/\Delta + (f_1 - f_0)z/\Delta \\ &\quad + (f_6 - f_4 - f_2 + f_0)y/\Delta + (f_2 - f_0)y/\Delta \\ &\quad + (f_4 - f_0) + f_0 \\ &= (f_7 - f_6 - f_5 + f_4)yz/\Delta^2 \\ &\quad + (f_6 - f_4)y/\Delta \\ &\quad + (f_5 - f_4)z/\Delta \\ &\quad + f_4. \end{aligned} \tag{323}$$

We discover that this is the same expression as equation (322) on the following substitutions:

$$f_0 = f_{000} \quad \rightarrow \quad f_{100} = f_4, \tag{324}$$
$$f_1 = f_{001} \quad \rightarrow \quad f_{101} = f_5, \tag{325}$$
$$f_2 = f_{010} \quad \rightarrow \quad f_{110} = f_6, \tag{326}$$
$$f_3 = f_{011} \quad \rightarrow \quad f_{111} = f_7, \tag{327}$$

which corresponds indeed to $(x = 0) \rightarrow (x = \Delta)$.

This property ensures that the field is not only interpolated within the cube, but also that the interpolation can be carried out the same way in the adjacent cube, and the values obtained should match on the interface. The interpolation can be extended continously, though not smoothly, across the whole computational domain.

To ensure that the interpolating formulas are correct, we should repeat the procedure for $y = 0$ and $y = \Delta$ and then for $z = 0$ and $z = \Delta$, as well. Instead we will test the interpolation against a sample of various functional expressions that should test its accuracy in various directions.

Simple Tests

The following Scheme script prototypes the interpolating utility and lets us test it against various sample functions.

```
;;
;; Interpolate
;;
;; the grid
;;
(define x_0 1)
(define y_0 2)
(define z_0 3)
(define Delta 1)
;;
;; sample functions
;;
(define f1
  (lambda (x y z)
    (* x y z)))
;;
(define f2
  (lambda (x y z)
    (+ (* 3 x y z) (* 2 x y) (* 4 y z) (* 5 z x) (* -2 x) (* -3 y) (* -4 z))))
;;
```

```
(define f3
  (lambda (x y z)
    (let* ((wavelength 30)
           (pi (* 2 (asin 1)))
           (k (/ (* 2 pi) wavelength)))
      (* (sin (* k x)) (sin (* k y)) (sin (* k z))))))
;;
(define f4
  (lambda (x y z)
    (sqrt (+ (* x x) (* y y) (* z z)))))
;;
;; evaluation of a function on a grid point
;;
(define f_grid
  (lambda (f i j k)
    (let ((x (+ x_0 (* i Delta)))
          (y (+ y_0 (* j Delta)))
          (z (+ z_0 (* k Delta))))
      (f x y z))))
;;
;; interpolate an arbitrary function from its
;; grid point values
;;
(define interpolate
  ;;
  ;; requires: f -- the function and x, y, z the arguments
  ;; external: x_0, y_0, z_0, Delta -- define the grid
  ;; returns: interpolated value of f from the eight grid points
  ;; surrounding x, y, z.
  ;;
  (lambda (f x y z)
    (let* (;;
           ;; the low node of the box to which (x, y, z) belongs
           ;;
           (nx (inexact->exact (floor (/ (- x x_0) Delta))))
           (ny (inexact->exact (floor (/ (- y y_0) Delta))))
           (nz (inexact->exact (floor (/ (- z z_0) Delta))))
           ;;
           ;; function values on the grid
           ;;
           (f0 (f_grid f nx ny nz))
           (f1 (f_grid f nx ny (1+ nz)))
           (f2 (f_grid f nx (1+ ny) nz))
           (f3 (f_grid f nx (1+ ny) (1+ nz)))
           (f4 (f_grid f (1+ nx) ny nz))
           (f5 (f_grid f (1+ nx) ny (1+ nz)))
           (f6 (f_grid f (1+ nx) (1+ ny) nz))
           (f7 (f_grid f (1+ nx) (1+ ny) (1+ nz)))
           ;;
           ;; interpolation coefficients
           ;;
           (c0 f0)
           (cz (/ (- f1 f0) Delta))
           (cy (/ (- f2 f0) Delta))
           (cx (/ (- f4 f0) Delta))
           (cyz (/ (+ f3 (- f2) (- f1) f0) (* Delta Delta)))
           (czx (/ (+ f5 (- f4) (- f1) f0) (* Delta Delta)))
           (cxy (/ (+ f6 (- f4) (- f2) f0) (* Delta Delta)))
           (cxyz (/ (+ f7 f1 f2 (- f3) f4 (- f5) (- f6) (- f0)) (* Delta Delta Delta)))
```

```
        ;;
        ;; cell's low corner
        ;;
        (x0 (+ x_0 (* nx Delta)))
        (y0 (+ y_0 (* ny Delta)))
        (z0 (+ z_0 (* nz Delta)))
        ;;
        ;; dx, dy and dz distances into the cell from the low corner
        ;;
        (dx (- x x0))
        (dy (- y y0))
        (dz (- z z0)))
    ;;
    ;; interpolated value
    ;;
    (+ (* cxyz dx dy dz) (* cyz dy dz) (* czx dz dx) (* cxy dx dy) (* cx dx) (* cy dy) (* cz dz) c0))))
```

Functions `f1`, `f2`, `f3` and `f4` can be used to test the interpolation procedure on. Functions `f1` and `f2` are of the same form as equation (301). In this case we would expect the interpolation to return exactly the same result as `f1` and `f2` at every point of the computational domain. Because $\boldsymbol{r}_0 = (1, 2, 3)$ and because $\Delta = 1$, values interpolated on integer locations, which correspond to grid nodes, must be always the same as those returned by the original functions.

The following log shows that this is indeed the case:

```
guile> (load "interpolate.scm")
guile> (f2 3 2 2)
74
guile> (interpolate f2 3 2 2)
74
guile> (f2 3.2 2.1 2.7)
110.252
guile> (interpolate f2 3.2 2.1 2.7)
110.252
guile>
```

Function `f3` is a three-dimensional sine of length 30. This time, interpolation between the nodes of the grid, the constant of which is one, will return approximate result for points that are inside grid cells, and exact results on grid nodes, for example:

```
guile> (f3 3 2 2)
0.097240075008608
guile> (interpolate f3 3 2 2)
0.097240075008608
guile> (f3 3.2 2.1 2.7)
0.141711110094061
guile> (interpolate f3 3.2 2.1 2.7)
0.14025808634249
guile>
```

Here, the interpolation is accurate to within 1%. At other points, for example, $(3.5, 2.5, 2.5)$, the accuracy gets a little worse, down to 1.6%, but it's still quite good. This example reminds us that in the finite difference world, we cannot expect better, unless we reduce the grid constant by an order of magnitude, say, from 1.0 to 0.1, but this would increase the computational and memory cost by a thousand times.

Function `f4` returns the distance of the point from $(0, 0, 0)$, namely

$$f_4(\boldsymbol{r}) = r. \tag{328}$$

The interpolated values in this case are also about 1% accurate:

```
guile> (f4 3.3 2.8 1.4)
```

```
4.54862616621766
guile> (interpolate f4 3.3 2.8 1.4)
4.59421917486799
guile> (/ (- 4.59421917486799 4.54862616621766) 4.54862616621766)
0.0100234679624687
guile>
```

The interpolation function discussed here is implemented as a C++ function *SCMinterpolate*, that can access any principal field of FORMS, in Section 8.6, page 203. The function is then loaded into Scheme in Section 8.7, page 207, as follows

```
scm_c_define_gsubr("interpolate", 4, 0, 0, (SCM (*) ()) SCMinterpolate);
```

The first argument of the function is a two character string that indicates the field to be accessed. The remaining three arguments are x, y and z of the location. Magnetic fields $\boldsymbol{H}$ and $\boldsymbol{B}$ are additionally interpolated in time between `new` and `old` instances.

The function can be invoked from a user defined `forms.post.iteration.lambda`, which does not take any arguments. If the lambda is defined, it is executed after every iteration, as implemented in Section 5.3, page 137, module ⟨ Iterate 60 ⟩. If the function returns `SCM_BOOL_F`, the iterations stop and the program exits.

The following example shows a simple invocation of *interpolate* in a Scheme script:

```
(define forms.post.iteration.lambda
  (lambda ()
    (format #t "post.iteration: time_e = ~a, dt = ~a~%" forms.time_e forms.dt)
    (let ((Ex (interpolate "Ex" 32.17 46.15 22.11))
          (Ey (interpolate "Ey" 32.17 46.15 22.11))
          (Ez (interpolate "Ez" 32.17 46.15 22.11)))
      (format #t "post.iteration: Ex = ~a~%" Ex)
      (format #t "post.iteration: Ey = ~a~%" Ey)
      (format #t "post.iteration: Ez = ~a~%" Ez))))
```

Volume Weighted Interpolation

Another simple scheme interpolates a value within the cube based on the volume of boxes formed between the point and the corners of the cube.

We enumerate field values on the cube corners as before, $f_0, \ldots, f_7$. For a given point (x, y, z) such that $x \in [x_0, x_1]$, $y \in [y_0, y_1]$, and $z \in [z_0, z_1]$, and

$$x_1 - x_0 = y_1 - y_0 = z_1 - z_0 = \Delta, \tag{329}$$

the corresponding box volumes are

$$\begin{aligned}
V_0 &= (x - x_0)(y - y_0)(z - z_0), && (330)\\
V_1 &= (x - x_0)(y - y_0)(z_1 - z), && (331)\\
V_2 &= (x - x_0)(y_1 - y)(z - z_0), && (332)\\
V_3 &= (x - x_0)(y_1 - y)(z_1 - z), && (333)\\
V_4 &= (x_1 - x)(y - y_0)(z - z_0), && (334)\\
V_5 &= (x_1 - x)(y - y_0)(z_1 - z), && (335)\\
V_6 &= (x_1 - x)(y_1 - y)(z - z_0), && (336)\\
V_7 &= (x_1 - x)(y_1 - y)(z_1 - z). && (337)
\end{aligned}$$

It is easy to see that

$$\sum_i V_i = (x_1 - x_0)(y_1 - y_0)(z_1 - z_0) = \Delta^3 = V_\Delta. \tag{338}$$

So V_i/V_Δ form a set of weights, but... not for the right corners. For example, if $(x, y, z) = (x_0, y_0, z_0)$, the corresponding $V_0 = 0$, but we want the weight to be 1. It is easy to see that V_7 would provide us with the right weight in this case, because it is V_Δ. So, the formula we arrive at is that the weight should be determined by the volume against the *opposite* corner:

$$W_0(x,y,z) = V_7/V_\Delta = (x_1 - x)(y_1 - y)(z_1 - z)/\Delta^3, \quad (339)$$
$$W_1(x,y,z) = V_6/V_\Delta = (x_1 - x)(y_1 - y)(z - z_0)/\Delta^3, \quad (340)$$
$$W_2(x,y,z) = V_5/V_\Delta = (x_1 - x)(y - y_0)(z_1 - z)/\Delta^3, \quad (341)$$
$$W_3(x,y,z) = V_4/V_\Delta = (x_1 - x)(y - y_0)(z - z_0)/\Delta^3, \quad (342)$$
$$W_4(x,y,z) = V_3/V_\Delta = (x - x_0)(y_1 - y)(z_1 - z)/\Delta^3, \quad (343)$$
$$W_5(x,y,z) = V_2/V_\Delta = (x - x_0)(y_1 - y)(z - z_0)/\Delta^3, \quad (344)$$
$$W_6(x,y,z) = V_1/V_\Delta = (x - x_0)(y - y_0)(z_1 - z)/\Delta^3, \quad (345)$$
$$W_7(x,y,z) = V_0/V_\Delta = (x - x_0)(y - y_0)(z - z_0)/\Delta^3. \quad (346)$$

And now

$$f(x,y,z) = \sum_i f_i W_i(x,y,z). \quad (347)$$

We test it by loading this script into Scheme:

```
;;
;; Interpolate
;;
;; the grid
;;
(define x_0 1)
(define y_0 2)
(define z_0 3)
(define Delta 1)
;;
;; sample functions
;;
(define f1
  (lambda (x y z)
    (* x y z)))
;;
(define f2
  (lambda (x y z)
    (+ (* 3 x y z) (* 2 x y) (* 4 y z) (* 5 z x) (* -2 x) (* -3 y) (* -4 z))))
;;
(define f3
  (lambda (x y z)
    (let* ((wavelength 30)
           (pi (* 2 (asin 1)))
           (k (/ (* 2 pi) wavelength)))
      (* (sin (* k x)) (sin (* k y)) (sin (* k z))))))
;;
(define f4
  (lambda (x y z)
    (sqrt (+ (* x x) (* y y) (* z z)))))
;;
;; evaluation of a function on a grid point
;;
(define f_grid
  (lambda (f i j k)
    (let ((x (+ x_0 (* i Delta)))
          (y (+ y_0 (* j Delta)))
          (z (+ z_0 (* k Delta))))
```

```
      (f x y z))))
;;
;; interpolate an arbitrary function from its
;; grid point values
;;
(define vinterpolate
  ;;
  ;; requires: f -- the function and x, y, z the arguments
  ;; external: x_0, y_0, z_0, Delta -- define the grid
  ;; returns: interpolated value of f from the eight grid points
  ;; surrounding x, y, z.
  ;;
  (lambda (f x y z)
    (let* (;;
           ;; the low node of the box to which (x, y, z) belongs
           ;;
           (nx (inexact->exact (floor (/ (- x x_0) Delta))))
           (ny (inexact->exact (floor (/ (- y y_0) Delta))))
           (nz (inexact->exact (floor (/ (- z z_0) Delta))))
           ;;
           ;; function values on the grid
           ;;
           (f0 (f_grid f nx ny nz))
           (f1 (f_grid f nx ny (1+ nz)))
           (f2 (f_grid f nx (1+ ny) nz))
           (f3 (f_grid f nx (1+ ny) (1+ nz)))
           (f4 (f_grid f (1+ nx) ny nz))
           (f5 (f_grid f (1+ nx) ny (1+ nz)))
           (f6 (f_grid f (1+ nx) (1+ ny) nz))
           (f7 (f_grid f (1+ nx) (1+ ny) (1+ nz)))
           ;;
           ;; cell's low corner
           ;;
           (x0 (+ x_0 (* nx Delta)))
           (y0 (+ y_0 (* ny Delta)))
           (z0 (+ z_0 (* nz Delta)))
           ;;
           ;; cell's high corner
           ;;
           (x1 (+ x0 Delta))
           (y1 (+ y0 Delta))
           (z1 (+ z0 Delta))
           ;;
           ;; weights
           ;;
           (one-by-delta-cube (/ 1.0 (* Delta Delta Delta)))
           (w0 (* (- x1 x) (- y1 y) (- z1 z) one-by-delta-cube))
           (w1 (* (- x1 x) (- y1 y) (- z z0) one-by-delta-cube))
           (w2 (* (- x1 x) (- y y0) (- z1 z) one-by-delta-cube))
           (w3 (* (- x1 x) (- y y0) (- z z0) one-by-delta-cube))
           (w4 (* (- x x0) (- y1 y) (- z1 z) one-by-delta-cube))
           (w5 (* (- x x0) (- y1 y) (- z z0) one-by-delta-cube))
           (w6 (* (- x x0) (- y y0) (- z1 z) one-by-delta-cube))
           (w7 (* (- x x0) (- y y0) (- z z0) one-by-delta-cube)))
      ;;
      ;; interpolated value
      ;;
      (+ (* f0 w0) (* f1 w1) (* f2 w2) (* f3 w3) (* f4 w4) (* f5 w5) (* f6 w6) (* f7 w7)))))
```

The script is similar to the one used previously, with the exception that function `interpolate` is now replaced

with function `vinterpolate`, which implements the volume weighted scheme discussed in this section.

When we run the script and evaluate `vinterpolate` we find that

```
guile> (load "vinterpolate.scm")
guile> (f2 3 2 2)
74
guile> (vinterpolate f2 3 2 2)
74.0
guile> (f2 3.2 2.1 2.7)
110.252
guile> (vinterpolate f2 3.2 2.1 2.7)
110.252
guile> (f3 3 2 2)
0.097240075008608
guile> (vinterpolate f3 3 2 2)
0.097240075008608
guile> (f3 3.2 2.1 2.7)
0.141711110094061
guile> (vinterpolate f3 3.2 2.1 2.7)
0.14025808634249
guile> (f4 3.3 2.8 1.4)
4.54862616621766
guile> (vinterpolate f4 3.3 2.8 1.4)
4.59421917486799
guile>
```

The results are exactly the same as those returned previously, which tells us that the volume weighted interpolation formula is the same as (301). That it should be so, follows from the fact that equation (301) has its coefficients determined *uniquely* by the eight values $f_0, \ldots, f_7$. The volume weighted equation (347) eventually expands into a formula with terms proportional to xyz, xy, yz, zx, x, y, and z, and with coefficients determined by $f_0, \ldots, f_7$, too, and so it must be identical.

We can see this also by evaluating the term proportional to, say, f_2 using equation (301). According to equations (314–321), f_2 is contributed by c_y, c_{yz}, c_{xy}, and c_{xyz}, as follows:

$$f_2 y/\Delta - f_2 yz/\Delta^2 - f_2 xy/\Delta^2 + f_2 xyz/\Delta^3 = f_2(y\Delta^2 - yz\Delta - xy\Delta + xyz)/\Delta^3. \tag{348}$$

In turn, equations (339–346) tell us that

$$\begin{aligned} W_2(x,y,z) &= V_5/V_\Delta = (x_1 - x)(y - y_0)(z_1 - z)/\Delta^3 \\ &= (\Delta - x)(y)(\Delta - z)/\Delta^3 \\ &= y(\Delta^2 - z\Delta - x\Delta + xz)/\Delta^3 \\ &= (y\Delta^2 - yz\Delta - xy\Delta + xyz)/\Delta^3, \end{aligned} \tag{349}$$

which is indeed the same as (348).

Other Interpolation Schemes

Balsara [1] proposes interpolation formulas, which are different from (301), and which are direction dependent, namely

$$B_x(x,y,z) = a_0 + a_x x + a_y y + a_z z + a_{xx}x^2 + a_{xy}xy + a_{xz}xz, \tag{350}$$

$$B_y(x,y,z) = b_0 + b_x x + b_y y + b_z z + b_{xy}xy + b_{yy}y^2 + b_{yz}yz, \tag{351}$$

$$B_z(x,y,z) = c_0 + c_x x + c_y y + c_z z + c_{xz}xz + c_{yz}yz + c_{zz}z^2. \tag{352}$$

For a fixed $y = y_0$ and $z = z_0$, for example, $B_x(x, y_0, z_0)$ is quadratic in x, whereas our formula (301) always results in linear dependence on each variable for the other two fixed. Interpolation (350–352) allows for divergence free field reconstruction in the Balsara AMR scheme for MHD.

Zakharian *et al.* [20] also resort to quadratic interpolations,[6] but theirs are split into two one-dimensional operations. And so, in order to obtain fine grid ghost cell values for E_z, which in their scheme is mounted in the center of the $\boldsymbol{e}_x \wedge \boldsymbol{e}_y$ face,[7] on the $\boldsymbol{e}_x \wedge \boldsymbol{e}_z$ interface, they carry out piecewise quadratic interpolation along the x-axis first:

$$\tilde{E}_z^f\left(i+\frac{1}{4},j,k-\frac{1}{2}\right)=\frac{5}{32}E_z^c\left(i-1,j,k-\frac{1}{2}\right)+\frac{15}{16}E_z^c\left(i,j,k-\frac{1}{2}\right)-\frac{3}{32}E_z^c\left(i+1,j,k-\frac{1}{2}\right), \tag{353}$$

where superscript f stands for "fine", and superscript c stands for "coarse". Then they carry out another quadratic interpolation normal to the interface, along the y-axis, using the temporary fine grid value found above:

$$E_z^g\left(i+\frac{1}{4},j+\frac{1}{4},k-\frac{1}{2}\right)=\frac{8}{15}\tilde{E}_z^f\left(i+\frac{1}{4},j,k-\frac{1}{2}\right)+\frac{2}{3}E_z^f\left(i+\frac{1}{4},j+\frac{3}{2},k-\frac{1}{2}\right)-\frac{1}{5}E_z^f\left(i+\frac{1}{4},j+\frac{5}{2},k-\frac{1}{2}\right), \tag{354}$$

where superscript g stands for "ghost".

This, like other higher order schemes, have the disadvantage that data needs to be collected from more than one cell to carry out the interpolation. For the purpose of flux evaluation across an arbitrary surface, this should not be necessary, but we will revisit the issue, when discussing multigrid operations.

[6] They do not derive the expressions, referring the reader to a future paper instead. But the paper has not been published so far. The methodology is said to be based on the two earlier papers that discussed two-dimensional multigrid FDTD, [21] and [22].

[7] In our scheme $\boldsymbol{B}$ is face center mounted and $\boldsymbol{E}$ is edge center mounted.

4 Guile Wrap

The construct seen in *main* boots Guile, which rides on top of the C++ program. This is supposed to be a more portable way of doing this, compared to a somewhat nicer *scm_init_guile*, given that Guile needs to know about the C++ stack.

Function *scm_boot_guile* never returns. Consequently, we defer everything else to *inner_main*, including the handling of exits and MPI startup.

Because the GH interface is now deprecated, we have to abstain from *gh_enter*.

Function *inner_main* takes three arguments. The first one is a pointer to any additional data that we may wish to provide it with. The data is passed to it through the last argument of *scm_boot_guile*, which here is simply Λ (meaning NULL).

The variable *current_level_ptr* is defined here as global. It is set by C++ Chombo functions and then used by C++ Scheme functions to access Chombo data. We also define here *levels_ptr*, which is a pointer to the **Vector** of levels, for interlevel operations that we may wish to orchestrate from within Scheme.

`EXIT_SUCCESS` is defined on *stdlib.h*. In this program I assume that it is zero. If for some totally pathological reason it is not zero on some future system, it should be redefined in the header file to be so.

52
```
#include "Forms.H"
static void inner_main(void *data, int argc, char *argv[]);
    /* The following is used to let Scheme draw on level grids. */
level *current_level_ptr;
Vector<level *> *levels_ptr;
int main(int argc, char *argv[])
{
  scm_boot_guile(argc, argv, inner_main, Λ);
      /* Never gets here */
  return (EXIT_SUCCESS);
}
```
⟨ Inner Main 53 ⟩

5 Code Outline

The code of *inner_main*, discussed in this chunk, is bracketed by a CPP # **ifdef** that checks if the code is going to be linked with MPI libraries. If so, *MPI_Init* is called at the very beginning and *MPI_Finalize* is called at the end. The remainder of the code is enclosed within a new scope defined in between. This is to ensure that all Chombo classes, including some that call MPI, go out of scope before *MPI_Finalize* is called (cf. [3], Section 7, "Parallel Programming with Chombo").

Within this new MPI scope we introduce the program, and check if the code has been linked with the 3D Chombo libraries. If not, we flag a problem and jump to exit. This is because some Chombo-style classes (our own and contributed) used by the code, for example, curl-refluxing, work in 3D only, but do not check for it fully.

Throughout the code we use *pout*( ), declared on `parstream.H`, rather than *cout* to print messages. In the parallel context *pout*( ) writes its output on process specific files.

exit_status is defined outside the MPI scope, because it has to be available before and after MPI has gone away, at the end of the program. We use *exit*( ) instead of **return** ( ), because, as we have mentioned above, *scm_boot_guile* that runs *inner_main* never returns, so we cannot return to *main*. We have to *exit*( ) here and now.

Once we get to it, the actual lines of *inner_main* are very simple.

First we read an input file, then we run additional initializations on Scheme, some of which require data provided on the input file. Then, using information provided, we build the multi-grid structure and fill it with data. Finally, we commence the iterations.

Although the code can run from a *restart* file to continue its computations, this is not visible at the top level. Instead, if a *restart* file exists and the user has requested its use, it is loaded by function *build_levels*, discussed in module ⟨ Function *build_levels* 62 ⟩, which is responsible for generating the multi-grid and filling it with initial data.

Similarly, not only images but checkpoint files as well are generated on request by function *dump_data*, which is discussed in module ⟨ Function *dump_data* 139 ⟩.

53 ⟨ Inner Main 53 ⟩ ≡

```
static void inner_main(void *data, int argc, char *argv[])
{
  int exit_status = EXIT_SUCCESS;
#ifdef CH_MPI
  MPI_Init(&argc, &argv);
#endif
  {
    pout( ) << argv[0] << ":" << endl;
    pout( ) << "\tVersion " << PROGRAM_VERSION << endl;
    pout( ) << "\tProcess " << procID( ) << endl;
    if (SpaceDim != 3) {
      pout( ) << "\tERROR(main): this program works in 3D only" << endl;
      exit_status = ERR_BAD_DIMENSION;
      goto exit;
    }
    initialize_scheme( );
    ⟨ Process an Input File 54 ⟩
    if (exit_status = post_initialize_scheme( )) {
      pout( ) << "\tERROR(main): failed to initialize Scheme" << endl;
      goto exit;
    }
    ⟨ Build a Multigrid 58 ⟩
    ⟨ Fill the Multigrid 59 ⟩
    ⟨ Iterate 60 ⟩
  }
exit:
```

```
#ifdef CH_MPI
    MPI_Finalize( );
#endif
    pout( ) << "\tFinished. Exiting..." << endl;
    exit(exit_status);
  }
```

This code is cited in chunks 64 and 137.

This code is used in chunk 52.

5.1 Process an Input File

This is a somewhat clumsy part of the code that could be relegated to a separate function. On the other hand, it doesn't do much, and, since CWEB lets us structure the code by other means, we keep it here for simplicity.

The first thing to do here is to check if an input file name has been given to us on the command line. We exit right away, if there is no name.

Once we have the name, we pass it to the **SCMParmParse** class on the object construction, and the constructor processes the file.

54 ⟨ Process an Input File 54 ⟩ ≡

```
char *input_file_name = Λ;
if (argc > 1) {
  input_file_name = argv[1];
  pout() ≪ "\tReading␣input␣from␣" ≪ argv[1] ≪ endl;
}
else {
  pout() ≪ "\tERROR(main):␣no␣input␣file␣specified" ≪ endl;
  pout() ≪ "\t\tUsage:␣" ≪ argv[0] ≪ "␣<inputfile>" ≪ endl;
  exit_status = ERR_NO_INPUT_FILE;
  goto exit;
}
SCMParmParse parameter_parser(argc − 2, argv + 2, Λ, input_file_name);
```

See also chunks 55 and 56.

This code is used in chunk 53.

¶ The input file has been opened and loaded into the program with the **SCMParmParse** constructor. Now we read verbosity and iteration parameters for *inner_main*.

The meaning of the parameters is as follows

watch_main Scheme name: *forms.watch.main*. This is a verbosity parameter. If set to 0, it makes *main* do the remainder of its work silently. If set to 1 or higher, it makes it chat about the progress of computation with verbosity proportional to its value.

number_of_steps Scheme name: *forms.iterate.number_of_steps*. The total number of iterations to be performed.

image_frequency Scheme name: *forms.iterate.image_frequency*. Images will be produced every *image_frequency* time steps.[8]

initial_label Scheme name: *forms.iterate.initial_label*. The initial label to use with dumped image and checkpoint files. If the job is to restart from a checkpoint file, the initial label should be set to an integer following the number of the last image dumped in the previous run, otherwise the previously produced images will be overwritten. A PBS script running the job should attend to this.

The *query* calls alter the default values for the parameters only if the required values are found on the input file. They do not throw errors if they are absent.

55 ⟨ Process an Input File 54 ⟩ +≡

```
SCMParmParse watch_parameter_parser("forms.watch");
SCMParmParse iterate_parameter_parser("forms.iterate");
SCMParmParse media_parameter_parser("forms.media");
int watch_main = false;
int number_of_steps = NUMBER_OF_STEPS;
int image_frequency = IMAGE_FREQUENCY;
int initial_label = INITIAL_LABEL;
int number_of_auxiliary_fields = N_AUXILIARIES;
```

[8] In our previous program SHAPES the meaning of this parameter was different and somewhat confusing. The images were dumped every *stride* ∗ *image_frequency* time steps. In this program *stride* will appear later, when level 0 is built.

```
watch_parameter_parser.query("main", watch_main);
iterate_parameter_parser.query("number_of_steps", number_of_steps);
iterate_parameter_parser.query("image_frequency", image_frequency);
iterate_parameter_parser.query("initial_label", initial_label);
media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
```

¶ Here function *inner_main* chats about what it's read, if it's been asked to do so.

56 ⟨ Process an Input File 54 ⟩ +≡

```
if (watch_main) {
  pout() ≪ "\t\twatch_main␣␣␣␣␣␣␣=␣" ≪ watch_main ≪ endl;
  pout() ≪ "\t\tnumber_of_steps␣␣=␣" ≪ number_of_steps ≪ endl;
  pout() ≪ "\t\timage_frequency␣␣=␣" ≪ image_frequency ≪ endl;
  pout() ≪ "\t\tinitial_label␣␣␣=␣" ≪ initial_label ≪ endl;
  pout() ≪ "\t\tauxiliary_fields␣=␣" ≪ number_of_auxiliary_fields ≪ endl;
}
```

5.2 Build Initial Condition

This is a very short and sweet part of the code, mostly because it's all hidden in *build_levels* and *fill_levels*.

The former builds level 0 grid first and then builds iteratively higher level grids. The latter fills the grids with an initial condition, which in this code is zero everywhere, and fills the *medium* field.

The *medium* field is handled as follows. The user provides a Scheme code to assign an integer medium index to each edge of the multigrid. When $\boldsymbol{D}$ needs to be converted to $\boldsymbol{E}$, another piece of user provided Scheme code is invoked. It is passed the medium index, $\boldsymbol{D}$, $\boldsymbol{E}$ and auxiliary fields at this location in its argument list, and then an updated $\boldsymbol{E}$ comes out.

This way the user has the greatest flexibility possible in defining the number of media, media distribution and media types. Whereas in SHAPES this stuff was hardwired into the code and various "models" provided, here, users co-program the code instead. The co-programming takes place on the interpreted level, which means that the program itself does not have to be recompiled every time the user provides new specifications.

What's inside *build_levels* and *fill_levels* is, unfortunately, somewhat complicated. Both functions return zero if everything has gone well, and a non-zero diagnostic if it hasn't. In this case, the diagnostic is transferred to *exit_status* and we jump to *exit*.

5.2.1 Build a Multigrid

58 ⟨ Build a Multigrid 58 ⟩ ≡

```
Vector⟨level ∗⟩ levels;
levels_ptr = &levels;
if (exit_status = build_levels(levels)) {
  pout( ) ≪ "\tERROR(main):␣Failed␣to␣build␣levels.␣Exiting..." ≪ endl;
  goto exit;
}
else {
  if (watch_main) {
    int actual_number_of_levels = levels.size( );
    pout( ) ≪ "\tCreated␣" ≪ actual_number_of_levels;
    if (actual_number_of_levels ≡ 1) pout( ) ≪ "␣level." ≪ endl;
    else pout( ) ≪ "␣levels." ≪ endl;
  }
}
```

This code is used in chunk 53.

5.2.2 Fill the Multigrid

59 ⟨ Fill the Multigrid 59 ⟩ ≡

```
if (exit_status = fill_levels(levels)) {
  pout( ) ≪ "\tERROR(main):␣Failed␣to␣fill␣levels.␣Exiting..." ≪ endl;
  goto exit;
}
else {
  if (watch_main) {
    int actual_number_of_levels = levels.size( );
    pout( ) ≪ "\tFilled␣" ≪ actual_number_of_levels;
    if (actual_number_of_levels ≡ 1) pout( ) ≪ "␣level." ≪ endl;
    else pout( ) ≪ "␣levels." ≪ endl;
  }
}
```

This code is used in chunk 53.

5.3 Iterate

This is the heart of the program. But even here we sweep all complexity into functions that are called within the loop to keep the mainlines clean.

The iteration loop takes care both of advancing the multi-level system and of dumping images and checkpoint files. The loop counter *count* is advanced in the loop header. The logical condition *time_to_dump_data* is calculated in the loop header too. The loop termination condition is such that iterations stop on the last image dumped, not on the actual number of iterations requested. This is so because it's pointless to iterate beyond the last image.

Within the loop itself we advance the electric field first, and then we advance the magnetic field. These are really mutually recursive functions that take care of a multi-level time step, so that advancing $\boldsymbol{E}$ at the bottom level (level 0) triggers advancing $\boldsymbol{H}$ and $\boldsymbol{E}$ at higher levels in a mutually recursive cascade and the same holds for advancing $\boldsymbol{H}$ at the bottom level.

Any problems here are handled by looking at function returns and jumping to *exit* if problems are detected.

When the *time_to_dump_data* condition is detected, a function is called that dumps data (images and checkpoint files) for the whole multi-grid.

> I thought of orchestrating this dance from within Scheme. This probably can be done for a sequential code, because we have the global *levels_ptr*. But I truly don't know if MPI calls would run correctly when invoked from within Scheme. Whatever context MPI sets up may not be visible to Scheme invoked C++ functions, and so the *exchange* methods of **LevelData** may fail to exchange data between MPI processes.
>
> This should not affect the various small Scheme functions that we are going to use in this program, because these are strictly local. It's not Scheme orchestrating C++. It's C++ running everything and only invoking Scheme *calculator* when needed.

60 ⟨ Iterate 60 ⟩ ≡

```
if (watch_main) pout() ≪ "\tIterating..." ≪ endl;
clock_t total_clock1, total_clock2, total_clock = (clock_t) 0;
clock_t inject_clock1, inject_clock2, inject_clock = (clock_t) 0;
clock_t convert_clock1, convert_clock2, convert_clock = (clock_t) 0;
clock_t advance_clock1, advance_clock2, advance_clock = (clock_t) 0;
clock_t output_clock1, output_clock2, output_clock = (clock_t) 0;
clock_t post_iteration_clock1, post_iteration_clock2, post_iteration_clock = (clock_t) 0;
clock_t post_all_clock1, post_all_clock2, post_all_clock = (clock_t) 0;
extern bool have_post_iteration_lambda;
extern bool have_post_all_lambda;
extern SCM post_iteration_lambda;
extern SCM post_all_lambda;
extern SCM t_e_var;
extern SCM dt_var;
SCMParmParse post_iteration_parser("forms.post.iterate");
total_clock1 = clock();
for (int label = initial_label, count = 1, time_to_dump_data = false;
(int)((count − 1)/image_frequency) ∗ image_frequency + image_frequency ≤ number_of_steps;
count++, time_to_dump_data = ¬(count % image_frequency)) {
  if (watch_main)
    pout() ≪ "\t\tcount = " ≪ count ≪ ", time_e = " ≪ levels[0]→time_e ≪ ", time_h = " ≪
        levels[0]→time_h ≪ endl;
  advance_clock1 = clock();
  if (exit_status = advance_d(levels[0])) {
    pout() ≪ "\tERROR(main): Failed to advance D. Exiting..." ≪ endl;
    goto exit;
  }
  advance_clock2 = clock();
```

```
advance_clock += advance_clock2 − advance_clock1;
inject_clock1 = clock();
if (exit_status = inject_d(levels[0])) {
  pout() ≪ "\tERROR(main):␣Failed␣to␣inject␣D.␣Exiting..." ≪ endl;
  goto exit;
}
inject_clock2 = clock();
inject_clock += inject_clock2 − inject_clock1;
convert_clock1 = clock();
if (exit_status = d_to_e(levels[0])) {
  pout() ≪ "\tERROR(main):␣Failed␣to␣convert␣D␣to␣E.␣Exiting..." ≪ endl;
  goto exit;
}
convert_clock2 = clock();
convert_clock += convert_clock2 − convert_clock1;
advance_clock1 = clock();
if (exit_status = advance_b(levels[0])) {
  pout() ≪ "\tERROR(main):␣Failed␣to␣advance␣B.␣Exiting..." ≪ endl;
  goto exit;
}
advance_clock2 = clock();
advance_clock += advance_clock2 − advance_clock1;
inject_clock1 = clock();
if (exit_status = inject_b(levels[0])) {
  pout() ≪ "\tERROR(main):␣Failed␣to␣inject␣B.␣Exiting..." ≪ endl;
  goto exit;
}
inject_clock2 = clock();
inject_clock += inject_clock2 − inject_clock1;
convert_clock1 = clock();
if (exit_status = b_to_h(levels[0])) {
  pout() ≪ "\tERROR(main):␣Failed␣to␣convert␣B␣to␣H.␣Exiting..." ≪ endl;
  goto exit;
}
convert_clock2 = clock();
convert_clock += convert_clock2 − convert_clock1;
if (have_post_iteration_lambda) {
  post_iteration_clock1 = clock();
      /* Update t_e and Δt on forms.time_e and forms.dt. We do this by reference avoiding the symbol
      table lookup. Any SCMParmParse object can be used for this, because the variables are not
      accessed by name. */
  post_iteration_parser.fastput(t_e_var, levels[0]→time_e);
  post_iteration_parser.fastput(dt_var, levels[0]→dt);
  SCM status = scm_call_0(post_iteration_lambda);
      /* scm_call_0 may return anything that's a valid SCM type. If it's a boolean false, we flag a
      problem and exit. */
  if (scm_is_bool(status)) {
    if (scm_is_false(status)) {
      pout() ≪ "\tERROR(main):␣forms.post.iteration.lambda␣returned␣#f" ≪ endl;
      goto exit;
    }
  }
  post_iteration_clock2 = clock();
  post_iteration_clock += post_iteration_clock2 − post_iteration_clock1;
```

```
    }     /* if (have_post_iteration_lambda) */
    if (time_to_dump_data) {
      if (watch_main) pout() ≪ "\t\tdumping␣data,␣file␣label␣" ≪ label ≪ endl;
      output_clock1 = clock();
      if (exit_status = dump_data(levels, label++)) {
        pout() ≪ "\tERROR(main):␣Failed␣to␣dump␣data.␣Exiting..." ≪ endl;
        goto exit;
      }
      output_clock2 = clock();
      output_clock += output_clock2 − output_clock1;
    }
  }     /* the iteration for loop */
  if (have_post_all_lambda) {
    post_all_clock1 = clock();
    SCM status = scm_call_0(post_all_lambda);
    if (scm_is_bool(status)) {
      if (scm_is_false(status)) {
        pout() ≪ "\tERROR(main):␣forms.post.all.lambda␣returned␣#f" ≪ endl;
        goto exit;
      }
    }
    post_all_clock2 = clock();
    post_all_clock += post_all_clock2 − post_all_clock1;
  }     /* if (have_post_all_lambda) */
  total_clock2 = clock();
  total_clock += total_clock2 − total_clock1;
  pout() ≪ "\tIterations␣completed␣in␣" ≪ (int) total_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
  pout() ≪ "\t\tAdvancing␣fields␣took␣" ≪ (int) advance_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
  pout() ≪ "\t\tSignal␣injection␣took␣" ≪ (int) inject_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
  pout() ≪ "\t\tMaterial␣physics␣took␣" ≪ (int) convert_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
  if (have_post_iteration_lambda)
    pout() ≪ "\t\tPost-iteration␣lambda␣" ≪ (int) post_iteration_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
  if (have_post_all_lambda)
    pout() ≪ "\t\tPost-all␣lambda␣" ≪ (int) post_all_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
  pout() ≪ "\t\tProducing␣output␣took␣" ≪ (int) output_clock/CLOCKS_PER_SEC ≪ "s" ≪ endl;
```

This code is cited in chunk 49.

This code is used in chunk 53.

6 Initialization

Here we define the content of file *Initialize.c*. Functions used to define the grid and the initial condition are described here.

61 ⟨ `Initialize.c` 61 ⟩ ≡

```
#include "Forms.H"
    /* Functions used to build initial condition */
⟨ Function build_levels 62 ⟩
⟨ Function fill_levels 74 ⟩
```

6.1 Build Levels

This function, *build_levels*, generates and distributes amongst MPI processes grids for all levels as requested by the user. We used to do this recursively in SHAPES, but this time all levels are constructed iteratively within the single body of this function. The resulting function is somewhat long, but really quite simple. A great deal of flexibility is provided by linking Scheme into the program. Because of this *build_levels* does not have to be as elaborate as it was in SHAPES.

The general layout of the function is very simple, as shown below, and the logic of each module is also simple, if at times tedious.

62 ⟨ Function *build_levels* 62 ⟩ ≡

```
int build_levels(Vector⟨level *⟩ &levels)
{
  ⟨ build_levels: variable definitions 63 ⟩
  ⟨ build_levels: read grid specifications 68 ⟩
  ⟨ build_levels: build level 0 70 ⟩
  ⟨ build_levels: build higher levels 71 ⟩
exit:
  if (watch_build_levels) pout() ≪ "build_levels: returning to caller" ≪ endl;
  return return_status;
}
```

This code is cited in chunk 53.

This code is used in chunk 61.

6.1.1 Variable Definitions

There is a fair number of variables to define here, and it gets pretty tedious to do so. But if this section is chopped into smaller chunks, the variables can be organized and explained in groups.

We begin with scalars. These are mostly integers and there's just one **Real** thrown in.

The meaning of the first two: *return_status* and *watch_build_levels* is obvious. The latter's value is read from the Scheme space, if not found there, it's set to *false*.

Then we have a group of important variables that specify the grid. They are

number_of_levels This is a number of grid levels to be generated altogether including level 0. This number is read from the Scheme space.

max_box_size This is a maximum size of the level 0 box. The level 0 domain will be subdivided into these. The boxes will then be distributed amongst the participating MPI processes. The actual dimensions of the level 0 grid are read into a vector that will be discussed in the next chunk.

refine_block_factor This is a smallest box size in cells of a subgrid to be generated that will be used for the subgrid **DisjointBoxLayout**.

refine_buffer_size This is a number of grid cells that separates its boundary with the coarser grid from its boundary with the finer grid. These two must not get too close. This variable is read from the Scheme space, but if it is too small, it is reset to the default `REFINE_BUFFER_SIZE` by Scheme routines that draw on the level's *tags* field.

refine_max_size This is a maximum box size, in cells, used to generate a subgrid. Similar to *max_box_size* for level 0.

refine_ratio This is a refinement ratio between grid levels. This variable cannot be changed in this version of the program and is preset to `REFINE_RATIO`, which is 2. Note that in 3-dimensions, the refinement ratio of 2 subdivides each cell into eight sub-cells–this is almost an order of magnitude increase in 3-D resolution, with the correspondingly high cost in storage and computation. So 2 is plenty.

The other reason we don't allow for this parameter to be user-adjusted is because we do not have field stitching mathematics developed for refinement ratios other than 2. Yet. ◀

refine_fill_ratio This parameter tells the regriding routine how tightly it should enclose the *tags* field in the newly generated subgrid. 1.0 means *very tightly,* 0.0 means *not tightly at all.* The lower this number, the larger the boxes used to generate the subgrid. If the *tags* field has a very intricate shape and this parameter is set to 1.0, the regriding routine will generate a large number of tiny boxes, with which the complicated boundary of the *tags* fiels will be drawn.

refine_base_level, *refine_top_level* These two parameters are used to tell the regriding routine which levels should be refined.

63 ⟨ *build_levels*: variable definitions 63 ⟩ ≡
```
int return_status = EXIT_SUCCESS;
int watch_build_levels = false;
int number_of_levels = N_LEVELS;
int max_box_size = MAX_BOX_SIZE;
int refine_block_factor = REFINE_BLOCK_FACTOR;
int refine_buffer_size = REFINE_BUFFER_SIZE;
int refine_max_size = REFINE_MAX_SIZE;
int refine_ratio = REFINE_RATIO;
int refine_base_level, refine_top_level;
Real refine_fill_ratio = REFINE_FILL_RATIO;
```
See also chunks 64, 65, 66, and 67.
This code is cited in chunk 71.
This code is used in chunk 62.

¶ Now we get to define various **Vector**s. They are all 3-dimensional, i.e., *SpaceDim* (let us remember that we have already checked if *SpaceDim* is three in module ⟨ Inner Main 53 ⟩, section 5, page 132).

n_cells This is a number of cells of the level 0 grid in each of its principal directions.

delta This used to be a vector, but I have decided to make it a scalar. It is a grid constant of the level 0 grid and it is the same for each principal direction of the grid, meaning that each grid cell is a cube. This is what our Fortran routines assume. This simplifies and speeds up computation, and there is no real advantage in making it different for every direction.

origin These are physical coordinates of the center of the $(0, 0, 0)$ cell of the level 0 grid.

pml_xyz_min These are *physical* coordinates of the low corner of the PML boundary box. The PML box, of course, must be contained entirely within the level 0 domain.

pml_xyz_max These are *physical* coordinates of the high corner of the PML boundary box.

signal_xyz_min These are *physical* coordinates of the low corner of the signal injection box. The signal injection box must be contained entirely within the PML box.

signal_xyz_max These are *physical* coordinates of the high corner of the signal injection box.

64 ⟨ *build_levels*: variable definitions 63 ⟩ +≡
```
Vector⟨int⟩ n_cells(SpaceDim, N_CELLS);
Real delta = DELTA;
Real stride = STRIDE;
Real t0 = T0;
Vector⟨Real⟩ origin(SpaceDim, X0);
Vector⟨Real⟩ pml_xyz_min(SpaceDim, PML_XYZ_MIN);
Vector⟨Real⟩ pml_xyz_max(SpaceDim, PML_XYZ_MAX);
Vector⟨Real⟩ signal_xyz_min(SpaceDim, SIGNAL_XYZ_MIN);
Vector⟨Real⟩ signal_xyz_max(SpaceDim, SIGNAL_XYZ_MAX);
```

¶ These are my **SCMParmParse** parsers for reading parameters from the Scheme space for `"grid"`, `"grid.level0"`, `"watch"`, `"pml"` and `"signal"` groups.

65 ⟨ *build_levels*: variable definitions 63 ⟩ +≡
```
SCMParmParse grid_parameter_parser("forms.grid");
SCMParmParse level0_parameter_parser("forms.grid.level0");
SCMParmParse watch_parameter_parser("forms.watch");
SCMParmParse pml_parameter_parser("forms.pml");
SCMParmParse signal_parameter_parser("forms.signal");
SCMParmParse iterate_parameter_parser("forms.iterate");
```

¶ These are pointers to levels.

current_level_ptr This is a global pointer variable that's been declared in section 4, page 131. Here the code will point it to the last fully defined level. Predefined drawing routines invoked by Scheme know about this variable and use it to draw on its *tags* field.

level_0_ptr This is a pointer to level 0 while it is being constructed and before it becomes *levels*[0].

new_level_ptr This is a pointer to a higher level while it is being constructed and before it is appended to the *levels* array.

66 ⟨ *build_levels*: variable definitions 63 ⟩ +≡
```
extern level *current_level_ptr;
level *level_0_ptr;
level *new_level_ptr;
```

¶ These are special variables that are used by the **BRMeshRefine** machine, which constructs boxes for the next refined level.

vector_of_tag_sets This is a vector of tag sets. Each vector entry corresponds to a level. Within this level we tag cells for refinement. The tags for a given level are assembled into a set, and the set is then appended to the vector. For example, *vector_of_tag_sets*[0] is a set of level 0 cells tagged for refinement.

vector_of_refinements This is a vector of refinement ratios between levels. In out case, it is simply $[2, 2, 2, 2, \ldots]$.

vector_of_vectors_of_boxes The boxes of each level are arranged into a vector, *vector_of_boxes*. Then the level vectors themselves are arranged into a vector, *vector_of_vectors_of_boxes*, that specifies the whole multi-level grid, currently constructed.

new_vector_of_vectors_of_boxes After the **BRMeshRefine** machine has done its job, it returns the new multi-level grid structure on the *new_vector_of_vectors_of_boxes*. From this we can pick up the last *vector_of_boxes*, it will be the newly created sub-grid, and distribute it over the MPI processes to balance the load.

67 ⟨ *build_levels*: variable definitions 63 ⟩ +≡
```
Vector⟨IntVectSet⟩ vector_of_tag_sets;
Vector⟨int⟩ vector_of_refinements;
Vector⟨Vector⟨Box⟩⟩ vector_of_vectors_of_boxes, new_vector_of_vectors_of_boxes;
```

6.1.2 Read Grid Specifications

So, now we get to the tedious part of parsing the Scheme space for the parameters that this function needs. We also check on this occasion if some parameters are sensibly defined.

The first one we read is *watch_build_levels*, which activates the chat. Then we read *number_of_levels*. This number must fit between `N_LEVELS_MIN`, which is 1, and `N_LEVELS_MAX`. Because building and advancing levels

is costly, both computationally and memory-wise, we restrict their numbers. We jump to exit if *number_of_levels* is bad.

Similarly we read *max_box_size* and ensure that it fits between `MAX_BOX_SIZE_MIN` and `MAX_BOX_SIZE_MAX`. On this occasion we also ensure that *max_box_size* is a power of 2. All sizes in FORMS should be a power of 2, preferably.

68 ⟨ *build_levels*: read grid specifications 68 ⟩ ≡

```
watch_parameter_parser.query("build_levels", watch_build_levels);
if (watch_build_levels) pout() ≪ "build_levels:" ≪ endl;
if (grid_parameter_parser.query("number_of_levels", number_of_levels)) {
  if (watch_build_levels) pout() ≪ "\tread:␣number_of_levels␣=␣" ≪ number_of_levels ≪ endl;
  if ((number_of_levels < N_LEVELS_MIN) ∨ (number_of_levels > N_LEVELS_MAX)) {
    pout() ≪ "\n\tERROR(build_levels):␣number_of_levels␣out␣of␣range" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
if (grid_parameter_parser.query("max_box_size", max_box_size)) {
  if (watch_build_levels) pout() ≪ "\tread:␣max_box_size␣=␣" ≪ max_box_size ≪ endl;
  if ((max_box_size < MAX_BOX_SIZE_MIN) ∨ (max_box_size > MAX_BOX_SIZE_MAX)) {
    pout() ≪ "\n\tERROR(build_levels):␣max_box_size␣out␣of␣range" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
  if (¬is_a_pwr_of_two(max_box_size)) {
    pout() ≪ "\n\tERROR(build_levels):␣max_box_size␣not␣a␣power␣of␣2" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
```

See also chunk 69.

This code is used in chunk 62.

¶ In this chunk we read array variables.

First we read number of cells in each direction, *n_cells*, which must fall between `N_CELLS_MIN` and `N_CELLS_MAX`, and which must be a power of two as well. Then we read grid constant *delta*. It must be a positive real number. Then we read the *origin* coordinates, no checking is done here.

The statements that follow read PML box corners, low and high, and signal injection box corners, low and high. No checking is done here, because we need to construct the level 0 grid first. We check if these parameters are correct in chunk ⟨ *build_levels*: build level 0 70 ⟩, section 6.1.3, page 146.

69 ⟨ *build_levels*: read grid specifications 68 ⟩ +≡

```
if (level0_parameter_parser.queryarr("cells", n_cells, 0, SpaceDim)) {
  if (watch_build_levels) pout() ≪ "\tread:␣n_cells␣=␣" ≪ n_cells ≪ endl;
  for (int i = 0; i < SpaceDim; i++) {
    if ((n_cells[i] < N_CELLS_MIN) ∨ (n_cells[i] > N_CELLS_MAX)) {
      pout() ≪ "\n\tERROR(build_levels):␣n_cells[" ≪ i ≪ "]␣out␣of␣range" ≪ endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
    if (¬is_a_pwr_of_two(n_cells[i])) {
      pout() ≪ "\n\tERROR(build_levels):␣n_cells[" ≪ i ≪ "]␣not␣a␣power␣of␣2" ≪ endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
```

```
    }
  }
  if (level0_parameter_parser.query("delta", delta)) {
    if (watch_build_levels) pout() ≪ "\tread: delta = " ≪ delta ≪ endl;
    if (delta ≤ 0) {
      pout() ≪ "\n\tERROR(build_levels): delta out of range" ≪ endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
  }
  if (iterate_parameter_parser.query("stride", stride)) {
    if (watch_build_levels) pout() ≪ "\tread: stride = " ≪ stride ≪ endl;
    if (stride < STRIDE) {
      pout() ≪ "\n\tERROR(build_levels): stride out of range" ≪ endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
  }
  if (iterate_parameter_parser.query("t0", t0)) {
    if (watch_build_levels) pout() ≪ "\tread: t0 = " ≪ t0 ≪ endl;
  }
        /* Here we allow any three real numbers. If queryarr doesn't crash, it means the input is correct. */
  if (level0_parameter_parser.queryarr("origin", origin, 0, SpaceDim))
    if (watch_build_levels) pout() ≪ "\tread: origin = " ≪ origin ≪ endl;
  if (pml_parameter_parser.queryarr("lo", pml_xyz_min, 0, SpaceDim))
    if (watch_build_levels) pout() ≪ "\tread: pml.lo = (" ≪ pml_xyz_min ≪ ")" ≪ endl;
  if (pml_parameter_parser.queryarr("hi", pml_xyz_max, 0, SpaceDim))
    if (watch_build_levels) pout() ≪ "\tread: pml.hi = (" ≪ pml_xyz_max ≪ ")" ≪ endl;
  if (signal_parameter_parser.queryarr("lo", signal_xyz_min, 0, SpaceDim))
    if (watch_build_levels) pout() ≪ "\tread: signal.lo = (" ≪ signal_xyz_min ≪ ")" ≪ endl;
  if (signal_parameter_parser.queryarr("hi", signal_xyz_max, 0, SpaceDim))
    if (watch_build_levels) pout() ≪ "\tread: signal.hi = (" ≪ signal_xyz_max ≪ ")" ≪ endl;
```

6.1.3 Build Level 0 Grid

Finally, all input read, we get to build the grid.

We begin by building the level 0 grid. This is done a little differently from building grids for higher levels. It is also simpler.

The first step is to allocate space for it with **new level**. Then we define the domain. It is a box in the **IntVect** space with its low corner located at $(0, 0, 0)$ and its high corner located at $(n\_cells[0], n\_cells[1], n\_cells[2])$. The domain's periodicity is default, i.e., non-periodic.

Having defined the domain, we split it into boxes of maximum side length given by *max_box_size*. Chombo function *domainSplit* does the job, and puts the result on *level_0_ptr→vector_of_boxes*. Chombo function *LoadBalance* is then invoked to distribute the boxes amongst participating MPI processes. The output of the function is *level_0_ptr→vector_of_processes*. With these two vectors, of boxes and processes, we can now define the disjoint box layout for level 0. This is the level 0 grid.

Once we're done with the grid, we fill the remaining parameters of the level, the *origin* and the *delta*.

It may happen that the actual grid constructed by *domainSplit* is not exactly as we have requested. The function may chop the grid here and there, while constructing the boxes. So we want to find out the real grid's low and high corners rather than assume them. This is done by function *effective_grid_bounds*, which is defined in module ⟨ Function *effective_grid_bounds* 150 ⟩, page 223, section 10.3. This is a non-trivial parallel function that involves some inter-process communication. The function fills *level_0_ptr→ijk_min*, *level_0_ptr→ijk_max*, *level_0_ptr→xyz_min* and *level_0_ptr→xyz_max* slots of the level structure.

Once we have returned we check if the PML and signal boxes, as read from the Scheme space, are correctly configured, and make appropriate assignments, if they are.

Eventually, having constructed the level, we append it to the *levels* vector. From this point onwards we refer to it as *levels*[0], or (∗*levels_ptr*)[0] (I wonder if this is going to work..., haven't tried this yet.)

70 ⟨ *build_levels*: build level 0 70 ⟩ ≡

```
level_0_ptr = new level;
level_0_ptr→domain.define(IntVect::Zero,
    (n_cells[0] − 1) ∗ BASISV(0) + (n_cells[1] − 1) ∗ BASISV(1) + (n_cells[2] − 1) ∗ BASISV(2));
domainSplit(level_0_ptr→domain, level_0_ptr→vector_of_boxes, max_box_size);
LoadBalance(level_0_ptr→vector_of_processes, level_0_ptr→vector_of_boxes);
level_0_ptr→box_layout.define(level_0_ptr→vector_of_boxes, level_0_ptr→vector_of_processes);
if (level_0_ptr→box_layout.isDisjoint()) {
  if (watch_build_levels > 1)
    pout() ≪ "\tdisjoint_box_layout␣=␣" ≪ endl ≪ level_0_ptr→box_layout ≪ endl;
}
else {
  pout() ≪ "\n\tERROR(build_levels):␣Failed␣to␣construct␣disjoint␣box␣layout␣for␣level␣0" ≪
      endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
level_0_ptr→origin.resize(SpaceDim);
level_0_ptr→origin = origin;
level_0_ptr→delta = delta;
level_0_ptr→dt = delta/stride;
level_0_ptr→time_e = t0;
level_0_ptr→time_h = t0 + level_0_ptr→dt/2;
level_0_ptr→number = 0;
    /∗ Find xyz_min, xyz_max, ijk_min and ijk_max. The domainSplit function may have changed them a
    little. ∗/
if (return_status = effective_grid_bounds(level_0_ptr)) {
  pout() ≪ "\n\tERROR(build_levels):␣Cannot␣find␣grid␣bounds" ≪ endl;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((pml_xyz_min[i] < level_0_ptr→xyz_min[i] + PML_MARGIN) ∨ (pml_xyz_max[i] >
      level_0_ptr→xyz_max[i] − PML_MARGIN)) {
    pout() ≪ "\n\tERROR(build_levels):␣PML␣margin␣too␣narrow" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
  if ((signal_xyz_min[i] < pml_xyz_min[i] + SIGNAL_MARGIN) ∨ (signal_xyz_max[i] >
      pml_xyz_max[i] − SIGNAL_MARGIN)) {
    pout() ≪ "\n\tERROR(build_levels):␣Signal␣margin␣too␣narrow" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
}
level_0_ptr→pml_xyz_min.resize(SpaceDim);
level_0_ptr→pml_xyz_min = pml_xyz_min;
level_0_ptr→pml_xyz_max.resize(SpaceDim);
level_0_ptr→pml_xyz_max = pml_xyz_max;
level_0_ptr→signal_xyz_min.resize(SpaceDim);
level_0_ptr→signal_xyz_min = signal_xyz_min;
level_0_ptr→signal_xyz_max.resize(SpaceDim);
level_0_ptr→signal_xyz_max = signal_xyz_max;
```

```
    /* Evaluate signal_ijk_lo and signal_ijk_hi. */
  for (int i = 0; i < SpaceDim; i++) {
    level_0_ptr→signal_ijk_lo[i] = (int) floor((signal_xyz_min[i] − origin[i])/delta + 0.5);
    level_0_ptr→signal_ijk_hi[i] = (int) floor((signal_xyz_max[i] − origin[i])/delta + 0.5);
  }
  levels.clear( );
  levels.push_back(level_0_ptr);
  if (watch_build_levels) {
    pout( ) ≪ "\tdomain␣=␣" ≪ "[" ≪ levels[0]→ijk_min[0] ≪ "," ≪ levels[0]→ijk_max[0] ≪ "]" ≪
        "␣x␣" ≪ "[" ≪ levels[0]→ijk_min[1] ≪ "," ≪ levels[0]→ijk_max[1] ≪ "]" ≪ "␣x␣" ≪ "[" ≪
        levels[0]→ijk_min[2] ≪ "," ≪ levels[0]→ijk_max[2] ≪ "]" ≪ endl;
    pout( ) ≪ "\tnumber␣of␣boxes␣=␣" ≪ levels[0]→box_layout.size( ) ≪ endl;
  }
```

This code is cited in chunks 19 and 69.

This code is used in chunk 62.

6.1.4 Build Higher Levels

Higher levels are now built iteratively within the **for** loop, rather than recursively, as was the case in SHAPES. And here is the loop. The loop lives inside an **if** statement, that checks if higher levels need to be built at all, and if we have adequate specifications provided to do so.

The latter check is simple: we just check if there is a bound variable called `"forms.grid.lambda"` defined in the Scheme space and if it's a procedure. Both are accomplished by the one call to the *isProcedure* method of the **SCMParmParse** class.

Assuming that we are to build higher levels and have some non-trivial instructions provided in the lambda, we read the basic refinement parameters from the Scheme space. These are *refine_fill_ratio*, *refine_block_factor*, *refine_buffer_size* and *refine_max_size*, all defined in chunk ⟨ *build_levels*: variable definitions 63 ⟩, page 142, section 6.1.1.

We do not check for their correctness here, which is an omission that will need to be rectified in future.

Then we truncate the three basic vectors that will be used in building subgrids: *vector_of_tag_sets*, *vector_of_refinements* and *vector_of_vectors_of_boxes*, and, at long last, we enter the loop. What happens inside it is deferred to module ⟨ *build_levels*: build next level 72 ⟩, page 148, section 6.1.4.

Since the vectors have not been used so far, they probably don't need to be truncated, because they're made of zero length when defined. So this is a "just in case" precaution.

71 ⟨ *build_levels*: build higher levels 71 ⟩ ≡

```
  if (number_of_levels > 1) {
    if (grid_parameter_parser.isProcedure("lambda")) {
      grid_parameter_parser.query("fill_ratio", refine_fill_ratio);
      grid_parameter_parser.query("block_factor", refine_block_factor);
      grid_parameter_parser.query("buffer_size", refine_buffer_size);
      grid_parameter_parser.query("max_size", refine_max_size);
      vector_of_tag_sets.clear( );
      vector_of_refinements.clear( );
      vector_of_vectors_of_boxes.clear( );
      for (int level_number = 1; level_number < number_of_levels; level_number++) {
        if (watch_build_levels) pout( ) ≪ "\n\tbuilding␣level␣" ≪ level_number ≪ endl;
        ⟨ build_levels: build next level 72 ⟩
      }
    }
    else {
      pout( ) ≪ "\n\tERROR(build_levels):␣no␣lambda" ≪ endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
```

}

This code is used in chunk 62.

Build Next Level

Here the fun begins.

The way a next level grid is made is as follows. First we tag selected cells of a current grid for refinement. The tagged cells are arranged into a set and this set is passed to a special machine, called **BRMeshRefine**, together with the grid specifications up to the current level. The machine makes the next level grid and returns it as a vector of boxes, because this is how a grid is specified in Chombo.

Now, the trick of this chunk is in the tagging. Tagging in SHAPES was a very clumsy process, which apart from being complicated was also inflexible. Tagging in FORMS is done by a Scheme closure, `"forms.grid.lambda"`, which user have to provide. The FORMS code is much simpler and much shorter, and yet even the most complex subgrids may be constructed.

Before we can invoke Scheme though, we have to prepare a few things.

First we set the global pointer *current_level_ptr* to point to the last level fully defined so far. The C++ routines that Scheme will invoke, communicate with the Chombo data structures by the means of this pointer.

Next we define the *tags* field, which the Scheme program will draw on. It is simply a field of integers defined on the last level's disjoint box layout. The *tags* field is then initialized to zero.

And now we simply ask Scheme to execute `"grid.level.lambda"` on the current level. As the Scheme closure is called on every **for** loop iteration *current_level_ptr* is updated to point to the last level fully built. It is on this level that we draw the outline for the next level.

Every Scheme evaluation effected by *scm_call_1* returns some Scheme value, which here we capture on *lambda_return*. These are all of type **SCM** in the C++ program that calls Scheme. The user closure may return anything, but here we check for just one possibility, namely that the returned value is boolean and *false*. This we interpret as a *failure* flag and on having encountered it here, we return to the caller raising an error.

Scheme does its job silently, unless explicitly asked to print something with *display* or *write*. But let us note that anything that Scheme prints will *not* go into the *pout*( ) output stream. To put Scheme output there, it must be translated into C strings first, then printed with *pout*( ).

There have been numerous changes to the Guile interface. This program assumes version 1.8.1 or later of Guile. In this version the general rule is

to Scheme *from* C use *scm_from* functions

from Scheme *to* C use *scm_to* functions

For example, *scm_from_int* takes a C integer and converts it to a Scheme integer. When calling *scm_call_1* we must ensure that the second argument is of the **SCM** type, exactly as it would be in the `(forms.grid.lambda ...)` statement of Scheme. We cannot just say *scm_call_0*(*lambda*, *level_number*).

In earlier versions of Guile, for example, 1.6.X, *scm_int2num* was used in place of *scm_from_int*. These older interfaces are now deprecated.

72 ⟨ *build_levels*: build next level 72 ⟩ ≡

```
/* current_level_ptr must be a global variable, because this is the simplest way to pass it to Scheme C procedures. */
current_level_ptr = levels[level_number − 1];
LevelData⟨BaseFab⟨int⟩⟩ &tags = current_level_ptr→tags;
DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
DataIterator data_iterator(box_layout);
    /* Create the tag field over the current level's box layout and initialize it to zero */
if (watch_build_levels) pout( ) ≪ "\tcreating␣tags␣LevelData" ≪ endl;
tags.define(box_layout, 1, 0 ∗ IntVect :: Unit);
if (tags.isDefined( )) {
  for (data_iterator.reset( ); data_iterator.ok( ); ++data_iterator) {
    tags[data_iterator( )].setVal(0, 0);
  }
```

```
  }
  else {
    pout() ≪ "\n\tERROR(build_levels): Failed to define tags field" ≪ endl;
    return_status = ERR_BUILD_LEVELS;
    goto exit;
  }
        /* Invoke Scheme to draw on the tags field. By now we know that forms.grid.lambda exists and is a
        closure. */
  SCM lambda = scm_variable_ref(scm_c_lookup("forms.grid.lambda"));
  SCM lambda_return = scm_call_1(lambda, scm_from_int(level_number));
  if (scm_is_bool(lambda_return)) {
    if (scm_is_false(lambda_return)) {
      pout() ≪ "\n\tERROR(build_levels): Lambda returned #f" ≪ endl;
      return_status = ERR_BUILD_LEVELS;
      goto exit;
    }
  }
```

See also chunk 73.

This code is cited in chunk 71.

This code is used in chunk 71.

¶ Once we have returned from Scheme, there should be something drawn on *current_level_ptr→tags*, which is a **LevelData⟨BaseFab⟨int⟩⟩**. "Something drawn", means that there will be ones in the *tags* field in some cells—this means drawing, whereas *tags* in other cells will remain zero.

But **BRMeshRefine** does not work with a **LevelData⟨BaseFab⟨int⟩⟩**. It wants a vector of **IntVectSet**s instead. So we need to convert the data on *tags* to an **IntVectSet** *tag_set*.

This is almost as tricky as finding the effective grid bounds and requires some communication between the MPI processes too. Function *make_tag_set*, defined in section 10.4, page 225, module ⟨ Function *make_tag_set* 158 ⟩, performs the conversion. So here we just call it and then proceed.

When *make_tag_set* returns, there should be something in *current_level_ptr→tag_set*, so we append it to *vector_of_tag_sets*. We append another *refine_ratio* to *vector_of_refinments* and *current_level_ptr→vector_of_boxes* to *vector_of_vectors_of_boxes*.

Now we are ready to bring the **BRMeshRefine** machine, and once it's created we ask it to refine the current level only. There is no need to redo the previously constructed subgrids. This is done by calling a **BRMeshRefine**::*regrid* method.

The method is quite slow. It may take a long while for it to return. From my simple tests just about all in this chunk runs like a rocket, but the code visibly halts on **BRMeshRefine**::*regrid*.

When **BRMeshRefine**::*regrid* returns finally, we must first check that it has done anything for starters. Has it created the next level? We check this by inspecting the size of *new_vector_of_vectors_of_boxes*. This vector should be longer by one entry than *vector_of_vectors_of_boxes*. But then we must also check that the last vector in *new_vector_of_vectors_of_boxes*, the one that would define the new level, is not of length zero.

These checks passed, we may be confident that **BRMeshRefine**::*regrid* has produced a valid new level subgrid, so at this stage we create the new level with *new_level_ptr* = **new level**. We have the vector of boxes for it, waiting in *new_vector_of_vectors_of_boxes*[*level_number*], but we must still distribute it amongst the MPI processes. We call function *LoadBalance* to do this, and then we define the **DisjointBoxLayout** of the new level.

In case this operation bumps out, we still check if the produced box layout is indeed disjoint and exit raising an error flag if it is not.

The new level is almost complete. We give it its level number, its domain, we tell it about the PML and signal boxes. We calculate its *delta* and its *origin*. The origin will have shifted according to the formula

$$\boldsymbol{\Delta}^{\text{new}} = \frac{\boldsymbol{\Delta}^{\text{current}}}{n_r} \tag{355}$$

$$\boldsymbol{r}_0^{\text{new}} = \boldsymbol{r}_0^{\text{current}} - \frac{n_r - 1}{2}\boldsymbol{\Delta}^{\text{new}}, \tag{356}$$

where n_r is the refinement ratio, $\boldsymbol{r}_0$ is the origin vector, and $\boldsymbol{\Delta}$ is the grid constant vector.

Figure 4 shows how the origin of the grid moves for $n_r = 4$. We see that we have to move it 3 halves of the refined lattice constant down and to the left. For $n_r = 2$ it would be only one half.

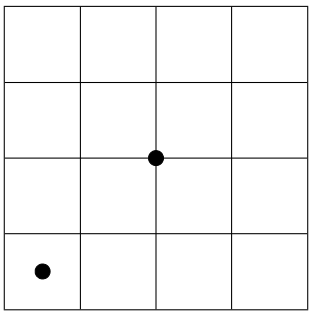


Figure 4: Movement of the grid origin on grid refinement for $n_r = 4$.

Finally, we find the real bounds of the new grid, both in the **IntVect** and in the physical space by calling *effective_grid_bounds*, and append the newly created level to the *levels* vector. From now on, we will refer to it by *levels*[*level_number*].

And this is it. This is how we make the multigrid.

```
73 ⟨ build_levels: build next level 72 ⟩ +≡
   make_tag_set(current_level_ptr);
   if (watch_build_levels) pout() ≪ "\tcreating␣vectors" ≪ endl;
   vector_of_tag_sets.push_back(current_level_ptr→tag_set);
   vector_of_refinements.push_back(refine_ratio);
   vector_of_vectors_of_boxes.push_back(current_level_ptr→vector_of_boxes);
   if (watch_build_levels) pout() ≪ "\tcreating␣BRMeshRefine" ≪ endl;
   BRMeshRefine mesh_refine(levels[0]→domain, vector_of_refinements, refine_fill_ratio, refine_block_factor,
       refine_buffer_size, refine_max_size);
   new_vector_of_vectors_of_boxes.clear();
   refine_base_level = current_level_ptr→number;
   refine_top_level = current_level_ptr→number;
   if (watch_build_levels) pout() ≪ "\tregriding" ≪ endl;
   mesh_refine.regrid(new_vector_of_vectors_of_boxes, vector_of_tag_sets, refine_base_level, refine_top_level,
       vector_of_vectors_of_boxes);
   if (new_vector_of_vectors_of_boxes.size() ≤ vector_of_vectors_of_boxes.size()) {
     pout() ≪ "\n\tERROR(build_levels):␣failed␣to␣generate␣vector␣of␣boxes␣for␣level␣" ≪
         level_number ≪ endl;
     return_status = ERR_BUILD_LEVELS;
     goto exit;
   }
   if (new_vector_of_vectors_of_boxes[level_number].size() ≡ 0) {
     pout() ≪ "\n\tERROR(build_levels):␣empty␣vector␣of␣boxes␣for␣level␣" ≪ level_number ≪ endl;
     return_status = ERR_BUILD_LEVELS;
     goto exit;
   }
   if (watch_build_levels) pout() ≪ "\tcreating␣new␣level" ≪ endl;
   new_level_ptr = new level;
   new_level_ptr→vector_of_boxes = new_vector_of_vectors_of_boxes[level_number];
   if (watch_build_levels) pout() ≪ "\tbalancing␣load" ≪ endl;
   LoadBalance(new_level_ptr→vector_of_processes, new_level_ptr→vector_of_boxes);
   if (watch_build_levels) pout() ≪ "\tcreating␣disjoint␣box␣layout" ≪ endl;
   new_level_ptr→box_layout.define(new_level_ptr→vector_of_boxes, new_level_ptr→vector_of_processes);
```

```
if (¬new_level_ptr→box_layout.isDisjoint( )) {
  pout( ) ≪ "\n\tERROR(build_levels):␣failed␣to␣construct␣disjoint␣box␣layout␣for␣level␣" ≪
      level_number ≪ endl;
  return_status = ERR_BUILD_LEVELS;
  goto exit;
}
new_level_ptr→number = level_number;
if (watch_build_levels) pout( ) ≪ "\trefining␣domain" ≪ endl;
new_level_ptr→domain = refine(current_level_ptr→domain, refine_ratio);
if (watch_build_levels) pout( ) ≪ "\tsmall␣items" ≪ endl;
new_level_ptr→pml_xyz_min.resize(SpaceDim);
new_level_ptr→pml_xyz_max.resize(SpaceDim);
new_level_ptr→signal_xyz_min.resize(SpaceDim);
new_level_ptr→signal_xyz_max.resize(SpaceDim);
new_level_ptr→pml_xyz_min = current_level_ptr→pml_xyz_min;
new_level_ptr→pml_xyz_max = current_level_ptr→pml_xyz_max;
new_level_ptr→signal_xyz_min = current_level_ptr→signal_xyz_min;
new_level_ptr→signal_xyz_max = current_level_ptr→signal_xyz_max;
new_level_ptr→delta = current_level_ptr→delta/refine_ratio;
new_level_ptr→dt = current_level_ptr→dt/(refine_ratio + 1);
new_level_ptr→time_e = t0;
new_level_ptr→time_h = t0 + new_level_ptr→dt/2;
new_level_ptr→origin.resize(SpaceDim);
for (int i = 0; i < SpaceDim; i++) {
  new_level_ptr→origin[i] = current_level_ptr→origin[i] − (refine_ratio − 1) ∗ new_level_ptr→delta/2.0;
}
effective_grid_bounds(new_level_ptr);
if (watch_build_levels) pout( ) ≪ "\tappending␣to␣the␣levels␣array" ≪ endl;
levels.push_back(new_level_ptr);
```

6.2 Fill Levels

Fill levels with data: initialize all arrays to zero, then invoke Scheme procedure to draw media distribution.

74 ⟨ Function *fill_levels* 74 ⟩ ≡

```
int fill_levels(Vector⟨level ∗⟩ &levels) { int return_status = EXIT_SUCCESS;
    int watch_fill_levels = false;
    int print_pml_arrays = false;
    int number_of_levels = N_LEVELS;
    int auxiliaries = N_AUXILIARIES;
    int number_of_precompute_fields = 0;
    int number_of_write_fields = 0;
    IntVect ghost_margin = N_GHOST_CELLS ∗ IntVect::Unit;
        /∗ Parsers used by the function ∗/
    SCMParmParse watch_parameter_parser("forms.watch");
    SCMParmParse grid_parameter_parser("forms.grid");
    SCMParmParse media_parameter_parser("forms.media");
    SCMParmParse signal_parameter_parser("forms.signal");
    SCMParmParse output_parameter_parser("forms.output");
    SCMParmParse pml_parameter_parser("forms.pml");
        /∗ Global variables ∗/
    extern level ∗current_level_ptr;
```

See also chunks 75, 76, 77, 78, 82, and 83.

This code is cited in chunk 18.

This code is used in chunk 61.

75 ¶⟨ Function *fill_levels* 74 ⟩ +≡

```
    /∗ Action ∗/
watch_parameter_parser.query("fill_levels", watch_fill_levels);
if (watch_fill_levels) pout() ≪ "fill_levels:" ≪ endl;
if (grid_parameter_parser.query("number_of_levels", number_of_levels)) {
  if (watch_fill_levels) pout() ≪ "\tread:␣number_of_levels␣=␣" ≪ number_of_levels ≪ endl;
  if ((number_of_levels < N_LEVELS_MIN) ∨ (number_of_levels > N_LEVELS_MAX)) {
    pout() ≪ "\n\tERROR(fill_levels):␣number_of_levels␣out␣of␣range" ≪ endl;
    return_status = ERR_FILL_LEVELS;
    goto exit;
  }
}
if (media_parameter_parser.query("number_of_auxiliary_fields", auxiliaries)) {
  if (watch_fill_levels) pout() ≪ "\tread:␣number_of_auxiliary_fields␣=␣" ≪ auxiliaries ≪ endl;
  if ((auxiliaries < N_AUXILIARIES_MIN) ∨ (auxiliaries > N_AUXILIARIES_MAX)) {
    pout() ≪ "\n\tERROR(fill_levels):␣number_of_auxiliary_fields␣out␣of␣range" ≪ endl;
    return_status = ERR_FILL_LEVELS;
    goto exit;
  }
}
```

76 ¶⟨ Function *fill_levels* 74 ⟩ +≡

```
    /∗ Allocate arrays for the data. ∗/
for (int n = 0; n < number_of_levels; n++) {
  DisjointBoxLayout &level_box_layout = levels[n]→box_layout;
      /∗ Iterators ∗/
      /∗ LayoutIterator contains all boxes of the layout on every processor, whether they belong to it or
      not, whereas DataIterator contains only the boxes that are managed by a given processor. ∗/
```

```
  levels[n]→data_iterator = level_box_layout.dataIterator( );
  levels[n]→layout_iterator = level_box_layout.layoutIterator( );
      /* CurlBox fields, D and E. Each field is defined with two components. Component 0 is for the new
      field and component 1 is for the old one. */
  if (watch_fill_levels) pout( ) ≪ "\tlevel␣" ≪ n ≪ ":" ≪ endl;
  if (watch_fill_levels) pout( ) ≪ "\t\tdefining␣E" ≪ flush;
  levels[n]→E.define(level_box_layout, 2, ghost_margin);
  if (watch_fill_levels) pout( ) ≪ "␣D" ≪ flush;
  levels[n]→D.define(level_box_layout, 2, ghost_margin);
      /* Auxiliary fields. The user specifies how many components will be needed. */
  if (auxiliaries) {
    if (watch_fill_levels) pout( ) ≪ "␣S" ≪ flush;
    levels[n]→S.define(level_box_layout, auxiliaries, ghost_margin);
  }
  if (watch_fill_levels) pout( ) ≪ "␣...␣done." ≪ endl ≪ flush;
      /* From this point onwards we can find about the number of auxiliary fields by invoking the
      levels[n]→S.nComp( ) method. This is actually a method inherited from the BoxLayoutData class. */
      /* EdgeFab⟨int⟩ fields */
  if (watch_fill_levels) pout( ) ≪ "\t\tdefining␣medium_E" ≪ flush;
  levels[n]→medium_E.define(level_box_layout, 1, ghost_margin);
      /* FaceFab⟨int⟩ fields */
  if (watch_fill_levels) pout( ) ≪ "␣medium_H" ≪ flush;
  levels[n]→medium_H.define(level_box_layout, 1, ghost_margin);
  if (watch_fill_levels) pout( ) ≪ "␣...␣done." ≪ endl ≪ flush;
      /* FluxBox fields */
  if (watch_fill_levels) pout( ) ≪ "\t\tdefining␣H" ≪ flush;
  levels[n]→H.define(level_box_layout, 2, ghost_margin);
  if (watch_fill_levels) pout( ) ≪ "␣B" ≪ flush;
  levels[n]→B.define(level_box_layout, 2, ghost_margin);
  if (watch_fill_levels) pout( ) ≪ "␣...␣done." ≪ endl ≪ flush;
      /* Output fields */
  number_of_write_fields = output_parameter_parser.countval("write");
  number_of_precompute_fields = output_parameter_parser.countval("precompute");
  if (number_of_write_fields ∧ number_of_precompute_fields) {
      /* These fields have no margin, because they are not exchanged between CPUs */
    if (watch_fill_levels) pout( ) ≪ "\t\tdefining␣output␣fields" ≪ flush;
    levels[n]→Out.define(level_box_layout, number_of_precompute_fields, IntVect::Zero);
    levels[n]→OutWrite.define(level_box_layout, number_of_write_fields, IntVect::Zero);
    if (watch_fill_levels) pout( ) ≪ "␣...␣done." ≪ endl ≪ flush;
  }
  if (watch_fill_levels) {
    pout( ) ≪ "\t\tE:␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→E.nComp( ) ≪ endl;
    pout( ) ≪ "\t\tD:␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→D.nComp( ) ≪ endl;
    pout( ) ≪ "\t\tH:␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→H.nComp( ) ≪ endl;
    pout( ) ≪ "\t\tB:␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→B.nComp( ) ≪ endl;
    if (auxiliaries) pout( ) ≪ "\t\tS:␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→S.nComp( ) ≪ endl;
    pout( ) ≪ "\t\tmedia␣␣#␣of␣components␣=␣" ≪ levels[n]→medium_E.nComp( ) ≪ endl;
    pout( ) ≪ "\t\t␣␣␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→medium_H.nComp( ) ≪ endl;
    if (number_of_precompute_fields ∧ number_of_write_fields) {
      pout( ) ≪ "\t\tOut:␣␣␣#␣of␣components␣=␣" ≪ levels[n]→Out.nComp( ) ≪ endl;
      pout( ) ≪ "\t\t␣␣␣␣␣␣␣#␣of␣components␣=␣" ≪ levels[n]→OutWrite.nComp( ) ≪ endl;
    }
  }
}     /* first loop over levels–allocations */
```

77 ¶⟨ Function *fill_levels* 74 ⟩ +≡

```
    /* Initialize the arrays */
  for (int n = 0; n < number_of_levels; n++) {
    DataIterator &data_iterator = levels[n]->data_iterator;
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      levels[n]->E[data_iterator()].setVal((Real) ZERO);
      levels[n]->D[data_iterator()].setVal((Real) ZERO);
      levels[n]->H[data_iterator()].setVal((Real) ZERO);
      levels[n]->B[data_iterator()].setVal((Real) ZERO);
      levels[n]->medium_E[data_iterator()].setVal((int) ZERO);
      levels[n]->medium_H[data_iterator()].setVal((int) ZERO);
      if (auxiliaries) levels[n]->S[data_iterator()].setVal((Real) ZERO);
      if (number_of_precompute_fields ∧ number_of_write_fields) {
        levels[n]->Out[data_iterator()].setVal((Real) ZERO);
        levels[n]->OutWrite[data_iterator()].setVal((Real) ZERO);
      }
    }
    for (int count = 0; count < 2; count++) {
      levels[n]->E_idx[count] = count;
      levels[n]->D_idx[count] = count;
      levels[n]->H_idx[count] = count;
      levels[n]->B_idx[count] = count;
    }
    if (watch_fill_levels) {
      pout() << "\tlevel " << n << ":" << endl;
      pout() << "\t\tAll Fabs initialized to zero." << endl;
    }
  }     /* second loop over levels–initializations */
```

¶ Here we attend to certain structures that are needed for level 0 calculations. Level 0 is different, because it has PMLs and signal injection.

78 ⟨ Function *fill_levels* 74 ⟩ +≡

```
    /* On level 0 we are going to mark UPML and scattered field regions to facilitate D to E and B to H
    conversions. */
  if (return_status = mark_regions(levels[0])) {
    pout() << "\n\tERROR(fill_levels): mark_regions failed" << endl;
    goto exit;
  }
  if (return_status = mark_TFR_faces(levels[0])) {
    pout() << "\n\tERROR(fill_levels:) mark_TFR_faces failed" << endl;
    goto exit;
  }
  if (return_status = fill_PML_arrays(levels[0])) {
    pout() << "\n\tERROR(fill_levels): fill_PML_arrays failed" << endl;
    goto exit;
  }
  pml_parameter_parser.query("print_arrays", print_pml_arrays);
  if (print_pml_arrays) {
    pout() << "\n\tPrinting PML arrays" << endl;
    ⟨ print Ckx arrays 79 ⟩
    ⟨ print Cky arrays 80 ⟩
    ⟨ print Ckz arrays 81 ⟩
  }
```

79 ¶⟨ print *Ckx* arrays 79 ⟩ ≡

```
pout( ) ≪ "\tC0x:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C0x.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C0x[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC1x:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C1x.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C1x[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC2x:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C2x.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C2x[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC3x:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C3x.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C3x[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC4x:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C4x.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C4x[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
```

This code is used in chunk 78.

80 ¶⟨ print *Cky* arrays 80 ⟩ ≡

```
pout( ) ≪ "\tC0y:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C0y.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C0y[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC1y:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C1y.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C1y[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC2y:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C2y.size( ); count++) {
  pout( ) ≪ "␣" ≪ levels[0]→C2y[count];
  if (¬(count % 6)) pout( ) ≪ endl ≪ "\t\t";
}
pout( ) ≪ endl;
pout( ) ≪ "\tC3y:" ≪ endl ≪ "\t\t";
for (int count = 0; count < levels[0]→C3y.size( ); count++) {
```

```
    pout() ≪ "␣" ≪ levels[0]→C3y[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
  pout() ≪ "\tC4y:" ≪ endl ≪ "\t\t";
  for (int count = 0; count < levels[0]→C4y.size(); count++) {
    pout() ≪ "␣" ≪ levels[0]→C4y[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
```

This code is used in chunk 78.

81 ¶⟨ print *Ckz* arrays 81 ⟩ ≡

```
  pout() ≪ "\tC0z:" ≪ endl ≪ "\t\t";
  for (int count = 0; count < levels[0]→C0z.size(); count++) {
    pout() ≪ "␣" ≪ levels[0]→C0z[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
  pout() ≪ "\tC1z:" ≪ endl ≪ "\t\t";
  for (int count = 0; count < levels[0]→C1z.size(); count++) {
    pout() ≪ "␣" ≪ levels[0]→C1z[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
  pout() ≪ "\tC2z:" ≪ endl ≪ "\t\t";
  for (int count = 0; count < levels[0]→C2z.size(); count++) {
    pout() ≪ "␣" ≪ levels[0]→C2z[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
  pout() ≪ "\tC3z:" ≪ endl ≪ "\t\t";
  for (int count = 0; count < levels[0]→C3z.size(); count++) {
    pout() ≪ "␣" ≪ levels[0]→C3z[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
  pout() ≪ "\tC4z:" ≪ endl ≪ "\t\t";
  for (int count = 0; count < levels[0]→C4z.size(); count++) {
    pout() ≪ "␣" ≪ levels[0]→C4z[count];
    if (¬(count % 6)) pout() ≪ endl ≪ "\t\t";
  }
  pout() ≪ endl;
```

This code is used in chunk 78.

82 ¶⟨ Function *fill_levels* 74 ⟩ +≡

```
    /* Now we have to draw media on levels[n]→medium by invoking the appropriate Scheme procedure. */
  if (media_parameter_parser.isProcedure("distribution.lambda")) {
    SCM lambda = scm_variable_ref(scm_c_lookup("forms.media.distribution.lambda"));
    for (int n = 0; n < number_of_levels; n++) {
      current_level_ptr = levels[n];
      SCM lambda_return = scm_call_0(lambda);
```

```
      if (scm_is_bool(lambda_return)) {
        if (scm_is_false(lambda_return)) {
          pout( ) ≪ "\n\tERROR(build_levels):␣Lambda␣returned␣#f" ≪ endl;
          return_status = ERR_FILL_LEVELS;
          goto exit;
        }
      }
    }
  }
  else {
    pout( ) ≪ "\n\tWARNING(build_levels):␣distribution␣lambda␣not␣provided" ≪ endl;
        /* This is not an error—users may wish to define media within the d_to_e script, or just propagate in
        vacuum. */
  }
```

83 ¶⟨ Function *fill_levels* 74 ⟩ +≡

```
exit:
  if (watch_fill_levels) pout( ) ≪ "fill_levels:␣returning␣to␣caller" ≪ endl;
  return return_status; }
```

7 Iteration

Here we define the content of file *Iteration.c*. Functions used in the iteration loop are defined here.

84 ⟨ `Iteration.c` 84 ⟩ ≡

```
#include <Forms.H>
```

```
    /* Evolving the D and E fields */
```

⟨ Function *advance_d* 85 ⟩
⟨ Function *advance_d_0* 86 ⟩
⟨ Function *advance_d_n* 87 ⟩
⟨ Function *inject_d* 88 ⟩
⟨ Function *d_to_e* 95 ⟩
⟨ Function *d_to_e_0* 96 ⟩
⟨ Function *d_to_e_0_0* 97 ⟩
⟨ Function *d_to_e_0_n* 98 ⟩
⟨ Function *d_to_e_n* 99 ⟩
⟨ Function *d_to_e_n_0* 100 ⟩
⟨ Function *d_to_e_n_n* 101 ⟩

```
    /* Evolving the B and H fields */
```

⟨ Function *advance_b* 105 ⟩
⟨ Function *advance_b_0* 106 ⟩
⟨ Function *advance_b_n* 107 ⟩
⟨ Function *inject_b* 108 ⟩
⟨ Function *b_to_h* 111 ⟩
⟨ Function *b_to_h_0* 112 ⟩
⟨ Function *b_to_h_n* 113 ⟩

7.1 Advance the D field

The main function here is just a simple wrapper that checks the level number and calls a more detailed wrapper. Level zero requires a more involved advance than a higher level, because here we need to attend to UPMLs and inject the signal.

Consequently, whereas a higher level advance is given by the relatively simple equations (85) through (87), level zero advance is given by the UPML equations (215) through (217), which have to be combined with the signal injection equations (138) through (161), and the additional UPML $\boldsymbol{D}$-to-$\boldsymbol{E}$ conversion equations (221) through (222).

85 ⟨ Function *advance_d* 85 ⟩ ≡

```
int advance_d(level *level_ptr)
{
  int watch_advance_d = false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_d", watch_advance_d);
  if (watch_advance_d) pout() ≪ "advance_d: level #" ≪ level_number ≪ endl;
  if (level_number ≡ 0)      /* level 0: special advance is needed here */
  {
    if (watch_advance_d) pout() ≪ "\tCalling advance_d_0" ≪ endl;
    if (return_status = advance_d_0(level_ptr)) {
      pout() ≪ "\tERROR(advance_d): Failed to advance level 0" ≪ endl;
      goto exit;
    }
  }
  else      /* not level 0: simple advance */
  {
    if (watch_advance_d) pout() ≪ "\tCalling advance_d_n" ≪ endl;
    if (return_status = advance_d_n(level_ptr)) {
      pout() ≪ "\tERROR(advance_d): Failed to advance level " ≪ level_number ≪ endl;
      goto exit;
    }
  }
  level_ptr→time_e += level_ptr→dt;
exit: return return_status;
}
```

This code is used in chunk 84.

7.1.1 Advance Level Zero (with UPML)

86 ⟨ Function *advance_d_0* 86 ⟩ ≡

```
int advance_d_0(level *level_ptr)
{
  int watch_advance_d = false;
  int watch_update_d = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator &data_iterator = level_ptr→data_iterator;
  int Cx_lo = -ghost_margin;
  int Cx_hi = (int) level_ptr→C1x.size() - ghost_margin;
  int Cy_lo = -ghost_margin;
  int Cy_hi = (int) level_ptr→C1y.size() - ghost_margin;
```

```
int Cz_lo = -ghost_margin;
int Cz_hi = (int) level_ptr->C1z.size() - ghost_margin;
Interval current_component;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("advance_d", watch_advance_d);
watch_parameter_parser.query("update_d", watch_update_d);
if (watch_advance_d) pout() << "\tadvance_d_0:" << endl;
if (level_number != 0) {
  pout() << "\t\tERROR(advance_d_0): This is not level zero" << endl;
  return_status = ERR_ADVANCE_D;
  goto exit;
}
current_component.define(level_ptr->H_idx[0], level_ptr->H_idx[0]);
level_ptr->H.exchange(current_component);
swap_idx(level_ptr->D_idx);      /* swap the new/old fields */
if (watch_advance_d > 1) pout() << "\t\tentering the box loop" << endl << flush;
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  FArrayBox &Dx = level_ptr->D[data_index][0];
  FArrayBox &Dy = level_ptr->D[data_index][1];
  FArrayBox &Dz = level_ptr->D[data_index][2];
  FArrayBox &Hx = level_ptr->H[data_index][0];
  FArrayBox &Hy = level_ptr->H[data_index][1];
  FArrayBox &Hz = level_ptr->H[data_index][2];
  Vector<Real> &C1x = level_ptr->C1x;
  Vector<Real> &C2x = level_ptr->C2x;
  Vector<Real> &C1y = level_ptr->C1y;
  Vector<Real> &C2y = level_ptr->C2y;
  Vector<Real> &C1z = level_ptr->C1z;
  Vector<Real> &C2z = level_ptr->C2z;
  if (watch_advance_d > 1)
    pout() << "\t\t\tcalling update_d_upml_ on box " << Dx.box() << " ... " << flush;
  update_d_upml_(
  &SpaceDim, &ghost_margin, &(level_ptr->D_idx[0]), &(level_ptr->H_idx[0]),
  &Cx_lo, &Cx_hi, &C1x[0], &C2x[0],
  &Cy_lo, &Cy_hi, &C1y[0], &C2y[0],
  &Cz_lo, &Cz_hi, &C1z[0], &C2z[0],
  &(Dx.loVect()[0]), &(Dx.loVect()[1]), &(Dx.loVect()[2]), &(Dx.hiVect()[0]), &(Dx.hiVect()[1]),
      &(Dx.hiVect()[2]), Dx.dataPtr(),
  &(Dy.loVect()[0]), &(Dy.loVect()[1]), &(Dy.loVect()[2]), &(Dy.hiVect()[0]), &(Dy.hiVect()[1]),
      &(Dy.hiVect()[2]), Dy.dataPtr(),
  &(Dz.loVect()[0]), &(Dz.loVect()[1]), &(Dz.loVect()[2]), &(Dz.hiVect()[0]), &(Dz.hiVect()[1]),
      &(Dz.hiVect()[2]), Dz.dataPtr(),
  &(Hx.loVect()[0]), &(Hx.loVect()[1]), &(Hx.loVect()[2]), &(Hx.hiVect()[0]), &(Hx.hiVect()[1]),
      &(Hx.hiVect()[2]), Hx.dataPtr(),
  &(Hy.loVect()[0]), &(Hy.loVect()[1]), &(Hy.loVect()[2]), &(Hy.hiVect()[0]), &(Hy.hiVect()[1]),
      &(Hy.hiVect()[2]), Hy.dataPtr(),
  &(Hz.loVect()[0]), &(Hz.loVect()[1]), &(Hz.loVect()[2]), &(Hz.hiVect()[0]), &(Hz.hiVect()[1]),
      &(Hz.hiVect()[2]), Hz.dataPtr(),
  &watch_update_d, &return_status
  );
  if (watch_advance_d > 1) pout() << "done." << endl << flush;
  if (return_status != EXIT_SUCCESS) {
    pout() << "\t\tERROR(advance_d_0): update_d_upml_ returned " << return_status << endl;
```

```
        goto exit;
      }
    }
  exit: return return_status;
  }
```

This code is used in chunk 84.

7.1.2 Advance Higher Levels

87 ⟨ Function *advance_d_n* 87 ⟩ ≡

```
int advance_d_n(level *level_ptr)
{
  int watch_advance_d = false;
  int watch_update_d = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  Real dt_by_dg = (level_ptr→dt)/(level_ptr→delta);
  DataIterator &data_iterator = level_ptr→data_iterator;
  Interval current_component;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_d", watch_advance_d);
  watch_parameter_parser.query("update_d", watch_update_d);
  if (watch_advance_d) pout() ≪ "\tadvance_d_n:" ≪ endl;
  if (level_number ≡ 0) {
    pout() ≪ "\t\tERROR(advance_d_n): invoked on level zero" ≪ endl;
    return_status = ERR_ADVANCE_D;
    goto exit;
  }
  current_component.define(level_ptr→H_idx[0], level_ptr→H_idx[0]);
  level_ptr→H.exchange(current_component);
  swap_idx(level_ptr→D_idx);      /* swap new/old fields */
      /* Loop over boxes */
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    FArrayBox &Dx = level_ptr→D[data_index][0];
    FArrayBox &Dy = level_ptr→D[data_index][1];
    FArrayBox &Dz = level_ptr→D[data_index][2];
    FArrayBox &Hx = level_ptr→H[data_index][0];
    FArrayBox &Hy = level_ptr→H[data_index][1];
    FArrayBox &Hz = level_ptr→H[data_index][2];
    update_d_(
    &SpaceDim, &ghost_margin,
    &(level_ptr→D_idx[0]), &(level_ptr→H_idx[0]),
    &dt_by_dg,
    &(Dx.loVect()[0]), &(Dx.loVect()[1]), &(Dx.loVect()[2]), &(Dx.hiVect()[0]), &(Dx.hiVect()[1]),
        &(Dx.hiVect()[2]), Dx.dataPtr(),
    &(Dy.loVect()[0]), &(Dy.loVect()[1]), &(Dy.loVect()[2]), &(Dy.hiVect()[0]), &(Dy.hiVect()[1]),
        &(Dy.hiVect()[2]), Dy.dataPtr(),
    &(Dz.loVect()[0]), &(Dz.loVect()[1]), &(Dz.loVect()[2]), &(Dz.hiVect()[0]), &(Dz.hiVect()[1]),
        &(Dz.hiVect()[2]), Dz.dataPtr(),
    &(Hx.loVect()[0]), &(Hx.loVect()[1]), &(Hx.loVect()[2]), &(Hx.hiVect()[0]), &(Hx.hiVect()[1]),
        &(Hx.hiVect()[2]), Hx.dataPtr(),
```

```
    &(Hy.loVect()[0]), &(Hy.loVect()[1]), &(Hy.loVect()[2]), &(Hy.hiVect()[0]), &(Hy.hiVect()[1]),
        &(Hy.hiVect()[2]), Hy.dataPtr(),
    &(Hz.loVect()[0]), &(Hz.loVect()[1]), &(Hz.loVect()[2]), &(Hz.hiVect()[0]), &(Hz.hiVect()[1]),
        &(Hz.hiVect()[2]), Hz.dataPtr(),
    &watch_update_d, &return_status
    );
    if (return_status ≠ EXIT_SUCCESS) {
      pout() ≪ "\t\tERROR(advance_d_n): update_d_ returned " ≪ return_status ≪ endl;
      goto exit;
    }
  }
exit: return return_status;
}
```

This code is used in chunk 84.

7.2 Inject the *D* field

88 ⟨ Function *inject_d* 88 ⟩ ≡

```
int inject_d(level *level_ptr)
{
    /* Inject the D field on the total/scattered field boundary. */
  int return_status = EXIT_SUCCESS;
  int watch_inject_d = (int) false;
  int garbage_collect = (int) false;
  SideIterator side_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse signal_parameter_parser("forms.signal");
  watch_parameter_parser.query("inject_d", watch_inject_d);
  signal_parameter_parser.query("garbage_collect", garbage_collect);
  if (watch_inject_d) pout() << "inject_d:" << endl;
  if (level_ptr->number != 0) {
    pout() << "\tERROR(inject_d): not level zero" << endl;
    return_status = ERR_INJECT_D;
    goto exit;
  }
  for (int dir = 0; dir < SpaceDim; dir++) {
    for (side_iterator.reset(); side_iterator.ok(); ++side_iterator) {
      Side::LoHiSide side = side_iterator();
      if (return_status = inject_d(level_ptr, dir, side)) {
        pout() << "\tERROR(inject_d): failed on dir = " << dir << ", side = " << side << endl;
        goto exit;
      }
    }
  }
  if (garbage_collect) scm_gc();
exit: return return_status;
}
```

See also chunk 89.

This code is used in chunk 84.

89 ¶⟨ Function *inject_d* 88 ⟩ +≡

```
int inject_d(level *level_ptr, int dir, Side::LoHiSide side)
{
  int return_status = EXIT_SUCCESS;
  int watch_inject_d = (int) false;
  int side_number = (int) side;
  int ghost_margin = N_GHOST_CELLS;
  IntVect ijk_lo = level_ptr->signal_ijk_lo;
  IntVect ijk_hi = level_ptr->signal_ijk_hi;
  Vector<Real> &origin = level_ptr->origin;
  Real &delta = level_ptr->delta;
  LevelData<CurlBox> &D = level_ptr->D;
  int &new_component = level_ptr->D_idx[0];
  Vector<DataIndex> &side_boxes = level_ptr->TFR_face_boxes[dir][side];
  Real dt_by_delta = level_ptr->dt/level_ptr->delta;
  Real time_h = level_ptr->time_h;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("inject_d", watch_inject_d);
  if (watch_inject_d) pout() << "\tinject_d: dir = " << dir << ", side = " << side << endl;
```

```
if (dir < 0 ∨ dir ≥ SpaceDim) {
  pout( ) ≪ "\t\tERROR(inject_d): bad direction" ≪ endl;
  return_status = ERR_INJECT_D;
  goto exit;
}
for (int count = 0; count < side_boxes.size( ); count++) {
    /* This loop picks up only the boxes that cross this side and that live on this processor. */
  DataIndex data_index = side_boxes[count];
  switch (dir) {
  case 0:     /* West–East */
    {
      FArrayBox &Dy = D[data_index][1];
      FArrayBox &Dz = D[data_index][2];
      IntVect Dy_lo = Dy.smallEnd( );
      IntVect Dy_hi = Dy.bigEnd( );
      IntVect Dz_lo = Dz.smallEnd( );
      IntVect Dz_hi = Dz.bigEnd( );
      inject_d_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_h, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(Dy_lo[0]), &(Dy_lo[1]), &(Dy_lo[2]), &(Dy_hi[0]), &(Dy_hi[1]),
          &(Dy_hi[2]), Dy.dataPtr( ), &(Dz_lo[0]), &(Dz_lo[1]), &(Dz_lo[2]), &(Dz_hi[0]), &(Dz_hi[1]),
          &(Dz_hi[2]), Dz.dataPtr( ), &watch_inject_d, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout( ) ≪ "\tERROR(inject_d): Fortran problem, dir = " ≪ dir ≪ endl;
        goto exit;
      }
    }
    break;
  case 1:     /* Front–Back */
    {
      FArrayBox &Dx = D[data_index][0];
      FArrayBox &Dz = D[data_index][2];
      IntVect Dx_lo = Dx.smallEnd( );
      IntVect Dx_hi = Dx.bigEnd( );
      IntVect Dz_lo = Dz.smallEnd( );
      IntVect Dz_hi = Dz.bigEnd( );
      inject_d_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_h, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(Dx_lo[0]), &(Dx_lo[1]), &(Dx_lo[2]), &(Dx_hi[0]), &(Dx_hi[1]),
          &(Dx_hi[2]), Dx.dataPtr( ), &(Dz_lo[0]), &(Dz_lo[1]), &(Dz_lo[2]), &(Dz_hi[0]), &(Dz_hi[1]),
          &(Dz_hi[2]), Dz.dataPtr( ), &watch_inject_d, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout( ) ≪ "\tERROR(inject_d): Fortran problem, dir = " ≪ dir ≪ endl;
        goto exit;
      }
    }
    break;
  case 2:     /* Bottom–Top */
    {
      FArrayBox &Dx = D[data_index][0];
      FArrayBox &Dy = D[data_index][1];
      IntVect Dx_lo = Dx.smallEnd( );
      IntVect Dx_hi = Dx.bigEnd( );
      IntVect Dy_lo = Dy.smallEnd( );
```

```
        IntVect Dy_hi = Dy.bigEnd( );
        inject_d_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
            &(origin[2]), &delta, &dt_by_delta, &time_h, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
            &(ijk_hi[1]), &(ijk_hi[2]), &(Dx_lo[0]), &(Dx_lo[1]), &(Dx_lo[2]), &(Dx_hi[0]), &(Dx_hi[1]),
            &(Dx_hi[2]), Dx.dataPtr( ), &(Dy_lo[0]), &(Dy_lo[1]), &(Dy_lo[2]), &(Dy_hi[0]), &(Dy_hi[1]),
            &(Dy_hi[2]), Dy.dataPtr( ), &watch_inject_d, &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout( ) ≪ "\tERROR(inject_d): Fortran problem, dir = " ≪ dir ≪ endl;
          goto exit;
        }
      }
      break;
    }     /* switch */
  }     /* for (int count = 0; count < side_boxes.size( ); count++) */
exit: return return_status;
}
```

7.2.1 Incident Field Functions

These functions must be compiled with plain C, because they're called by Fortran. When they are compiled with C++, the loader fails.

For this to work seamlessly with the Makefile, changes must be made to $(CHOMBO_HOME)/*mk*/*Make.rules*, namely, a $(*_lib_config*) (%. *o*) : %. *c* rule must be added in the "rules to build objects for executable programs" section, and, possibly, rules may have to be added to the section that runs *makedepend*. It's rather messy and it is an ommission on the side of Chombo engineers not to have provided them for plain C.

Magnetic Field Injection

These three functions, *hx_inc_*, *hy_inc_*, *hz_inc_*, are called by the Fortran subroutine *inject_d_*. They are basically C wrappers for the Scheme call that implements the injection functions $E_x(\zeta)$, $E_y(\zeta)$ and $E_z(\zeta)$, where $\zeta = \boldsymbol{n} \cdot \boldsymbol{r} - t$. The wrapper implements $\boldsymbol{H} = \boldsymbol{n} \times \boldsymbol{E}(\zeta)$.

In explicit notation this is

$$\begin{aligned} H_x &= n_y E_z - n_z E_y, \\ H_y &= n_z E_x - n_x E_z, \\ H_z &= n_x E_y - n_y E_x. \end{aligned}$$

$\boldsymbol{n}$ is obtained from the main program globals, where it is put by the Scheme initialization routine.

Note that because these three functions are callable from Fortran, the arguments are *addresses* of x, y, z, and t, not their actual values. Same applies to *ex_inc_*, *ey_inc_*, and *ez_inc_*.

91 ⟨ Injection.c_std 91 ⟩ ≡

```
#include <REAL.H>
#include <libguile.h>
#include "./Injection.h"
Real hx_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
  extern Real ny, nz;
  return (ny * ez_inc_(x, y, z, t) − nz * ey_inc_(x, y, z, t));
}
Real hy_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
  extern Real nx, nz;
```

```
  return (nz * ex_inc_(x, y, z, t) - nx * ez_inc_(x, y, z, t));
}
Real hz_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
  extern Real nx, ny;
  return (nx * ey_inc_(x, y, z, t) - ny * ex_inc_(x, y, z, t));
}
```

See also chunk 92.

Electric Field Injection

The signal injection is defined by the user in the $\boldsymbol{E}$ terms. The user specifies the direction of the signal by providing vector $\boldsymbol{n}$. The vector does not have to be normalized, because the program normalizes it.

The user also has to provide three functions that define $E_x(\zeta)$, $E_y(\zeta)$ and $E_z(\zeta)$, where $\zeta = \boldsymbol{n} \cdot \boldsymbol{r} - t$. Vector $\boldsymbol{E}$ should be perpendicular to $\boldsymbol{n}$. In case the user forgets about it, the program enforces this anyway. This, however, is costly, so the user, who's certain that $\forall_\zeta \boldsymbol{E}(\zeta) \perp \boldsymbol{n}$, may tell the program that this is the case, whereupon a global variable *normalized* will be set to *true*, and the perpendicularization of $\boldsymbol{E}$ will be omitted.

92 ⟨ Injection.c_std 91 ⟩ +≡

```
Real ex_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
    /* global variables */
  extern Real nx, ny, nz;
  extern SCM ex_lambda, ey_lambda, ez_lambda;
  extern int normalized;
      /* local variables */
  Real zeta;
  Real ex = 0.0, ey, ez;
  Real e_times_n;
  zeta = nx * (*x) + ny * (*y) + nz * (*z) - (*t);
  ex = scm_to_double(scm_call_1(ex_lambda, scm_from_double(zeta)));
  if (¬normalized) {
    ey = scm_to_double(scm_call_1(ey_lambda, scm_from_double(zeta)));
    ez = scm_to_double(scm_call_1(ez_lambda, scm_from_double(zeta)));
    e_times_n = ex * nx + ey * ny + ez * nz;
    ex = ex - e_times_n * nx;
  }
  return ex;
}
Real ey_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
    /* global variables */
  extern Real nx, ny, nz;
  extern SCM ex_lambda, ey_lambda, ez_lambda;
  extern int normalized;
      /* local variables */
  Real zeta;
  Real ex, ey = 0.0, ez;
  Real e_times_n;
  zeta = nx * (*x) + ny * (*y) + nz * (*z) - (*t);
  ey = scm_to_double(scm_call_1(ey_lambda, scm_from_double(zeta)));
  if (¬normalized) {
    ex = scm_to_double(scm_call_1(ex_lambda, scm_from_double(zeta)));
    ez = scm_to_double(scm_call_1(ez_lambda, scm_from_double(zeta)));
```

```
    e_times_n = ex * nx + ey * ny + ez * nz;
    ey = ey - e_times_n * ny;
  }
  return ey;
}
Real ez_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t)
{
    /* global variables */
  extern Real nx, ny, nz;
  extern SCM ex_lambda, ey_lambda, ez_lambda;
  extern int normalized;
    /* local variables */
  Real zeta;
  Real ex, ey, ez = 0.0;
  Real e_times_n;
  zeta = nx * (*x) + ny * (*y) + nz * (*z) - (*t);
  ez = scm_to_double(scm_call_1(ez_lambda, scm_from_double(zeta)));
  if (¬normalized) {
    ex = scm_to_double(scm_call_1(ex_lambda, scm_from_double(zeta)));
    ey = scm_to_double(scm_call_1(ey_lambda, scm_from_double(zeta)));
    e_times_n = ex * nx + ey * ny + ez * nz;
    ez = ez - e_times_n * nz;
  }
  return ez;
}
```

Injection Includes

93 ⟨ Injection.h_std 93 ⟩ ≡

```
Real hx_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real hy_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real hz_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real ex_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real ey_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
Real ez_inc_(const Real *const x, const Real *const y, const Real *const z, const Real *const t);
```

7.3 Convert *D* to *E*

7.3.1 Function `d_to_e`

95 ⟨ Function *d_to_e* 95 ⟩ ≡

```
int d_to_e(level *level_ptr)
{
  int watch_d_to_e = (int) false;
  int garbage_collect = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  if (watch_d_to_e) {
    pout( ) ≪ "\tlevel_number = " ≪ level_number ≪ endl;
  }
  if (level_number ≡ 0) {
    if (return_status = d_to_e_0(level_ptr)) {
      pout( ) ≪ "\tERROR(d_to_e): failure in d_to_e_0: " ≪ return_status ≪ endl;
      goto exit;
    }
  }
  else {
    if (return_status = d_to_e_n(level_ptr)) {
      pout( ) ≪ "\tERROR(d_to_e): failure in d_to_e_n: " ≪ return_status ≪ endl;
      goto exit;
    }
  }
  media_parameter_parser.query("garbage_collect", garbage_collect);
  if (garbage_collect) scm_gc( );
exit:
  if (watch_d_to_e) pout( ) ≪ "d_to_e: returning to caller" ≪ endl;
  return return_status;
}
```

This code is used in chunk 84.

7.3.2 Function `d_to_e_0`

96 ⟨ Function *d_to_e_0* 96 ⟩ ≡

```
int d_to_e_0(level *level_ptr)
{
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  int number_of_auxiliary_fields = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
  if (watch_d_to_e) {
    pout( ) ≪ "\td_to_e_0:" ≪ endl;
    pout( ) ≪ "\t\tnumber_of_auxiliary_fields = " ≪ number_of_auxiliary_fields ≪ endl;
```

```
    }
    if (level_number ≠ 0) {
      pout( ) ≪ "\t\tERROR(d_to_e_0): not level zero" ≪ endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    if (number_of_auxiliary_fields) {
      if (return_status = d_to_e_0_n(level_ptr)) {
        pout( ) ≪ "\t\tERROR(d_to_e_0): failure in d_to_e_0_n" ≪ endl;
        goto exit;
      }
    }
    else {
      if (return_status = d_to_e_0_0(level_ptr)) {
        pout( ) ≪ "\t\tERROR(d_to_e_0): failure in d_to_e_0_0" ≪ endl;
        goto exit;
      }
    }
  exit:
    if (watch_d_to_e) pout( ) ≪ "\td_to_e_0: returning to caller" ≪ endl;
    return return_status;
  }
```

This code is cited in chunk 18.

This code is used in chunk 84.

7.3.3 Function `d_to_e_0_0`

97 ⟨ Function *d_to_e_0_0* 97 ⟩ ≡

```
  int d_to_e_0_0(level *level_ptr)
  {
      /* Convert D to E in every cell of this level zero with no auxiliary fields. */
    int watch_d_to_e = (int) false;
    int return_status = EXIT_SUCCESS;
    int ghost_margin = N_GHOST_CELLS;
    int level_number = level_ptr→number;
    int number_of_auxiliary_fields = 0;
    DataIterator data_iterator = level_ptr→data_iterator;
    SCMParmParse watch_parameter_parser("forms.watch");
    SCMParmParse media_parameter_parser("forms.media");
    watch_parameter_parser.query("d_to_e", watch_d_to_e);
    media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
    if (watch_d_to_e) pout( ) ≪ "\t\td_to_e_0_0:" ≪ endl;
    if (level_number) {
      pout( ) ≪ "\t\t\tERROR(d_to_e_0_0): not level zero" ≪ endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    if (number_of_auxiliary_fields) {
      pout( ) ≪ "\t\t\tERROR(d_to_e_0_0): auxiliary fields needed" ≪ endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    swap_idx(level_ptr→E_idx);      /* swap the new/old field index */
    for (data_iterator.reset( ); data_iterator.ok( ); ++data_iterator) {
```

```
    DataIndex data_index = data_iterator( );
    CurlBox &E = level_ptr→E[data_index];
    CurlBox &D = level_ptr→D[data_index];
    EdgeFab⟨int⟩ &medium_E = level_ptr→medium_E[data_index];
    Real &time_e = level_ptr→time_e;
    Real &dt = level_ptr→dt;
    Vector⟨Real⟩ &C0x = level_ptr→C0x;
    Vector⟨Real⟩ &C1x = level_ptr→C1x;
    Vector⟨Real⟩ &C3x = level_ptr→C3x;
    Vector⟨Real⟩ &C4x = level_ptr→C4x;
    Vector⟨Real⟩ &C0y = level_ptr→C0y;
    Vector⟨Real⟩ &C1y = level_ptr→C1y;
    Vector⟨Real⟩ &C3y = level_ptr→C3y;
    Vector⟨Real⟩ &C4y = level_ptr→C4y;
    Vector⟨Real⟩ &C0z = level_ptr→C0z;
    Vector⟨Real⟩ &C1z = level_ptr→C1z;
    Vector⟨Real⟩ &C3z = level_ptr→C3z;
    Vector⟨Real⟩ &C4z = level_ptr→C4z;
    int Cx_lo = -ghost_margin;
    int Cx_hi = (int) C1x.size( ) - ghost_margin;
    int Cy_lo = -ghost_margin;
    int Cy_hi = (int) C1y.size( ) - ghost_margin;
    int Cz_lo = -ghost_margin;
    int Cz_hi = (int) C1z.size( ) - ghost_margin;
    for (int dir = 0; dir < SpaceDim; dir++) {
      FArrayBox &E_dir = E[dir];
      FArrayBox &D_dir = D[dir];
      BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
      d_to_e_0_0_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→E_idx[0]), &(level_ptr→D_idx[0]), &time_e,
          &dt, &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0], &Cy_lo, &Cy_hi, &C0y[0], &C1y[0],
          &C3y[0], &C4y[0], &Cz_lo, &Cz_hi, &C0z[0], &C1z[0], &C3z[0], &C4z[0], &(D_dir.loVect( )[0]),
          &(D_dir.loVect( )[1]), &(D_dir.loVect( )[2]), &(D_dir.hiVect( )[0]), &(D_dir.hiVect( )[1]),
          &(D_dir.hiVect( )[2]), D_dir.dataPtr( ), E_dir.dataPtr( ), medium_E_dir.dataPtr( ), &watch_d_to_e,
          &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout( ) ≪ "\t\t\tERROR(d_to_e_0_0): failure in d_to_e_0_0_: " ≪ return_status ≪ endl;
        goto exit;
      }
    }     /* for (int dir = 0; dir < SpaceDim; dir++) */
  }     /* for (data_iterator.reset( ); data_iterator.ok( ); ++data_iterator) */
exit:
  if (watch_d_to_e) pout( ) ≪ "\t\td_to_e_0_0: returning to caller" ≪ endl;
  return return_status;
}
```

This code is used in chunk 84.

7.3.4 Function `d_to_e_0_n`

98 ⟨ Function *d_to_e_0_n* 98 ⟩ ≡

```
int d_to_e_0_n(level *level_ptr)
{
    /* Convert D to E in every cell of this level zero with auxiliary fields. */
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
```

```
int ghost_margin = N_GHOST_CELLS;
int level_number = level_ptr→number;
int number_of_auxiliary_fields = 0;
DataIterator data_iterator = level_ptr→data_iterator;
SCMParmParse watch_parameter_parser("forms.watch");
SCMParmParse media_parameter_parser("forms.media");
watch_parameter_parser.query("d_to_e", watch_d_to_e);
media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
if (watch_d_to_e) pout() ≪ "\t\td_to_e_0_n:" ≪ endl;
if (level_number) {
  pout() ≪ "\t\t\tERROR(d_to_e_0_n): not level zero" ≪ endl;
  return_status = ERR_D_TO_E;
  goto exit;
}
if (¬number_of_auxiliary_fields) {
  pout() ≪ "\t\t\tERROR(d_to_e_0_n): no auxiliary fields" ≪ endl;
  return_status = ERR_D_TO_E;
  goto exit;
}
swap_idx(level_ptr→E_idx);      /* swap the new/old field index */
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  CurlBox &E = level_ptr→E[data_index];
  CurlBox &D = level_ptr→D[data_index];
  CurlBox &S = level_ptr→S[data_index];
  EdgeFab⟨int⟩ &medium_E = level_ptr→medium_E[data_index];
  Real &time_e = level_ptr→time_e;
  Real &dt = level_ptr→dt;
  Vector⟨Real⟩ &C0x = level_ptr→C0x;
  Vector⟨Real⟩ &C1x = level_ptr→C1x;
  Vector⟨Real⟩ &C3x = level_ptr→C3x;
  Vector⟨Real⟩ &C4x = level_ptr→C4x;
  Vector⟨Real⟩ &C0y = level_ptr→C0y;
  Vector⟨Real⟩ &C1y = level_ptr→C1y;
  Vector⟨Real⟩ &C3y = level_ptr→C3y;
  Vector⟨Real⟩ &C4y = level_ptr→C4y;
  Vector⟨Real⟩ &C0z = level_ptr→C0z;
  Vector⟨Real⟩ &C1z = level_ptr→C1z;
  Vector⟨Real⟩ &C3z = level_ptr→C3z;
  Vector⟨Real⟩ &C4z = level_ptr→C4z;
  int Cx_lo = −ghost_margin;
  int Cx_hi = (int) C1x.size() − ghost_margin;
  int Cy_lo = −ghost_margin;
  int Cy_hi = (int) C1y.size() − ghost_margin;
  int Cz_lo = −ghost_margin;
  int Cz_hi = (int) C1z.size() − ghost_margin;
  if (S.nComp() ≠ number_of_auxiliary_fields) {
    pout() ≪ "\t\t\tERROR(d_to_e_0_n): bad number of S components" ≪ endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  for (int dir = 0; dir < SpaceDim; dir++) {
    FArrayBox &E_dir = E[dir];
    FArrayBox &D_dir = D[dir];
```

```
      FArrayBox &S_dir = S[dir];
      BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
      d_to_e_0_n_(&SpaceDim, &dir, &ghost_margin, &number_of_auxiliary_fields, &(level_ptr→E_idx[0]),
          &(level_ptr→D_idx[0]), &time_e, &dt, &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0],
          &Cy_lo, &Cy_hi, &C0y[0], &C1y[0], &C3y[0], &C4y[0], &Cz_lo, &Cz_hi, &C0z[0], &C1z[0],
          &C3z[0], &C4z[0], &(D_dir.loVect()[0]), &(D_dir.loVect()[1]), &(D_dir.loVect()[2]),
          &(D_dir.hiVect()[0]), &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]), D_dir.dataPtr(),
          E_dir.dataPtr(), S_dir.dataPtr(), medium_E_dir.dataPtr(), &watch_d_to_e, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() ≪ "\t\t\tERROR(d_to_e_0_n): failure in d_to_e_0_n_: " ≪ return_status ≪ endl;
        goto exit;
      }
    }    /* for (int dir = 0; dir < SpaceDim; dir++) */
  }    /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
exit:
  if (watch_d_to_e) pout() ≪ "\t\td_to_e_0_n: returning to caller" ≪ endl;
  return return_status;
}
```

This code is cited in chunk 18.

This code is used in chunk 84.

7.3.5 Function `d_to_e_n`

99 ⟨ Function *d_to_e_n* 99 ⟩ ≡

```
int d_to_e_n(level *level_ptr)
{
  int watch_d_to_e = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  int number_of_auxiliary_fields = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("d_to_e", watch_d_to_e);
  media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
  if (watch_d_to_e) {
    pout() ≪ "\td_to_e_n:" ≪ endl;
    pout() ≪ "\t\tnumber_of_auxiliary_fields = " ≪ number_of_auxiliary_fields ≪ endl;
  }
  if (level_number ≡ 0) {
    pout() ≪ "\t\tERROR(d_to_e_n): invoked on level zero" ≪ endl;
    return_status = ERR_D_TO_E;
    goto exit;
  }
  if (number_of_auxiliary_fields) {
    if (return_status = d_to_e_n_n(level_ptr)) {
      pout() ≪ "\t\tERROR(d_to_e_n): failure in d_to_e_n_n" ≪ endl;
      goto exit;
    }
  }
  else {
    if (return_status = d_to_e_n_0(level_ptr)) {
      pout() ≪ "\t\tERROR(d_to_e_n): failure in d_to_e_n_0" ≪ endl;
      goto exit;
    }
```

```
    }
  exit:
    if (watch_d_to_e) pout() ≪ "\td_to_e_n:␣returning␣to␣caller" ≪ endl;
    return return_status;
  }
```

This code is used in chunk 84.

7.3.6 Function `d_to_e_n_0`

100 ⟨ Function *d_to_e_n_0* 100 ⟩ ≡

```
  int d_to_e_n_0(level *level_ptr)
  {
      /* Convert D to E in every cell of this level (other than zero) with no auxiliary fields. */
    int watch_d_to_e = (int) false;
    int return_status = EXIT_SUCCESS;
    int ghost_margin = N_GHOST_CELLS;
    int level_number = level_ptr→number;
    int number_of_auxiliary_fields = 0;
    DataIterator data_iterator = level_ptr→data_iterator;
    SCMParmParse watch_parameter_parser("forms.watch");
    SCMParmParse media_parameter_parser("forms.media");
    watch_parameter_parser.query("d_to_e", watch_d_to_e);
    media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
    if (watch_d_to_e) pout() ≪ "\t\td_to_e_n_0:" ≪ endl;
    if (¬level_number) {
      pout() ≪ "\t\t\tERROR(d_to_e_n_0):␣invoked␣on␣level␣zero" ≪ endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    if (number_of_auxiliary_fields) {
      pout() ≪ "\t\t\tERROR(d_to_e_n_0):␣auxiliary␣fields␣needed" ≪ endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    swap_idx(level_ptr→E_idx);
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      CurlBox &E = level_ptr→E[data_index];
      CurlBox &D = level_ptr→D[data_index];
      EdgeFab⟨int⟩ &medium_E = level_ptr→medium_E[data_index];
      Real &time_e = level_ptr→time_e;
      Real &dt = level_ptr→dt;
      for (int dir = 0; dir < SpaceDim; dir++) {
        FArrayBox &E_dir = E[dir];
        FArrayBox &D_dir = D[dir];
        BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
        d_to_e_n_0_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→E_idx[0]), &(level_ptr→D_idx[0]), &time_e,
            &dt, &(D_dir.loVect()[0]), &(D_dir.loVect()[1]), &(D_dir.loVect()[2]), &(D_dir.hiVect()[0]),
            &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]), D_dir.dataPtr(), E_dir.dataPtr(),
            medium_E_dir.dataPtr(), &watch_d_to_e, &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() ≪ "\t\t\tERROR(d_to_e_n_0):␣failure␣in␣d_to_e_n_0_:␣" ≪ return_status ≪ endl;
          goto exit;
```

```
        }
      }     /* for (int dir = 0; dir < SpaceDim; dir++) */
    }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  exit:
    if (watch_d_to_e) pout() << "\t\td_to_e_n_0: returning to caller" << endl;
    return return_status;
  }
```

This code is used in chunk 84.

7.3.7 Function `d_to_e_n_n`

101 ⟨ Function *d_to_e_n_n* 101 ⟩ ≡

```
  int d_to_e_n_n(level *level_ptr)
  {
      /* Convert D to E in every cell of this level (other than zero) with auxiliary fields. */
    int watch_d_to_e = (int) false;
    int return_status = EXIT_SUCCESS;
    int ghost_margin = N_GHOST_CELLS;
    int level_number = level_ptr->number;
    int number_of_auxiliary_fields = 0;
    DataIterator data_iterator = level_ptr->data_iterator;
    SCMParmParse watch_parameter_parser("forms.watch");
    SCMParmParse media_parameter_parser("forms.media");
    watch_parameter_parser.query("d_to_e", watch_d_to_e);
    media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields);
    if (watch_d_to_e) pout() << "\t\td_to_e_n_n:" << endl;
    if (¬level_number) {
      pout() << "\t\t\tERROR(d_to_e_n_n): invoked on level zero" << endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    if (¬number_of_auxiliary_fields) {
      pout() << "\t\t\tERROR(d_to_e_n_n): no auxiliary fields" << endl;
      return_status = ERR_D_TO_E;
      goto exit;
    }
    swap_idx(level_ptr->E_idx);
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      DataIndex data_index = data_iterator();
      CurlBox &E = level_ptr->E[data_index];
      CurlBox &D = level_ptr->D[data_index];
      CurlBox &S = level_ptr->S[data_index];
      EdgeFab<int> &medium_E = level_ptr->medium_E[data_index];
      Real &time_e = level_ptr->time_e;
      Real &dt = level_ptr->dt;
      if (S.nComp() ≠ number_of_auxiliary_fields) {
        pout() << "\t\t\tERROR(d_to_e_n_n): bad number of S components" << endl;
        return_status = ERR_D_TO_E;
        goto exit;
      }
      for (int dir = 0; dir < SpaceDim; dir++) {
        FArrayBox &E_dir = E[dir];
        FArrayBox &D_dir = D[dir];
        FArrayBox &S_dir = S[dir];
```

```
        BaseFab⟨int⟩ &medium_E_dir = medium_E[dir];
        d_to_e_n_n_(&SpaceDim, &dir, &ghost_margin, &number_of_auxiliary_fields, &(level_ptr→E_idx[0]),
            &(level_ptr→D_idx[0]), &time_e, &dt, &(D_dir.loVect()[0]), &(D_dir.loVect()[1]),
            &(D_dir.loVect()[2]), &(D_dir.hiVect()[0]), &(D_dir.hiVect()[1]), &(D_dir.hiVect()[2]),
            D_dir.dataPtr(), E_dir.dataPtr(), S_dir.dataPtr(), medium_E_dir.dataPtr(), &watch_d_to_e,
            &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout() ≪ "\t\t\tERROR(d_to_e_n_n): failure in d_to_e_n_n_: " ≪ return_status ≪ endl;
          goto exit;
        }
      }     /* for (int dir = 0; dir < SpaceDim; dir++) */
    }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
  exit:
    if (watch_d_to_e) pout() ≪ "\t\td_to_e_n_n: returning to caller" ≪ endl;
    return return_status;
  }
```

This code is used in chunk 84.

7.3.8 Scheme Driven *D*-to-*E* Conversion

The following two plain C functions are called from Fortran in the **case default** clause of the *select***case** (*medium_E*(i, j, k)) switch. The functions prepare things for calling Scheme and then call it. Because all that has to be done here consists of side-evaluations on $\boldsymbol{E}$, $\boldsymbol{E}_{\text{old}}$, $\boldsymbol{D}$ and $\boldsymbol{D}_{\text{old}}$ we have to feed these into Scheme globals that must be defined beforehand. The functions *do not* check for the presence of the definitions though, because there is simply no time for this. This has to run as fast as possible. It is the responsibility of the programmer, meaning *me*, to ensure that a correct Scheme environment has been created before these functions are invoked.

The first function is *scm_d_to_e_0_*. In this form the function does not expect to be passed any auxiliary fields.

102 ⟨ Conversion.c_std 102 ⟩ ≡

```
#include <REAL.H>
#include <libguile.h>
#include "./Conversion.h"      /* This must be plain C, so we cannot call SCMParmParse. */
  Real scm_d_to_e_0_(const int *const direction, const int *const medium, const Real *const E_old, const
          Real *const D, const Real *const D_old, const Real *const t, const Real *const dt)
  {
    extern SCM E_old_var, D_var, D_old_var, medium_var, direction_var, t_e_var, dt_var;
    extern SCM e_lambda;
    Real E_ret;
    scm_variable_set_x(D_var, scm_from_double(*D));
    scm_variable_set_x(E_old_var, scm_from_double(*E_old));
    scm_variable_set_x(D_old_var, scm_from_double(*D_old));
    scm_variable_set_x(medium_var, scm_from_int(*medium));
    scm_variable_set_x(direction_var, scm_from_int(*direction));
    scm_variable_set_x(t_e_var, scm_from_int(*t));
    scm_variable_set_x(dt_var, scm_from_int(*dt));
    E_ret = scm_to_double(scm_call_0(e_lambda));
    return E_ret;
  }
```

See also chunk 103.

This code is cited in chunk 18.

¶ The second function is much like the first one, but it handles additionally auxiliary fields.

103 ⟨ Conversion.c_std 102 ⟩ +≡

```
Real scm_d_to_e_n_(const int *const direction, const int *const n_aux_fields, const int
        *const medium, const Real *const E_old, const Real *const D, const Real *const D_old, Real
        *const S, const Real *const t, const Real *const dt)
{
  extern SCM E_old_var, D_var, D_old_var, S_var_ref, medium_var, direction_var, t_e_var, dt_var;
  extern SCM e_lambda;
  Real E_ret;
  Real *S_ptr;
  int count;
  scm_variable_set_x(D_var, scm_from_double(*D));
  scm_variable_set_x(E_old_var, scm_from_double(*E_old));
  scm_variable_set_x(D_old_var, scm_from_double(*D_old));
  scm_variable_set_x(medium_var, scm_from_int(*medium));
  scm_variable_set_x(direction_var, scm_from_int(*direction));
  scm_variable_set_x(t_e_var, scm_from_int(*t));
  scm_variable_set_x(dt_var, scm_from_int(*dt));
  S_ptr = S;
  for (count = 0; count < *n_aux_fields; count++)
    scm_uniform_vector_set_x(S_var_ref, scm_from_int(count), scm_from_double((double) *S_ptr++));
  E_ret = scm_to_double(scm_call_0(e_lambda));
  S_ptr = S;
  for (count = 0; count < *n_aux_fields; count++)
    *S_ptr++ = (Real) scm_to_double(scm_uniform_vector_ref(S_var_ref, scm_from_int(count)));
  return E_ret;
}
```

¶ And this is a header file for "Conversion.c", just declaring both functions and EXIT_SUCCESS.

104 ⟨ Conversion.h_std 104 ⟩ ≡

```
#ifdef EXIT_SUCCESS
#undef EXIT_SUCCESS
#endif
# define EXIT_SUCCESS 0
  Real scm_d_to_e_0_(const int *const direction, const int *const medium, const Real *const E_old, const
      Real *const D, const Real *const D_old, const Real *const t, const Real *const dt);
  Real scm_d_to_e_n_(const int *const direction, const int *const n_aux_fields, const int
      *const medium, const Real *const E_old, const Real *const D, const Real *const D_old, Real
      *const S, const Real *const t, const Real *const dt);
```

7.4 Advance the B field

105 ⟨ Function *advance_b* 105 ⟩ ≡

```
int advance_b(level *level_ptr)
{
  int watch_advance_b = false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_b", watch_advance_b);
  if (watch_advance_b) pout() ≪ "advance_b: level #" ≪ level_number ≪ endl;
  if (level_number ≡ 0)      /* level 0: special advance is needed here */
  {
    if (watch_advance_b) pout() ≪ "\tCalling advance_b_0" ≪ endl;
    if (return_status = advance_b_0(level_ptr)) {
      pout() ≪ "\tERROR(advance_b): Failed to advance level 0" ≪ endl;
      goto exit;
    }
  }
  else      /* not level 0: simple advance */
  {
    if (watch_advance_b) pout() ≪ "\tCalling advance_b_n" ≪ endl;
    if (return_status = advance_b_n(level_ptr)) {
      pout() ≪ "\tERROR(advance_b): Failed to advance level " ≪ level_number ≪ endl;
      goto exit;
    }
  }
  level_ptr→time_h += level_ptr→dt;
exit: return return_status;
}
```

This code is used in chunk 84.

7.4.1 Advance Level Zero (with UPML)

106 ⟨ Function *advance_b_0* 106 ⟩ ≡

```
int advance_b_0(level *level_ptr)
{
  int watch_advance_b = false;
  int watch_update_b = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator &data_iterator = level_ptr→data_iterator;
  int Cx_lo = −ghost_margin;
  int Cx_hi = (int) level_ptr→C1x.size() − ghost_margin;
  int Cy_lo = −ghost_margin;
  int Cy_hi = (int) level_ptr→C1y.size() − ghost_margin;
  int Cz_lo = −ghost_margin;
  int Cz_hi = (int) level_ptr→C1z.size() − ghost_margin;
  Interval current_component;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_b", watch_advance_b);
  watch_parameter_parser.query("update_b", watch_update_b);
  if (watch_advance_b) pout() ≪ "\tadvance_b_0:" ≪ endl;
```

```
  if (level_number ≠ 0) {
    pout( ) ≪ "\t\tERROR(advance_b_0):␣This␣is␣not␣level␣zero" ≪ endl;
    return_status = ERR_ADVANCE_B;
    goto exit;
  }
  current_component.define(level_ptr→E_idx[0], level_ptr→E_idx[0]);
  level_ptr→E.exchange(current_component);
  swap_idx(level_ptr→B_idx);
  if (watch_advance_b > 1) pout( ) ≪ "\t\tentering␣the␣box␣loop" ≪ endl ≪ flush;
  for (data_iterator.reset( ); data_iterator.ok( ); ++data_iterator) {
    DataIndex data_index = data_iterator( );
    FArrayBox &Bx = level_ptr→B[data_index][0];
    FArrayBox &By = level_ptr→B[data_index][1];
    FArrayBox &Bz = level_ptr→B[data_index][2];
    FArrayBox &Ex = level_ptr→E[data_index][0];
    FArrayBox &Ey = level_ptr→E[data_index][1];
    FArrayBox &Ez = level_ptr→E[data_index][2];
    Vector⟨Real⟩ &C1x = level_ptr→C1x;
    Vector⟨Real⟩ &C2x = level_ptr→C2x;
    Vector⟨Real⟩ &C1y = level_ptr→C1y;
    Vector⟨Real⟩ &C2y = level_ptr→C2y;
    Vector⟨Real⟩ &C1z = level_ptr→C1z;
    Vector⟨Real⟩ &C2z = level_ptr→C2z;
    if (watch_advance_b > 1)
      pout( ) ≪ "\t\t\tcalling␣update_b_upml_␣on␣box␣" ≪ Bx.box( ) ≪ "␣...␣" ≪ flush;
    update_b_upml_(
    &SpaceDim, &ghost_margin,
    &(level_ptr→B_idx[0]), &(level_ptr→E_idx[0]),
    &Cx_lo, &Cx_hi, &C1x[0], &C2x[0],
    &Cy_lo, &Cy_hi, &C1y[0], &C2y[0],
    &Cz_lo, &Cz_hi, &C1z[0], &C2z[0],
    &(Bx.loVect( )[0]), &(Bx.loVect( )[1]), &(Bx.loVect( )[2]), &(Bx.hiVect( )[0]), &(Bx.hiVect( )[1]),
        &(Bx.hiVect( )[2]), Bx.dataPtr( ),
    &(By.loVect( )[0]), &(By.loVect( )[1]), &(By.loVect( )[2]), &(By.hiVect( )[0]), &(By.hiVect( )[1]),
        &(By.hiVect( )[2]), By.dataPtr( ),
    &(Bz.loVect( )[0]), &(Bz.loVect( )[1]), &(Bz.loVect( )[2]), &(Bz.hiVect( )[0]), &(Bz.hiVect( )[1]),
        &(Bz.hiVect( )[2]), Bz.dataPtr( ),
    &(Ex.loVect( )[0]), &(Ex.loVect( )[1]), &(Ex.loVect( )[2]), &(Ex.hiVect( )[0]), &(Ex.hiVect( )[1]),
        &(Ex.hiVect( )[2]), Ex.dataPtr( ),
    &(Ey.loVect( )[0]), &(Ey.loVect( )[1]), &(Ey.loVect( )[2]), &(Ey.hiVect( )[0]), &(Ey.hiVect( )[1]),
        &(Ey.hiVect( )[2]), Ey.dataPtr( ),
    &(Ez.loVect( )[0]), &(Ez.loVect( )[1]), &(Ez.loVect( )[2]), &(Ez.hiVect( )[0]), &(Ez.hiVect( )[1]),
        &(Ez.hiVect( )[2]), Ez.dataPtr( ),
    &watch_update_b, &return_status
    );
    if (watch_advance_b > 1) pout( ) ≪ "done." ≪ endl ≪ flush;
    if (return_status ≠ EXIT_SUCCESS) {
      pout( ) ≪ "\t\tERROR(advance_b_0):␣update_b_upml_␣returned␣" ≪ return_status ≪ endl;
      goto exit;
    }
  }
exit: return return_status;
}
```

This code is used in chunk 84.

7.4.2 Advance Higher Levels

107 ⟨ Function *advance_b_n* 107 ⟩ ≡

```
int advance_b_n(level *level_ptr)
{
  int watch_advance_b = false;
  int watch_update_b = false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  Real dt_by_dg = (level_ptr→dt)/(level_ptr→delta);
  DataIterator &data_iterator = level_ptr→data_iterator;
  Interval current_component;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("advance_b", watch_advance_b);
  watch_parameter_parser.query("update_b", watch_update_b);
  if (watch_advance_b) pout() ≪ "\tadvance_b_n:" ≪ endl;
  if (level_number ≡ 0) {
    pout() ≪ "\t\tERROR(advance_b_n): invoked on level zero" ≪ endl;
    return_status = ERR_ADVANCE_B;
    goto exit;
  }
  current_component.define(level_ptr→E_idx[0], level_ptr→E_idx[0]);
  level_ptr→E.exchange(current_component);
  swap_idx(level_ptr→B_idx);
      /* Loop over boxes */
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    FArrayBox &Bx = level_ptr→B[data_index][0];
    FArrayBox &By = level_ptr→B[data_index][1];
    FArrayBox &Bz = level_ptr→B[data_index][2];
    FArrayBox &Ex = level_ptr→E[data_index][0];
    FArrayBox &Ey = level_ptr→E[data_index][1];
    FArrayBox &Ez = level_ptr→E[data_index][2];
    update_b_(
    &SpaceDim, &ghost_margin,
    &(level_ptr→B_idx[0]), &(level_ptr→E_idx[0]),
    &dt_by_dg,
    &(Bx.loVect()[0]), &(Bx.loVect()[1]), &(Bx.loVect()[2]), &(Bx.hiVect()[0]), &(Bx.hiVect()[1]),
        &(Bx.hiVect()[2]), Bx.dataPtr(),
    &(By.loVect()[0]), &(By.loVect()[1]), &(By.loVect()[2]), &(By.hiVect()[0]), &(By.hiVect()[1]),
        &(By.hiVect()[2]), By.dataPtr(),
    &(Bz.loVect()[0]), &(Bz.loVect()[1]), &(Bz.loVect()[2]), &(Bz.hiVect()[0]), &(Bz.hiVect()[1]),
        &(Bz.hiVect()[2]), Bz.dataPtr(),
    &(Ex.loVect()[0]), &(Ex.loVect()[1]), &(Ex.loVect()[2]), &(Ex.hiVect()[0]), &(Ex.hiVect()[1]),
        &(Ex.hiVect()[2]), Ex.dataPtr(),
    &(Ey.loVect()[0]), &(Ey.loVect()[1]), &(Ey.loVect()[2]), &(Ey.hiVect()[0]), &(Ey.hiVect()[1]),
        &(Ey.hiVect()[2]), Ey.dataPtr(),
    &(Ez.loVect()[0]), &(Ez.loVect()[1]), &(Ez.loVect()[2]), &(Ez.hiVect()[0]), &(Ez.hiVect()[1]),
        &(Ez.hiVect()[2]), Ez.dataPtr(),
    &watch_update_b, &return_status
    );
    if (return_status ≠ EXIT_SUCCESS) {
      pout() ≪ "\t\tERROR(advance_b_n): update_b_ returned " ≪ return_status ≪ endl;
      goto exit;
```

```
        }
      }
    exit: return return_status;
    }
```
This code is used in chunk 84.

7.5 Inject the B field

108 ⟨ Function *inject_b* 108 ⟩ ≡

```
int inject_b(level *level_ptr)
{
    /* Inject the B field on the total/scattered field boundary. */
  int return_status = EXIT_SUCCESS;
  int watch_inject_b = (int) false;
  int garbage_collect = (int) false;
  SideIterator side_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse signal_parameter_parser("forms.signal");
  watch_parameter_parser.query("inject_b", watch_inject_b);
  signal_parameter_parser.query("garbage_collect", garbage_collect);
  if (watch_inject_b) pout() << "inject_b:" << endl;
  if (level_ptr->number != 0) {
    pout() << "\tERROR(inject_b): not level zero" << endl;
    return_status = ERR_INJECT_B;
    goto exit;
  }
  for (int dir = 0; dir < SpaceDim; dir++) {
    for (side_iterator.reset(); side_iterator.ok(); ++side_iterator) {
      Side::LoHiSide side = side_iterator();
      if (return_status = inject_b(level_ptr, dir, side)) {
        pout() << "\tERROR(inject_b): failed on dir = " << dir << ", side = " << side << endl;
        goto exit;
      }
    }
  }
  if (garbage_collect) scm_gc();
exit: return return_status;
}
```

See also chunk 109.

This code is used in chunk 84.

109 ¶⟨ Function *inject_b* 108 ⟩ +≡

```
int inject_b(level *level_ptr, int dir, Side::LoHiSide side)
{
  int return_status = EXIT_SUCCESS;
  int watch_inject_b = (int) false;
  int side_number = (int) side;
  int ghost_margin = N_GHOST_CELLS;
  IntVect ijk_lo = level_ptr->signal_ijk_lo;
  IntVect ijk_hi = level_ptr->signal_ijk_hi;
  Vector<Real> &origin = level_ptr->origin;
  Real &delta = level_ptr->delta;
  LevelData<FluxBox> &B = level_ptr->B;
  int &new_component = level_ptr->B_idx[0];
  Vector<DataIndex> &side_boxes = level_ptr->TFR_face_boxes[dir][side];
  Real dt_by_delta = level_ptr->dt/level_ptr->delta;
  Real time_e = level_ptr->time_e;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("inject_b", watch_inject_b);
  if (watch_inject_b) pout() << "\tinject_b: dir = " << dir << ", side = " << side << endl;
```

```
if (dir < 0 ∨ dir ≥ SpaceDim) {
  pout( ) ≪ "\t\tERROR(inject_b):␣bad␣direction" ≪ endl;
  return_status = ERR_INJECT_B;
  goto exit;
}
for (int count = 0; count < side_boxes.size( ); count++) {
    /* This loop picks up only the boxes that cross this side and that live on this processor. */
  DataIndex data_index = side_boxes[count];
  switch (dir) {
  case 0:     /* West-East */
    {
      FArrayBox &By = B[data_index][1];
      FArrayBox &Bz = B[data_index][2];
      IntVect By_lo = By.smallEnd( );
      IntVect By_hi = By.bigEnd( );
      IntVect Bz_lo = Bz.smallEnd( );
      IntVect Bz_hi = Bz.bigEnd( );
      inject_b_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_e, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(By_lo[0]), &(By_lo[1]), &(By_lo[2]), &(By_hi[0]), &(By_hi[1]),
          &(By_hi[2]), By.dataPtr( ), &(Bz_lo[0]), &(Bz_lo[1]), &(Bz_lo[2]), &(Bz_hi[0]), &(Bz_hi[1]),
          &(Bz_hi[2]), Bz.dataPtr( ), &watch_inject_b, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout( ) ≪ "\tERROR(inject_d):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
        goto exit;
      }
    }
    break;
  case 1:     /* Front-Back */
    {
      FArrayBox &Bx = B[data_index][0];
      FArrayBox &Bz = B[data_index][2];
      IntVect Bx_lo = Bx.smallEnd( );
      IntVect Bx_hi = Bx.bigEnd( );
      IntVect Bz_lo = Bz.smallEnd( );
      IntVect Bz_hi = Bz.bigEnd( );
      inject_b_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
          &(origin[2]), &delta, &dt_by_delta, &time_e, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
          &(ijk_hi[1]), &(ijk_hi[2]), &(Bx_lo[0]), &(Bx_lo[1]), &(Bx_lo[2]), &(Bx_hi[0]), &(Bx_hi[1]),
          &(Bx_hi[2]), Bx.dataPtr( ), &(Bz_lo[0]), &(Bz_lo[1]), &(Bz_lo[2]), &(Bz_hi[0]), &(Bz_hi[1]),
          &(Bz_hi[2]), Bz.dataPtr( ), &watch_inject_b, &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout( ) ≪ "\tERROR(inject_d):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
        goto exit;
      }
    }
    break;
  case 2:     /* Bottom-Top */
    {
      FArrayBox &Bx = B[data_index][0];
      FArrayBox &By = B[data_index][1];
      IntVect Bx_lo = Bx.smallEnd( );
      IntVect Bx_hi = Bx.bigEnd( );
      IntVect By_lo = By.smallEnd( );
```

```
        IntVect By_hi = By.bigEnd( );
        inject_b_(&dir, &side_number, &ghost_margin, &new_component, &(origin[0]), &(origin[1]),
            &(origin[2]), &delta, &dt_by_delta, &time_e, &(ijk_lo[0]), &(ijk_lo[1]), &(ijk_lo[2]), &(ijk_hi[0]),
            &(ijk_hi[1]), &(ijk_hi[2]), &(Bx_lo[0]), &(Bx_lo[1]), &(Bx_lo[2]), &(Bx_hi[0]), &(Bx_hi[1]),
            &(Bx_hi[2]), Bx.dataPtr( ), &(By_lo[0]), &(By_lo[1]), &(By_lo[2]), &(By_hi[0]), &(By_hi[1]),
            &(By_hi[2]), By.dataPtr( ), &watch_inject_b, &return_status);
        if (return_status ≠ EXIT_SUCCESS) {
          pout( ) ≪ "\tERROR(inject_d):␣Fortran␣problem,␣dir␣=␣" ≪ dir ≪ endl;
          goto exit;
        }
      }
      break;
    }     /* switch (dir) */
  }     /* for (int count = 0; count < side_boxes.size( ); count++) */
exit: return return_status;
}
```

7.6 Convert B to H

7.6.1 Function b_to_h

111 ⟨ Function *b_to_h* 111 ⟩ ≡

```
int b_to_h(level *level_ptr)
{
  int watch_b_to_h = (int) false;
  int garbage_collect = (int) false;
  int return_status = EXIT_SUCCESS;
  int level_number = level_ptr→number;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse media_parameter_parser("forms.media");
  watch_parameter_parser.query("b_to_h", watch_b_to_h);
  if (watch_b_to_h) {
    pout( ) ≪ "\tb_to_h:" ≪ endl;
    pout( ) ≪ "\tlevel_number␣=␣" ≪ level_number ≪ endl;
  }
  if (level_number ≡ 0) {
    if (return_status = b_to_h_0(level_ptr)) {
      pout( ) ≪ "\tERROR(b_to_h):␣failure␣in␣b_to_h_0:␣" ≪ return_status ≪ endl;
      goto exit;
    }
  }
  else {
    if (return_status = b_to_h_n(level_ptr)) {
      pout( ) ≪ "\tERROR(b_to_h):␣failure␣in␣b_to_h_n:␣" ≪ return_status ≪ endl;
      goto exit;
    }
  }
  media_parameter_parser.query("garbage_collect", garbage_collect);
  if (garbage_collect) scm_gc( );
exit:
  if (watch_b_to_h) pout( ) ≪ "b_to_h:␣returning␣to␣caller" ≪ endl;
  return return_status;
}
```

This code is used in chunk 84.

7.6.2 Function b_to_h_0

112 ⟨ Function *b_to_h_0* 112 ⟩ ≡

```
int b_to_h_0(level *level_ptr)
{
    /* Convert B to H in every cell of this level zero. */
  int watch_b_to_h = (int) false;
  int return_status = EXIT_SUCCESS;
  int ghost_margin = N_GHOST_CELLS;
  int level_number = level_ptr→number;
  DataIterator data_iterator = level_ptr→data_iterator;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("b_to_h", watch_b_to_h);
  if (watch_b_to_h) pout( ) ≪ "\tb_to_h_0:" ≪ endl;
  if (level_number) {
    pout( ) ≪ "\t\tERROR(b_to_h_0):␣not␣level␣zero" ≪ endl;
```

```
    return_status = ERR_B_TO_H;
    goto exit;
  }
  swap_idx(level_ptr→H_idx);
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    FluxBox &H = level_ptr→H[data_index];
    FluxBox &B = level_ptr→B[data_index];
    FaceFab⟨int⟩ &medium_H = level_ptr→medium_H[data_index];
    Real &time_h = level_ptr→time_h;
    Real &dt = level_ptr→dt;
    Vector⟨Real⟩ &C0x = level_ptr→C0x;
    Vector⟨Real⟩ &C1x = level_ptr→C1x;
    Vector⟨Real⟩ &C3x = level_ptr→C3x;
    Vector⟨Real⟩ &C4x = level_ptr→C4x;
    Vector⟨Real⟩ &C0y = level_ptr→C0y;
    Vector⟨Real⟩ &C1y = level_ptr→C1y;
    Vector⟨Real⟩ &C3y = level_ptr→C3y;
    Vector⟨Real⟩ &C4y = level_ptr→C4y;
    Vector⟨Real⟩ &C0z = level_ptr→C0z;
    Vector⟨Real⟩ &C1z = level_ptr→C1z;
    Vector⟨Real⟩ &C3z = level_ptr→C3z;
    Vector⟨Real⟩ &C4z = level_ptr→C4z;
    int Cx_lo = -ghost_margin;
    int Cx_hi = (int) C1x.size() - ghost_margin;
    int Cy_lo = -ghost_margin;
    int Cy_hi = (int) C1y.size() - ghost_margin;
    int Cz_lo = -ghost_margin;
    int Cz_hi = (int) C1z.size() - ghost_margin;
    for (int dir = 0; dir < SpaceDim; dir++) {
      FArrayBox &H_dir = H[dir];
      FArrayBox &B_dir = B[dir];
      BaseFab⟨int⟩ &medium_H_dir = medium_H[dir];
      b_to_h_0_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→H_idx[0]), &(level_ptr→B_idx[0]), &time_h, &dt,
          &Cx_lo, &Cx_hi, &C0x[0], &C1x[0], &C3x[0], &C4x[0], &Cy_lo, &Cy_hi, &C0y[0], &C1y[0],
          &C3y[0], &C4y[0], &Cz_lo, &Cz_hi, &C0z[0], &C1z[0], &C3z[0], &C4z[0], &(B_dir.loVect()[0]),
          &(B_dir.loVect()[1]), &(B_dir.loVect()[2]), &(B_dir.hiVect()[0]), &(B_dir.hiVect()[1]),
          &(B_dir.hiVect()[2]), B_dir.dataPtr(), H_dir.dataPtr(), medium_H_dir.dataPtr(), &watch_b_to_h,
          &return_status);
      if (return_status ≠ EXIT_SUCCESS) {
        pout() ≪ "\t\t\tERROR(b_to_h_0): failure in b_to_h_0_: " ≪ return_status ≪ endl;
        goto exit;
      }
    }     /* for (int dir = 0; dir < SpaceDim; dir++) */
  }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
exit:
  if (watch_b_to_h) pout() ≪ "\tb_to_h_0: returning to caller" ≪ endl;
  return return_status;
}
```

This code is used in chunk 84.

7.6.3 Function `b_to_h_n`

```
113  ⟨ Function b_to_h_n 113 ⟩ ≡
       int b_to_h_n(level *level_ptr)
       {
           /* Convert B to H in every cell of this level (other than zero). */
         int watch_b_to_h = (int) false;
         int return_status = EXIT_SUCCESS;
         int ghost_margin = N_GHOST_CELLS;
         int level_number = level_ptr→number;
         DataIterator data_iterator = level_ptr→data_iterator;
         SCMParmParse watch_parameter_parser("forms.watch");
         watch_parameter_parser.query("b_to_h", watch_b_to_h);
         if (watch_b_to_h) pout() << "\tb_to_h_n:" << endl;
         if (¬level_number) {
           pout() << "\t\tERROR(b_to_h_n): invoked on level zero" << endl;
           return_status = ERR_B_TO_H;
           goto exit;
         }
         swap_idx(level_ptr→H_idx);
         for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
           DataIndex data_index = data_iterator();
           FluxBox &H = level_ptr→H[data_index];
           FluxBox &B = level_ptr→B[data_index];
           FaceFab⟨int⟩ &medium_H = level_ptr→medium_H[data_index];
           Real &time_h = level_ptr→time_h;
           Real &dt = level_ptr→dt;
           for (int dir = 0; dir < SpaceDim; dir++) {
             FArrayBox &H_dir = H[dir];
             FArrayBox &B_dir = B[dir];
             BaseFab⟨int⟩ &medium_H_dir = medium_H[dir];
             b_to_h_n_(&SpaceDim, &dir, &ghost_margin, &(level_ptr→H_idx[0]), &(level_ptr→B_idx[0]), &time_h, &dt,
                 &(B_dir.loVect()[0]), &(B_dir.loVect()[1]), &(B_dir.loVect()[2]), &(B_dir.hiVect()[0]),
                 &(B_dir.hiVect()[1]), &(B_dir.hiVect()[2]), B_dir.dataPtr(), H_dir.dataPtr(),
                 medium_H_dir.dataPtr(), &watch_b_to_h, &return_status);
             if (return_status ≠ EXIT_SUCCESS) {
               pout() << "\t\t\tERROR(b_to_h_n): failure in b_to_h_n_: " << return_status << endl;
               goto exit;
             }
           }     /* for (int dir = 0; dir < SpaceDim; dir++) */
         }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
       exit:
         if (watch_b_to_h) pout() << "\tb_to_h_n: returning to caller" << endl;
         return return_status;
       }
```

This code is used in chunk 84.

8 Scheme Functions

114 ⟨ Scheme.c 114 ⟩ ≡
```
#include "Forms.H"
```
⟨ Function *draw_box* 115 ⟩
⟨ Function *draw_ball* 121 ⟩
⟨ Function *draw_ellipsoid* 127 ⟩
⟨ Auxiliary Scheme Procedures 134 ⟩
⟨ MPI Functions 135 ⟩
⟨ Interpolation 136 ⟩
⟨ Initialize Scheme 137 ⟩

8.1 Draw Box

This is just a wrapper, in this new version, calling an appropriate C++ utility. Its arguments must be two *scm_f64vector*s, and an optional integer, the *color*, with which to draw. If the *color* is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following this one.

115 ⟨ Function *draw_box* 115 ⟩ ≡

```
SCM draw_box(SCM lower_corner, SCM upper_corner, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_lower_corner(SpaceDim, 0.0);
  Vector⟨Real⟩ C_upper_corner(SpaceDim, 0.0);
  bool draw_on_media = false;
  int C_color = 0;
  int watch_draw_box = 0;
  ⟨ draw_box: introduce yourself 116 ⟩
  ⟨ draw_box: read, check, and translate arguments 117 ⟩
  ⟨ draw_box: call appropriate utility 118 ⟩
exit:
  if (watch_draw_box) pout( ) ≪ "draw_box: returning to caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_box 119 ⟩
⟨ Function draw_tags_box 120 ⟩
```

This code is cited in chunk 236.

This code is used in chunk 114.

116 ¶⟨ *draw_box*: introduce yourself 116 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_box", watch_draw_box);
if (watch_draw_box) pout( ) ≪ "draw_box:" ≪ endl;
```

This code is used in chunk 115.

117 ¶⟨ *draw_box*: read, check, and translate arguments 117 ⟩ ≡

```
    /* Are the two first arguments vectors of length SpaceDim, and can they be used for the two corners of
    a box? */
if (scm_is_true(scm_f64vector_p(lower_corner)) ∧ scm_is_true(scm_f64vector_p(upper_corner)) ∧
      (scm_to_int(scm_f64vector_length(lower_corner)) ≡ SpaceDim) ∧
      (scm_to_int(scm_f64vector_length(upper_corner)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_lower_corner[i] = scm_to_double(scm_f64vector_ref(lower_corner, scm_from_int(i)));
    C_upper_corner[i] = scm_to_double(scm_f64vector_ref(upper_corner, scm_from_int(i)));
  }
  if (watch_draw_box) pout( ) ≪ "\tbox corners: #f64(" ≪ C_lower_corner ≪ ") and #f64(" ≪
      C_upper_corner ≪ ")" ≪ endl;
}
else {
  pout( ) ≪ "\tERROR(draw_box): arguments must be two #f64(a b c) vectors" ≪ endl;
  return_status = false;
  goto exit;
}
```

```
for (int i = 0; i < SpaceDim; i++) {
  if (¬(C_lower_corner[i] < C_upper_corner[i])) {
    pout( ) ≪ "\tERROR(draw_box): corners ill-defined" ≪ endl;
    return_status = false;
    goto exit;
  }
}
      /* Is the third, optional, argument provided, and can it be used for color? */
if (SCM_UNBNDP(color)) {
  draw_on_media = false;
}
else {
  draw_on_media = true;
  if (scm_is_integer(color)) {
    C_color = scm_to_int(color);
    if (C_color < 0) {
      pout( ) ≪ "\tERROR(draw_box): negative colors are reserved" ≪ endl;
      return_status = false;
      goto exit;
    }
  }
  else {
    pout( ) ≪ "\tERROR(draw_box): drawing color must be an integer" ≪ endl;
    return_status = false;
    goto exit;
  }
}
```

This code is used in chunk 115.

118 ¶⟨ *draw_box*: call appropriate utility 118 ⟩ ≡

```
if (draw_on_media) return_status = draw_media_box(C_lower_corner, C_upper_corner, C_color);
else return_status = draw_tags_box(C_lower_corner, C_upper_corner);
```

This code is used in chunk 115.

8.1.1 Drawing Box on Media

119 ⟨ Function *draw_media_box* 119 ⟩ ≡

```
int draw_media_box(Vector⟨Real⟩ &C_lower_corner, Vector⟨Real⟩ &C_upper_corner, int &C_color)
{
  extern level *current_level_ptr;
  Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
  Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
  Vector⟨Real⟩ &origin = current_level_ptr→origin;
  Real &delta = current_level_ptr→delta;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator &data_iterator = current_level_ptr→data_iterator;
  IntVect IV_lower_corner = IntVect::Zero;
  IntVect IV_upper_corner = IntVect::Zero;
  int buffer_size = REFINE_BUFFER_SIZE;
  int ghost_margin = N_GHOST_CELLS;
  int watch_draw_box = false;
  int return_status = true;      /* this is for Scheme handling */
  Box the_box;
  SCMParmParse watch_parameter_parser("forms.watch");
```

```
watch_parameter_parser.query("draw_box", watch_draw_box);
if (watch_draw_box) pout() << "\tdraw_media_box:" << endl;
if ((C_lower_corner.size() != SpaceDim) || (C_upper_corner.size() != SpaceDim)) {
  pout() << "\t\tERROR(draw_media_box): badly sized corner vectors" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((signal_xyz_min[i] + buffer_size * delta > C_lower_corner[i]) || (signal_xyz_max[i] - buffer_size * delta <
       C_upper_corner[i])) {
    pout() << "\tERROR(draw_box): box too close to signal injection boundary" << endl;
    return_status = false;
    goto exit;
  }
}
      /* Find approximate IntVect representation of the box corners and pad it. */
for (int i = 0; i < SpaceDim; i++) {
  IV_lower_corner[i] = (int) floor((C_lower_corner[i] - origin[i])/delta + 0.5 + EPSILON) - buffer_size;
  IV_upper_corner[i] = (int) ceil((C_upper_corner[i] - origin[i])/delta - 0.5 - EPSILON) + buffer_size;
}
the_box.define(IV_lower_corner, IV_upper_corner);
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (the_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr->medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_box_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(IV_lower_corner[0]),
          &(IV_upper_corner[0]), &(C_lower_corner[0]), &(C_upper_corner[0]), &C_color,
          &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
          medium_E_dir.dataPtr(), &watch_draw_box, &return_status);
      if (!return_status) {
        pout() << "\t\tERROR(f_draw_box_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr->medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
      f_draw_box_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(IV_lower_corner[0]),
          &(IV_upper_corner[0]), &(C_lower_corner[0]), &(C_upper_corner[0]), &C_color,
          &(H_box_dir.type()[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
          medium_H_dir.dataPtr(), &watch_draw_box, &return_status);
      if (!return_status) {
        pout() << "\t\tERROR(f_draw_box_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
    }
  }
}
```

```
exit: return return_status;
}
```
This code is used in chunk 115.

8.1.2 Drawing Box on Tags

This function can be done very similarly to the one that draws on the media, because *f_draw_box_* is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel anyway.

120 ⟨ Function *draw_tags_box* 120 ⟩ ≡
```
int draw_tags_box(Vector⟨Real⟩ &C_lower_corner, Vector⟨Real⟩ &C_upper_corner)
{
  pout() ≪ "Do not push it. Drawing on tags is not implemented yet." ≪ endl;
  return false;
}
```
This code is used in chunk 115.

8.2 Draw Ball

This is just a wrapper, calling an appropriate C++ utility. Its arguments must be an *scm_f64vector* (the center of the ball), a **double** (its radius), and an optional integer, the *color* with which to draw. If the *color* is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following this one.

121 ⟨ Function *draw_ball* 121 ⟩ ≡

```
SCM draw_ball(SCM center, SCM radius, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_center(SpaceDim, 0.0);
  Real C_radius;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_ball = 0;
  ⟨ draw_ball: introduce yourself 122 ⟩
  ⟨ draw_ball: read, check, and translate arguments 123 ⟩
  ⟨ draw_ball: call appropriate utility 124 ⟩
exit:
  if (watch_draw_ball) pout( ) ≪ "draw_ball:␣returning␣to␣caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_ball 125 ⟩
⟨ Function draw_tags_ball 126 ⟩
```

This code is used in chunk 114.

122 ¶⟨ *draw_ball*: introduce yourself 122 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ball", watch_draw_ball);
if (watch_draw_ball) pout( ) ≪ "draw_ball:" ≪ endl;
```

This code is used in chunk 121.

123 ¶⟨ *draw_ball*: read, check, and translate arguments 123 ⟩ ≡

```
    /* Is the first argument vector of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(center)) ∧ (scm_to_int(scm_f64vector_length(center)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++)
    C_center[i] = scm_to_double(scm_f64vector_ref(center, scm_from_int(i)));
  if (watch_draw_ball) pout( ) ≪ "\tball␣center:␣#f64(" ≪ C_center ≪ ")" ≪ endl;
}
else {
  pout( ) ≪ "\tERROR(draw_ball):␣" ≪ "first␣argument␣must␣be␣an␣#f64(a␣b␣c)␣v\
      ector␣(center)" ≪ endl;
  return_status = false;
  goto exit;
}
if (scm_is_real(radius)) {
  C_radius = scm_to_double(radius);
  if (watch_draw_ball) pout( ) ≪ "\tradius:␣␣␣␣␣␣␣" ≪ C_radius ≪ endl;
}
else {
```

```
    pout() ≪ "\tERROR(draw_ball): " ≪ "second argument must be a real (center)" ≪ endl;
    return_status = false;
    goto exit;
  }
        /* Is the third, optional, argument provided, and can it be used for color? */
  if (SCM_UNBNDP(color)) {
    draw_on_media = false;
  }
  else {
    draw_on_media = true;
    if (scm_is_integer(color)) {
      C_color = scm_to_int(color);
      if (C_color < 0) {
        pout() ≪ "\tERROR(draw_ball): negative colors are reserved" ≪ endl;
        return_status = false;
        goto exit;
      }
    }
    else {
      pout() ≪ "\tERROR(draw_ball): drawing color must be an integer" ≪ endl;
      return_status = false;
      goto exit;
    }
  }
```

This code is used in chunk 121.

124 ¶⟨ *draw_ball*: call appropriate utility 124 ⟩ ≡

```
  if (draw_on_media) return_status = draw_media_ball(C_center, C_radius, C_color);
  else return_status = draw_tags_ball(C_center, C_radius);
```

This code is used in chunk 121.

8.2.1 Drawing Ball on Media

125 ⟨ Function *draw_media_ball* 125 ⟩ ≡

```
  int draw_media_ball(Vector⟨Real⟩ &C_center, Real &C_radius, int &C_color)
  {
    extern level *current_level_ptr;
    Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
    Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
    Vector⟨Real⟩ &origin = current_level_ptr→origin;
    Real &delta = current_level_ptr→delta;
    DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
    DataIterator &data_iterator = current_level_ptr→data_iterator;
    IntVect IV_lower_corner = IntVect::Zero;
    IntVect IV_upper_corner = IntVect::Zero;
    int buffer_size = REFINE_BUFFER_SIZE;
    int ghost_margin = N_GHOST_CELLS;
    int watch_draw_ball = false;
    int return_status = true;      /* this is for Scheme handling */
    Box the_box;
    SCMParmParse watch_parameter_parser("forms.watch");
```

```
watch_parameter_parser.query("draw_ball", watch_draw_ball);
if (watch_draw_ball) pout() << "\tdraw_media_ball:" << endl;
if (C_center.size() != SpaceDim) {
  pout() << "\t\tERROR(draw_media_ball): badly sized center vector" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((signal_xyz_min[i]+buffer_size*delta > C_center[i]-C_radius)∨(signal_xyz_max[i]-buffer_size*delta <
       C_center[i] + C_radius)) {
    pout() << "\tERROR(draw_ball): ball too close to signal injection boundary" << endl;
    return_status = false;
    goto exit;
  }
}
      /* Find approximate IntVect representation of the ball box corners and pad it. */
for (int i = 0; i < SpaceDim; i++) {
  IV_lower_corner[i] = (int) floor((C_center[i] - C_radius - origin[i])/delta + 0.5 + EPSILON) - buffer_size;
  IV_upper_corner[i] = (int) ceil((C_center[i] + C_radius - origin[i])/delta - 0.5 - EPSILON) + buffer_size;
}
the_box.define(IV_lower_corner, IV_upper_corner);
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (the_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr->medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_ball_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(IV_lower_corner[0]),
          &(IV_upper_corner[0]), &(C_center[0]), &C_radius, &C_color, &(E_box_dir.type()[0]), &(E_lo[0]),
          &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]), medium_E_dir.dataPtr(),
          &watch_draw_ball, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_ball_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr->medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
      f_draw_ball_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(IV_lower_corner[0]),
          &(IV_upper_corner[0]), &(C_center[0]), &C_radius, &C_color, &(H_box_dir.type()[0]),
          &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]), medium_H_dir.dataPtr(),
          &watch_draw_ball, &return_status);
      if (¬return_status) {
        pout() << "\t\tERROR(f_draw_ball_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
    }
  }
}
```

```
exit: return return_status;
}
```
This code is used in chunk 121.

8.2.2 Drawing Ball on Tags

This function can be done very similarly to the one that draws a ball on the media, because *f_draw_ball_* is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel anyway.

126 ⟨ Function *draw_tags_ball* 126 ⟩ ≡
```
int draw_tags_ball(Vector⟨Real⟩ &C_center, Real &C_radius)
{
  pout() ≪ "Do not push it. Drawing on tags is not implemented yet." ≪ endl;
  return false;
}
```
This code is used in chunk 121.

8.3 Draw Ellipsoid

This is just a wrapper, calling an appropriate C++ utility. Its arguments must be two *scm_f64vector*s (the focal points of the ellipsoid), a **double** (the sum of distances from a surface point to the focal points), and an optional integer, the *color* with which to draw. If the *color* is provided, the function draws on media with the *color* specified. If not, the function draws on tags with *color* = 1.

The two pure C++ utilities, one for drawing on media and one for drawing on tags, are then specified following this one.

127 ⟨ Function *draw_ellipsoid* 127 ⟩ ≡

```
SCM draw_ellipsoid(SCM focus_1, SCM focus_2, SCM sum, SCM color)
{
  bool return_status = true;
  Vector⟨Real⟩ C_focus_1(SpaceDim, 0.0), C_focus_2(SpaceDim, 0.0);
  Real C_sum;
  int C_color = 0;
  bool draw_on_media = false;
  int watch_draw_ellipsoid = 0;
  ⟨ draw_ellipsoid: introduce yourself 128 ⟩
  ⟨ draw_ellipsoid: read, check, and translate arguments 129 ⟩
  ⟨ draw_ellipsoid: call appropriate utility 130 ⟩
exit:
  if (watch_draw_ellipsoid) pout() ≪ "draw_ellipsoid: returning to caller" ≪ endl;
  return scm_from_bool(return_status);
}
⟨ Function draw_media_ellipsoid 131 ⟩
⟨ Function draw_tags_ellipsoid 132 ⟩
```

This code is used in chunk 114.

128 ¶⟨ *draw_ellipsoid*: introduce yourself 128 ⟩ ≡

```
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ellipsoid", watch_draw_ellipsoid);
if (watch_draw_ellipsoid) pout() ≪ "draw_ellipsoid:" ≪ endl;
```

This code is used in chunk 127.

129 ¶⟨ *draw_ellipsoid*: read, check, and translate arguments 129 ⟩ ≡

```
    /* Is the first two arguments vectors of length SpaceDim? */
if (scm_is_true(scm_f64vector_p(focus_1)) ∧ (scm_to_int(scm_f64vector_length(focus_1)) ≡ SpaceDim) ∧
      scm_is_true(scm_f64vector_p(focus_2)) ∧ (scm_to_int(scm_f64vector_length(focus_2)) ≡ SpaceDim)) {
  for (int i = 0; i < SpaceDim; i++) {
    C_focus_1[i] = scm_to_double(scm_f64vector_ref(focus_1, scm_from_int(i)));
    C_focus_2[i] = scm_to_double(scm_f64vector_ref(focus_2, scm_from_int(i)));
  }
  if (watch_draw_ellipsoid) {
    pout() ≪ "\tellipsoid focus_1: #f64(" ≪ C_focus_1 ≪ ")" ≪ endl;
    pout() ≪ "\t          focus_2: #f64(" ≪ C_focus_2 ≪ ")" ≪ endl;
  }
}
else {
  pout() ≪ "\tERROR(draw_ellipsoid): " ≪ "first two arguments must be #f64(a b c) vectors" ≪
      endl;
  return_status = false;
  goto exit;
```

```
}
if (scm_is_real(sum)) {
  C_sum = scm_to_double(sum);
  if (watch_draw_ellipsoid) pout() ≪ "\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sum:␣" ≪ C_sum ≪ endl;
}
else {
  pout() ≪ "\tERROR(draw_ellipsoid):␣" ≪ "third␣argument␣must␣be␣a␣real␣(center)" ≪ endl;
  return_status = false;
  goto exit;
}
      /* Is the fourth, optional, argument provided, and can it be used for color? */
if (SCM_UNBNDP(color)) {
  draw_on_media = false;
}
else {
  draw_on_media = true;
  if (scm_is_integer(color)) {
    C_color = scm_to_int(color);
    if (C_color < 0) {
      pout() ≪ "\tERROR(draw_ellipsoid):␣negative␣colors␣are␣reserved" ≪ endl;
      return_status = false;
      goto exit;
    }
  }
  else {
    pout() ≪ "\tERROR(draw_ellipsoid):␣drawing␣color␣must␣be␣an␣integer" ≪ endl;
    return_status = false;
    goto exit;
  }
}
```

This code is used in chunk 127.

130 ¶⟨ *draw_ellipsoid*: call appropriate utility 130 ⟩ ≡

```
if (draw_on_media) return_status = draw_media_ellipsoid(C_focus_1, C_focus_2, C_sum, C_color);
else return_status = draw_tags_ellipsoid(C_focus_1, C_focus_2, C_sum);
```

This code is used in chunk 127.

8.3.1 Drawing Ellipsoid on Media

131 ⟨ Function *draw_media_ellipsoid* 131 ⟩ ≡

```
int draw_media_ellipsoid(Vector⟨Real⟩ &C_focus_1, Vector⟨Real⟩ &C_focus_2, Real &C_sum, int
        &C_color)
{
  extern level *current_level_ptr;
  Vector⟨Real⟩ &signal_xyz_min = current_level_ptr→signal_xyz_min;
  Vector⟨Real⟩ &signal_xyz_max = current_level_ptr→signal_xyz_max;
  Vector⟨Real⟩ &origin = current_level_ptr→origin;
  Real &delta = current_level_ptr→delta;
  DisjointBoxLayout &box_layout = current_level_ptr→box_layout;
  DataIterator &data_iterator = current_level_ptr→data_iterator;
  IntVect IV_lower_corner = IntVect::Zero;
  IntVect IV_upper_corner = IntVect::Zero;
  int buffer_size = REFINE_BUFFER_SIZE;
```

```
int ghost_margin = N_GHOST_CELLS;
int watch_draw_ellipsoid = false;
int return_status = true;      /* this is for Scheme handling */
Vector<Real> C_center(SpaceDim, 0.0);
Box the_box;
SCMParmParse watch_parameter_parser("forms.watch");
watch_parameter_parser.query("draw_ellipsoid", watch_draw_ellipsoid);
if (watch_draw_ellipsoid) pout() << "\tdraw_media_ellipsoid:" << endl;
if ((C_focus_1.size() != SpaceDim) || (C_focus_2.size() != SpaceDim)) {
  pout() << "\t\tERROR(draw_media_ellipsoid): badly sized focus vectors" << endl;
  return_status = false;
  goto exit;
}
for (int i = 0; i < SpaceDim; i++) {
  if ((signal_xyz_min[i]+buffer_size*delta > C_focus_1[i]-C_sum)||(signal_xyz_max[i]-buffer_size*delta <
       C_focus_1[i] + C_sum) || (signal_xyz_min[i] + buffer_size * delta >
       C_focus_2[i] - C_sum) || (signal_xyz_max[i] - buffer_size * delta < C_focus_2[i] + C_sum)) {
    pout() << "\tERROR(draw_ellipsoid): ellipsoid too close to signal injection boundary" <<
       endl;
    return_status = false;
    goto exit;
  }
}
      /* Find approximate IntVect representation of the ellipsoid box corners and pad it. */
for (int i = 0; i < SpaceDim; i++) C_center[i] = (C_focus_1[i] + C_focus_2[i])/2.0;
for (int i = 0; i < SpaceDim; i++) {
  IV_lower_corner[i] = (int) floor((C_center[i]- C_sum/2.0- origin[i])/delta+0.5+EPSILON)- buffer_size;
  IV_upper_corner[i] = (int) ceil((C_center[i]+ C_sum/2.0- origin[i])/delta-0.5-EPSILON)+ buffer_size;
}
the_box.define(IV_lower_corner, IV_upper_corner);
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  DataIndex data_index = data_iterator();
  Box layout_box = box_layout.get(data_index);
  if (the_box.intersects(layout_box)) {
      /* This box may need work. This will be ultimately decided by the Fortran routine. */
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = current_level_ptr->medium_E[data_index][dir];
      Box E_box_dir = medium_E_dir.box();
      IntVect E_lo = E_box_dir.smallEnd();
      IntVect E_hi = E_box_dir.bigEnd();
      f_draw_ellipsoid_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(IV_lower_corner[0]),
          &(IV_upper_corner[0]), &(C_focus_1[0]), &(C_focus_2[0]), &C_sum, &C_color,
          &(E_box_dir.type()[0]), &(E_lo[0]), &(E_lo[1]), &(E_lo[2]), &(E_hi[0]), &(E_hi[1]), &(E_hi[2]),
          medium_E_dir.dataPtr(), &watch_draw_ellipsoid, &return_status);
      if (!return_status) {
        pout() << "\t\tERROR(f_draw_ellipsoid_): Fortran problem, dir = " << dir << endl;
        goto exit;
      }
      BaseFab<int> &medium_H_dir = current_level_ptr->medium_H[data_index][dir];
      Box H_box_dir = medium_H_dir.box();
      IntVect H_lo = H_box_dir.smallEnd();
      IntVect H_hi = H_box_dir.bigEnd();
      f_draw_ellipsoid_(&SpaceDim, &ghost_margin, &(origin[0]), &delta, &(IV_lower_corner[0]),
          &(IV_upper_corner[0]), &(C_focus_1[0]), &(C_focus_2[0]), &C_sum, &C_color,
```

```
        &(H_box_dir.type( )[0]), &(H_lo[0]), &(H_lo[1]), &(H_lo[2]), &(H_hi[0]), &(H_hi[1]), &(H_hi[2]),
        medium_H_dir.dataPtr( ), &watch_draw_ellipsoid, &return_status);
      if (¬return_status) {
        pout( ) ≪ "\t\tERROR(f_draw_ellipsoid_): Fortran problem, dir = " ≪ dir ≪ endl;
        goto exit;
      }
    }
  }
}
exit: return return_status;
}
```

This code is used in chunk 127.

8.3.2 Drawing Ellipsoid on Tags

This function can be done very similarly to the one that draws a ellipsoid on the media, because *f_draw_ellipsoid_* is universal enough to do this. For the time being, we skip this, because the current version of the code does not do multilevel anyway.

132 ⟨ Function *draw_tags_ellipsoid* 132 ⟩ ≡

```
int draw_tags_ellipsoid(Vector⟨Real⟩ &C_focus_1, Vector⟨Real⟩ &C_focus_2, Real &C_sum)
{
  pout( ) ≪ "Do not push it. Drawing on tags is not implemented yet." ≪ endl;
  return false;
}
```

This code is used in chunk 127.

8.4 Auxiliary Scheme Procedures

8.4.1 Is a Symbol Defined

Function *does_scm_symbol_exist*, borrowed from [8] with one small modification that fixes the guile-1.8.1 note about deprecated features, does a job that is similar to *scm_c_lookup*( )[9] with one difference: it does not bomb out if the symbol is not found, just returns *false*.

Implementation notes:

scm_str2symbol This function is defined on `"libguile/discouraged.h"`. I have replaced it with *scm_string_to_symbol*.

scm_sym2var This is a Guile internal, defined on `"libguile/modules.h"`.

scm_current_module_lookup_closure( ) Ditto.

This function is also declared and defined in **SCMParmParse** as a protected member. See Section 12.1, page 252.

134 ⟨ Auxiliary Scheme Procedures 134 ⟩ ≡

```
bool does_scm_symbol_exist(const char *name)
{
  SCM symbol;
  SCM variable;
  assert(name ≠ (const char *) Λ);
  symbol = scm_string_to_symbol(scm_from_locale_string(name));
      /* Lookup this symbol in the current module lookup closure, but do not create it automatically if it
      does not exist. This is what the third argument to this function is about. Return SCM_BOOL_F if this is
      the case. */
  variable = scm_sym2var(symbol, scm_current_module_lookup_closure( ), SCM_BOOL_F);
  return (variable ≠ SCM_BOOL_F);
}
```

This code is used in chunk 114.

[9] Defined in "Accessing Modules from C" in "The Guile Module System" section of the "Modules" chapter.

8.5 Functions for MPI Connectivity

These are wrappers around Chombo functions that themselves are wrappers around MPI functions. Nothing special. The purpose is to extend the most basic MPI functionality to the Scheme layer of the program. Expected use is in user defined flux functions and similar. See Section 3.5.3, page 119 for an example of their use.

135 ⟨ MPI Functions 135 ⟩ ≡

```
SCM SCMnumProc( )
{     /* Return number of processes in the MPI pool. */
  int pool_size = 0;
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  pool_size = numProc( );
  if (watch_scheme) pout( ) << "\tSCMnumProc: procs in pool = " << pool_size << endl;
  return (scm_from_int(pool_size));
}
SCM SCMprocID( )
{     /* Return this process' rank. */
  int rank = 0;
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  rank = procID( );
  if (watch_scheme) pout( ) << "\tSCMprocID: my rank = " << rank << endl;
  return (scm_from_int(rank));
}
SCM SCMbarrier( )
{     /* Synchronize all processes. */
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  if (watch_scheme) pout( ) << "\tSCMBarrier: synchronizing... ";
  barrier( );
  if (watch_scheme) pout( ) << "done." << endl;
  return (SCM_BOOL_T);
}
SCM SCMuniqueProc( )
{     /* Return a unique process ID for this task. */     /* The only task defined is compute. */
  int watch_scheme = 0;
  int unique_process = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  unique_process = uniqueProc(SerialTask::compute);
  if (watch_scheme) pout( ) << "\tSCMuniqueProc: unique proc rank = " << unique_process << endl;
  return (scm_from_int(unique_process));
}
SCM SCMgather(SCM collection_vector, SCM item, SCM destination)
{     /* Gather items from all processes on the collection_vector. */     /* Only the destination process
       does this. */     /* Defined for item being Scheme integer or real only. */
  int watch_scheme = 0;
  int root_process_rank = scm_to_int(destination);
  int number_of_processes = numProc( );
```

```
  int my_rank = procID( );
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  if (watch_scheme) pout( ) ≪ "SCMgather:" ≪ endl;      /* Is collection_vector appropriately sized? */
  if (scm_c_vector_length(collection_vector) < number_of_processes) {
    pout( ) ≪ "\tSCMgather: ERROR: data vector size too small." ≪ endl;
    return (SCM_BOOL_F);
  }
  else if (scm_is_real(item)) {
      /* Scheme reals will answer positively to integer? if they are also integers, so we cannot capture
        this. We treat them all like doubles. */
    Vector⟨Real⟩ vector_of_items(number_of_processes, (Real) 0.0);
    Real datum = (Real) scm_to_double(item);
    if (watch_scheme) pout( ) ≪ "\tSCMgather: sending " ≪ datum ≪ endl;
    gather(vector_of_items, datum, root_process_rank);
    if (my_rank ≡ root_process_rank) {
      for (int count = 0; count < number_of_processes; count++)
        scm_c_vector_set_x(collection_vector, count, scm_from_double((double) vector_of_items[count]));
      if (watch_scheme) pout( ) ≪ "\tSCMgather: received " ≪ vector_of_items ≪ endl;
    }
    return (SCM_BOOL_T);
  }
  else {
    pout( ) ≪ "\tSCMgather: ERROR: cannot gather this type." ≪ endl;
    return (SCM_BOOL_F);
  }
}
SCM SCMbroadcast(SCM item, SCM source)
{     /* The source process broadcasts item to all processes. */
      /* The item may be Scheme integer or real. */
  int watch_scheme = 0;
  int source_process_rank = scm_to_int(source);
  int my_rank = procID( );
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
  if (watch_scheme) pout( ) ≪ "SCMbroadcast:" ≪ endl;
  if (scm_is_real(item)) {
      /* Scheme reals will answer positively to integer? if they are also integers, so we can't capture this
        condition. We treat all numbers like doubles. */
    Real receive = (Real) scm_to_double(item);
    if (watch_scheme ∧ (my_rank ≡ source_process_rank))
      pout( ) ≪ "\tSCMbroadcast: sending: " ≪ receive ≪ endl;
    broadcast(receive, source_process_rank);
    if (watch_scheme) pout( ) ≪ "\tSCMbroadcast: received: " ≪ receive ≪ endl;
    return (scm_from_double(receive));
  }
  else {
    pout( ) ≪ "\tSCMbroadcast: ERROR: cannot broadcast this type." ≪ endl;
    return (SCM_BOOL_F);
  }
}
```

This code is cited in chunks 13 and 46.

This code is used in chunk 114.

8.6 Interpolation

Here we implement functions discussed in Section 3.5.4, page 121.

136 ⟨ Interpolation 136 ⟩ ≡

```
SCM SCMinterpolate(SCM scm_name, SCM scm_x, SCM scm_y, SCM scm_z)
{
    /* Return the interpolated value of the scm_name field, which must be a two character string such as
       "Ex", "Hy", or "Bz", at (x, y, z), which are expected to be passed as three Scheme reals—these map
       directly onto C's doubles. The interpolation is restricted to level 0 at present. */
  extern Vector<level *> *levels_ptr;
  level *level_0_ptr = (*levels_ptr)[0];
  Vector<Real> &origin = level_0_ptr->origin;
  Real &delta = level_0_ptr->delta;
  DisjointBoxLayout &box_layout = level_0_ptr->box_layout;
  DataIterator &data_iterator = level_0_ptr->data_iterator;
  Vector<Real> position(SpaceDim);
  string name = "AA";
  int name_length = 2;
  int watch_scheme = 0;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("scheme", watch_scheme);
      /* Here we check if the input paramers, scm_name, scm_x, scm_y, and scm_z are correct. We abort
      returning SCM_BOOL_F if not. */
  if (scm_is_real(scm_x)) {
    position[0] = (Real) scm_to_double(scm_x);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: x = " << position[0] << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: x must be real" << endl;
    return (SCM_BOOL_F);
  }
  if (scm_is_real(scm_y)) {
    position[1] = (Real) scm_to_double(scm_y);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: y = " << position[1] << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: y must be real" << endl;
    return (SCM_BOOL_F);
  }
  if (scm_is_real(scm_z)) {
    position[2] = (Real) scm_to_double(scm_z);
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: z = " << position[2] << endl;
  }
  else {
    pout() << "\tSCMinterpolate: ERROR: z must be real" << endl;
    return (SCM_BOOL_F);
  }
  if (scm_is_string(scm_name)) {
    char buffer[name_length + 1];
    buffer[name_length] = (char) 0;
    int count = scm_to_locale_stringbuf(scm_name, buffer, name_length);
    if (count ≠ name_length) {
      pout() << "\tSCMinterpolate: ERROR: bad field name length (must be 2)" << endl;
      return (SCM_BOOL_F);
```

```
  }
  if (watch_scheme > 1) pout() << "\tSCMinterpolate: buffer = " << buffer << endl;
  name.replace(0, 2, buffer);
  if (watch_scheme > 1) pout() << "\tSCMinterpolate: name = " << name << endl;
}
else {
  pout() << "\tSCMinterpolate: ERROR: field name must be string" << endl;
  return (SCM_BOOL_F);
}
      /* Loop over all boxes under this process control. */
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  FArrayBox *field;
  int current_idx, old_idx;
  bool need_old_field = false;
  DataIndex data_index = data_iterator();
      /* We match the name against all fields that we know about. If we find a match, we point field to
      the field and set current_idx to the current index of the field. For magnetic fields, we also set the
      old_idx, because it will be needed for time interpolation. */
  if (name == "Ex") {
    field = &(level_0_ptr->E[data_index][0]);
    current_idx = level_0_ptr->E_idx[0];
  }
  else if (name == "Ey") {
    field = &(level_0_ptr->E[data_index][1]);
    current_idx = level_0_ptr->E_idx[0];
  }
  else if (name == "Ez") {
    field = &(level_0_ptr->E[data_index][2]);
    current_idx = level_0_ptr->E_idx[0];
  }
  else if (name == "Dx") {
    field = &(level_0_ptr->D[data_index][0]);
    current_idx = level_0_ptr->D_idx[0];
  }
  else if (name == "Dy") {
    field = &(level_0_ptr->D[data_index][1]);
    current_idx = level_0_ptr->D_idx[0];
  }
  else if (name == "Dz") {
    field = &(level_0_ptr->D[data_index][2]);
    current_idx = level_0_ptr->D_idx[0];
  }
  else if (name == "Hx") {
    field = &(level_0_ptr->H[data_index][0]);
    current_idx = level_0_ptr->H_idx[0];
    old_idx = level_0_ptr->H_idx[1];
    need_old_field = true;
  }
  else if (name == "Hy") {
    field = &(level_0_ptr->H[data_index][1]);
    current_idx = level_0_ptr->H_idx[0];
    old_idx = level_0_ptr->H_idx[1];
    need_old_field = true;
  }
```

```
else if (name ≡ "Hz") {
  field = &(level_0_ptr→H[data_index][2]);
  current_idx = level_0_ptr→H_idx[0];
  old_idx = level_0_ptr→H_idx[1];
  need_old_field = true;
}
else if (name ≡ "Bx") {
  field = &(level_0_ptr→B[data_index][0]);
  current_idx = level_0_ptr→B_idx[0];
  old_idx = level_0_ptr→B_idx[1];
  need_old_field = true;
}
else if (name ≡ "By") {
  field = &(level_0_ptr→B[data_index][1]);
  current_idx = level_0_ptr→B_idx[0];
  old_idx = level_0_ptr→B_idx[1];
  need_old_field = true;
}
else if (name ≡ "Bz") {
  field = &(level_0_ptr→B[data_index][2]);
  current_idx = level_0_ptr→B_idx[0];
  old_idx = level_0_ptr→B_idx[1];
  need_old_field = true;
}
else {
  pout( ) ≪ "\tSCMinterpolate: ERROR: field name not recognized" ≪ endl;
  return (SCM_BOOL_F);
}
```

/* Find the low corner of the cell from which to obtain data. Shrink the field box, so as to drop the ghost regions, and check if the cell's low corner fits in the box. Proceed only, if it does, otherwise go to the next *data_index*. Why do we do all of this here only, and not up-front? This is because we need to know about the type of the box—even to evaluate its low corner—and this we can extract from the specific **FArrayBox** only. */

```
Box full_box = field→box( );
IntVect low_corner, box_type = full_box.type( );
for (int i = 0; i < SpaceDim; i++)
  low_corner[i] = (int) floor((position[i] − (origin[i] − 0.5 ∗ box_type[i] ∗ delta))/delta);
Box proper_box = grow(full_box, (−1) ∗ N_GHOST_CELLS);
for (int i = 0; i < SpaceDim; i++)
  if (box_type[i]) proper_box.enclosedCells(i);
if (proper_box.contains(low_corner)) {
  Vector⟨IntVect⟩ corner(8);
  Vector⟨Real⟩ f(8);
  Real one_by_delta = 1.0/delta;
  Real one_by_delta_square = one_by_delta ∗ one_by_delta;
  Real one_by_delta_cube = one_by_delta_square ∗ one_by_delta;
  if (watch_scheme > 1)
    pout( ) ≪ "\tSCMinterpolate: found my cell, interpolating..." ≪ endl;
  corner[0] = low_corner;      /* (0, 0, 0) */
  corner[1] = low_corner + BASISV(2);      /* (0, 0, 1) */
  corner[2] = low_corner + BASISV(1);      /* (0, 1, 0) */
  corner[3] = low_corner + BASISV(1) + BASISV(2);      /* (0, 1, 1) */
  corner[4] = low_corner + BASISV(0);      /* (1, 0, 0) */
  corner[5] = low_corner + BASISV(0) + BASISV(2);      /* (1, 0, 1) */
```

```
        corner[6] = low_corner + BASISV(0) + BASISV(1);      /* (1, 1, 0) */
        corner[7] = low_corner + BASISV(0) + BASISV(1) + BASISV(2);      /* (1, 1, 1) */
            /* Compare (above and below) with equations (302–309) in Section 3.5.4, page 121. */
        for (int i = 0; i < 8; i++) f[i] = field->get(corner[i], current_idx);
            /* For B and H interpolate in time as well. */
        if (need_old_field)
          for (int i = 0; i < 8; i++) f[i] = 0.5 * (f[i] + field->get(corner[i], old_idx));
            /* Compare with equations (314–321) in Section 3.5.4, page 121. */
        Real c0 = f[0];
        Real cz = (f[1] - f[0]) * one_by_delta;
        Real cy = (f[2] - f[0]) * one_by_delta;
        Real cx = (f[4] - f[0]) * one_by_delta;
        Real cyz = (f[3] - f[2] - f[1] + f[0]) * one_by_delta_square;
        Real czx = (f[5] - f[4] - f[1] + f[0]) * one_by_delta_square;
        Real cxy = (f[6] - f[4] - f[2] + f[0]) * one_by_delta_square;
        Real cxyz = (f[7] + f[1] + f[2] - f[3] + f[4] - f[5] - f[6] - f[0]) * one_by_delta_cube;
        Real x0 = origin[0] + (low_corner[0] - 0.5 * box_type[0]) * delta;
        Real y0 = origin[1] + (low_corner[1] - 0.5 * box_type[1]) * delta;
        Real z0 = origin[2] + (low_corner[2] - 0.5 * box_type[2]) * delta;
        Real dx = position[0] - x0;
        Real dy = position[1] - y0;
        Real dz = position[2] - z0;
            /* Compare with equation 301 in Section 3.5.4, page 121. */
        Real f_int = cxyz * dx * dy * dz + cyz * dy * dz + czx * dz * dx + cxy * dx * dy + cx * dx + cy * dy + cz * dz + c0;
        if (watch_scheme > 1) pout() << "\tSCMinterpolate: returning " << f_int << endl;
        return (scm_from_double((double) f_int));
      }     /* if (proper_box.contains(low_corner)) */
    }     /* for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) */
    if (watch_scheme > 1) pout() << "\tSCMinterpolate: cell not found, returning zero" << endl;
        /* Perhaps we should return SCM_BOOL_F instead and leave it to the caller to sort things out? But
        then, how will the caller know why it's SCM_BOOL_F? Some other marker is needed here. */
    return (scm_from_double((double) 0.0));
  }
```

This code is cited in chunks 13 and 33.

This code is used in chunk 114.

8.7 Initialize Scheme

Here we condition the Scheme process on its start up. This is done *after* the input file is read.

We begin by registering the C++ implemented Scheme procedures with Scheme. This, as was pointed out in Section 5, page 132, module ⟨ Inner Main 53 ⟩, must be done at the beginning of the program, so that the procedures become available to Scheme before any user code is processed.

Also, on this occasion we create certain bindings and initialize certain variables—especially vectors. These can be used in lambda expressions that will be provided by users.

We *define* primitive subroutines, written in C++, by calling *scm_c_define_gsubr*. The arguments to *scm_c_define_gsubr* are as follows:

`"draw-box"` This is the Scheme name of the function.

$2, 2, 0$ Here we tell Scheme that the function has two required arguments, two optional arguments, and zero *rest* arguments.

(**SCM**(∗)( ))*draw_box* This is the C++ name of the function, and it is also *cast* here on what *scm_c_define_gsubr* expects in this slot.

> In the "Tutorial Introduction to Guile" by David Drysdale [5] *gh_new_procedure* from the currently deprecated GH toolkit was used to register C-functions with Scheme. The Guile-1.8.1 documentation recommends using *scm_c_define_gsubr* instead, but the example that is provided does not work. Luckily, Michael Gran shows how to do this in [8]. It turns out that an explicit cast to (**SCM**(∗)( )) is needed in front of the function C-name. The compiler complains otherwise.

137 ⟨ Initialize Scheme 137 ⟩ ≡

```
    /* global variables */
  Real nx, ny, nz;
  int normalized;
  bool have_post_iteration_lambda, have_post_all_lambda;
  SCM ex_lambda, ey_lambda, ez_lambda, e_lambda, post_iteration_lambda, post_all_lambda;
  SCM E_var, E_old_var, D_var, D_old_var, S_var, S_var_ref;
  SCM medium_var, direction_var, t_e_var, dt_var;
  int initialize_scheme( )
  {
    /* To be run before the first instantiation of SCMParmParse class, which reads the input file. */
    int return_status = EXIT_SUCCESS;
      /* Register new Scheme functions */
      /* ... Functions for drawing on media and tag arrays */
    scm_c_define_gsubr("draw-box", 2, 1, 0, (SCM(*)( ))draw_box);
    scm_c_define_gsubr("draw-ball", 2, 1, 0, (SCM(*)( ))draw_ball);
    scm_c_define_gsubr("draw-ellipsoid", 3, 1, 0, (SCM(*)( ))draw_ellipsoid);
      /* ... Functions for MPI operations */
    scm_c_define_gsubr("num-proc", 0, 0, 0, (SCM(*)( ))SCMnumProc);
    scm_c_define_gsubr("proc-id", 0, 0, 0, (SCM(*)( ))SCMprocID);
    scm_c_define_gsubr("barrier", 0, 0, 0, (SCM(*)( ))SCMbarrier);
    scm_c_define_gsubr("unique-proc", 0, 0, 0, (SCM(*)( ))SCMuniqueProc);
    scm_c_define_gsubr("gather", 3, 0, 0, (SCM(*)( ))SCMgather);
    scm_c_define_gsubr("broadcast", 2, 0, 0, (SCM(*)( ))SCMbroadcast);
      /* ... Functions to access field data */
    scm_c_define_gsubr("interpolate", 4, 0, 0, (SCM(*)( ))SCMinterpolate);
      /* End of function registration section */
      /* Define some Scheme symbols to be used by lambdas. This part used to live inside what is
      now called post_initialize_scheme( ), but we want it up-front, so that the symbols are available to
      user defined pre-iteration operations. Pointers to the associated variables are extracted in the
      post_initialize_scheme( ) below, because SCMParmParse::variable function is needed to do this. */
```

```
  scm_c_define("forms.spacedim", scm_from_int(SpaceDim));
  scm_c_eval_string("(define␣forms.E␣0.0)\n␣␣␣␣␣␣␣␣(define␣forms.E_old␣0.0)\n␣␣␣␣\
      ␣␣␣(define␣forms.D␣0.0)\n␣␣␣␣␣␣␣␣(define␣forms.D_old␣0.0)\n␣␣␣␣␣␣␣␣(define␣forms.m\
      edium␣0)\n␣␣␣␣␣␣␣␣(define␣forms.direction␣0)\n␣␣␣␣␣␣␣␣(define␣forms.time_e␣0.0)\n␣\
      ␣␣␣␣␣␣(define␣forms.dt␣0.1)\n");
  return return_status;
}
int post_initialize_scheme( )
{
  SCMParmParse forms_parser("forms");
  SCMParmParse watch_parser("forms.watch");
  int watch_initialize_scheme = (int) false;
  int return_status = EXIT_SUCCESS;
  watch_parser.query("initialize_scheme", watch_initialize_scheme);
  if (watch_initialize_scheme) {
    pout( ) << "initialize_scheme:" << endl;
    pout( ) << "\tloading␣C++␣functions" << endl;
  }
  if (watch_initialize_scheme) pout( ) << "\tdefining␣parameters" << endl;
  if (SpaceDim ≠ 3) {
    pout( ) << "\tERROR(initialize_scheme):␣SpaceDim␣=/=␣3" << endl;
    return_status = ERR_INITIALIZE_SCHEME;
    goto exit;
  }
  else {
      /* Define Scheme variables to be used by C-lambdas. This part used to live inside one of the if
        statements below, but it's needed globally. The symbols themselves are now defined up-front before
        the first SCMParmParse is invoked to read the input file. */
    E_var = forms_parser.variable("E");
    E_old_var = forms_parser.variable("E_old");
    D_var = forms_parser.variable("D");
    D_old_var = forms_parser.variable("D_old");
    medium_var = forms_parser.variable("medium");
    direction_var = forms_parser.variable("direction");
    t_e_var = forms_parser.variable("time_e");
    dt_var = forms_parser.variable("dt");
      /* Read the signal group */
    if (watch_initialize_scheme) pout( ) << "\treading␣the␣signal␣group␣..." << flush;
        /* Read n and transfer the data to a global variable. */
    SCMParmParse signal_parameter_parser("forms.signal");
    Vector<Real> n(SpaceDim);
    if (¬signal_parameter_parser.queryarr("direction", n, 0, SpaceDim)) {
      pout( ) << "\n\tERROR(initialize_scheme):␣no␣signal␣propagation␣direction" << endl;
      return_status = ERR_INITIALIZE_SCHEME;
      goto exit;
    }
    else {
      /* have acquired n */
      Real length = sqrt(n[0] * n[0] + n[1] * n[1] + n[2] * n[2]);
      if (length > 0.0) {
        extern Real nx, ny, nz;
        nx = n[0]/length;
        ny = n[1]/length;
        nz = n[2]/length;
```

```
    if (watch_initialize_scheme) pout() ≪ "␣direction," ≪ flush;
  }
  else {
    pout() ≪ "\n\tERROR(initialize_scheme):␣zero␣direction" ≪ endl;
    return_status = ERR_INITIALIZE_SCHEME;
    goto exit;
  }
}
if (¬signal_parameter_parser.query("normalized", normalized)) {
  pout() ≪ "\n\tWARNING(initialize_scheme):␣signal␣not␣normalized" ≪ endl;
  normalized = (int) false;
}
else if (¬normalized) {
  pout() ≪ "\n\tWARNING(initialize_scheme):␣signal␣not␣normalized" ≪ endl;
}
else {
  if (watch_initialize_scheme) pout() ≪ "␣normalization," ≪ flush;
}
if (¬signal_parameter_parser.isProcedure("ex_lambda")) {
  pout() ≪ "\n\tERROR(initialize_scheme):␣ex_lambda␣not␣provided" ≪ endl;
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
  ex_lambda = scm_variable_ref(signal_parameter_parser.variable("ex_lambda"));
  if (watch_initialize_scheme) pout() ≪ "␣ex_lambda," ≪ flush;
}
if (¬signal_parameter_parser.isProcedure("ey_lambda")) {
  pout() ≪ "\n\tERROR(initialize_scheme):␣ey_lambda␣not␣provided" ≪ endl;
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
  ey_lambda = scm_variable_ref(signal_parameter_parser.variable("ey_lambda"));
  if (watch_initialize_scheme) pout() ≪ "␣ey_lambda," ≪ flush;
}
if (¬signal_parameter_parser.isProcedure("ez_lambda")) {
  pout() ≪ "\n\tERROR(initialize_scheme):␣ez_lambda␣not␣provided" ≪ endl;
  return_status = ERR_INITIALIZE_SCHEME;
  goto exit;
}
else {
  ez_lambda = scm_variable_ref(signal_parameter_parser.variable("ez_lambda"));
  if (watch_initialize_scheme) pout() ≪ "␣ez_lambda" ≪ endl ≪ flush;
}
      /* Read the media group */
if (watch_initialize_scheme) pout() ≪ "\treading␣the␣media␣group␣..." ≪ flush;
SCMParmParse media_parameter_parser("forms.media");
int number_of_auxiliary_fields = 0;
if (¬media_parameter_parser.query("number_of_auxiliary_fields", number_of_auxiliary_fields)) {
  pout() ≪ "\n\tWARNING(initialize_scheme):␣number_of_auxiliary_fields␣not␣defined" ≪
      endl;
}
else {
```

```
    if (watch_initialize_scheme) pout() ≪ "␣number_of_auxiliary_fields" ≪ flush;
  }
  if (¬media_parameter_parser.isProcedure("distribution.lambda")) {
    pout() ≪ "\n\tWARNING(initialize_scheme):␣distribution␣lambda␣not␣provided" ≪ endl;
  }
  else {
    if (watch_initialize_scheme) pout() ≪ "␣distribution.lambda" ≪ flush;
    if (¬media_parameter_parser.isProcedure("model.e_lambda")) {
      pout() ≪ "\n\tERROR(initialize_scheme):␣model.e_lambda␣not␣provided" ≪ endl;
      return_status = ERR_INITIALIZE_SCHEME;
      goto exit;
    }
    else {
      e_lambda = scm_variable_ref(media_parameter_parser.variable("model.e_lambda"));
      if (watch_initialize_scheme) pout() ≪ "␣model.e_lambda" ≪ flush;
      if (number_of_auxiliary_fields) {
        char command[BUFSIZ];
        sprintf(command, "(define␣forms.S␣(make-f64vector␣%d␣0.0))\n",
            number_of_auxiliary_fields);
        scm_c_eval_string(command);
        S_var = forms_parser.variable("S");
        S_var_ref = scm_variable_ref(S_var);
      }
    }    /* if (¬media_parameter_parser.isProcedure("model.e_lambda")) else clause */
  }    /* if (¬media_parameter_parser.isProcedure("distribution.lambda")) else clause */
       /* Read the post group */
  if (watch_initialize_scheme) pout() ≪ "\treading␣the␣post␣group␣..." ≪ endl;
  SCMParmParse post_parser("forms.post");
  if (¬post_parser.isProcedure("iteration.lambda")) {
    pout() ≪ "\n\tWARNING(initialize_scheme):␣post-iteration␣lambda␣not␣provided" ≪ endl;
    have_post_iteration_lambda = false;
  }
  else {
    post_iteration_lambda = scm_variable_ref(post_parser.variable("iteration.lambda"));
    have_post_iteration_lambda = true;
  }    /* if (¬post_parser.isProcedure("iteration.lambda")) else clause */
  if (¬post_parser.isProcedure("all.lambda")) {
    pout() ≪ "\n\tWARNING(initialize_scheme):␣post-all␣lambda␣not␣provided" ≪ endl;
    have_post_all_lambda = false;
  }
  else {
    post_all_lambda = scm_variable_ref(post_parser.variable("all.lambda"));
    have_post_all_lambda = true;
  }    /* if (¬post_parser.isProcedure("all.lambda")) else clause */
}    /* if (SpaceDim ≠ 3) { } else */
exit:
  if (watch_initialize_scheme) pout() ≪ "initialize_scheme:␣returning␣to␣caller" ≪ endl;
  return return_status;
}
```

This code is cited in chunks 18 and 46.

This code is used in chunk 114.

9 IO Functions

138 ⟨ IO.c 138 ⟩ ≡
```
#include "Forms.H"
```
⟨ Function *dump_data* 139 ⟩

9.1 Dump Data

139 ⟨ Function *dump_data* 139 ⟩ ≡

```
int dump_data(Vector⟨level ∗⟩ &levels, int label)
{
  char file_name[BUFSIZ];
  char format[BUFSIZ];
  string root;
  int number_of_precomputes = 0;
  int number_of_outputs = 0;
  int number_of_digits = 0;
  int number_of_levels = levels.size( );
  int return_status = EXIT_SUCCESS;
  int watch_dump_data = (int) false;
  int report_outliers = (int) false;
  int Ex_idx = −1, Ey_idx = −1, Ez_idx = −1, Dx_idx = −1, Dy_idx = −1, Dz_idx = −1, Hx_idx = −1,
      Hy_idx = −1, Hz_idx = −1, Bx_idx = −1, By_idx = −1, Bz_idx = −1, En_idx = −1, Px_idx = −1,
      Py_idx = −1, Pz_idx = −1;
  Vector⟨string⟩ vector_of_precomputes, vector_of_outputs;
  Vector⟨DisjointBoxLayout⟩ vector_of_box_layouts;
  Vector⟨LevelData⟨FArrayBox⟩ ∗⟩ vector_of_ptrs_to_level_data;
  Vector⟨int⟩ vector_of_refinements;
  vector_of_precomputes.clear( );
  vector_of_outputs.clear( );
  vector_of_box_layouts.clear( );
  vector_of_ptrs_to_level_data.clear( );
  vector_of_refinements.clear( );
  SCMParmParse output_parameter_parser("forms.output");
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("dump_data", watch_dump_data);
  if (watch_dump_data) pout( ) ≪ "\t\tdump_data:" ≪ endl;
  output_parameter_parser.query("report_outliers", report_outliers);
  if ((number_of_precomputes = output_parameter_parser.countval("precompute")) ≤ 0) goto exit;
  if ((number_of_outputs = output_parameter_parser.countval("write")) ≤ 0) goto exit;
      /∗ If we're here, it means that we have something to output and that we have something to
      precompute. ∗/
  output_parameter_parser.getarr("precompute", vector_of_precomputes, 0, number_of_precomputes);
  output_parameter_parser.getarr("write", vector_of_outputs, 0, number_of_outputs);
  if (¬output_parameter_parser.query("filename.root", root)) root = FORMS_OUTPUT;
  if (¬output_parameter_parser.query("filename.number_of_digits", number_of_digits))
    number_of_digits = OUTPUT_NUMBER_OF_DIGITS;
  sprintf (format, "%%s_%%%d.%dd.hdf5", number_of_digits, number_of_digits);
  sprintf (file_name, format, root.c_str( ), label);
  if (watch_dump_data) pout( ) ≪ "\t\t\tprecomputing␣" ≪ vector_of_precomputes ≪ endl;
      /∗ Now we precompute the fields specified ∗/
  for (int n = 0; n < number_of_levels; n++) {
    DataIterator &data_iterator = levels[n]→data_iterator;
    int &E_idx = levels[n]→E_idx[0];
    int &D_idx = levels[n]→D_idx[0];
    int &B_idx = levels[n]→B_idx[0];
    int &H_idx = levels[n]→H_idx[0];
    for (data_iterator.reset( ); data_iterator.ok( ); ++data_iterator) {
      DataIndex data_index = data_iterator( );
      CurlBox &E = levels[n]→E[data_index];
```

```
      CurlBox &D = levels[n]→D[data_index];
      FluxBox &B = levels[n]→B[data_index];
      FluxBox &H = levels[n]→H[data_index];
      FArrayBox &Out = levels[n]→Out[data_index];
      FArrayBox &OutWrite = levels[n]→OutWrite[data_index];
      if (watch_dump_data) {
        pout() ≪ "\t\t\tbox:␣" ≪ Out.box() ≪ endl;
        pout() ≪ "\t\t\t\t" ≪ flush;
      }
      for (int count = 0; count < number_of_precomputes; count++) {
        string field = vector_of_precomputes[count];
        if (watch_dump_data) pout() ≪ field ≪ "␣" ≪ flush;
        ⟨ dump_data: precompute field 141 ⟩
      }
      if (watch_dump_data) pout() ≪ endl;
          /* Alas, this is not the end of the story. Having precomputed the fields, we now must select the
          ones for writing and assemble them in OutWrite. It should not be necessary to do this with
          copy, but right now this is how I do it, because WriteAMRHierarchyHDF5 can only output the
          whole LevelData and not specified components of it. I will have to write my own version of it
          to optimize this part of the code. */
      if (watch_dump_data) {
        pout() ≪ "\t\t\t\tAssembling␣fields␣for␣writing␣out" ≪ endl;
        pout() ≪ "\t\t\t\t" ≪ flush;
      }
      for (int count = 0; count < number_of_outputs; count++) {
        string field = vector_of_outputs[count];
        if (watch_dump_data) pout() ≪ field ≪ "␣" ≪ flush;
        ⟨ dump_data: extract field from precomputes 146 ⟩
      }
      if (watch_dump_data) pout() ≪ endl;
    }
    vector_of_box_layouts.push_back(levels[n]→box_layout);
    vector_of_ptrs_to_level_data.push_back(&(levels[n]→OutWrite));
    vector_of_refinements.push_back(REFINE_RATIO);
  }
  if (watch_dump_data) {
    pout() ≪ "\t\t\tWriting␣" ≪ vector_of_outputs ≪ "on␣" ≪ file_name ≪ endl;
    pout() ≪ "\t\t\t\trefinements␣␣" ≪ vector_of_refinements ≪ endl;
    pout() ≪ "\t\t\t\tdomain␣␣␣␣␣␣␣" ≪ levels[0]→domain ≪ endl;
    pout() ≪ "\t\t\t\tdelta␣␣␣␣␣␣␣␣" ≪ levels[0]→delta ≪ endl;
    pout() ≪ "\t\t\t\tdt␣␣␣␣␣␣␣␣␣␣" ≪ levels[0]→dt ≪ endl;
    pout() ≪ "\t\t\t\ttime_e␣␣␣␣␣␣␣" ≪ levels[0]→time_e ≪ endl;
    pout() ≪ "\t\t\t\tlevels␣␣␣␣␣␣␣" ≪ number_of_levels ≪ endl;
    pout() ≪ "\t\t\t\t\tlayouts␣␣␣␣" ≪ vector_of_box_layouts.size() ≪ endl;
    pout() ≪ "\t\t\t\t\tdatasets␣␣␣" ≪ vector_of_ptrs_to_level_data.size() ≪ endl;
    pout() ≪ "\t\t\t\t\tcomponents␣" ≪ vector_of_ptrs_to_level_data[0]→nComp() ≪ endl;
  }
  if (report_outliers) {
    pout() ≪ "\t\t\t\toutliers:" ≪ endl;
    for (int level_number = 0; level_number < number_of_levels; level_number++) {
      LevelData⟨FArrayBox⟩ &OutWrite = *(vector_of_ptrs_to_level_data[level_number]);
      DataIterator &data_iterator = levels[level_number]→data_iterator;
      pout() ≪ "\t\t\t\t\tlevel␣" ≪ level_number ≪ endl;
      for (int count = 0; count < number_of_outputs; count++) {
```

```
        pout() ≪ "\t\t\t\t\t\t" ≪ vector_of_outputs[count] ≪ "␣" ≪ flush;
        Real minimum = 0.0, maximum = 0.0;
        IntVect minimum_location = IntVect::Zero, maximum_location = IntVect::Zero;
        for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
          FArrayBox &field = OutWrite[data_iterator()];
          Real local_minimum, local_maximum;
          IntVect local_minimum_location, local_maximum_location;
          local_minimum = field.min(count);
          local_minimum_location = field.minIndex(count);
          local_maximum = field.max(count);
          local_maximum_location = field.maxIndex(count);
          if (local_minimum < minimum) {
            minimum = local_minimum;
            minimum_location = local_minimum_location;
          }
          if (local_maximum > maximum) {
            maximum = local_maximum;
            maximum_location = local_maximum_location;
          }
        }
        pout() ≪ minimum ≪ "␣at␣" ≪ minimum_location ≪ ",␣" ≪ maximum ≪ "␣at␣" ≪
            maximum_location ≪ endl;
      }
    }
  }
  WriteAMRHierarchyHDF5(file_name, vector_of_box_layouts, vector_of_ptrs_to_level_data, vector_of_outputs,
      levels[0]→domain, levels[0]→delta, levels[0]→dt, levels[0]→time_e, vector_of_refinements, number_of_levels);
  if (watch_dump_data) pout() ≪ "\t\t\t...␣done." ≪ endl;
exit: return return_status;
}
```

This code is cited in chunk 53.

This code is used in chunk 138.

9.2 Precompute Fields for Output

9.2.1 Space Interpolate Electric Fields

141 ⟨ *dump_data*: precompute field 141 ⟩ ≡

```
if ((field ≡ "Ex") ∨ (field ≡ "Ey") ∨ (field ≡ "Ez")) {
  int dir;
  if (field ≡ "Ex") {
    dir = 0;
    Ex_idx = count;
  }
  else if (field ≡ "Ey") {
    dir = 1;
    Ey_idx = count;
  }
  else {
    dir = 2;
    Ez_idx = count;
  }
  from_edge_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &E_idx, &(Out.loVect()[0]),
      &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]), &(Out.hiVect()[2]),
      Out.dataPtr(), &(E[dir].loVect()[0]), &(E[dir].loVect()[1]), &(E[dir].loVect()[2]),
      &(E[dir].hiVect()[0]), &(E[dir].hiVect()[1]), &(E[dir].hiVect()[2]), E[dir].dataPtr(),
      &watch_dump_data, &return_status);
  if (return_status ≠ EXIT_SUCCESS) {
    pout() ≪ "\n\t\t\tERROR(dump_data): from_edge_to_center_ returned " ≪ return_status ≪ endl;
    goto exit;
  }
}
else if ((field ≡ "Dx") ∨ (field ≡ "Dy") ∨ (field ≡ "Dz")) {
  int dir;
  if (field ≡ "Dx") {
    dir = 0;
    Dx_idx = count;
  }
  else if (field ≡ "Dy") {
    dir = 1;
    Dy_idx = count;
  }
  else {
    dir = 2;
    Dz_idx = count;
  }
  from_edge_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &D_idx, &(Out.loVect()[0]),
      &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]), &(Out.hiVect()[2]),
      Out.dataPtr(), &(D[dir].loVect()[0]), &(D[dir].loVect()[1]), &(D[dir].loVect()[2]),
      &(D[dir].hiVect()[0]), &(D[dir].hiVect()[1]), &(D[dir].hiVect()[2]), D[dir].dataPtr(),
      &watch_dump_data, &return_status);
  if (return_status ≠ EXIT_SUCCESS) {
    pout() ≪ "\n\t\t\tERROR(dump_data): from_edge_to_center_ returned " ≪ return_status ≪ endl;
    goto exit;
  }
}
```

See also chunks 142, 143, 144, and 145.

This code is used in chunk 139.

9.2.2 Space and Time Interpolate Magnetic Fields

142 ⟨ *dump_data*: precompute field 141 ⟩ +≡

```
else
  if ((field ≡ "Bx") ∨ (field ≡ "By") ∨ (field ≡ "Bz")) {
    int dir;
    if (field ≡ "Bx") {
      dir = 0;
      Bx_idx = count;
    }
    else if (field ≡ "By") {
      dir = 1;
      By_idx = count;
    }
    else {
      dir = 2;
      Bz_idx = count;
    }
    from_face_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &B_idx, &(Out.loVect()[0]),
        &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
        &(Out.hiVect()[2]), Out.dataPtr(), &(B[dir].loVect()[0]), &(B[dir].loVect()[1]),
        &(B[dir].loVect()[2]), &(B[dir].hiVect()[0]), &(B[dir].hiVect()[1]), &(B[dir].hiVect()[2]),
        B[dir].dataPtr(), &watch_dump_data, &return_status);
    if (return_status ≠ EXIT_SUCCESS) {
      pout() ≪ "\n\t\t\tERROR(dump_data): from_face_to_center_ returned " ≪ return_status ≪
          endl;
      goto exit;
    }
  }
  else if ((field ≡ "Hx") ∨ (field ≡ "Hy") ∨ (field ≡ "Hz")) {
    int dir;
    if (field ≡ "Hx") {
      dir = 0;
      Hx_idx = count;
    }
    else if (field ≡ "Hy") {
      dir = 1;
      Hy_idx = count;
    }
    else {
      dir = 2;
      Hz_idx = count;
    }
    from_face_to_center_(&SpaceDim, &dir, &number_of_precomputes, &count, &H_idx, &(Out.loVect()[0]),
        &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
        &(Out.hiVect()[2]), Out.dataPtr(), &(H[dir].loVect()[0]), &(H[dir].loVect()[1]),
        &(H[dir].loVect()[2]), &(H[dir].hiVect()[0]), &(H[dir].hiVect()[1]), &(H[dir].hiVect()[2]),
        H[dir].dataPtr(), &watch_dump_data, &return_status);
    if (return_status ≠ EXIT_SUCCESS) {
      pout() ≪ "\n\t\t\tERROR(dump_data): from_face_to_center_ returned " ≪ return_status ≪
          endl;
      goto exit;
    }
  }
```

9.2.3 Evaluate Energy

143 ⟨ *dump_data*: precompute field 141 ⟩ +≡

```
else
  if (field ≡ "En") {
    En_idx = count;
        /* For this field we need to have E and B available. */
    if ((Ex_idx ≡ −1) ∨ (Ey_idx ≡ −1) ∨ (Ez_idx ≡ −1) ∨ (Bx_idx ≡ −1) ∨ (By_idx ≡ −1) ∨ (Bz_idx ≡ −1)) {
      pout( ) ≪ "\n\t\t\tERROR(dump_data): procompute E and B before Energy" ≪ endl;
      return_status = ERR_DUMP_DATA;
      goto exit;
    }
    evaluate_energy_(&SpaceDim, &number_of_precomputes, &count, &Ex_idx, &Ey_idx, &Ez_idx, &Bx_idx,
        &By_idx, &Bz_idx, &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
        &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
    if (return_status ≠ EXIT_SUCCESS) {
      pout( ) ≪ "\n\t\t\tERROR(dump_data): evaluate_energy_ returned " ≪ return_status ≪ endl;
      goto exit;
    }
  }
```

9.2.4 Evaluate Poynting Vector

144 ⟨ *dump_data*: precompute field 141 ⟩ +≡

```
else
  if ((field ≡ "Px") ∨ (field ≡ "Py") ∨ (field ≡ "Pz")) {
    int dir;
    if (field ≡ "Px") {
      dir = 0;
      Px_idx = count;
      if ((Ey_idx ≡ −1) ∨ (Hz_idx ≡ −1) ∨ (Ez_idx ≡ −1) ∨ (Hy_idx ≡ −1)) {
        pout( ) ≪ "\n\t\t\tERROR(dump_data): precompute Ey, Hz, Ez and Hy" ≪ " before Px" ≪
            endl;
        return_status = ERR_DUMP_DATA;
        goto exit;
      }
      evaluate_flow_(&SpaceDim, &number_of_precomputes, &count, &Ey_idx, &Hz_idx, &Ez_idx, &Hy_idx,
          &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
          &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
    }
    else if (field ≡ "Py") {
      dir = 1;
      Py_idx = count;
      if ((Ez_idx ≡ −1) ∨ (Hx_idx ≡ −1) ∨ (Ex_idx ≡ −1) ∨ (Hz_idx ≡ −1)) {
        pout( ) ≪ "\n\t\t\tERROR(dump_data): precompute Ez, Hx, Ex and Hz" ≪ " before Py" ≪
            endl;
        return_status = ERR_DUMP_DATA;
        goto exit;
      }
      evaluate_flow_(&SpaceDim, &number_of_precomputes, &count, &Ez_idx, &Hx_idx, &Ex_idx, &Hz_idx,
          &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
          &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
    }
    else {
```

```
    dir = 2;
    Pz_idx = count;
    if ((Ex_idx ≡ −1) ∨ (Hy_idx ≡ −1) ∨ (Ey_idx ≡ −1) ∨ (Hx_idx ≡ −1)) {
      pout( ) ≪ "\n\t\t\tERROR(dump_data): precompute Ex, Hy, Ey and Hx" ≪ " before Pz" ≪
          endl;
      return_status = ERR_DUMP_DATA;
      goto exit;
    }
    evaluate_flow_(&SpaceDim, &number_of_precomputes, &count, &Ex_idx, &Hy_idx, &Ey_idx, &Hx_idx,
        &(Out.loVect( )[0]), &(Out.loVect( )[1]), &(Out.loVect( )[2]), &(Out.hiVect( )[0]),
        &(Out.hiVect( )[1]), &(Out.hiVect( )[2]), Out.dataPtr( ), &watch_dump_data, &return_status);
  }
  if (return_status ≠ EXIT_SUCCESS) {
    pout( ) ≪ "\n\t\t\tERROR(dump_data): evaluate_flow_ returned " ≪ return_status ≪ endl;
    goto exit;
  }
}
```

9.2.5 Evaluate User Defined Fields

145 ⟨ *dump_data*: precompute field 141 ⟩ +≡

```
else { }
```

9.3 Transfer Fields to the Output Object

146 ⟨ *dump_data*: extract field from precomputes 146 ⟩ ≡

```
    /* Find the field's index in the precomputes arrays */
  int index = -1;
  for (int i = 0; i < number_of_precomputes; i++)
    if (vector_of_precomputes[i] ≡ field) index = i;
  if (index < 0) {
    pout() ≪ "\n\t\t\tERROR(dump_data): field " ≪ field ≪ " not precomputed" ≪ endl;
    return_status = ERR_DUMP_DATA;
    goto exit;
  }
    /* Now copy from Out[index] to OutWrite[count]. I use my own Fortran copy function here, because
       I don't quite trust Chombo's FArrayBox::copy. */
  output_field_copy_(&SpaceDim, &number_of_precomputes, &index, &number_of_outputs, &count,
      &(Out.loVect()[0]), &(Out.loVect()[1]), &(Out.loVect()[2]), &(Out.hiVect()[0]), &(Out.hiVect()[1]),
      &(Out.hiVect()[2]), Out.dataPtr(), OutWrite.dataPtr(), &watch_dump_data, &return_status);
  if (return_status ≠ EXIT_SUCCESS) {
    pout() ≪ "\n\t\t\tERROR(dump_data): output_field_copy_ returned " ≪ return_status ≪ endl;
    goto exit;
  }
```

This code is used in chunk 139.

10 Auxiliary Functions

147 ⟨ Auxiliary.c 147 ⟩ ≡
```
#include "Forms.H"
```
⟨ Function *is_a_pwr_of_two* 148 ⟩
⟨ Function *is_file_readable* 149 ⟩
⟨ Function *effective_grid_bounds* 150 ⟩
⟨ Function *make_tag_set* 158 ⟩
⟨ Function *upml_sigma* 166 ⟩
⟨ Function *mark_regions* 167 ⟩
⟨ Function *complement* 168 ⟩
⟨ Function *mark_TFR_faces* 169 ⟩
⟨ Function *fill_PML_arrays* 171 ⟩
⟨ Function *swap_idx* 172 ⟩

10.1 Is It a Power of Two

This is a little charming function that checks if a given integer, call it n is a power of 2. If it is then $n = 2^m$ where m is some other integer, and thus $m = \log_2 n$.

A logarithm of base 2 can be expressed in terms of natural logarithms as follows. Let x be such a number that $\mathrm{e}^x = 2$. Then $n = 2^m = (\mathrm{e}^x)^m = \mathrm{e}^{xm}$. Therefore $xm = \ln n$. But since $x = \ln 2$, we find that

$$m = \frac{\ln n}{\ln 2} = \log_2 n \tag{357}$$

So, now if m is an integer then

$$\mathrm{floor}\left(\frac{\ln n}{\ln 2}\right) = \left(\frac{\ln n}{\ln 2}\right) = m \tag{358}$$

and we should get that

$$n = \mathrm{floor}\left(2^{\mathrm{floor}(\ln n/\ln 2)}\right) = 2^m \tag{359}$$

But if m is not an integer, the "floors" will chop the non-integer part and we won't get the equality. Hence, testing for the equality is equivalent to testing if n is a power of 2.

148 ⟨ Function *is_a_pwr_of_two* 148 ⟩ ≡

```
int is_a_pwr_of_two(int n)
{
  const double eps = 0.001;
  int n_compare;      /* A slight problem here with floating point arithmetic, hence the dirty trick with the
      epsilon. This needs to be watched... */
  n_compare = (int)(floor(pow(2.0, (log((double) n + eps)/log((double) 2)))));
  pout() << "\tn␣=␣" << n << ",␣n_compare␣=␣" << n_compare << endl;      /* Intel's icpc objected to
      comparing integers and floats, so we try a clean int-to-int comparison here. */
  return (n == n_compare);
}
```

This code is used in chunk 147.

10.2 Is File Readable

This is a simple function that returns the file size in bytes if the file named in its argument is readable and NO (zero) otherwise. If it so happens that the file is readable, but its size is zero, NO is returned as well.

We first call the UNIX *stat* function on the file, and if this fails, we return NO.

Otherwise we check if the file is regular, i.e., not a device file and not a directory either.[10] If the file is not regular, we return NO.

Otherwise we attempt to open the file with the C++ **ifstream** :: *open*( ). If it fails we return NO.

Otherwise, we close the file and return its size in bytes.

Why not try **ifstream** :: *open*( ) right away? This is because the *open*( ) would work on a directory or a device file, and we don't want this.

149 ⟨ Function *is_file_readable* 149 ⟩ ≡

```
off_t is_file_readable(const char *file_name)
{
  struct stat file_status;
  off_t exit_status;
  ifstream file_stream;
  if (stat(file_name, &file_status) < 0) {
    pout( ) ≪ "\tERROR(is_file_readable):␣cannot␣stat␣" ≪ file_name ≪ endl;
    pout( ) ≪ "\t\tError␣number␣" ≪ errno ≪ endl;
    exit_status = (off_t) NO;
    goto exit;
  }
  if (¬(S_ISREG(file_status.st_mode))) {
    pout( ) ≪ "\tERROR(is_file_readable):␣file␣" ≪ file_name ≪ "␣is␣not␣regular" ≪ endl;
    exit_status = (off_t) NO;
    goto exit;
  }
  file_stream.open(file_name);
  if (file_stream.fail( )) {
    pout( ) ≪ "\tERROR(is_file_readable):␣file␣" ≪ file_name ≪
        "␣cannot␣be␣opened␣for␣reading" ≪ endl;
    exit_status = (off_t) NO;
    goto exit;
  }
  file_stream.close( );
  exit_status = file_status.st_size;
exit: return exit_status;
}
```

This code is used in chunk 147.

[10] (Will a soft link show as a regular file? Hopefully not.

10.3 Effective Grid Bounds

We just scan through all the boxes of the layout, find min and max, and this is it. Because we use **LayoutIterator** every process scans through all the boxes of the box layout, including the ones that live on other processes.

Why do I need this silly function at all? This is because when *domainSplit* is called on the top level grid, it may sometimes generate a grid that does not cover the original domain exactly. Calling this function lets me find out what the actual computational domain is.

I used to mess about with parallel version of this function, not quite understanding that it wasn't at all necessary, because every process has full knowledge of the *whole* box layout, which can be accessed by extracting **LayoutIterator** from it, and then iterating over the layout.

150 ⟨ Function *effective_grid_bounds* 150 ⟩ ≡

```
int effective_grid_bounds(level *level_ptr) {
    int return_status = EXIT_SUCCESS;
    int watch_effective_grid_bounds = false;
    SCMParmParse watch_parameter_parser("forms.watch");
```

See also chunks 151, 152, 153, 154, 155, 156, and 157.

This code is cited in chunks 70 and 158.

This code is used in chunk 147.

¶ These two **IntVect**s correspond to the lower and upper corners of a box within which the given subgrid is embedded tightly.

The *min*s are all initialized to the largest integer possible and the *max*es are all initialized to the lowest (most negative) integer possible.

151 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
IntVect
ijk_min(D_DECL(numeric_limits⟨int⟩::max(), numeric_limits⟨int⟩::max(), numeric_limits⟨int⟩::max())),
ijk_max(D_DECL(numeric_limits⟨int⟩::min(), numeric_limits⟨int⟩::min(), numeric_limits⟨int⟩::min()));
```

¶ Vectors *xyz_min* and *xyz_max* are for the physical (as opposed to **IntVect**) coordinates of *ijk_min* and *ijk_max*.

152 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
Vector⟨Real⟩ xyz_min(SpaceDim, 0.0), xyz_max(SpaceDim, 0.0);
```

¶ These two variables are the box layout of the current level and the layout iterator associated with it.

153 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
DisjointBoxLayout &box_layout = level_ptr→box_layout;
LayoutIterator layout_iterator = box_layout.layoutIterator();
```

¶ Action starts in this chunk. The first thing, as usual, is to check how much we should be telling the user about our actions.

154 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
    /* Action */
watch_parameter_parser.query("effective_grid_bounds", watch_effective_grid_bounds);
if (watch_effective_grid_bounds) pout() ≪ "effective_grid_bounds:" ≪ endl;
```

¶ Now, we loop over all the boxes of the layout and find the *smallEnd*( ) and the *bigEnd*( ) of each. Then compare with what we have in *ijk_min* and *ijk_max* and adjust the latter two vectors to hold the min and max values always.

155 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
    /* Find the extremal values for i, j, and k */
  for (layout_iterator.reset( ); layout_iterator.ok( ); ++layout_iterator) {
    Box box = box_layout.get(layout_iterator( ));
    IntVect lo_vector = box.smallEnd( );
    IntVect hi_vector = box.bigEnd( );
    for (int i = 0; i < SpaceDim; i++) {
      if (lo_vector[i] < ijk_min[i]) ijk_min[i] = lo_vector[i];
      if (hi_vector[i] > ijk_max[i]) ijk_max[i] = hi_vector[i];
    }
  }
  if (watch_effective_grid_bounds)
    pout( ) << "\tijk_min = " << ijk_min << ", ijk_max = " << ijk_max << endl;
```

¶ Each process, having found *ijk_min* and *ijk_max*, can use its own knowledge of *origin* and *delta* to evaluate physical coordinates that correspond to the min and max **IntVect**s of this level.

156 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
  for (int i = 0; i < SpaceDim; i++) {
    xyz_min[i] = level_ptr->origin[i] + ijk_min[i] * level_ptr->delta;
    xyz_max[i] = level_ptr->origin[i] + ijk_max[i] * level_ptr->delta;
  }
  level_ptr->ijk_min = ijk_min;
  level_ptr->ijk_max = ijk_max;
  level_ptr->xyz_min.resize(SpaceDim);
  level_ptr->xyz_min = xyz_min;
  level_ptr->xyz_max.resize(SpaceDim);
  level_ptr->xyz_max = xyz_max;
  if (watch_effective_grid_bounds)
    pout( ) << "\tlevel_ptr->xyz_min = (" << level_ptr->xyz_min << "), level_ptr->xyz_max = (" <<
        level_ptr->xyz_max << ")" << endl;
```

¶ And this is it. We return to the caller

157 ⟨ Function *effective_grid_bounds* 150 ⟩ +≡

```
  if (watch_effective_grid_bounds) pout( ) << "effective_grid_bounds: returning to caller" << endl;
  return return_status;
  }
```

10.4 Make Tag Set

The effect of calling *scm_c_eval_string*(*lambda*) is some pattern drawn on the **level** . *tags* field. But this is not what **BRMeshRefine** wants. The latter wants to see a *set* of cells, an **IntVectSet**. Such a set is easy to construct. All we need to do is to run through every **BaseFab**⟨**int**⟩ of the **level** . *tags* field, then run through its every cell, and if the value of the field in the cell is 1, then we add the cell to the set, otherwise, we don't.

But this will produce a set of cells for a given MPI process that is limited to what the process can see of the whole grid. So we have to carry out a procedure similar to the one outlined in *effective_grid_bounds*, module ⟨ Function *effective_grid_bounds* 150 ⟩, section 10.3, page 223. We have to collect the sets from all MPI processes, combined them into a single set that contains all tagged cells, and then return the set to the processes.

We begin by presenting the interface of the function. It takes a pointer to a **level** as its argument and returns an integer: `EXIT_SUCCESS` if all has worked fine. The first three internal variables of the functions are as we have seen before in other functions.

158 ⟨ Function *make_tag_set* 158 ⟩ ≡
```
int make_tag_set(level *level_ptr) {
    int return_status = EXIT_SUCCESS;
    int watch_make_tag_set = false;
    SCMParmParse watch_parameter_parser("forms.watch");
```
See also chunks 159, 160, 161, 162, 163, 164, and 165.
This code is cited in chunk 73.
This code is used in chunk 147.

¶

tags This is the tagged **BaseFab**⟨**int**⟩ field of the level.

local_tag_set This is the set which every MPI process is going to construct on its own by scanning its portion of the *tags* field.

tag_set This is the global set of tagged cells that is the sum of all *local_tag_set*s.

159 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
LevelData⟨BaseFab⟨int⟩⟩ &tags = level_ptr→tags;
IntVectSet local_tag_set;
IntVectSet &tag_set = level_ptr→tag_set;
```

¶

box_layout This is the disjoint box layout of the current level.

data_iterator This is the level data iterator associated with the current level's box layout.

160 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
DisjointBoxLayout &box_layout = level_ptr→box_layout;
DataIterator data_iterator(box_layout);
```

¶

root_process This is a process that will collect sets from all other MPI processes, and merge them.

number_of_processes The number of processes in the current MPI pool.

vector_of_tag_sets The *root_process* will collect sets sent to it from other processes on this vector.

161 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
int root_process = uniqueProc(SerialTask::compute);
int number_of_processes = numProc( );
Vector⟨IntVectSet⟩ vector_of_tag_sets(number_of_processes);
```

¶ Action starts here. We begin by checking the verbosity level.

162 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
watch_parameter_parser.query("make_tag_set", watch_make_tag_set);
if (watch_make_tag_set) pout() ≪ "make_tag_set:" ≪ endl;
```

¶ Here each MPI process collects tagged cells and puts them in its *local_tag_set*. The iteration is over the **BaseFab**s that the process looks after. For each **BaseFab**⟨**int**⟩, here referred to as *local_tags*, we extract its **Box**, then loop over its cells and add the cell to the set if *local_tags*(*cell*) is other than zero.

163 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
local_tag_set.makeEmpty();
if (watch_make_tag_set) pout() ≪ "\tProcess␣" ≪ procID() ≪ ":␣collecting␣tags␣locally" ≪ endl;
for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
  BaseFab⟨int⟩ &local_tags = tags[data_iterator()];
  Box tag_box = local_tags.box();
  BoxIterator box_iterator(tag_box);
  for (box_iterator.reset(); box_iterator.ok(); ++box_iterator) {
    IntVect cell = box_iterator();
    if (local_tags(cell)) local_tag_set |= cell;
  }
}
```

¶ Here the *root_process* gathers *local_tag_set*s from all MPI processes on its *vector_of_tag_sets*. Then it adds them all to *tag_set*, which has been emptied just before this operation, and sends the latter back to the processes.

164 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
tag_set.makeEmpty();
if (watch_make_tag_set) pout() ≪ "\tExchanging␣data..." ≪ endl;
gather(vector_of_tag_sets, local_tag_set, root_process);
if (procID() ≡ root_process)
  for (int count = 0; count < number_of_processes; count++) tag_set |= vector_of_tag_sets[count];
broadcast(tag_set, root_process);
if (watch_make_tag_set) pout() ≪ "\tProcess␣" ≪ procID() ≪ ":␣" ≪ level_ptr→tag_set.numPts() ≪
    "␣points␣in␣tag_set" ≪ endl;
```

¶ And this it. We return to the caller.

165 ⟨ Function *make_tag_set* 158 ⟩ +≡
```
if (watch_make_tag_set) pout() ≪ "make_tag_set:␣returning␣to␣caller" ≪ endl;
return return_status;
}
```

10.5 The UPML sigma

Here we evaluate [7]

$$\sigma(x) = \frac{\sigma_{\max} \left| x - x_0 \right|^m}{d^m}, \tag{360}$$

where x_0 is the position of the PML boundary, m is an experimental parameter, which seems to return best results for $m = 4$, and d is the width of the UPML layer.

166 ⟨ Function *upml_sigma* 166 ⟩ ≡

```
Real upml_sigma(int count, int ghost_margin, Real origin, Real delta, Real x_min, Real x_max, Real
        pml_min, Real pml_max, Real sigma_max)
{
  const int m = PML_M;
  int index = count − ghost_margin;
  Real position = index ∗ delta + origin;
  if (position < pml_min) {
    Real width = pml_min − x_min;
    Real depth = pml_min − position;
    if (depth ≥ width) return sigma_max;
    else return (sigma_max ∗ pow(depth/width, m));
  }
  else if (position > pml_max) {
    Real width = x_max − pml_max;
    Real depth = position − pml_max;
    if (depth ≥ width) return sigma_max;
    else return (sigma_max ∗ pow(depth/width, m));
  }
  else return 0.0;
}
```

This code is used in chunk 147.

10.6 Marking Regions

This function is slow, because drawing data on **BaseFab**⟨**int**⟩ with *setVal* is slow, and because we calculate and compare coordinates for *every* edge and for every face of the 3D grid (of which there are six per cell altogether). Luckily, we only have to do this once.

The way to speed up this and similar functions would be to redo the computations within Fortran, thus avoiding *setVal*s.

167 ⟨ Function *mark_regions* 167 ⟩ ≡

```
int mark_regions(level *level_ptr)
{    /* Mark UPML and scattered field regions on the medium_E and medium_H fields. The UPML
        region gets medium value of PML_MARKER (-2) and the scattered field region gets medium value of
        SCATTERED_FIELD_MARKER (-1). */
  SCMParmParse watch_parameter_parser("forms.watch");
  int watch_mark_regions = (int) false;
  int return_value = OK;
  const int pml_marker = PML_MARKER;
  const int scattered_field_marker = SCATTERED_FIELD_MARKER;
  Vector<Real> &pml_xyz_min = level_ptr->pml_xyz_min;
  Vector<Real> &pml_xyz_max = level_ptr->pml_xyz_max;
  Vector<Real> &signal_xyz_min = level_ptr->signal_xyz_min;
  Vector<Real> &signal_xyz_max = level_ptr->signal_xyz_max;
  Vector<Real> &origin = level_ptr->origin;
  Real &delta = level_ptr->delta;
  LevelData<EdgeFab<int>> &medium_E = level_ptr->medium_E;
  LevelData<FaceFab<int>> &medium_H = level_ptr->medium_H;
  DataIterator &data_iterator = level_ptr->data_iterator;
  watch_parameter_parser.query("mark_regions", watch_mark_regions);
  if (watch_mark_regions) pout() << "\tmark_regions: " << endl;
  if (level_ptr->number != 0) {
    pout() << "\t\tERROR(mark_regions): cannot mark on level " << level_ptr->number << endl;
    return_value = ERR_MARK_REGIONS;
    goto exit;
  }
        /* Mark medium_E */
  if (watch_mark_regions) pout() << "\t\tmarking edges" << endl;
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    for (int dir = 0; dir < SpaceDim; dir++) {
      BaseFab<int> &medium_E_dir = medium_E[data_iterator()][dir];
      Box medium_E_dir_box = medium_E_dir.box();
      BoxIterator box_iterator(medium_E_dir_box);
      IntVect box_type = medium_E_dir_box.type();
      if (watch_mark_regions > 1) {
        pout() << "\t\tdir = " << dir << " box = " << medium_E_dir_box << endl;
      }
      for (box_iterator.reset(); box_iterator.ok(); ++box_iterator) {
        IntVect cell = box_iterator();
        Box cell_box(cell, cell, box_type);
        Vector<Real> position(SpaceDim);
        for (int i = 0; i < SpaceDim; i++) {
          position[i] = origin[i] + (cell[i] - 0.5 * box_type[i]) * delta;
        }
        if ((position[0] <= pml_xyz_min[0]) ∨ (pml_xyz_max[0] <= position[0]) ∨ (position[1] <= pml_xyz_min[1]) ∨
            (pml_xyz_max[1] <= position[1]) ∨ (position[2] <= pml_xyz_min[2]) ∨ (pml_xyz_max[2] <= position[2]))
          {
```

```
            medium_E_dir.setVal(pml_marker, cell_box, 0);
          }
          else if ((position[0] ≤ signal_xyz_min[0]) ∨ (signal_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤
              signal_xyz_min[1]) ∨ (signal_xyz_max[1] ≤ position[1]) ∨ (position[2] ≤
              signal_xyz_min[2]) ∨ (signal_xyz_max[2] ≤ position[2])) {
            medium_E_dir.setVal(scattered_field_marker, cell_box, 0);
          }
        }
      }
    }
        /* Mark medium_H */
    if (watch_mark_regions) pout() ≪ "\t\tmarking faces" ≪ endl;
    for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
      for (int dir = 0; dir < SpaceDim; dir++) {
        BaseFab⟨int⟩ &medium_H_dir = medium_H[data_iterator()][dir];
        Box medium_H_dir_box = medium_H_dir.box();
        BoxIterator box_iterator(medium_H_dir_box);
        IntVect box_type = medium_H_dir_box.type();
        if (watch_mark_regions > 1) {
          pout() ≪ "\t\tdir = " ≪ dir ≪ " box = " ≪ medium_H_dir_box ≪ endl;
        }
        for (box_iterator.reset(); box_iterator.ok(); ++box_iterator) {
          IntVect cell = box_iterator();
          Box cell_box(cell, cell, box_type);
          Vector⟨Real⟩ position(SpaceDim);
          for (int i = 0; i < SpaceDim; i++) {
            position[i] = origin[i] + (cell[i] − 0.5 ∗ box_type[i]) ∗ delta;
          }
          if ((position[0] ≤ pml_xyz_min[0]) ∨ (pml_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤ pml_xyz_min[1]) ∨
              (pml_xyz_max[1] ≤ position[1])∨(position[2] ≤ pml_xyz_min[2])∨(pml_xyz_max[2] ≤ position[2]))
            {
            medium_H_dir.setVal(pml_marker, cell_box, 0);
          }
          else if ((position[0] ≤ signal_xyz_min[0]) ∨ (signal_xyz_max[0] ≤ position[0]) ∨ (position[1] ≤
              signal_xyz_min[1]) ∨ (signal_xyz_max[1] ≤ position[1]) ∨ (position[2] ≤
              signal_xyz_min[2]) ∨ (signal_xyz_max[2] ≤ position[2])) {
            medium_H_dir.setVal(scattered_field_marker, cell_box, 0);
          }
        }
      }
    }
  exit:
    if (watch_mark_regions) pout() ≪ "\tmark_regions: returning to caller" ≪ endl;
    return return_value;
  }
```

This code is used in chunk 147.

10.7 Return Third Direction

Return a complementary third direction if two different directions are given. Flag (negative) error otherwise.

168 ⟨ Function *complement* 168 ⟩ ≡

```
int complement(int i, int j)
{
  int return_value;
      /* Works for SpaceDim ≡ 3 only */
  if (((i ≡ 0) ∧ (j ≡ 1)) ∨ ((i ≡ 1) ∧ (j ≡ 0))) return_value = 2;
  else if (((i ≡ 0) ∧ (j ≡ 2)) ∨ ((i ≡ 2) ∧ (j ≡ 0))) return_value = 1;
  else if (((i ≡ 1) ∧ (j ≡ 2)) ∨ ((i ≡ 2) ∧ (j ≡ 1))) return_value = 0;
  else {
    pout( ) ≪ "\tERROR(complement):␣bad␣arguments" ≪ endl;
    return_value = ERR_COMPLEMENT;
  }
  return return_value;
}
```

This code is used in chunk 147.

10.8 Mark Total Field Region Faces

Fill the **Vector**⟨**Vector**⟨**Vector**⟨**DataIndex**⟩⟩⟩ *total_field_side_boxes* slot of *levels*[0].

This form of the function is merely a wrapper. The function just invokes itself six times with direction and side specified.

169 ⟨ Function *mark_TFR_faces* 169 ⟩ ≡

```
int mark_TFR_faces(level *level_ptr)
{
  int return_status = EXIT_SUCCESS;
  int watch_mark_TFR_faces = (int) false;
  SideIterator side_iterator;
  IntVect &signal_ijk_lo = level_ptr→signal_ijk_lo;
  IntVect &signal_ijk_hi = level_ptr→signal_ijk_hi;
  Vector⟨Vector⟨Vector⟨DataIndex⟩⟩⟩ &TFR_face_boxes = level_ptr→TFR_face_boxes;
  SCMParmParse watch_parameter_parser("forms.watch");
  watch_parameter_parser.query("mark_TFR_faces", watch_mark_TFR_faces);
  if (watch_mark_TFR_faces) {
    pout() ≪ "\tmark_TFR_faces:" ≪ endl;
    if (watch_mark_TFR_faces > 1) {
      pout() ≪ "\t\tsignal_ijk_lo␣=␣" ≪ signal_ijk_lo ≪ endl;
      pout() ≪ "\t\tsignal_ijk_hi␣=␣" ≪ signal_ijk_hi ≪ endl;
    }
  }
  if (watch_mark_TFR_faces) pout() ≪ "\t\tresizing␣TFR_face_boxes" ≪ endl;
  TFR_face_boxes.resize(SpaceDim);
  for (int dir = 0; dir < SpaceDim; dir++) TFR_face_boxes[dir].resize(2);
  for (int dir = 0; dir < SpaceDim; dir++) {
    for (side_iterator.reset(); side_iterator.ok(); ++side_iterator) {
      Side::LoHiSide side = side_iterator();
      if (watch_mark_TFR_faces > 1)
        pout() ≪ "\t\tcalling␣mark_TFR_faces:␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪ endl;
      if (return_status = mark_TFR_faces(level_ptr, dir, side)) {
        pout() ≪ "\t\tERROR(mart_TFR_faces):␣failed␣on␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪
            endl;
        goto exit;
      }
      else {
        if (watch_mark_TFR_faces > 1) {
          Vector⟨DataIndex⟩ &face_boxes = level_ptr→TFR_face_boxes[dir][side];
          DisjointBoxLayout &box_layout = level_ptr→box_layout;
          pout() ≪ "\t\tSelected␣" ≪ face_boxes.size() ≪ "␣boxes␣out␣of␣" ≪
              box_layout.numBoxes(procID()) ≪ endl;
        }
        if (watch_mark_TFR_faces > 2) {
          Vector⟨DataIndex⟩ &face_boxes = level_ptr→TFR_face_boxes[dir][side];
          DisjointBoxLayout &box_layout = level_ptr→box_layout;
          pout() ≪ "\t\tTFR␣boxes␣for␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪ ":" ≪ endl;
          for (int count = 0; count < face_boxes.size(); count++) {
            DataIndex &data_index = face_boxes[count];
            Box the_box = box_layout.get(data_index);
            pout() ≪ "\t\t\t" ≪ the_box ≪ endl;
          }
        }
      }
```

```
    }
  }
exit:
  if (watch_mark_TFR_faces) pout() ≪ "\tmark_TFR_faces:␣returning␣to␣caller" ≪ endl;
  return return_status;
}
```

See also chunk 170.

This code is used in chunk 147.

¶ Here we perform the task for a specific direction and side.

First we select the index that corresponds to the plane. It's called *ns*. Then we enter a loop over all boxes of the domain and for each check if it intersects with *ns* in direction *dir*.

If it does, we check if it fits in the total field region in the other two directions. If it does then only we push it onto the *TFR_face_boxes*[*dir*][*side*] vector.

170 ⟨ Function *mark_TFR_faces* 169 ⟩ +≡

```
int mark_TFR_faces(level *level_ptr, int dir, Side::LoHiSide side)
{
  int return_status = EXIT_SUCCESS;
  int watch_mark_TFR_faces = (int) false;
  SCMParmParse watch_parameter_parser("forms.watch");
  IntVect &signal_ijk_lo = level_ptr→signal_ijk_lo;
  IntVect &signal_ijk_hi = level_ptr→signal_ijk_hi;
  int ns = (side ≡ Side::Lo) ? signal_ijk_lo[dir] : signal_ijk_hi[dir];
  watch_parameter_parser.query("mark_TFR_faces", watch_mark_TFR_faces);
  if (watch_mark_TFR_faces) {
    pout() ≪ "\t\tmark_TFR_faces,␣dir␣=␣" ≪ dir ≪ ",␣side␣=␣" ≪ side ≪ endl;
    if (watch_mark_TFR_faces > 1) {
      for (int i = 0; i < SpaceDim; i++) {
        if (i ≡ dir) pout() ≪ "\t\t\tns␣=␣" ≪ ns ≪ endl;
        else pout() ≪ "\t\t\tlo␣=␣" ≪ signal_ijk_lo[i] ≪ ",␣hi␣=␣" ≪ signal_ijk_hi[i] ≪ endl;
      }
    }
  }
  DataIterator &data_iterator = level_ptr→data_iterator;
  DisjointBoxLayout &box_layout = level_ptr→box_layout;
  Vector⟨DataIndex⟩ &face_boxes = level_ptr→TFR_face_boxes[dir][side];
  face_boxes.clear();
  for (data_iterator.reset(); data_iterator.ok(); ++data_iterator) {
    DataIndex data_index = data_iterator();
    Box the_box = box_layout.get(data_index);
      /* This box has to be enlarged to cover all subboxes of the EdgeFab or FaceFab for this dir. */
    Box enlarged_box(surroundingNodes(the_box));
    IntVect small_end = enlarged_box.smallEnd();
    IntVect big_end = enlarged_box.bigEnd();
    if ((small_end[dir] ≤ ns) ∧ (ns ≤ big_end[dir]))      /* is on the plane */
    {
      vector⟨bool⟩ is_in(2, false);
      int count = 0;
      for (int i = 0; i < SpaceDim; i++) {
        if (i ≠ dir)
          if ((big_end[i] ≥ signal_ijk_lo[i]) ∧ (small_end[i] ≤ signal_ijk_hi[i])) is_in[count++] = true;
      }
```

```
        if (is_in[0] ∧ is_in[1]) face_boxes.push_back(data_index);
      }
    }
    return return_status;
}
```

10.9 Fill PML Arrays

171 ⟨ Function *fill_PML_arrays* 171 ⟩ ≡

```
int fill_PML_arrays(level *level_ptr)
{
  int return_status = EXIT_SUCCESS;
  int watch_fill_PML_arrays = (int) false;
  int level_number = level_ptr→number;
  int const ghost_margin = N_GHOST_CELLS;
  Real sigma_max = PML_SIGMA_MAX;
  SCMParmParse watch_parameter_parser("forms.watch");
  SCMParmParse pml_parameter_parser("forms.pml");
  IntVect &ijk_min = level_ptr→ijk_min;
  IntVect &ijk_max = level_ptr→ijk_max;
  Vector⟨Real⟩ &xyz_min = level_ptr→xyz_min;
  Vector⟨Real⟩ &xyz_max = level_ptr→xyz_max;
  Vector⟨Real⟩ &pml_xyz_min = level_ptr→pml_xyz_min;
  Vector⟨Real⟩ &pml_xyz_max = level_ptr→pml_xyz_max;
  Vector⟨Real⟩ &origin = level_ptr→origin;
  const Real &delta = level_ptr→delta;
  const Real &dt = level_ptr→dt;
  Vector⟨Real⟩ &C0x = level_ptr→C0x;
  Vector⟨Real⟩ &C1x = level_ptr→C1x;
  Vector⟨Real⟩ &C2x = level_ptr→C2x;
  Vector⟨Real⟩ &C3x = level_ptr→C3x;
  Vector⟨Real⟩ &C4x = level_ptr→C4x;
  Vector⟨Real⟩ &C0y = level_ptr→C0y;
  Vector⟨Real⟩ &C1y = level_ptr→C1y;
  Vector⟨Real⟩ &C2y = level_ptr→C2y;
  Vector⟨Real⟩ &C3y = level_ptr→C3y;
  Vector⟨Real⟩ &C4y = level_ptr→C4y;
  Vector⟨Real⟩ &C0z = level_ptr→C0z;
  Vector⟨Real⟩ &C1z = level_ptr→C1z;
  Vector⟨Real⟩ &C2z = level_ptr→C2z;
  Vector⟨Real⟩ &C3z = level_ptr→C3z;
  Vector⟨Real⟩ &C4z = level_ptr→C4z;
  int Cx_length = ijk_max[0] − ijk_min[0] + 1 + 2 ∗ ghost_margin;
  int Cy_length = ijk_max[1] − ijk_min[1] + 1 + 2 ∗ ghost_margin;
  int Cz_length = ijk_max[2] − ijk_min[2] + 1 + 2 ∗ ghost_margin;
  watch_parameter_parser.query("fill_PML_arrays", watch_fill_PML_arrays);
  pml_parameter_parser.query("sigma_max", sigma_max);
  if (watch_fill_PML_arrays) {
    pout() ≪ "\tfill_PML_arrays:" ≪ endl;
  }
  if (level_number ≠ 0) {
    pout() ≪ "\t\tERROR(fill_PML_arrays):␣level␣number␣not␣equal␣zero" ≪ endl;
    return_status = ERR_PML_ARRAYS;
    goto exit;
  }
  C0x.resize(Cx_length);
  C1x.resize(Cx_length);
  C2x.resize(Cx_length);
  C3x.resize(Cx_length);
  C4x.resize(Cx_length);
  C0y.resize(Cy_length);
```

```
  C1y.resize(Cy_length);
  C2y.resize(Cy_length);
  C3y.resize(Cy_length);
  C4y.resize(Cy_length);
  C0z.resize(Cz_length);
  C1z.resize(Cz_length);
  C2z.resize(Cz_length);
  C3z.resize(Cz_length);
  C4z.resize(Cz_length);
  if (watch_fill_PML_arrays) pout() << "\t\t\tcalling upml_sigma(count, " << ghost_margin << ", "
      << origin[0] << ", " << delta << ", " << xyz_min[0] << ", " << xyz_max[0] << ", " << pml_xyz_min[0] <<
      ", " << pml_xyz_max[0] << ", " << sigma_max << ")" << endl;
  for (int count = 0; count < Cx_length; count++) {
    Real sigma = upml_sigma(count, ghost_margin, origin[0], delta, xyz_min[0], xyz_max[0], pml_xyz_min[0],
        pml_xyz_max[0], sigma_max);
    C3x[count] = 2.0 + sigma * dt;
    C4x[count] = 2.0 - sigma * dt;
    C0x[count] = 1.0/C3x[count];
    C1x[count] = C4x[count] * C0x[count];
    C2x[count] = 2 * dt/delta * C0x[count];
  }
  if (watch_fill_PML_arrays) pout() << "\t\t\tCx: " << C0x.size() << " entries initialized" << endl;
  if (watch_fill_PML_arrays) pout() << "\t\t\tcalling upml_sigma(count, " << ghost_margin << ", "
      << origin[1] << ", " << delta << ", " << xyz_min[1] << ", " << xyz_max[1] << ", " << pml_xyz_min[1] <<
      ", " << pml_xyz_max[1] << ", " << sigma_max << ")" << endl;
  for (int count = 0; count < Cy_length; count++) {
    Real sigma = upml_sigma(count, ghost_margin, origin[1], delta, xyz_min[1], xyz_max[1], pml_xyz_min[1],
        pml_xyz_max[1], sigma_max);
    C3y[count] = 2.0 + sigma * dt;
    C4y[count] = 2.0 - sigma * dt;
    C0y[count] = 1.0/C3y[count];
    C1y[count] = C4y[count] * C0y[count];
    C2y[count] = 2 * dt/delta * C0y[count];
  }
  if (watch_fill_PML_arrays) pout() << "\t\t\tCy: " << C0y.size() << " entries initialized" << endl;
  if (watch_fill_PML_arrays) pout() << "\t\t\tcalling upml_sigma(count, " << ghost_margin << ", "
      << origin[2] << ", " << delta << ", " << xyz_min[2] << ", " << xyz_max[2] << ", " << pml_xyz_min[2] <<
      ", " << pml_xyz_max[2] << ", " << sigma_max << ")" << endl;
  for (int count = 0; count < Cz_length; count++) {
    Real sigma = upml_sigma(count, ghost_margin, origin[2], delta, xyz_min[2], xyz_max[2], pml_xyz_min[2],
        pml_xyz_max[2], sigma_max);
    C3z[count] = 2.0 + sigma * dt;
    C4z[count] = 2.0 - sigma * dt;
    C0z[count] = 1.0/C3z[count];
    C1z[count] = C4z[count] * C0z[count];
    C2z[count] = 2 * dt/delta * C0z[count];
  }
  if (watch_fill_PML_arrays) pout() << "\t\t\tCz: " << C0z.size() << " entries initialized" << endl;
  if (watch_fill_PML_arrays) pout() << "\t\tdone." << endl;
exit: return return_status;
}
```

This code is used in chunk 147.

10.10 Swap New-Old Indexes

172 ⟨ Function *swap_idx* 172 ⟩ ≡

```
void swap_idx(int idx_array[2])
{
  int hold = idx_array[0];
  idx_array[0] = idx_array[1];
  idx_array[1] = hold;
}
```

This code is used in chunk 147.

11 Header File

This is a listing of the header file. The actual file that is dumped by `tangle` is called `Forms.hw`. This file is then additionally massaged by `sed` to remove CWEB line numbers, comments, and empty lines. Finally, `astyle` is called to prettify it and the output is written on `Forms.H` eventually.

Although CWEB-related line numbers are lost in the process, the resulting code is cleaner and can be exported on its own. If references to the CWEB source are needed, these steps should be removed from the `Makefile` and `Forms.hw` simply renamed to `Forms.H`.

173 ⟨ Forms.hw 173 ⟩ ≡

```
#ifndef _FORMS_H_
# define _FORMS_H_
# define PROGRAM_AUTHOR "Zdzislaw␣(Gustav)␣Meglicki,␣Indiana␣University"
# define PROGRAM_VERSION "$Id:␣Includes.w,v␣1.63␣2008/06/02␣21:04:51␣gustav␣Exp␣$"
      /* C includes */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#ifndef CH_Linux
  extern int errno;
#endif
      /* Guile includes */
#include <libguile.h>
      /* C++ includes */
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>
#include <fstream>
#include <ios>
#include <vector>
#include <cmath>
#include <cstdlib>
  using std::cout;
  using std::cin;
  using std::cerr;
  using std::endl;
  using std::flush;
  using std::setw;
  using std::numeric_limits;
  using std::ifstream;
  using std::string;
```

See also chunks 174, 175, 176, 177, 179, 181, 182, 184, 185, 186, 188, 189, and 190.

11.1 Chombo Includes

174 ⟨ Forms.hw 173 ⟩ +≡

```
    /* Standard Chombo includes */
#include <parstream.H>      /* pout( ) lives here */
#include <SPACE.H>      /* SpaceDim lives here */
#include <SPMD.H>      /* procID( ) and numProc( ) live here */
#include <IntVectSet.H>
#include <REAL.H>
#include <Box.H>
#include <DisjointBoxLayout.H>
#include <FArrayBox.H>
#include <FluxBox.H>
#include <LevelData.H>
#include <BRMeshRefine.H>
#include <LoadBalance.H>
#include <BoxIterator.H>
#include <LoHiSide.H>
#include <AMRIO.H>
    /* My Chombo extensions: includes */
#include <SCMParmParse.H>
#include <EdgeFab.H>
#include <FaceFab.H>
#include <CurlBox.H>
```

11.2 Program Constants

175 ⟨ Forms.hw 173 ⟩ +≡

```
    /* Exit statuses */
#ifdef EXIT_SUCCESS
#undef EXIT_SUCCESS
#endif
# define EXIT_SUCCESS 0
# define ERR_COMPLEMENT -1
# define ERR_BAD_DIMENSION 1
# define ERR_NO_INPUT_FILE 2
# define ERR_BUILD_LEVELS 3
# define ERR_FILL_LEVELS 4
# define ERR_ADVANCE_D 5
# define ERR_ADVANCE_B 6
# define ERR_DUMP_DATA 7
# define ERR_INITIALIZE_SCHEME 8
# define ERR_D_TO_E 9
# define ERR_INJECT_D 10
# define ERR_B_TO_H 11
# define ERR_INJECT_B 12
# define ERR_MARK_REGIONS 30
# define ERR_PML_ARRAYS 31
    /* Other program specific constants */
#ifdef NO
#undef NO
#endif
# define NO 0
#ifdef YES
#undef YES
#endif
# define YES 1
#ifdef OK
#undef OK
#endif
# define OK 0
#ifdef OUCH
#undef OUCH
#endif
# define OUCH 1
#ifdef ZERO
#undef ZERO
#endif
# define ZERO 0
#ifdef ONE
#undef ONE
#endif
# define ONE 1
#ifdef EPSILON
#undef EPSILON
#endif
# define EPSILON 0.0001
    /* Default iteration specifications */
# define NUMBER_OF_STEPS 100
# define IMAGE_FREQUENCY 10
```

```
# define STRIDE 2
# define TO 0.0
# define INITIAL_LABEL 1
    /* Default grid specifications */
# define N_CELLS 128
# define N_CELLS_MIN 8
# define N_CELLS_MAX 1024
# define DELTA 1.0
# define XO 0.0
# define PML_XYZ_MIN 10
# define PML_XYZ_MAX 118
# define PML_MARGIN 3.0
# define PML_M 4
# define PML_SIGMA_MAX 10.0
# define SIGNAL_XYZ_MIN 20
# define SIGNAL_XYZ_MAX 108
# define SIGNAL_MARGIN 3.0    /* PML_MARGIN and SIGNAL_MARGIN should really be 7.0, but we make them
    less temporarily for test purposes. */
# define N_LEVELS 1
# define N_LEVELS_MIN 1
# define N_LEVELS_MAX 10
# define N_GHOST_CELLS 2
# define N_AUXILIARIES 0
# define N_AUXILIARIES_MIN 0
# define N_AUXILIARIES_MAX 10
# define MAX_BOX_SIZE 64
# define MAX_BOX_SIZE_MIN 4
# define MAX_BOX_SIZE_MAX 1024
# define LAMBDA_MIN_SIZE 5
# define REFINE_RATIO 2
# define REFINE_BLOCK_FACTOR 2
# define REFINE_BUFFER_SIZE 4
# define REFINE_MAX_SIZE 50
# define REFINE_FILL_RATIO 1.0    /* REFINE_BUFFER_SIZE should really be 8, but here we make it smaller
    for test purposes. */    /* Region markers */
# define PML_MARKER -2
# define SCATTERED_FIELD_MARKER -1
# define FORMS_OUTPUT "forms_output"
# define OUTPUT_NUMBER_OF_DIGITS 3
```

11.3 C++ Functions

176 ⟨ Forms.hw 173 ⟩ +≡
⟨ Structure level 191 ⟩
int *build_levels*(**Vector**⟨**level** ∗⟩ &*levels*);
int *fill_levels*(**Vector**⟨**level** ∗⟩ &*levels*);
int *mark_regions*(**level** ∗*level_ptr*);
int *advance_d*(**level** ∗*level_ptr*);
int *advance_d_0*(**level** ∗*level_ptr*);
int *advance_d_n*(**level** ∗*level_ptr*);
int *inject_d*(**level** ∗*level_ptr*);
int *inject_d*(**level** ∗*level_ptr*, **int** *dir*, **Side** :: *LoHiSide side*);
int *d_to_e*(**level** ∗*level_ptr*);
int *d_to_e_0*(**level** ∗*level_ptr*);
int *d_to_e_0_0*(**level** ∗*level_ptr*);
int *d_to_e_0_n*(**level** ∗*level_ptr*);
int *d_to_e_n*(**level** ∗*level_ptr*);
int *d_to_e_n_0*(**level** ∗*level_ptr*);
int *d_to_e_n_n*(**level** ∗*level_ptr*);
int *advance_b*(**level** ∗*level_ptr*);
int *advance_b_0*(**level** ∗*level_ptr*);
int *advance_b_n*(**level** ∗*level_ptr*);
int *inject_b*(**level** ∗*level_ptr*);
int *inject_b*(**level** ∗*level_ptr*, **int** *dir*, **Side** :: *LoHiSide side*);
int *b_to_h*(**level** ∗*level_ptr*);
int *b_to_h_0*(**level** ∗*level_ptr*);
int *b_to_h_n*(**level** ∗*level_ptr*);
int *dump_data*(**Vector**⟨**level** ∗⟩ &*levels*, **int** *label*);
int *is_a_pwr_of_two*(**int** *n*);
off_t *is_file_readable*(**const char** ∗*file_name*);
int *effective_grid_bounds*(**level** ∗*level_ptr*);
int *make_tag_set*(**level** ∗*level_ptr*);
Real *upml_sigma*(**int** *count*, **int** *ghost_margin*, **Real** *origin*, **Real** *delta*, **Real** *x_min*, **Real** *x_max*, **Real** *pml_min*, **Real** *pml_max*, **Real** *sigma_max*);
int *mark_TFR_faces*(**level** ∗*level_ptr*);
int *mark_TFR_faces*(**level** ∗*level_ptr*, **int** *dir*, **Side** :: *LoHiSide side*);
int *fill_PML_arrays*(**level** ∗*level_ptr*);
void *swap_idx*(**int** *idx_array*[2]);

11.4 Scheme Functions

177 ⟨ Forms.hw 173 ⟩ +≡

```
    /* Scheme stuff */
  bool does_scm_symbol_exist(const char *name);
  int initialize_scheme( );
  int post_initialize_scheme( );
    /* Drawing figures */
    /* A box */
  SCM draw_box(SCM low_corner, SCM high_corner, SCM color);
  int draw_media_box(Vector⟨Real⟩ &low_corner, Vector⟨Real⟩ &high_corner, int &color);
  int draw_tags_box(Vector⟨Real⟩ &low_corner, Vector⟨Real⟩ &high_corner);
  extern "C"
  {
    void f_draw_box_(
    const int *const SpaceDim, const int *const ghost_margin, const Real *const origin, const
        Real *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const
        Real *const low_corner, const Real *const high_corner, const int *const color, const int
        *const box_type, const int *const i_lo, const int *const j_lo, const int *const k_lo, const int
        *const i_hi, const int *const j_hi, const int *const k_hi, const int *const medium, const int
        *const verbosity, const int *const return_status);
  }
    /* A ball */
  SCM draw_ball(SCM center, SCM radius, SCM color);
  int draw_media_ball(Vector⟨Real⟩ &center, Real &radius, int &color);
  int draw_tags_ball(Vector⟨Real⟩ &center, Real &radius);
  extern "C"
  {
    void f_draw_ball_(const int *const SpaceDim, const int *const margin, const Real *const origin, const
        Real *const delta, const int *const iv_lower_corner, const int *const iv_upper_corner, const Real
        *const center, const Real *const radius, const int *const color, const int *const box_type, const
        int *const i_lo, const int *const j_lo, const int *const k_lo, const int *const i_hi, const int
        *const j_hi, const int *const k_hi, const int *const medium, const int *const verbosity, const
        int *const return_status);
  }
    /* An ellipsoid */
  SCM draw_ellipsoid(SCM focus_1, SCM focus_2, SCM sum, SCM color);
  int draw_media_ellipsoid(Vector⟨Real⟩ &focus_1, Vector⟨Real⟩ &focus_2, Real &sum, int &color);
  int draw_tags_ellipsoid(Vector⟨Real⟩ &focus_1, Vector⟨Real⟩ &focus_2, Real &radius);
  extern "C"
  {
    void f_draw_ellipsoid_(const int *const SpaceDim, const int *const margin, const Real
        *const origin, const Real *const delta, const int *const iv_lower_corner, const int
        *const iv_upper_corner, const Real *const focus_1, const Real *const focus_2, const Real
        *const sum, const int *const color, const int *const box_type, const int *const i_lo, const
        int *const j_lo, const int *const k_lo, const int *const i_hi, const int *const j_hi, const int
        *const k_hi, const int *const medium, const int *const verbosity, const int *const return_status);
  }
    /* This is now declared in SCMParmParse, which is already included above: bool
    does_scm_symbol_exist(const char *name); */
```

11.5 Fortran Functions

11.5.1 Update Functions

179 ⟨ Forms.hw 173 ⟩ +≡

```
/* Fortran stuff */
extern "C"
{
  void update_d_(
  const int *const SpaceDim, const int *const ghost_margin, const int *const D_idx, const int
      *const H_idx, const Real *const dt_by_dg,
  const int *const imin_Dx, const int *const jmin_Dx, const int *const kmin_Dx, const int
      *const imax_Dx, const int *const jmax_Dx, const int *const kmax_Dx, const Real *const Dx,
  const int *const imin_Dy, const int *const jmin_Dy, const int *const kmin_Dy, const int
      *const imax_Dy, const int *const jmax_Dy, const int *const kmax_Dy, const Real *const Dy,
  const int *const imin_Dz, const int *const jmin_Dz, const int *const kmin_Dz, const int
      *const imax_Dz, const int *const jmax_Dz, const int *const kmax_Dz, const Real *const Dz,
  const int *const imin_Hx, const int *const jmin_Hx, const int *const kmin_Hx, const int
      *const imax_Hx, const int *const jmax_Hx, const int *const kmax_Hx, const Real *const Hx,
  const int *const imin_Hy, const int *const jmin_Hy, const int *const kmin_Hy, const int
      *const imax_Hy, const int *const jmax_Hy, const int *const kmax_Hy, const Real *const Hy,
  const int *const imin_Hz, const int *const jmin_Hz, const int *const kmin_Hz, const int
      *const imax_Hz, const int *const jmax_Hz, const int *const kmax_Hz, const Real *const Hz,
  const int *const fortran_verbose, const int *const return_status
  );
}
extern "C"
{
  void update_d_upml_(
  const int *const SpaceDim, const int *const ghost_margin, const int *const D_idx, const int
      *const H_idx,
  const int *const imin_Cx, const int *const imax_Cx, const Real *const C1x, const Real
      *const C2x,
  const int *const jmin_Cy, const int *const jmax_Cy, const Real *const C1y, const Real
      *const C2y,
  const int *const kmin_Cz, const int *const kmax_Cz, const Real *const C1z, const Real
      *const C2z,
  const int *const imin_Dx, const int *const jmin_Dx, const int *const kmin_Dx, const int
      *const imax_Dx, const int *const jmax_Dx, const int *const kmax_Dx, const Real *const Dx,
  const int *const imin_Dy, const int *const jmin_Dy, const int *const kmin_Dy, const int
      *const imax_Dy, const int *const jmax_Dy, const int *const kmax_Dy, const Real *const Dy,
  const int *const imin_Dz, const int *const jmin_Dz, const int *const kmin_Dz, const int
      *const imax_Dz, const int *const jmax_Dz, const int *const kmax_Dz, const Real *const Dz,
  const int *const imin_Hx, const int *const jmin_Hx, const int *const kmin_Hx, const int
      *const imax_Hx, const int *const jmax_Hx, const int *const kmax_Hx, const Real *const Hx,
  const int *const imin_Hy, const int *const jmin_Hy, const int *const kmin_Hy, const int
      *const imax_Hy, const int *const jmax_Hy, const int *const kmax_Hy, const Real *const Hy,
  const int *const imin_Hz, const int *const jmin_Hz, const int *const kmin_Hz, const int
      *const imax_Hz, const int *const jmax_Hz, const int *const kmax_Hz, const Real *const Hz,
  const int *const fortran_verbose, const int *const return_status
  );
}
extern "C"
{
  void update_b_(
```

```
  const int *const SpaceDim, const int *const ghost_margin, const int *const B_idx, const int
      *const E_idx, const Real *const dt_by_dg,
  const int *const imin_Bx, const int *const jmin_Bx, const int *const kmin_Bx, const int
      *const imax_Bx, const int *const jmax_Bx, const int *const kmax_Bx, const Real *const Bx,
  const int *const imin_By, const int *const jmin_By, const int *const kmin_By, const int
      *const imax_By, const int *const jmax_By, const int *const kmax_By, const Real *const By,
  const int *const imin_Bz, const int *const jmin_Bz, const int *const kmin_Bz, const int
      *const imax_Bz, const int *const jmax_Bz, const int *const kmax_Bz, const Real *const Bz,
  const int *const imin_Ex, const int *const jmin_Ex, const int *const kmin_Ex, const int
      *const imax_Ex, const int *const jmax_Ex, const int *const kmax_Ex, const Real *const Ex,
  const int *const imin_Ey, const int *const jmin_Ey, const int *const kmin_Ey, const int
      *const imax_Ey, const int *const jmax_Ey, const int *const kmax_Ey, const Real *const Ey,
  const int *const imin_Ez, const int *const jmin_Ez, const int *const kmin_Ez, const int
      *const imax_Ez, const int *const jmax_Ez, const int *const kmax_Ez, const Real *const Ez,
  const int *const fortran_verbose, const int *const return_status
  );
}
extern "C"
{
  void update_b_upml_(
  const int *const SpaceDim, const int *const ghost_margin,
  const int *const B_idx, const int *const E_idx,
  const int *const imin_Cx, const int *const imax_Cx, const Real *const C1x, const Real
      *const C2x,
  const int *const jmin_Cy, const int *const jmax_Cy, const Real *const C1y, const Real
      *const C2y,
  const int *const kmin_Cz, const int *const kmax_Cz, const Real *const C1z, const Real
      *const C2z,
  const int *const imin_Bx, const int *const jmin_Bx, const int *const kmin_Bx, const int
      *const imax_Bx, const int *const jmax_Bx, const int *const kmax_Bx, const Real *const Bx,
  const int *const imin_By, const int *const jmin_By, const int *const kmin_By, const int
      *const imax_By, const int *const jmax_By, const int *const kmax_By, const Real *const By,
  const int *const imin_Bz, const int *const jmin_Bz, const int *const kmin_Bz, const int
      *const imax_Bz, const int *const jmax_Bz, const int *const kmax_Bz, const Real *const Bz,
  const int *const imin_Ex, const int *const jmin_Ex, const int *const kmin_Ex, const int
      *const imax_Ex, const int *const jmax_Ex, const int *const kmax_Ex, const Real *const Ex,
  const int *const imin_Ey, const int *const jmin_Ey, const int *const kmin_Ey, const int
      *const imax_Ey, const int *const jmax_Ey, const int *const kmax_Ey, const Real *const Ey,
  const int *const imin_Ez, const int *const jmin_Ez, const int *const kmin_Ez, const int
      *const imax_Ez, const int *const jmax_Ez, const int *const kmax_Ez, const Real *const Ez,
  const int *const fortran_verbose, const int *const return_status
  );
}
```

11.5.2 *D*-to-*E* Conversion Functions

Level Zero Functions

181 ⟨ Forms.hw 173 ⟩ +≡

```
  extern "C"
  {
    void d_to_e_0_0_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
        int *const E_idx, const int *const D_idx, const Real *const time_e, const Real *const dt, const
        int *const imin_Cx, const int *const imax_Cx, const Real *const C0x, const Real
```

```
        *const C1x, const Real *const C3x, const Real *const C4x, const int *const imin_Cy, const int
        *const imax_Cy, const Real *const C0y, const Real *const C1y, const Real *const C3y, const
        Real *const C4y, const int *const imin_Cz, const int *const imax_Cz, const Real
        *const C0z, const Real *const C1z, const Real *const C3z, const Real *const C4z, const
        int *const imin_D, const int *const jmin_D, const int *const kmin_D, const int
        *const imax_D, const int *const jmax_D, const int *const kmax_D, const Real *const D, const
        Real *const E, const int *const medium_E, const int *const watch_d_to_e, const int
        *const return_status);
    }
    extern "C"
    {
      void d_to_e_0_n_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
          int *const number_of_auxiliary_fields, const int *const E_idx, const int *const D_idx, const Real
          *const time_e, const Real *const dt, const int *const imin_Cx, const int *const imax_Cx, const
          Real *const C0x, const Real *const C1x, const Real *const C3x, const Real *const C4x, const
          int *const imin_Cy, const int *const imax_Cy, const Real *const C0y, const Real
          *const C1y, const Real *const C3y, const Real *const C4y, const int *const imin_Cz, const
          int *const imax_Cz, const Real *const C0z, const Real *const C1z, const Real
          *const C3z, const Real *const C4z, const int *const imin_D, const int *const jmin_D, const
          int *const kmin_D, const int *const imax_D, const int *const jmax_D, const int
          *const kmax_D, const Real *const D, const Real *const E, const Real *const S, const int
          *const medium_E, const int *const watch_d_to_e, const int *const return_status);
    }
```

Higher Level Functions

182 ⟨ Forms.hw 173 ⟩ +≡

```
    extern "C"
    {
      void d_to_e_n_0_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
          int *const E_idx, const int *const D_idx, const Real *const time_e, const Real *const dt, const
          int *const imin_D, const int *const jmin_D, const int *const kmin_D, const int
          *const imax_D, const int *const jmax_D, const int *const kmax_D, const Real *const D, const
          Real *const E, const int *const medium_E, const int *const watch_d_to_e, const int
          *const return_status);
    }
    extern "C"
    {
      void d_to_e_n_n_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
          int *const number_of_auxiliary_fields, const int *const E_idx, const int *const D_idx, const Real
          *const time_e, const Real *const dt, const int *const imin_D, const int *const jmin_D, const
          int *const kmin_D, const int *const imax_D, const int *const jmax_D, const int
          *const kmax_D, const Real *const D, const Real *const E, const Real *const S, const int
          *const medium_E, const int *const watch_d_to_e, const int *const return_status);
    }
```

11.5.3 *B*-to-*H* Conversion Functions

Level Zero Functions

184 ⟨ Forms.hw 173 ⟩ +≡

```
    extern "C"
    {
```

```
  void b_to_h_0_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
      int *const H_idx, const int *const B_idx, const Real *const time_h, const Real *const dt, const
      int *const imin_Cx, const int *const imax_Cx, const Real *const C0x, const Real
      *const C1x, const Real *const C3x, const Real *const C4x, const int *const imin_Cy, const int
      *const imax_Cy, const Real *const C0y, const Real *const C1y, const Real *const C3y, const
      Real *const C4y, const int *const imin_Cz, const int *const imax_Cz, const Real
      *const C0z, const Real *const C1z, const Real *const C3z, const Real *const C4z, const int
      *const imin_B, const int *const jmin_B, const int *const kmin_B, const int *const imax_B, const
      int *const jmax_B, const int *const kmax_B, const Real *const B, const Real *const H, const
      int *const medium_H, const int *const watch_b_to_h, const int *const return_status);
}
```

Higher Level Functions

185 ⟨ Forms.hw 173 ⟩ +≡

```
extern "C"
{
  void b_to_h_n_(const int *const space_dim, const int *const dir, const int *const ghost_margin, const
      int *const H_idx, const int *const B_idx, const Real *const time_h, const Real
      *const dt, const int *const imin_B, const int *const jmin_B, const int *const kmin_B, const int
      *const imax_B, const int *const jmax_B, const int *const kmax_B, const Real *const B, const
      Real *const H, const int *const medium_H, const int *const watch_b_to_h, const int
      *const return_status);
}
```

11.5.4 Signal Injection Functions

186 ⟨ Forms.hw 173 ⟩ +≡

```
extern "C"
{
  void inject_d_(const int *const dir, const int *const side_number, const int *const margin, const
      int *const new_component, const Real *const x_0, const Real *const y_0, const
      Real *const z_0, const Real *const delta, const Real *const dt_by_delta, const Real
      *const time_h, const int *const i_pml_lo, const int *const j_pml_lo, const int *const k_pml_lo,
      const int *const i_pml_hi, const int *const j_pml_hi, const int *const k_pml_hi, const
      int *const i_Dx_lo, const int *const j_Dx_lo, const int *const k_Dx_lo, const int
      *const i_Dx_hi, const int *const j_Dx_hi, const int *const k_Dx_hi, const Real *const Dx, const
      int *const i_Dy_lo, const int *const j_Dy_lo, const int *const k_Dy_lo, const int
      *const i_Dy_hi, const int *const j_Dy_hi, const int *const k_Dy_hi, const Real *const Dy, const
      int *const watch_inject_d, const int *const return_status);
}
extern "C"
{
  void inject_b_(const int *const dir, const int *const side_number, const int *const margin, const
      int *const new_component, const Real *const x_0, const Real *const y_0, const
      Real *const z_0, const Real *const delta, const Real *const dt_by_delta, const Real
      *const time_e, const int *const i_pml_lo, const int *const j_pml_lo, const int *const k_pml_lo,
      const int *const i_pml_hi, const int *const j_pml_hi, const int *const k_pml_hi, const
      int *const i_Bx_lo, const int *const j_Bx_lo, const int *const k_Bx_lo, const int
      *const i_Bx_hi, const int *const j_Bx_hi, const int *const k_Bx_hi, const Real *const Bx, const
      int *const i_By_lo, const int *const j_By_lo, const int *const k_By_lo, const int
      *const i_By_hi, const int *const j_By_hi, const int *const k_By_hi, const Real *const By, const
      int *const watch_inject_b, const int *const return_status);
}
```

11.5.5 Output Functions

Interpolation Functions

188 ⟨ Forms.hw 173 ⟩ +≡

```
extern "C"
{
  void from_edge_to_center_(const int *const spacedim, const int *const dir, const int
      *const n_out, const int *const count, const int *const idx, const int *const i_out_lo, const
      int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const out, const int *const i_fld_lo, const
      int *const j_fld_lo, const int *const k_fld_lo, const int *const i_fld_hi, const int
      *const j_fld_hi, const int *const k_fld_hi, const Real *const fld, const int *const watch, const
      int *const status);
}
extern "C"
{
  void from_face_to_center_(const int *const spacedim, const int *const dir, const int
      *const n_out, const int *const count, const int *const idx, const int *const i_out_lo, const
      int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const out, const int *const i_fld_lo, const
      int *const j_fld_lo, const int *const k_fld_lo, const int *const i_fld_hi, const int
      *const j_fld_hi, const int *const k_fld_hi, const Real *const fld, const int *const watch, const
      int *const status);
}
```

Energy and Flux Functions

189 ⟨ Forms.hw 173 ⟩ +≡

```
extern "C"
{
  void evaluate_energy_(const int *const spacedim, const int *const n_out, const int *const count, const
      int *const Ex_idx, const int *const Ey_idx, const int *const Ez_idx, const int
      *const Bx_idx, const int *const By_idx, const int *const Bz_idx, const int *const i_out_lo, const
      int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
      *const j_out_hi, const int *const k_out_hi, const Real *const out, const int *const watch, const
      int *const status);
}
extern "C"
{
  void evaluate_flow_(const int *const spacedim, const int *const n_out, const int *const count, const int
      *const Ey_idx, const int *const Bz_idx, const int *const Ez_idx, const int *const By_idx, const
      int *const i_out_lo, const int *const j_out_lo, const int *const k_out_lo, const int
      *const i_out_hi, const int *const j_out_hi, const int *const k_out_hi, const Real *const out, const
      int *const watch, const int *const status);
}
```

Copy Functions

190 ⟨ Forms.hw 173 ⟩ +≡

```
extern "C"
{
  void output_field_copy_(const int *const spacedim, const int *const n_pcmp, const int
      *const idx_pcmp, const int *const n_outp, const int *const idx_outp, const int *const i_out_lo,
```

```
        const int *const j_out_lo, const int *const k_out_lo, const int *const i_out_hi, const int
        *const j_out_hi, const int *const k_out_hi, const Real *const pcmp, const Real *const outp, const
        int *const watch, const int *const status);
  }
#endif
```

11.6 Structure level

This is the central structure of the program. We'll have to work on this, because as it is, it is too clumsy. This has been copied from SHAPES.

191 ⟨ Structure level 191 ⟩ ≡

```
struct level {
    /* level number */
  int number;
    /* electric and magnetic time and time step */
  Real time_e, time_h, dt;
    /* physics regions: UPML boundaries and signal injection */
  Vector⟨Real⟩ pml_xyz_min;
  Vector⟨Real⟩ pml_xyz_max;
  Vector⟨Real⟩ signal_xyz_min;
  Vector⟨Real⟩ signal_xyz_max;
  IntVect signal_ijk_lo;
  IntVect signal_ijk_hi;
    /* general grid parameters */
  Box domain;
  Vector⟨Real⟩ origin;
  Real delta;     /* identical grid spacing in every direction */
  IntVect ijk_min;
  IntVect ijk_max;
  Vector⟨Real⟩ xyz_min;
  Vector⟨Real⟩ xyz_max;
    /* level distribution and structure */
  Vector⟨Box⟩ vector_of_boxes;
  Vector⟨int⟩ vector_of_processes;
  DisjointBoxLayout box_layout;
  LayoutIterator layout_iterator;
  DataIterator data_iterator;
  LevelData⟨BaseFab⟨int⟩⟩ tags;
  IntVectSet tag_set;
  Vector⟨Vector⟨Vector⟨DataIndex⟩⟩⟩ TFR_face_boxes;
    /* fields */
    /* magnetic */
  LevelData⟨FluxBox⟩ H, B;
    /* electric and electric-auxiliary */
  LevelData⟨CurlBox⟩ E, D, S;
    /* output */
  LevelData⟨FArrayBox⟩ Out, OutWrite;
    /* media */
  LevelData⟨EdgeFab⟨int⟩⟩ medium_E;
  LevelData⟨FaceFab⟨int⟩⟩ medium_H;
    /* old/new field index */
  int E_idx[2], D_idx[2], B_idx[2], H_idx[2];
    /* UPMLs */
  Vector⟨Real⟩ C0x;
  Vector⟨Real⟩ C1x;
  Vector⟨Real⟩ C2x;
  Vector⟨Real⟩ C3x;
  Vector⟨Real⟩ C4x;
  Vector⟨Real⟩ C0y;
  Vector⟨Real⟩ C1y;
  Vector⟨Real⟩ C2y;
```

```
    Vector⟨Real⟩ C3y;
    Vector⟨Real⟩ C4y;
    Vector⟨Real⟩ C0z;
    Vector⟨Real⟩ C1z;
    Vector⟨Real⟩ C2z;
    Vector⟨Real⟩ C3z;
    Vector⟨Real⟩ C4z;
  };
```

This code is cited in chunk 18.

This code is used in chunk 176.

12 Chombo Extensions

The extensions are needed to handle face, e.g., $\boldsymbol{B}$, and edge, e.g., $\boldsymbol{E}$, mounted data, also to orchestrate the movement of such data between levels, with special attention paid to level boundaries. Throughout all these operations we must ensure that the condition $\boldsymbol{\nabla} \cdot \boldsymbol{E} = \boldsymbol{\nabla} \cdot \boldsymbol{B} = 0$ is maintained.

The extensions are as follows

SCMParmParse Section 12.1, page 252—a class, similar to Chombo **ParmParse** that parses data from a Scheme environment.

EdgeFab$\langle$**T**$\rangle$ Section 12.2, page 300—a class template for handling edge mounted fields.

headers Section 12.2.1, page 302

implementation Section 12.2.2, page 305

FaceFab$\langle$**T**$\rangle$ Section 12.3, page 316—a class template for handling face mounted fields.

headers Section 12.3.1, page 316

implementation Section 12.3.2, page 317

CurlBox Section 12.4, page 323—a class for handling edge mounted **FArrayBox** fields.

headers Section 12.4.1, page 323

implementation Section 12.4.2, page 324

12.1 Scheme Parser

This class, **SCMParmParse**, is similar to Chombo **ParmParse**, but instead of parsing stuff from a file, it parses it from Scheme that runs within this program. So, the idea is that we have just one file that contains Scheme defines and code, we read it into the Scheme subprocess right at the beginning and then the program picks up what it wants from Scheme by calling various **SCMParmParse** methods.

Although **SCMParmParse** can be thought of as a Scheme utility, it can be also viewed as a Chombo extension. For this reason, it gets its own separate code file, header file and is placed here in the Chombo extensions section.

12.1.1 SCMParmParse Class Headers

194 ⟨ SCMParmParse.hw 194 ⟩ ≡

```
#ifndef CH_SCMPARMPARSE_H
# define CH_SCMPARMPARSE_H
      /* Standard UNIX includes */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
      /* C++ includes */
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>
#include <fstream>
#include <ios>
#include <vector>
#include <cassert>
#include <cstdlib>
      /* Guile includes */
#include <libguile.h>
      /* Chombo include */
#ifdef CH_SPACEDIM
#include <MayDay.H>
#include <Misc.H>
#include <Vector.H>
#include <parstream.H>     /* contains pout() */
#include <SPMD.H>     /* includes "mpi.h" if needed */
#endif
      /* Error codes */
# define ERR_PARFILE_NULL 121
# define ERR_PARFILE_NOT_READABLE 122
# define ERR_PARFILE_LOAD_FAILED 123
      /* Common functions are from the std class */
  using std::cout;
  using std::cerr;
  using std::endl;
  using std::flush;
  using std::setw;
  using std::numeric_limits;
  using std::ifstream;
  using std::string;
  using std::vector;
```

```
  using std::min;
  using std::max;
      /* The class declaration */
  class SCMParmParse {
  public: ⟨ SCMParmParse headers: constructors and destructors 195 ⟩
    ⟨ SCMParmParse headers: queries and extractors 196 ⟩
  protected: ⟨ SCMParmParse headers: protected members 197 ⟩
  };
#endif     /* CH_SCMPARMPARSE_H */
```

SCMParmParse Constructors and Destructors

195 ⟨ **SCMParmParse** headers: constructors and destructors 195 ⟩ ≡

```
  SCMParmParse(int argc, char **argv, const char *prefix = Λ, const char *parfile = Λ);
  SCMParmParse(const char *prefix = Λ);
  ~SCMParmParse( );
```

This code is used in chunk 194.

SCMParmParse Queries and Extractors

196 ⟨ **SCMParmParse** headers: queries and extractors 196 ⟩ ≡

```
  bool contains(const char *name);
  bool contains(const string &name);
  bool isProcedure(const char *name);
  bool isClosure(const char *name);
  bool isThunk(const char *name);
  SCM variable(const char *name);
  int countval(const char *name);
      /* getting or putting integers */
  void fastget(SCM variable, int &ref);
  void fastput(SCM variable, const int ref);
  void get(const char *name, int &ref);
  void put(const char *name, const int ref);
  bool query(const char *name, int &ref);
  bool queryput(const char *name, const int ref);
      /* getting or putting floats */
  void fastget(SCM variable, float &ref);
  void fastput(SCM variable, const float ref);
  void get(const char *name, float &ref);
  void put(const char *name, const float ref);
  bool query(const char *name, float &ref);
  bool queryput(const char *name, const float ref);
      /* getting or putting doubles */
  void fastget(SCM variable, double &ref);
  void fastput(SCM variable, const double ref);
  void get(const char *name, double &ref);
  void put(const char *name, const double ref);
  bool query(const char *name, double &ref);
  bool queryput(const char *name, const double ref);
      /* getting or putting strings */
  void fastget(SCM variable, string &ref);
  void get(const char *name, string &ref);
  void fastput(SCM variable, const string ref);
```

```
  void fastput(SCM variable, const char *ref);
  void put(const char *name, const string ref);
  void put(const char *name, const char *ref);
  bool query(const char *name, string &ref);
  bool queryput(const char *name, const string ref);
  bool queryput(const char *name, const char *ref);
      /* vector of ints */
  void fastgetarr(SCM variable, vector⟨int⟩ &ref);
  void fastputarr(SCM variable, vector⟨int⟩ &ref);
#ifdef CH_SPACEDIM
  void fastgetarr(SCM variable, Vector⟨int⟩ &ref);
  void fastputarr(SCM variable, Vector⟨int⟩ &ref);
#endif
  void getarr(const char *name, vector⟨int⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, vector⟨int⟩ &ref, int start_ix, int num_val);
      /* vector of floats */
  void fastgetarr(SCM variable, vector⟨float⟩ &ref);
  void fastputarr(SCM variable, vector⟨float⟩ &ref);
#ifdef CH_SPACEDIM
  void fastgetarr(SCM variable, Vector⟨float⟩ &ref);
  void fastputarr(SCM variable, Vector⟨float⟩ &ref);
#endif
  void getarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val);
      /* vector of double */
  void fastgetarr(SCM variable, vector⟨double⟩ &ref);
  void fastputarr(SCM variable, vector⟨double⟩ &ref);
#ifdef CH_SPACEDIM
  void fastgetarr(SCM variable, Vector⟨double⟩ &ref);
  void fastputarr(SCM variable, Vector⟨double⟩ &ref);
#endif
  void getarr(const char *name, vector⟨double⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, vector⟨double⟩ &ref, int start_ix, int num_val);
      /* vector of strings */
  void getarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val);
#ifdef CH_SPACEDIM
  void getarr(const char *name, Vector⟨int⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, Vector⟨int⟩ &ref, int start_ix, int num_val);
  void getarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val);
  void getarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val);
  bool queryarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val);
  void getarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val);
#endif
  void print_prefix( );
```

This code is used in chunk 194.

SCMParmParse Protected Members

197 ⟨ **SCMParmParse** headers: protected members 197 ⟩ ≡

```
  string prefix;
  bool does_scm_symbol_exist(const char *name);
```

off_t *is_file_readable*(**const char** ∗*file_name*);

This code is used in chunk 194.

12.1.2 SCMParmParse Class Implementation

198 ⟨ SCMParmParse.c 198 ⟩ ≡

```
#include <SCMParmParse.H>
```

⟨ **SCMParmParse** implementation: constructors and destructors 199 ⟩
⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩
⟨ **SCMParmParse** implementation: protected members 216 ⟩

SCMParmParse Constructors and Destructors

199 ⟨ **SCMParmParse** implementation: constructors and destructors 199 ⟩ ≡

```
SCMParmParse::SCMParmParse(int argc, char **argv, const char *c_prefix, const char *parfile)
{
  if (parfile ≡ Λ) {
#ifdef CH_SPACEDIM
    pout() ≪ "\n\tERROR(SCMParmParse::SCMParmParse): NULL file name" ≪ endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::SCMParmParse): NULL file name" ≪ endl;
#endif
#ifdef CH_MPI
    MPI_Abort(Chombo_MPI::comm, ERR_PARFILE_NULL);
#else
    abort();
#endif
  }
  if (¬is_file_readable(parfile)) {
#ifdef CH_SPACEDIM
    pout() ≪ "\n\tERROR(SCMParmParse::SCMParmParse): file " ≪ parfile ≪
        " cannot be opened for reading" ≪ endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::SCMParmParse): file " ≪ parfile ≪
        " cannot be opened for reading" ≪ endl;
#endif
#ifdef CH_MPI
    MPI_Abort(Chombo_MPI::comm, ERR_PARFILE_NOT_READABLE);
#else
    abort();
#endif
  }
  SCM status = scm_c_primitive_load(parfile);
  if (scm_is_false(status)) {
#ifdef CH_SPACEDIM
    pout() ≪ "\n\tERROR(SCMParmParse::SCMParmParse): primitive load failed" ≪ endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::SCMParmParse): primitive load failed" ≪ endl;
#endif
#ifdef CH_MPI
    MPI_Abort(Chombo_MPI::comm, ERR_PARFILE_LOAD_FAILED);
#else
    abort();
```

```
#endif
    }
    else {
      prefix.clear();
      if (c_prefix) prefix += c_prefix;
    }
  }
  SCMParmParse::SCMParmParse(const char *c_prefix)
  {
    prefix.clear();
    if (c_prefix) prefix += c_prefix;
  }
  SCMParmParse::~SCMParmParse()
  {
    prefix.clear();
  }
```

This code is cited in chunk 6.

This code is used in chunk 198.

SCMParmParse Queries and Extractors

200 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ ≡

```
  bool SCMParmParse::contains(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::contains): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::contains): NULL name" << endl;
#endif
      return false;
    }
    return does_scm_symbol_exist(fullname.c_str());
  }
  bool SCMParmParse::contains(const string &name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (¬name.empty()) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::contains): empty name" << endl;
```

```
#else
      cerr ≪ "\n\tERROR(SCMParmParse::contains):␣empty␣name" ≪ endl;
#endif
      return false;
    }
    return does_scm_symbol_exist(fullname.c_str( ));
  }
```

See also chunks 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, and 215.

This code is used in chunk 198.

¶ Here we go beyond what Chombo does and check, for certain things Schemish.

201 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
  bool SCMParmParse::isProcedure(const char *name)
  {
    string fullname;
    fullname.clear( );
    if (¬prefix.empty( )) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(SCMParmParse::isProcedure):␣NULL␣name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::isProcedure):␣NULL␣name" ≪ endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str( ))) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
      return scm_to_bool(scm_procedure_p(variable));
    }
    else {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(SCMParmParse::isProcedure):␣symbol␣does␣not␣exist" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::isProcedure):␣symbol␣does␣not␣exist" ≪ endl;
#endif
      return false;
    }
  }
  bool SCMParmParse::isClosure(const char *name)
  {
    string fullname;
    fullname.clear( );
    if (¬prefix.empty( )) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
```

```
      pout() << "\n\tERROR(SCMParmParse::isClosure): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isClosure): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      return scm_to_bool(scm_closure_p(variable));
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isClosure): symbol does not exist" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isClosure): symbol does not exist" << endl;
#endif
      return false;
    }
  }
  bool SCMParmParse::isThunk(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isThunk): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isThunk): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      return scm_to_bool(scm_thunk_p(variable));
    }
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#endif
      return false;
    }
  }
  SCM SCMParmParse::variable(const char *name)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
```

```
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::variable): NULL name" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::variable): NULL name" << endl;
#endif
    return SCM_UNDEFINED;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    return scm_c_lookup(fullname.c_str());
  }
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::isThunk): symbol does not exist" << endl;
#endif
    return SCM_UNDEFINED;
  }
}
```

202 ¶⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
int SCMParmParse::countval(const char *name)
{
  int count = 0;
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::countval): NULL name" << endl;
#else
    cerr << "\n\tERROR(SCMParmParse::countval): NULL name" << endl;
#endif
    return -1;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_number_p(variable)) ∨ scm_is_true(scm_string_p(variable))) {
      count = 1;
    }
    else      /* some compound variables */
    {
        /* We test for
```

1. list

2. vector

3. uniform vector

4. array—incluedes typed arrays

```
*/
      if (scm_is_true(scm_list_p(variable))) count = scm_to_int(scm_length(variable));
      else if (scm_is_true(scm_vector_p(variable))) count = scm_to_int(scm_vector_length(variable));
      else if (scm_is_true(scm_uniform_vector_p(variable)))
        count = scm_to_int(scm_uniform_vector_length(variable));
      else if (scm_is_true(scm_array_p(variable, SCM_UNDEFINED))) {
        int rank = (int) scm_c_array_rank(variable);
        count = 1;
        SCM dimensions = scm_array_dimensions(variable);
        for (int dim = 0; dim < rank; dim++) {
          SCM item = scm_list_ref(dimensions, scm_from_int(dim));
          if (scm_is_true(scm_number_p(item))) count = count * scm_to_int(item);
          else {
            int lo = scm_to_int(scm_list_ref(item, scm_from_int(0)));
            int hi = scm_to_int(scm_list_ref(item, scm_from_int(1)));
            count = count * (hi - lo + 1);
          }
        }
      }
    }
    return count;
  }
```

¶ *fastget*, *get*, *fastput*, *put*, *query* and *queryput* on simple variables. **int** case first.

203 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
  void SCMParmParse::fastget(SCM variable, int &ref)
  {
    ref = scm_to_int(scm_variable_ref(variable));
  }
  void SCMParmParse::get(const char *name, int &ref)
  {
      /* Checks, but does not tell. Prevents crash. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#endif
      return;
```

```
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_integer(variable)) ref = (int) scm_to_double(variable);
    }
  }
  void SCMParmParse::fastput(SCM variable, const int ref)
  {
    scm_variable_set_x(variable, scm_from_int(ref));
  }
  void SCMParmParse::put(const char *name, const int ref)
  {     /* Put ref into name if the Scheme variable named so exists and is an integer. Bail out silently
         otherwise. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::put): NULL name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::put): NULL name" ≪ endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_integer(scm_variable_ref(variable))) scm_variable_set_x(variable, scm_from_int(ref));
    }
  }
  bool SCMParmParse::query(const char *name, int &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::query): NULL name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::query): NULL name" ≪ endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_integer(variable)) {
```

```
        ref = (int) scm_to_double(variable);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const int ref)
  {
    /* Puts the ref value into name. Checks and returns false if problems are encountered. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::queryput): NULL name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryput): NULL name" ≪ endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_integer(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_int(ref));
        return true;
      }
      else {
        return false;
      }
    }
    else {
      return false;
    }
  }
```

¶ This is the same as above, but **int** is replaced with **float**.

204 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
  void SCMParmParse::fastget(SCM variable, float &ref)
  {
    ref = (float) scm_to_double(scm_variable_ref(variable));
  }
  void SCMParmParse::get(const char *name, float &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
```

```
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::get): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_real(variable)) ref = (float) scm_to_double(variable);
    }
  }
  void SCMParmParse::fastput(SCM variable, const float ref)
  {
    scm_variable_set_x(variable, scm_from_double((double) ref));
  }
  void SCMParmParse::put(const char *name, const float ref)
  {    /* Put ref into name if the Scheme variable named so exists and is an integer. Bail out silently
       otherwise. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_real(scm_variable_ref(variable)))
        scm_variable_set_x(variable, scm_from_double((double) ref));
    }
  }
  bool SCMParmParse::query(const char *name, float &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
```

```
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_real(variable)) {
        ref = (float) scm_to_double(variable);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const float ref)
  {
      /* Puts the ref value into name. Checks and returns false if problems are encountered. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_real(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_double((double) ref));
        return true;
      }
      else {
        return false;
      }
    }
    else {
      return false;
    }
  }
```

¶ This is the same as above, but **float** is replaced with **double**.

205 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::fastget(SCM variable, double &ref)
{
  ref = scm_to_double(scm_variable_ref(variable));
}
void SCMParmParse::get(const char *name, double &ref)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() ≪ "\n\tERROR(SCMParmParse::get): NULL name" ≪ endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::get): NULL name" ≪ endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_real(variable)) ref = scm_to_double(variable);
  }
}
void SCMParmParse::fastput(SCM variable, const double ref)
{
  scm_variable_set_x(variable, scm_from_double(ref));
}
void SCMParmParse::put(const char *name, const double ref)
{   /* Put ref into name if the Scheme variable named so exists and is an integer. Bail out silently
       otherwise. */
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() ≪ "\n\tERROR(SCMParmParse::put): NULL name" ≪ endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::put): NULL name" ≪ endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_c_lookup(fullname.c_str());
    if (scm_is_real(scm_variable_ref(variable))) scm_variable_set_x(variable, scm_from_double(ref));
  }
```

```
  }
  bool SCMParmParse::query(const char *name, double &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_real(variable)) {
        ref = scm_to_double(variable);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const double ref)
  {
      /* Puts the ref value into name. Checks and returns false if problems are encountered. */
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::queryput): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_real(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_double(ref));
        return true;
      }
      else {
        return false;
```

```
    }
  }
  else {
    return false;
  }
}
```

¶ This is similar to the above, but this time we get or put a string.

206 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::fastget(SCM variable, string &ref)
{
  char *buffer = scm_to_locale_string(scm_variable_ref(variable));
  ref.clear();
  ref += buffer;
  free(buffer);
}
void SCMParmParse::get(const char *name, string &ref)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() ≪ "\n\tERROR(SCMParmParse::get): NULL name" ≪ endl;
#else
    cerr ≪ "\n\tERROR(SCMParmParse::get): NULL name" ≪ endl;
#endif
    return;
  }
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_string(variable)) {
      char *buffer = scm_to_locale_string(variable);
      ref.clear();
      ref += buffer;
      free(buffer);
    }
  }
}
void SCMParmParse::fastput(SCM variable, const string ref)
{
  scm_variable_set_x(variable, scm_from_locale_string(ref.c_str()));
}
void SCMParmParse::fastput(SCM variable, const char *ref)
{
  scm_variable_set_x(variable, scm_from_locale_string(ref));
}
```

```
  void SCMParmParse::put(const char *name,const string ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_string(scm_variable_ref(variable))) {
        scm_variable_set_x(variable,scm_from_locale_string(ref.c_str()));
      }
    }
  }
  void SCMParmParse::put(const char *name,const char *ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::put): NULL name" << endl;
#endif
      return;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_string(scm_variable_ref(variable))) {
        scm_variable_set_x(variable,scm_from_locale_string(ref));
      }
    }
  }
  bool SCMParmParse::query(const char *name,string &ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
```

```
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
      if (scm_is_string(variable)) {
        char *buffer = scm_to_locale_string(variable);
        ref.clear();
        ref += buffer;
        free(buffer);
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const string ref)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::query): NULL name" << endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_c_lookup(fullname.c_str());
      if (scm_is_string(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_locale_string(ref.c_str()));
        return true;
      }
      else return false;
    }
    else return false;
  }
  bool SCMParmParse::queryput(const char *name, const char *ref)
  {
```

```
    string fullname;
    fullname.clear( );
    if (¬prefix.empty( )) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(SCMParmParse::query): NULL name" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::query): NULL name" ≪ endl;
#endif
      return false;
    }
    if (does_scm_symbol_exist(fullname.c_str( ))) {
      SCM variable = scm_c_lookup(fullname.c_str( ));
      if (scm_is_string(scm_variable_ref(variable))) {
        scm_variable_set_x(variable, scm_from_locale_string(ref));
        return true;
      }
      else return false;
    }
    else return false;
  }
```

¶ Process lists, vectors and uniform vectors of **int**s.

207 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
  void SCMParmParse::fastgetarr(SCM variable, vector⟨int⟩ &ref)
  {    /* Transfer data from a uniform Scheme vector to vector⟨int⟩. Both vectors are assumed to be of the
       same size. All vector content is transferred. Nothin is checked. Will crash if anything is wrong. */
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size( );
    for (int count = 0; count < length; count++) {
      ref[count] = scm_to_int(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
    }
  }
  void SCMParmParse::fastputarr(SCM variable, vector⟨int⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size( );
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_int(ref[count]));
    }
  }
#ifdef CH_SPACEDIM
  void SCMParmParse::fastgetarr(SCM variable, Vector⟨int⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size( );
    for (int count = 0; count < length; count++) {
      ref[count] = scm_to_int(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
```

```
    }
  }
  void SCMParmParse::fastputarr(SCM variable, Vector<int> &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_int(ref[count]));
    }
  }
#endif
  void SCMParmParse::getarr(const char *name, vector<int> &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::getarr): NULL name" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): NULL name" << endl;
#endif
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

   1. a list

   2. a vector

   3. a uniform vector

   */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::getarr): list value not an integer" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::getarr): list value not an integer" << endl;
#endif
```

```
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else
#ifdef CH_SPACEDIM
            pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣an␣integer" ≪ endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣an␣integer" ≪ endl;
#endif
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else
#ifdef CH_SPACEDIM
            pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣uniform␣vector␣value␣not␣an␣integer" ≪
                endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::getarr):␣uniform␣vector␣value␣not␣an␣integer" ≪
                endl;
#endif
        }
      }
    }
    else
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪ endl;
#endif
  }
  exit: return;
  }
  bool SCMParmParse::queryarr(const char *name, vector⟨int⟩ &ref, int start_ix, int num_val)
  {
    string fullname;
    bool return_value = true;
```

```
    fullname.clear( );
    if (¬prefix.empty( )) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear( );
    if (does_scm_symbol_exist(fullname.c_str( ))) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
          /* This may be either

   1. a list

   2. a vector

   3. a uniform vector

   */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣an␣integer" ≪ endl;
#else
              cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣an␣integer" ≪ endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
```

```
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not an integer" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not an integer" << endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 <= start_ix) ∧ (start_ix <= length - 1)) {
          int num_val_available = length - start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not\
                  an integer" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not an integer" <<
                  endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else {
#ifdef CH_SPACEDIM
        pout() << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
            endl;
#else
        cerr << "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" <<
            endl;
#endif
        return_value = false;
        goto exit;
      }
    }
    else return_value = false;
  exit: return return_value;
  }
```

¶ Process lists, vectors and uniform vectors of **float**s.

```
208 ⟨ SCMParmParse implementation: queries and extractors 200 ⟩ +≡
  void SCMParmParse::fastgetarr(SCM variable, vector⟨float⟩ &ref)
  {     /* Transfer data from a uniform Scheme vector to vector⟨float⟩. Both vectors are assumed to be of
         the same size. All vector content is transferred. Nothin is checked. Will crash if anything is wrong. */
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      ref[count] = (float) scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
    }
  }
  void SCMParmParse::fastputarr(SCM variable, vector⟨float⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double((double) ref[count]));
    }
  }
#ifdef CH_SPACEDIM
  void SCMParmParse::fastgetarr(SCM variable, Vector⟨float⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      ref[count] = (float) scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
    }
  }
  void SCMParmParse::fastputarr(SCM variable, Vector⟨float⟩ &ref)
  {
    SCM variable_ref = scm_variable_ref(variable);
    int length = ref.size();
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double((double) ref[count]));
    }
  }
#endif
  void SCMParmParse::getarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::getarr): name empty" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::getarr): name empty" ≪ endl;
#endif
      goto exit;
    }
```

```
    ref.clear( );
    if (does_scm_symbol_exist(fullname.c_str( ))) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
          /* This may be either

   1. a list

   2. a vector

   3. a uniform vector

   */
        if (scm_is_true(scm_list_p(variable))) {
          int length = scm_to_int(scm_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
              else
#ifdef CH_SPACEDIM
                pout( ) ≪ "\n\tERROR(SCMParmParse::getarr): list value not a float" ≪ endl;
#else
                cerr ≪ "\n\tERROR(SCMParmParse::getarr): list value not a float" ≪ endl;
#endif
            }
          }
        }
        else if (scm_is_vector(variable)) {
          int length = scm_to_int(scm_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
              else
#ifdef CH_SPACEDIM
                pout( ) ≪ "\n\tERROR(SCMParmParse::getarr): vector value not a float" ≪ endl;
#else
                cerr ≪ "\n\tERROR(SCMParmParse::getarr): vector value not a float" ≪ endl;
#endif
            }
          }
        }
        else if (scm_is_true(scm_uniform_vector_p(variable))) {
          int length = scm_to_int(scm_uniform_vector_length(variable));
          if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
```

```
            if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() ≪ "\n\tERROR(SCMParmParse::getarr): uniform vector value not a float" ≪
                  endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::getarr): uniform vector value not a float" ≪ endl;
#endif
          }
        }
      }
      else
#ifdef CH_SPACEDIM
        pout() ≪ "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" ≪
            endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" ≪ endl;
#endif
    }
  exit: return;
  }
  bool SCMParmParse::queryarr(const char *name, vector⟨float⟩ &ref, int start_ix, int num_val)
  {
    string fullname;
    bool return_value = true;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr): name empty" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr): name empty" ≪ endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either
  1. a list
  2. a vector
  3. a uniform vector
  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
```

```
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else {
#ifdef CH_SPACEDIM
            pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a float" << endl;
#else
            cerr << "\n\tERROR(SCMParmParse::queryarr): list value not a float" << endl;
#endif
            return_value = false;
            goto exit;
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 <= start_ix) && (start_ix <= length - 1)) {
          int num_val_available = length - start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): vector value not a float" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not a float" << endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 <= start_ix) && (start_ix <= length - 1)) {
          int num_val_available = length - start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a float" <<
                  endl;
#else
              cerr << "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a float" <<
                  endl;
#endif
              return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
#endif
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

¶ Process lists, vectors and uniform vectors of **double**s.

209 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::fastgetarr(SCM variable, vector⟨double⟩ &ref)
{   /* Transfer data from a uniform Scheme vector to vector⟨double⟩. Both vectors are assumed to be of
       the same size. All vector content is transferred. Nothin is checked. Will crash if anything is wrong. */
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    ref[count] = scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
  }
}
void SCMParmParse::fastputarr(SCM variable, vector⟨double⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double(ref[count]));
  }
}
#ifdef CH_SPACEDIM
void SCMParmParse::fastgetarr(SCM variable, Vector⟨double⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
  for (int count = 0; count < length; count++) {
    ref[count] = scm_to_double(scm_uniform_vector_ref(variable_ref, scm_from_int(count)));
  }
}
void SCMParmParse::fastputarr(SCM variable, Vector⟨double⟩ &ref)
{
  SCM variable_ref = scm_variable_ref(variable);
  int length = ref.size();
```

```
    for (int count = 0; count < length; count++) {
      scm_uniform_vector_set_x(variable_ref, scm_from_int(count), scm_from_double(ref[count]));
    }
  }
#endif
  void SCMParmParse::getarr(const char *name, vector<double> &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#endif
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

   1. a list

   2. a vector

   3. a uniform vector

   */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = scm_to_double(item);
            else
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::getarr): list value not a double" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::getarr): list value not a double" << endl;
#endif
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
```

```
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else
#ifdef CH_SPACEDIM
            pout() << "\n\tERROR(SCMParmParse::getarr): vector value not a double" << endl;
#else
            cerr << "\n\tERROR(SCMParmParse::getarr): vector value not a double" << endl;
#endif
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 <= start_ix) && (start_ix <= length - 1)) {
        int num_val_available = length - start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else
#ifdef CH_SPACEDIM
            pout() << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a double" <<
                endl;
#else
            cerr << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a double" << endl;
#endif
        }
      }
    }
    else
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" <<
          endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" << endl;
#endif
  }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, vector<double> &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
#ifdef CH_SPACEDIM
    pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
```

```
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear( );
    if (does_scm_symbol_exist(fullname.c_str( ))) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
          /* This may be either

  1. a list

  2. a vector

  3. a uniform vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣a␣double" ≪ endl;
#else
              cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣a␣double" ≪ endl;
#endif
              return_value = false;
              goto exit;
            }
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_real(item)) ref[count] = scm_to_double(item);
            else {
#ifdef CH_SPACEDIM
              pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr):␣vector␣value␣not␣a␣double" ≪ endl;
#else
              cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣vector␣value␣not␣a␣double" ≪ endl;
#endif
              return_value = false;
              goto exit;
            }
```

```
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
#ifdef CH_SPACEDIM
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣uniform␣vector␣value␣not␣a␣double" ≪
                endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣uniform␣vector␣value␣not␣a␣double" ≪
                endl;
#endif
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
#endif
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

¶ Process lists and vectors of **string**s.

210 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
```

```
#ifdef CH_SPACEDIM
      pout() << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#else
      cerr << "\n\tERROR(SCMParmParse::getarr): name empty" << endl;
#endif
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 <= start_ix) ∧ (start_ix <= length - 1)) {
          int num_val_available = length - start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_string(item)) {
              char *buffer = scm_to_locale_string(item);
              ref[count].clear();
              ref[count] += buffer;
              free(buffer);
            }
            else
#ifdef CH_SPACEDIM
              pout() << "\n\tERROR(SCMParmParse::getarr): list value not a string" << endl;
#else
              cerr << "\n\tERROR(SCMParmParse::getarr): list value not a string" << endl;
#endif
            }
          }
        }
        else if (scm_is_vector(variable)) {
          int length = scm_to_int(scm_vector_length(variable));
          if ((0 <= start_ix) ∧ (start_ix <= length - 1)) {
            int num_val_available = length - start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_string(item)) {
                char *buffer = scm_to_locale_string(item);
                ref[count].clear();
                ref[count] += buffer;
                free(buffer);
              }
              else
#ifdef CH_SPACEDIM
```

```
            pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣a␣string" ≪ endl;
#else
            cerr ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣a␣string" ≪ endl;
#endif
          }
        }
      }
      else
#ifdef CH_SPACEDIM
        pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list␣or␣vector" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list␣or␣vector" ≪ endl;
#endif
    }
  exit: return;
  }
  bool SCMParmParse::queryarr(const char *name, vector⟨string⟩ &ref, int start_ix, int num_val)
  {
    string fullname;
    bool return_value = true;
    fullname.clear();
    if (¬prefix.empty()) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
#ifdef CH_SPACEDIM
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
#else
      cerr ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
#endif
      return_value = false;
      goto exit;
    }
    ref.clear();
    if (does_scm_symbol_exist(fullname.c_str())) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
          /* This may be either

  1. a list

  2. a vector

  */
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_string(item)) {
              char *buffer = scm_to_locale_string(item);
```

```
                ref[count].clear( );
                ref[count] += buffer;
                free(buffer);
              }
              else {
#ifdef CH_SPACEDIM
                pout( ) << "\n\tERROR(SCMParmParse::queryarr): list value not a string" << endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): list value not a string" << endl;
#endif
                return_value = false;
                goto exit;
              }
            }
          }
        }
        else if (scm_is_vector(variable)) {
          int length = scm_to_int(scm_vector_length(variable));
          if ((0 <= start_ix) ∧ (start_ix <= length − 1)) {
            int num_val_available = length − start_ix;
            ref.resize(min(num_val_available, num_val));
            for (int count = 0; count < min(num_val_available, num_val); count++) {
              SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
              if (scm_is_string(item)) {
                char *buffer = scm_to_locale_string(item);
                ref[count].clear( );
                ref[count] += buffer;
                free(buffer);
              }
              else {
#ifdef CH_SPACEDIM
                pout( ) << "\n\tERROR(SCMParmParse::queryarr): vector value not a string" << endl;
#else
                cerr << "\n\tERROR(SCMParmParse::queryarr): vector value not a string" << endl;
#endif
                return_value = false;
                goto exit;
              }
            }
          }
        }
        else {
#ifdef CH_SPACEDIM
          pout( ) << "\n\tERROR(SCMParmParse::queryarr): not a list or vector" << endl;
#else
          cerr << "\n\tERROR(SCMParmParse::queryarr): not a list or vector" << endl;
#endif
          return_value = false;
          goto exit;
        }
      }
      else return_value = false;
    exit: return return_value;
    }
```

¶ Trivial overload: **vector** → **Vector** and *cerr* → *pout*( ).

211 ⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
#ifdef CH_SPACEDIM
  void SCMParmParse::getarr(const char *name, Vector<int> &ref, int start_ix, int num_val)
  {
    string fullname;
    fullname.clear( );
    if (¬prefix.empty( )) {
      fullname += prefix;
      fullname += ".";
    }
    if (name) fullname += name;
    else {
      pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣name␣empty" ≪ endl;
      goto exit;
    }
    ref.clear( );
    if (does_scm_symbol_exist(fullname.c_str( ))) {
      SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
      if (scm_is_true(scm_list_p(variable))) {
        int length = scm_to_int(scm_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣list␣value␣not␣an␣integer" ≪ endl;
          }
        }
      }
      else if (scm_is_vector(variable)) {
        int length = scm_to_int(scm_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
            else
              pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣an␣integer" ≪ endl;
          }
        }
      }
      else if (scm_is_true(scm_uniform_vector_p(variable))) {
        int length = scm_to_int(scm_uniform_vector_length(variable));
        if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
          int num_val_available = length − start_ix;
          ref.resize(min(num_val_available, num_val));
          for (int count = 0; count < min(num_val_available, num_val); count++) {
            SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
            if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
```

```
        else
          pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣uniform␣vector␣value␣not␣an␣integer" ≪
              endl;
      }
    }
  }
  else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
      endl;
}
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector⟨int⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
    return_value = false;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else {
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣an␣integer" ≪ endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
```

```
        else {
          pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣vector␣value␣not␣an␣integer" ≪ endl;
          return_value = false;
          goto exit;
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_integer(item)) ref[count] = (int) scm_to_double(item);
          else {
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣uniform␣vector␣value␣not\
                ␣an␣integer" ≪ endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else {
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

212 ¶⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣name␣empty" ≪ endl;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
```

```
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr): list value not a float" ≪ endl;
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr): vector value not a float" ≪ endl;
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
          else
            pout() ≪ "\n\tERROR(SCMParmParse::getarr): uniform vector value not a float" ≪
                endl;
        }
      }
    }
    else pout() ≪ "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" ≪
        endl;
  }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector⟨float⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
```

```
if (name) fullname += name;
else {
  pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣name␣empty" ≪ endl;
  return_value = false;
  goto exit;
}
ref.clear();
if (does_scm_symbol_exist(fullname.c_str())) {
  SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
  if (scm_is_true(scm_list_p(variable))) {
    int length = scm_to_int(scm_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
        else {
          pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣list␣value␣not␣a␣float" ≪ endl;
          return_value = false;
          goto exit;
        }
      }
    }
  }
  else if (scm_is_vector(variable)) {
    int length = scm_to_int(scm_vector_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
        else {
          pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣vector␣value␣not␣a␣float" ≪ endl;
          return_value = false;
          goto exit;
        }
      }
    }
  }
  else if (scm_is_true(scm_uniform_vector_p(variable))) {
    int length = scm_to_int(scm_uniform_vector_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = (float) scm_to_double(item);
        else {
          pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣uniform␣vector␣value␣not␣a␣float" ≪
              endl;
          return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else {
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list,␣vector␣or␣uniform␣vector" ≪
          endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

213 ¶⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, Vector⟨double⟩ &ref, int start_ix, int num_val)
{
  string fullname;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣name␣empty" ≪ endl;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else pout() ≪ "\n\tERROR(SCMParmParse::getarr):␣list␣value␣not␣a␣double" ≪ endl;
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
```

```
        if (scm_is_real(item)) ref[count] = scm_to_double(item);
        else pout() << "\n\tERROR(SCMParmParse::getarr): vector value not a double" << endl;
      }
    }
  }
  else if (scm_is_true(scm_uniform_vector_p(variable))) {
    int length = scm_to_int(scm_uniform_vector_length(variable));
    if ((0 <= start_ix) && (start_ix <= length - 1)) {
      int num_val_available = length - start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_real(item)) ref[count] = scm_to_double(item);
        else
          pout() << "\n\tERROR(SCMParmParse::getarr): uniform vector value not a double" <<
              endl;
      }
    }
  }
  else pout() << "\n\tERROR(SCMParmParse::getarr): not a list, vector or uniform vector" <<
      endl;
}
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector<double> &ref, int start_ix, int num_val)
{
  string fullname;
  bool return_value = true;
  fullname.clear();
  if (¬prefix.empty()) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout() << "\n\tERROR(SCMParmParse::queryarr): name empty" << endl;
    return_value = false;
    goto exit;
  }
  ref.clear();
  if (does_scm_symbol_exist(fullname.c_str())) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str()));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 <= start_ix) && (start_ix <= length - 1)) {
        int num_val_available = length - start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
            pout() << "\n\tERROR(SCMParmParse::queryarr): list value not a double" << endl;
            return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr): vector value not a double" ≪ endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else if (scm_is_true(scm_uniform_vector_p(variable))) {
      int length = scm_to_int(scm_uniform_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_uniform_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_real(item)) ref[count] = scm_to_double(item);
          else {
            pout() ≪ "\n\tERROR(SCMParmParse::queryarr): uniform vector value not a double" ≪
                endl;
            return_value = false;
            goto exit;
          }
        }
      }
    }
    else {
      pout() ≪ "\n\tERROR(SCMParmParse::queryarr): not a list, vector or uniform vector" ≪
          endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
```

214 ¶⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::getarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val)
{
  string fullname;
```

```
  fullname.clear( );
  if (¬prefix.empty( )) {
    fullname += prefix;
    fullname += ".";
  }
  if (name) fullname += name;
  else {
    pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣name␣empty" ≪ endl;
    goto exit;
  }
  ref.clear( );
  if (does_scm_symbol_exist(fullname.c_str( ))) {
    SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
    if (scm_is_true(scm_list_p(variable))) {
      int length = scm_to_int(scm_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_string(item)) {
            char *buffer = scm_to_locale_string(item);
            ref[count].clear( );
            ref[count] += buffer;
            free(buffer);
          }
          else pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣list␣value␣not␣a␣string" ≪ endl;
        }
      }
    }
    else if (scm_is_vector(variable)) {
      int length = scm_to_int(scm_vector_length(variable));
      if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
        int num_val_available = length − start_ix;
        ref.resize(min(num_val_available, num_val));
        for (int count = 0; count < min(num_val_available, num_val); count++) {
          SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
          if (scm_is_string(item)) {
            char *buffer = scm_to_locale_string(item);
            ref[count].clear( );
            ref[count] += buffer;
            free(buffer);
          }
          else pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣vector␣value␣not␣a␣string" ≪ endl;
        }
      }
    }
    else pout( ) ≪ "\n\tERROR(SCMParmParse::getarr):␣not␣a␣list␣or␣vector" ≪ endl;
  }
exit: return;
}
bool SCMParmParse::queryarr(const char *name, Vector⟨string⟩ &ref, int start_ix, int num_val)
{
```

```
string fullname;
bool return_value = true;
fullname.clear( );
if (¬prefix.empty( )) {
  fullname += prefix;
  fullname += ".";
}
if (name) fullname += name;
else {
  pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr): name empty" ≪ endl;
  return_value = false;
  goto exit;
}
ref.clear( );
if (does_scm_symbol_exist(fullname.c_str( ))) {
  SCM variable = scm_variable_ref(scm_c_lookup(fullname.c_str( )));
  if (scm_is_true(scm_list_p(variable))) {
    int length = scm_to_int(scm_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_list_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_string(item)) {
          char *buffer = scm_to_locale_string(item);
          ref[count].clear( );
          ref[count] += buffer;
          free(buffer);
        }
        else {
          pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr): list value not a string" ≪ endl;
          return_value = false;
          goto exit;
        }
      }
    }
  }
  else if (scm_is_vector(variable)) {
    int length = scm_to_int(scm_vector_length(variable));
    if ((0 ≤ start_ix) ∧ (start_ix ≤ length − 1)) {
      int num_val_available = length − start_ix;
      ref.resize(min(num_val_available, num_val));
      for (int count = 0; count < min(num_val_available, num_val); count++) {
        SCM item = scm_vector_ref(variable, scm_from_int(start_ix + count));
        if (scm_is_string(item)) {
          char *buffer = scm_to_locale_string(item);
          ref[count].clear( );
          ref[count] += buffer;
          free(buffer);
        }
        else {
          pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr): vector value not a string" ≪ endl;
          return_value = false;
```

```
            goto exit;
          }
        }
      }
    }
    else {
      pout( ) ≪ "\n\tERROR(SCMParmParse::queryarr):␣not␣a␣list␣or␣vector" ≪ endl;
      return_value = false;
      goto exit;
    }
  }
  else return_value = false;
exit: return return_value;
}
#endif     /* CH_SPACEDIM */
```

215 ¶⟨ **SCMParmParse** implementation: queries and extractors 200 ⟩ +≡

```
void SCMParmParse::print_prefix( )
{
#ifdef CH_SPACEDIM
  pout( ) ≪ "SCMParmParse::prefix␣=␣" ≪ prefix ≪ endl;
#else
  cout ≪ "SCMParmParse::prefix␣=␣" ≪ prefix ≪ endl;
#endif
}
```

SCMParmParse Protected Members

Function *does_scm_symbol_exist*, borrowed from [8] with one small modification that fixes the guile-1.8.1 note about deprecated features, does a job that is similar to *scm_c_lookup*( )[11] with one difference: it does not bomb out if the symbol is not found, just returns *false*.

Implementation notes:

scm_str2symbol This function is defined on `"libguile/discouraged.h"`. I have replaced it with *scm_string_to_symbol*.

scm_sym2var This is a Guile internal, defined on `"libguile/modules.h"`.

scm_current_module_lookup_closure( ) Ditto.

216 ⟨ **SCMParmParse** implementation: protected members 216 ⟩ ≡

```
  /* This function comes from [8], with some modifications to account for guile-1.8.1 and our own needs. */
bool SCMParmParse::does_scm_symbol_exist(const char *name)
{
  SCM symbol;
  SCM variable;
  assert(name ≠ (const char *) Λ);
  symbol = scm_string_to_symbol(scm_from_locale_string(name));
    /* Lookup this symbol in the current module lookup closure, but do not create it automatically if it
    does not exist. This is what the third argument to this function is about. Return SCM_BOOL_F if this is
    the case. */
  variable = scm_sym2var(symbol, scm_current_module_lookup_closure( ), SCM_BOOL_F);
```

[11] Defined in "Accessing Modules from C" in "The Guile Module System" section of the "Modules" chapter.

```
  return (variable ≠ SCM_BOOL_F);
}
```

See also chunk 217.

This code is used in chunk 198.

¶ This is a simple function that returns the file size in bytes if the file named in its argument is readable and *false* (zero) otherwise. If it so happens that the file is readable, but its size is zero, *false* is returned as well.

We first call the UNIX *stat* function on the file, and if this fails, we return *false*.

Otherwise we check if the file is regular, i.e., not a device file and not a directory either.[12] If the file is not regular, we return *false*.

Otherwise we attempt to open the file with the C++ **ifstream**::*open*( ). If it fails we return *false*.

Otherwise, we close the file and return its size in bytes.

Why not try **ifstream**::*open*( ) right away? This is because the *open*( ) would work on a directory or a device file, and we don't want this.

This function is an almost exact copy of *is_file_readable* declared and defined in the FORMS code, with the exception that it uses *cerr* instead of *pout*( ) if CH_SPACEDIM is not defined, because we want it to work outside Chombo as well. Also, we use *false* instead of NO.

217 ⟨ **SCMParmParse** implementation: protected members 216 ⟩ +≡

```
  off_t SCMParmParse::is_file_readable(const char *file_name)
  {
    struct stat file_status;
    off_t exit_status;
    ifstream file_stream;
    if (stat(file_name, &file_status) < 0) {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(is_file_readable): cannot stat " ≪ file_name ≪ endl;
      pout( ) ≪ "\t\tError number " ≪ errno ≪ endl;
#else
      cerr ≪ "\n\tERROR(is_file_readable): cannot stat " ≪ file_name ≪ endl;
      cerr ≪ "\t\tError number " ≪ errno ≪ endl;
#endif
      exit_status = (off_t) false;
      goto exit;
    }
    if (¬(S_ISREG(file_status.st_mode))) {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(is_file_readable): file " ≪ file_name ≪ " is not regular" ≪ endl;
#else
      cerr ≪ "\n\tERROR(is_file_readable): file " ≪ file_name ≪ " is not regular" ≪ endl;
#endif
      exit_status = (off_t) false;
      goto exit;
    }
    file_stream.open(file_name);
    if (file_stream.fail( )) {
#ifdef CH_SPACEDIM
      pout( ) ≪ "\n\tERROR(is_file_readable): file " ≪ file_name ≪
          " cannot be opened for reading" ≪ endl;
#else
      cerr ≪ "\n\tERROR(is_file_readable): file " ≪ file_name ≪
          " cannot be opened for reading" ≪ endl;
#endif
```

[12] (Will a soft link show as a regular file? Hopefully not.

```
    exit_status = (off_t) false;
    goto exit;
  }
  file_stream.close( );
  exit_status = file_status.st_size;
exit: return exit_status;
}
```

12.2 EdgeFab⟨T⟩ Class Template

The **EdgeFab**⟨**T**⟩ class template [13] is similar to **FluxBox**, with the exception that fields are mounted on cell edges, not on faces, and that the fields can be of various numeric types, **Real**, **int**, or anything else that the **BaseFab** template can accept.

Figure 5 shows the geometric difference between a **FluxBox** and an **EdgeFab**⟨**T**⟩ cell.

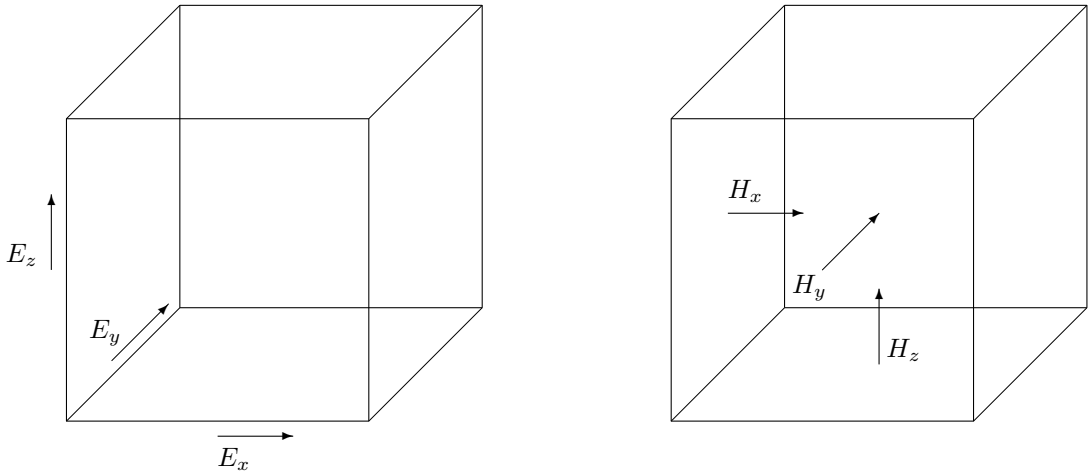


Figure 5: An **EdgeFab**⟨**T**⟩ cell on the left versus a **FluxBox** cell on the right.

In a **FluxBox**, fields are fluxes through the faces of each cell. And so,

H_x is attached in the middle of the $\boldsymbol{e}_y \times \boldsymbol{e}_z$ face, and is of type (NODE, CELL, CELL),

H_y is attached in the middle of the $\boldsymbol{e}_z \times \boldsymbol{e}_y$ face, and is of type (CELL, NODE, CELL), and

H_z is attached in the middle of the $\boldsymbol{e}_x \times \boldsymbol{e}_y$ face, and is of type (CELL, CELL, NODE).

On the other hand, in an **EdgeFab**⟨**T**⟩, fields are defined on the edges of each cell, so that curls, which in themselves are fluxes, can be evaluated for each cell face. And so,

E_x is attached in the middle of the $\boldsymbol{e}_x$ edge, and is of type (CELL, NODE, NODE),

E_y is attached in the middle of the $\boldsymbol{e}_y$ edge, and is of type (NODE, CELL, NODE), and

E_z is attached in the middle of the $\boldsymbol{e}_z$ edge, and is of type (NODE, NODE, CELL).

In summary, we find that in a **FluxBox** fields are NODE mounted in their directions, and (CELL, CELL) mounted in the other two directions, whereas in an **EdgeFab**⟨**T**⟩ it's the other way round: fields are CELL mounted in their directions, and (NODE, NODE) mounted in the remaining two directions.

This has obvious ramifications in terms of how data is to be moved between levels for these two complementary classes, because their geometric alignment between the two levels will be different—and different from cell-mounted fields, too. At the same time, we will have to ensure that divergence is not accidentally generated neither for the $\boldsymbol{B}$, nor for the $\boldsymbol{E}$ fields in the process.

We can easily make use of the CELL and NODE designation, to find the exact location of field attachment in the physical space. The **IntVect** numbering of the cells corresponds to cell centers. At the same time we have that CELL = 0 and NODE = 1, and the specific indexing type of a box is returned by **IntVect Box**::*type*( ).

For example, for the H_x field, the box type should be (NODE, CELL, CELL) = (1, 0, 0). The point of attachment of a field in a cell (i, j, k) of type *type* is

$$\boldsymbol{r}_{\text{field}} = \boldsymbol{r}_0 \quad + \left(i - \frac{\text{type}_x}{2}\right) \Delta_x \boldsymbol{e}_x \quad + \left(j - \frac{\text{type}_y}{2}\right) \Delta_y \boldsymbol{e}_y \quad + \left(k - \frac{\text{type}_z}{2}\right) \Delta_z \boldsymbol{e}_z \tag{361}$$

[13] This template derives from the **FluxBox** class. I have also consulted the **EdgeDataBox** code by Dan Martin.

And for H_x and $\boldsymbol{r}_0 = (0,0,0)$ this becomes

$$\boldsymbol{r}_{H_x}(i,j,k) = \left(i - \frac{1}{2}\right)\Delta_x \boldsymbol{e}_x + j\Delta_y \boldsymbol{e}_y + k\Delta_z \boldsymbol{e}_z$$

At the same time for E_x and $\boldsymbol{r}_0 = (0,0,0)$ this becomes

$$\boldsymbol{r}_{E_x}(i,j,k) = i\Delta_x \boldsymbol{e}_x + \left(j - \frac{1}{2}\right)\Delta_y \boldsymbol{e}_y + \left(k - \frac{1}{2}\right)\Delta_z \boldsymbol{e}_z$$

These simple formulae will become useful when defining media layouts and movements of data between levels.

Although both **FluxBox** and **EdgeFab**$\langle$**T**$\rangle$ are defined on a cell centered box of some dimensions, the fields, (H_x, H_y, H_z) for **FluxBox** and (E_x, E_y, E_z) for **EdgeFab**$\langle$**T**$\rangle$ each have their own boxes associated with them. The boxes are typed, depending on the field, as above, and they are also *sized* so as to include fluxes or curls over the whole box, which means adding layers of data "on the other side" of the box. These would not normally be included, if we were to define fields simply on the original cells of a cell centered box, merely re-typed to mount data on faces or edges, with the effect that we could not evaluate full fluxes or full curls associated with the box using the box data alone. [14]

For example, consider a cell centered box, let us call it *cell_centered_box*: $((0,0,0)(2,2,2)(0,0,0))$. This is how Chombo would write the box out to *cout* or *pout*( ). The box has its cells numbered from $(0,0,0)$ through $(2,2,2)$ and the field mounting points in all its cells are in cell centres, $(0,0,0)$.

In order to include fluxes through all its surfaces, we need to add a layer of cells on the right hand side for the H_x field, a layer of cells on the far side for the H_y field and a layer of cells on the high side for the H_z field.

If we were to define **FluxBox** B(*cell_centered_box*), this would result in packing three **FArrayBox**es into B. One for H_x, one for H_y and one for H_z. Each of these **FArrayBox**es can be extracted from B by calling a **FluxBox**::*getFlux*(*dir*) method. If we were to inspect the actual boxes that would be returned with these fields, we would find that the boxes have been resized and typed so as to incorporate data on the other side and so as to attach the data to cell faces.

And so,

B.*getFlux*(0).*box*( ) returns $((0,0,0)(3,2,2)(1,0,0))$,

B.*getFlux*(1).*box*( ) returns $((0,0,0)(2,3,2)(0,1,0))$, and

B.*getFlux*(2).*box*( ) returns $((0,0,0)(2,2,3)(0,0,1))$.

Similarly for **EdgeFab** the internal boxes associated with fields (E_x, E_y, E_z) must be resized and typed so as to let us calculate curls over the box's surface.

If we were to define **EdgeFab**$\langle$**Real**$\rangle$ E(*cell_centered_box*), this would result in packing three **BaseFab**$\langle$**Real**$\rangle$s into E. One for E_x, one for E_y and one for E_z. Each of these **BaseFab**$\langle$**Real**$\rangle$s can be extracted from E by calling an **EdgeFab**$\langle$**Real**$\rangle$::*getEdgeData*(*dir*) method. If we were to inspect the actual boxes that would be returned with these fields, we would find that the boxes have been resized and typed so as to incorporate data "on the other edge" and so that the fields would be attached to the edges as well.

And so,

E.*getEdgeData*(0).*box*( ) returns $((0,0,0)(2,3,3)(0,1,1))$,

E.*getEdgeData*(1).*box*( ) returns $((0,0,0)(3,2,3)(1,0,1))$, and

E.*getEdgeData*(2).*box*( ) returns $((0,0,0)(3,3,2)(1,1,0))$.

Let us note here that the first box, $((0,0,0)(2,3,3)(0,1,1))$ contains *all* four $\boldsymbol{e}_x$ edges of the box. They are $([0..2],0,0)$, $([0..2],3,0)$, $([0..2],0,3)$, and $([0..2],3,3)$. Similarly for the other boxes. It is therefore possible to evaluate curls over all six faces of the original cell-centered box.

The face and edge data are *shared* with adjacent boxes. A programmer, i.e., me, must ensure that the shared data is always identical.

[14] This is another difference between, say, **FArrayBox**, which *could* be defined on edge or face centered cells as well, and **FluxBox** or **EdgeFab**$\langle$**T**$\rangle$. This fundamental difference implies that, unfortunately, the internal boxes of **FluxBox** and **EdgeFab**$\langle$**T**$\rangle$ cannot be arranged into a **DisjointBoxLayout**, because they aren't disjoint. They must overlap on the box-box boundaries. But the basic cell-centered boxes of both classes, of course, can be arranged into a **DisjointBoxLayout**.

12.2.1 EdgeFab⟨T⟩ Headers

This is the header file, that should be written on `"EdgeFab.H"` eventually. Ctangle will write it on `"EdgeFab.hw"`, for further processing with `sed` and `astyle`.

The whole content of the file is enclosed in the # **ifdef** .. # **endif** brackets, to ensure that the declarations aren't read into the compilation stream more than once.

The implementation is treated like a header file as well and is included after the declarations.

We do not include `"FArrayBox.H"`, because the class is based on the **FArrayBox**'s parent **BaseFab**⟨**T**⟩. Here we lose C++ arithmetic that we can do directly on **FArrayBox**es, but nothing else. In particular, **BaseFab**⟨**T**⟩ provides the same Fortran-style data layout as **FArrayBox**.

219 ⟨ `EdgeFab.hw` 219 ⟩ ≡
```
#ifndef CH_EDGEFAB_H
# define CH_EDGEFAB_H
#include <Box.H>
#include <Vector.H>
#include <BaseFab.H>
#include <REAL.H>
  ⟨ EdgeFab: class template declaration 220 ⟩
#include <EdgeFabImplem.H>
#endif     /* CH_EDGEFAB_H */
```
This code is cited in chunk 228.

¶ The class template **EdgeFab**⟨**T**⟩ is declared with public, protected and private members. These are discussed below in their own chunks.

220 ⟨ **EdgeFab**: class template declaration 220 ⟩ ≡
```
template<class T> class EdgeFab {
public: ⟨ EdgeFab: public members 221 ⟩
protected: ⟨ EdgeFab: protected members 226 ⟩
private: ⟨ EdgeFab: private members 227 ⟩
};
```
This code is used in chunk 219.

Public Members

This is the public image of the class template and it is here that we are going to discuss what various methods are supposed to do, then in the implementation part, we'll discuss how they go about it.

EdgeFab( ) This is a default constructor, which in this case is not going to construct much, because it doesn't have enough data. But it sets an internal class parameter *m_nComp* to −1, which flags to all other class methods that the object is not fully defined.

EdgeFab(const Box &*box*, **int** *nComp* = 1) This form of the constructor builds an **EdgeFab** on a *cell centered box*, also referred to as the object's *domain,* attaching an *uninitialized nComp* component field at each edge site of the *box*, including its far edges. The default value for *nComp* is 1. The electrodynamic $\boldsymbol{E} = (E_x, E_y, E_z)$ field has *nComp* = 1, because there is just one component, E_x, E_y, or E_z attached to the edge of each cell. Values of *nComp* greater than 1 are for tensor fields. But we could, for example, make use of multiple components here to encode both $\boldsymbol{D}$ and $\boldsymbol{E}$ on the same **EdgeFab**⟨**Real**⟩. We could even pack auxiliary fields $\boldsymbol{S}_n, n = 1, \ldots, k$ that arise from time integration of $\boldsymbol{E}$ (these are various induced "currents"). Or we could choose to keep $\boldsymbol{D}$ and $\boldsymbol{E}$ separate, but lump all $\boldsymbol{S}_n$ together.

∼**EdgeFab**( ) The default destructor. It calls *clear*( ), which deletes all data associated with the **EdgeFab** fields, resets internal pointers to Λ, and resets *m_nComp* to −1, so that all other class methods know that the object is no longer fully defined.

221 ⟨ **EdgeFab**: public members 221 ⟩ ≡
```
EdgeFab( );
EdgeFab(const Box &box, int nComp = 1);
~EdgeFab( );
```
See also chunks 222, 223, 224, and 225.
This code is used in chunk 220.

Constructors and Destructors

The actual constructors and destructors of the class:

resize(**const Box** &*box*, **int** *nComp* = 1) This function either allocates all space required for the **BaseFab**⟨**T**⟩ fields that are going to live within the **EdgeFab**, or resizes the object's domain, and its internal boxes together with it, and, if requested, also the number of field components for an existing **EdgeFab**, always making sure that there is enough space available for data. *box* is the possibly new cell-centered domain over which to span the field, and *nComp* is the number of the field's components.

define(**const Box** &*box*, **int** *nComp* = 1) This is the real constructor method of the class. It creates *SpaceDim* multicomponent (*nComp*) edge mounted *uninitialized* **BaseFab**⟨**T**⟩s over the object's domain *box*, providing each **BaseFab**⟨**T**⟩ with its own specially sized and typed box, so as to include fields attached to all edges of the domain, including the far ones.

clear( ) This is the effective destructor of the class. It wipes everything out, deallocates space, resets internal pointers to Λ, and resets *m_nComp* to −1.

222 ⟨ **EdgeFab**: public members 221 ⟩ +≡
```
void resize(const Box &box, int nComp = 1);
void define(const Box &box, int nComp = 1);
void clear( );
```

Extractors

Now we have a range of *extractors,* i.e., functions that extract data from **EdgeFab**:

nComp( ) This function returns the number of field components.

box( ) This function returns the cell-centered box that the **EdgeFab** is built on.

getEdgeData(**const int** *dir*) This function returns the **BaseFab**⟨**T**⟩ that is associated with the edge in direction *dir*. The two versions have an identical code inside. The **const** one doesn't allow the programmer to modify the returned field, which is enforced by the declaration. This same function can be also invoked simply by using the [ ] operator, which makes an **EdgeFab**⟨**T**⟩ look like an array of three **BaseFab**⟨**T**⟩s.

223 ⟨ **EdgeFab**: public members 221 ⟩ +≡
```
int nComp( ) const;
const Box &box( ) const;
BaseFab⟨T⟩ &getEdgeData(const int dir);
const BaseFab⟨T⟩ &getEdgeData(const int dir) const;
BaseFab⟨T⟩ &operator[](const int dir);
const BaseFab⟨T⟩ &operator[](const int dir) const;
```

Modifiers

The modifiers of the class fall in two groups:

setVal( ) These methods set the value of the **EdgeFab**⟨**T**⟩ fields, either all components of all of them, to some real value, or all components in some specified direction *dir*, or some specified range of the components in a specified direction, on the whole domain of the defined object or on a *box* that is contained by the domain.

The *setVal*( ) methods invoke the **BaseFab**⟨**T**⟩::*setVal*( ) methods internally.

copy( ) These methods *copy* data from an argument **EdgeFab**⟨**T**⟩ to *this* **EdgeFab**⟨**T**⟩. As their semantics are somewhat subtle, here we discuss them in more detail.

copy(**const EdgeFab**⟨**T**⟩ &*src*) This method copies the whole content of the argument **EdgeFab**⟨**T**⟩ into this **EdgeFab**⟨**T**⟩, on the intersection of the two **BaseFab**⟨**T**⟩ domains.

copy(**const EdgeFab**⟨**T**⟩ &*src*, **const int** *srcComp*, **const int** *destComp*, **const int** *numComp*) This method copies as above, but the copy is restricted to the specified range of components.

copy(**const EdgeFab**⟨**T**⟩ &*src*, **const int** *dir*, **const int** *srcComp*, **const int** *destComp*, **const int** *numComp*) This method copies as above, but the copy is additionally restricted to a specified direction, for example, E_x only.

copy(**const Box** &*RegionFrom*, **const Interval** &*Cdest*, **const Box** &*RegionTo*, **const EdgeFab**⟨**T**⟩ &*src*, **const In** This method copies edge data that is sourced from the intersection of the *RegionFrom* box with the source **EdgeFab**⟨**T**⟩ domain to the intersection of the *RegionTo* box with the domain of *this* box. The copy moves data between the *Csrc* and *Cdest* component intervals.

copy(**const Box** &*Region*, **const Interval** &*Cdest*, **const EdgeFab**⟨**T**⟩ &*src*, **const Interval** &*Csrc*) This method copies as above with the difference that the destination box is the same as the source box, here referred to as *Region*. The data movement is restricted to the intersection of *Region* with the domains of the source and *this* **EdgeFab**⟨**T**⟩s.

shift(**const IntVect** &*v*) This function does nothing to the field data. Instead, it shifts all boxes within the **EdgeFab**⟨**T**⟩ by *v*, which is a lightweight operation. Yet, this has the effect of pushing the whole **EdgeFab**⟨**T**⟩ by *v*, because the way the field data is associated with an actual grid point is through the definition of boxes on which the data is spanned. Chombo does not define the actual grid and does not bind the data to specific grid points other than through the data layout within its internal arrays, which is Fortran-like rather than C-like.

224 ⟨ **EdgeFab**: public members 221 ⟩ +≡

```
void setVal(const T val);
void setVal(const T val, const int dir);
void setVal(const T val, const int dir, const int startComp, const int nComp);
void setVal(const T val, const Box &box);
void setVal(const T val, const Box &box, const int dir, const int startComp, const int nComp);
void copy(const EdgeFab<T> &src);
void copy(const EdgeFab<T> &src, const int srcComp, const int destComp, const int numComp);
void copy(const EdgeFab<T> &src, const int dir, const int srcComp, const int destComp, const int
    numComp);
void copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const EdgeFab<T>
    &src, const Interval &Csrc);
void copy(const Box &Region, const Interval &Cdest, const EdgeFab<T> &src, const Interval &Csrc);
EdgeFab<T> &shift(const IntVect &v);
```

Transfer Helpers

These four functions are used in transferring data between MPI processes. The way it's done, generally, is as follows: the data from box edges are packed into a buffer by calling *linearOut*. The buffer is then MPI transmitted to the destination process (the **LevelData**⟨ ⟩ . *exchange*( ) does this). There, it is unpacked back onto box edges by calling *linearIn*.

size(**const Box** &*box*, **const Interval** &*comps*) This method returns the size, in bytes, of data in components *comps* attached to all cell edges associated with the cell centered box *box*. It's purpose is to provide sizing for transfer buffers.

linearOut(**void** ∗*buf*, **const Box** &*R*, **const Interval** &*comps*) This method writes all edge data associated with all cells of **Box** *R* and resident in components *comps* onto a preallocated and presized buffer *buf*.

linearIn(**void** ∗*buf*, **const Box** &*R*, **const Interval** &*comps*) This method unpacks edge data that has arrived in buffer *buf* onto all cells of **Box** *R* and writes them into components *comps*.

preAllocatable( ) A small helper function used by the above.

225 ⟨ **EdgeFab**: public members 221 ⟩ +≡
```
int size(const Box &box, const Interval &comps) const;
void linearOut(void *buf, const Box &R, const Interval &comps) const;
void linearIn(void *buf, const Box &R, const Interval &comps);
static int preAllocatable( )
{
  return 0;
}
```

Protected Members

m_box This is the cell centered box, also referred to as the object's *domain*, the **EdgeFab**⟨**T**⟩ is built on.

m_nComp This is the number of components mounted on each cell edge.

m_edgeData This is a vector of *SpaceDim* **BaseFab**⟨**T**⟩s that stores the field data.

226 ⟨ **EdgeFab**: protected members 226 ⟩ ≡
```
Box m_box;
int m_nComp;
Vector<BaseFab<T> *> m_edgeData;
```
This code is used in chunk 220.

Private Members

These two are for copying the whole **EdgeFab** onto another one, as in *EdgeFab_1* = *EdgeFab_2*. They are declared here, but they're not defined in the code, so they can't be used.

227 ⟨ **EdgeFab**: private members 227 ⟩ ≡
```
EdgeFab(const EdgeFab<T> &);
EdgeFab<T> &operator=(const EdgeFab<T> &);
```
This code is used in chunk 220.

12.2.2 EdgeFab⟨T⟩ Implementation

Now, we get to implement the function templates declared in Section 12.2.1. Because the whole class is defined as a template, this file is also a header file, to be included in `"EdgeFab.H"`, as discussed in module ⟨ `EdgeFab.hw` 219 ⟩, page 302.

Its own structure is bracketed by # **ifdef** .. # **endif** clauses that ensure that the definitions will not be read into the compilation stream more than once. The file must ultimately become `"EdgeFabImplem.H"`.

228 ⟨ `EdgeFabImplem.hw` 228 ⟩ ≡
```
#ifndef CH_EDGEFABIMPLEM_H
# define CH_EDGEFABIMPLEM_H
  ⟨ EdgeFab template implementation 229 ⟩
#endif      /* CH_EDGEFABIMPLEM_H */
```

Extractors

We begin by implementing the two trivial methods, *nComp*( ) and *box*( ). The first one returns the value of *m_nComp*, which stores the number of components to be attached at any given edge. The second one returns the value of *m_box*, which stores the cell centered box the **EdgeFab** is built on.

229 ⟨ **EdgeFab** template implementation 229 ⟩ ≡

```
template⟨class T⟩ int EdgeFab⟨T⟩::nComp( ) const
{
  return m_nComp;
}
template⟨class T⟩ const Box &EdgeFab⟨T⟩::box( ) const
{
  return m_box;
}
```

See also chunks 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 244, 245, and 246.

This code is used in chunk 228.

Get Edge Data

The next step is to define the *getEdgeData*( ) methods, i.e., functions that return the **BaseFab**⟨**T**⟩s that constitute the **EdgeFab**⟨**T**⟩.

The first form returns a pointer to the corresponding **BaseFab**⟨**T**⟩ associated with a given direction *dir*. It begins, in the style of other Chombo functions, with *assert*s that check if

1. the number of components is positive (if it is not, it means that the **EdgeFab**⟨**T**⟩ is empty),
2. the *dir* value fits between zero and *SpaceDim*, and
3. the *m_edgeData*[*dir*], which is itself a pointer to **BaseFab**⟨**T**⟩, is not Λ (i.e., not `NULL`).

> Using *assert*, which is a function from the C Library, is a somewhat crude method of handling errors. When an expression, that is an argument of *assert*, evaluates to *false*, the function prints a message, which is quite informative, so this is good, and then calls *abort*. In short, *assert* exits are rather unceremonious and cannot be caught. ◀
>
> Chombo used to disable *assert*s unless it was explicitly compiled with the `DEBUG` switch. In its latest version Chombo divorces *assert*s from `DEBUG` and binds them instead to `OPT`.

Having *assert*ed that everything is OK, *getEdgeData* returns the object that *m_edgeData*[*dir*] points to, that is, the corresponding **BaseFab**⟨**T**⟩. The return is *by reference,* which means that no data is copied.

The second form of *getEdgeData* has exactly the same stuff inside, but is declared differently. The **const** after the declaration says that this is an *inspector,* i.e., a function that does not modify the state of an **EdgeFab**⟨**T**⟩. The **const** in front of **BaseFab**⟨**T**⟩ & says, more specifically, that this **BaseFab**⟨**T**⟩ must not change. In this form the *getEdgeData* method may be used, for example, inside the *copy* method.

The third and the fourth forms are the same as calling *getEdgeData* in one of the two forms above, but a [ ] operator is used instead. So, if *a_curl_box* is an **EdgeFab**⟨**T**⟩, we can extract its **BaseFab**⟨**T**⟩ components simply by *a_curl_box*[*dir*] instead of *a_curl_box*.*getEdgeData*(*dir*).

230 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ BaseFab⟨T⟩ &EdgeFab⟨T⟩::getEdgeData(const int dir)
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
```

```
template⟨class T⟩ const BaseFab⟨T⟩ &EdgeFab⟨T⟩::getEdgeData(const int dir) const
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
template⟨class T⟩ BaseFab⟨T⟩ &EdgeFab⟨T⟩::operator[](const int dir)
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
template⟨class T⟩ const BaseFab⟨T⟩ &EdgeFab⟨T⟩::operator[](const int dir) const
{
  assert(m_nComp > 0);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  return *m_edgeData[dir];
}
```

Constructors and Destructors

In this section we discuss constructors and destructors of the class.

The simplest default constructor is **EdgeFab⟨T⟩::EdgeFab( )**. Because it knows nothing of the box or the number of components, all it does is to allocate a **Vector** *m_edgeData*, make it *SpaceDim* entries long and set its entries to Λ. The other internal variable of the class, *m_nComp* is set to −1, which flags to other class methods that the object is not fully defined.

When **EdgeFab⟨T⟩::EdgeFab( )** is called with a box and a number of components, *m_edgeData* is prepared as before, but this time we call *define*(*a_box*, *a_nComp*) to complete the construction.

231 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ EdgeFab⟨T⟩::EdgeFab( )
: m_edgeData(SpaceDim, Λ) {
  m_nComp = −1;
}
template⟨class T⟩ EdgeFab⟨T⟩::EdgeFab(const Box &a_box, int a_nComp)
: m_edgeData(SpaceDim, Λ) {
  define(a_box, a_nComp);
}
```

¶ The destructor of the class is **EdgeFab⟨T⟩::∼EdgeFab( )**. It simply calls *clear*( ), which completes the job.

Function *clear*( ), in turn, deletes all the *m_edgeData*[*dir*] **BaseFab⟨T⟩**s, if they're not Λ, and resets the *m_edgeData*[*dir*] pointers to Λ. *m_nComp* is reset to −1, as it was in the **EdgeFab( )** constructor, which flags to all class methods that this object is now empty. Finally, the box itself, *m_box*, is reset to an empty box, which is what **Box( )** returns when called without arguments.

232 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ EdgeFab⟨T⟩::∼EdgeFab( )
{
  clear( );
}
```

```
template⟨class T⟩ void EdgeFab⟨T⟩::clear()
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    if (m_edgeData[dir] ≠ Λ) {
      delete m_edgeData[dir];
      m_edgeData[dir] = Λ;
    }
  }
  m_nComp = −1;
  m_box = Box();
}
```

¶ **EdgeFab⟨T⟩**::*define*( ) is the actual constructor of the class, behind the wrappers. It takes *a_box* and *a_nComp* as its arguments.

The first thing that happens after a sanity check for *a_nComp* is that *a_box* gets copied on the internal variable *m_box* and *a_nComp* is copied on *m_nComp*. And so, these two variables are now accessible to all other class methods. The currently empty vector of **BaseFab⟨T⟩**s, is resized to *SpaceDim* in preparation for space allocations.

Now we enter a loop over space directions, we copy *m_box* onto *edgeBox* and adjust the *edgeBox* for this particular **BaseFab⟨T⟩**. The first adjustment is to convert the original cell-centered *m_box* to a node box in *all* directions. Function *surroundingNodes* not only changes the type of the box, but also increases *all* components of the upper corner of the box by one. But we don't want the box to be stretched and typed so in *all* directions. We only want it modified in directions that are perpendicular to *dir*. So, the next adjustment, *enclosedCells*(*dir*), returns the box to cell centered in the *dir* direction and reduces its high corner *dir* coordinate by one too, thus delivering us a box exactly as is needed for the edge mounted field in this direction.

For comparison, the **FluxBox** code in this place performs just one operation, which is ◀

Box *edgeBox*(*surroundingNodes*(*m_box*, *dir*));

When converting the **FluxBox** code to the **EdgeFab⟨T⟩** code we need to replace every occurrence of the above with

Box *edgeBox*(*surroundingNodes*(*m_box*)); *edgeBox*.*enclosedCells*(*dir*);

We shall flag in the following every time we do so with the ◀ sign, because this is important and a failure to do so would produce an incorrect code.

Having constructed the right box, we now create a new **BaseFab⟨T⟩** with *m_nComp* components on this box, and place a pointer to it in the *m_edgeData* array.

The method creates and structures space for data, and hooks it into the **EdgeFab⟨T⟩** object, but it does not initialize the data.

233 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::define(const Box &a_box, int a_nComp)
{
  assert(a_nComp > 0);
  m_box = a_box;
  m_nComp = a_nComp;
  m_edgeData.resize(SpaceDim);
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box edgeBox(surroundingNodes(m_box));
    edgeBox.enclosedCells(dir);
    BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(edgeBox, m_nComp);
    m_edgeData[dir] = newFabPtr;
  }
}
```

¶ **EdgeFab**⟨**T**⟩::*resize*( ) is quite similar to *define*. It could be called *redefine* instead. Basically, it lets us substitute a new box in place of the old one, and/or change the number of components on a field that may have already been defined.

But we first check if *m_nComp* is less than zero, which would be the sign that this object hasn't been constructed yet. In this case, we simply refer the matter to *define*.

If the object has been defined, we check that the new *a_nComp* is sound, then transfer the new *a_box* and *a_nComp* to the internal class variables *m_box* and *m_nComp* respectively. Then we enter a loop over directions.

And the first thing we do within this loop is to adjust the cell centered *a_box* for the edge-mounted field, as we did in *define*. ◀

With the box suitably adjusted we now have two possible cases.

1. There is already a **BaseFab**⟨**T**⟩ defined for this direction. In this case we simply ask it to **BaseFab**⟨**T**⟩::*resize*( ) itself, and give it the new *edgeBox* and the new *m_nComp*.

2. There is no **BaseFab**⟨**T**⟩ defined for this direction, in which case we define it and place the pointer to it in the *m_edgeData* vector.

> If an **EdgeFab** had been cleared, *m_nComp* would be reset to −1, so in this case *define* would be called. This situation assumes that *m_nComp* > 0, but, for some reason, the **BaseFab**⟨**T**⟩ alone has been cleared, e.g., by a programmer, and reset to Λ.

In case (1) the **BaseFab**⟨**T**⟩::*resize*( ) always results in the memory allocated for this **BaseFab** to *grow* and never shrink. The Chombo documentation also recommends that this method be used for managing temporary space, in situations when calling for a new allocation would be too costly.

234 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::resize(const Box &a_box, int a_nComp)
{
  if (m_nComp < 0) {
    define(a_box, a_nComp);
  }
  else {
    assert(a_nComp > 0);
    m_box = a_box;
    m_nComp = a_nComp;
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box edgeBox(surroundingNodes(m_box));
      edgeBox.enclosedCells(dir);
      if (m_edgeData[dir] ≠ Λ) {
        m_edgeData[dir]→resize(edgeBox, m_nComp);
      }
      else {
        BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(edgeBox, m_nComp);
        m_edgeData[dir] = newFabPtr;
      }
    }
  }
}
```

Modifiers

Now we have two basic modifiers of the class. The first one, *setVal*, just sets the whole **EdgeFab** to some value. This can be trimmed a little, to set only a specific range of components or only the fields associated with a specific direction or both.

The first three forms are pretty trivial

setVal(**const T** *val*) Set all components of all **EdgeFab**⟨**T**⟩ fields to *val*.

setVal(**const T** *val*, **int** *dir*) Set all components of **EdgeFab**⟨**T**⟩ fields associated with direction *dir* to *val*.

setVal(**const T** *val*, **const int** *dir*, **const int** *startComp*, **const int** *nComp*) Set the range of components of **EdgeFab**⟨**T**⟩ fields associated with direction *dir* to *val*.

They all work by first testing the sanity of parameters passed with *assert*s, and then by calling the **BaseFab**⟨**T**⟩::*setVal*( ) method to do the job.

235 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val)
{
  assert(m_nComp > 0);
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]→setVal(val);
  }
}
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, int dir)
{
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  m_edgeData[dir]→setVal(val);
}
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, const int dir, const int startComp, const
        int nComp)
{
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  for (int comp = startComp; comp < startComp + nComp; comp++) {
    m_edgeData[dir]→setVal(val, comp);
  }
}
```

¶ Now we get into a somewhat trickier version of *setVal*, because this time we set it on a **Box** *box*. For this to work the **Box** in question must be a sub-box of *m_box*. We could relax this condition and simply set the **EdgeFab** to *val* on an overlap of the *box* and *m_box*, if it is not empty, but Chombo semantics do not go this far. So the programmer has to check for this explicitly before calling *setVal*, as we did in module ⟨ Function *draw_box* 115 ⟩ of Section 8.1. We check whether *box* ⊂ *m_box* with an *assert*.

Then we get into a loop over directions. For each direction we make sure that there is a corresponding **BaseFab**⟨**T**⟩ defined, and then we modify the **Box** passed to make it into an edge-**Box** that is shaped specifically for this direction. Once the *box* has been tweaked, we call the **BaseFab**⟨**T**⟩::*setVal*( ) method to carry out the operation. ◀

In the second form of *setVal*( ) the programmer may specify the direction and the range of components additionally. In this case we additionally check if the direction provided and the components range are within the bounds.

236 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, const Box &box)
{
  assert(m_box.contains(box));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box edgeBox(surroundingNodes(box));
```

```
    edgeBox.enclosedCells(dir);
    m_edgeData[dir]→setVal(val, edgeBox, 0, m_nComp);
  }
}
template⟨class T⟩ void EdgeFab⟨T⟩::setVal(const T val, const Box &box, const int dir, const int
        startComp, const int nComp)
{
  assert(m_box.contains(box));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
  Box edgeBox(surroundingNodes(box));
  edgeBox.enclosedCells(dir);
  m_edgeData[dir]→setVal(val, edgeBox, startComp, nComp);
}
```

Copy

EdgeFab⟨T⟩::*copy*( ) is an important group of modifiers that deserves its own subsection. As is the case with *setVal*( ), here we also defer the actual action to **BaseFab⟨T⟩::***copy*( ), but tweaking input, and checking with *assert*s still has to be done before **BaseFab⟨T⟩::***copy*( ) can be invoked.

The first version of **EdgeFab⟨T⟩::***copy*( ) copies data from one **EdgeFab⟨T⟩** to another one. Both have to have the same number of components. The **BaseFab⟨T⟩::***copy*( ) invoked here performs this operation on the *intersection* of the domains of both **EdgeFab⟨T⟩**s.

> The original **FluxBox** code had an *assert* here that checked if the source and *this* **EdgeFab** domains were identical, meaning of the same size, location and type. This was too strong an assert and it would fail unnecessarily making it impossible to copy between **EdgeFab⟨T⟩**s that are defined on overlapping domains, so I deleted it. I also extended the remaining *assert* to include a check for both **EdgeFab⟨T⟩**s being fully defined.

237 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const EdgeFab⟨T⟩ &src)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0) ∧ (src.nComp( ) ≡ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]→copy(src[dir]);
  }
}
```

¶ This second form of **EdgeFab⟨T⟩::***copy*( ) copies data from a range of source components into a range of destination components on *this* **EdgeFab**, and on the overlap of the two **EdgeFab⟨T⟩** domains, using **BaseFab⟨T⟩::***copy*( ).

> I have modifed the *assert* checks here, because the original ones ◀
>
> *assert*(*srcComp* ∗ *destComp* > −1); *assert*(*srcComp* + *numComp* ≤ *src.nComp*( ));
> *assert*(*destComp* + *numComp* ≤ *m_nvar*);
>
> inherited from **FluxBox** didn't do enough checking. The first check here was strange, because it would not detect an error if *srcComp* and *destComp* were both negative.
>
> The new *assert*s check if both **EdgeFab⟨T⟩**s are fully defined, then check if both *srcComp* and *destComp* are positive and finally check if we'll stay within the component range within both **EdgeFab⟨T⟩**s.

Having run our sanity checks, we loop over the directions. For each direction we check that the destination **BaseFab**⟨**T**⟩ exists and invoke **BaseFab**⟨**T**⟩::*copy*( ) on *src*[*dir*].

238 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const EdgeFab⟨T⟩ &src, const int srcComp, const int
        destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    m_edgeData[dir]→copy(srcFab, srcComp, destComp, numComp);
  }
}
```

¶ In this version of *copy*( ) we let a programmer additionally specify the direction. Otherwise it is like the version above. I have carried the same *assert* checks as above and added a check for the *dir* as well.

239 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const EdgeFab⟨T⟩ &src, const int dir, const int
        srcComp, const int destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  const BaseFab⟨T⟩ &srcFab = src[dir];
  m_edgeData[dir]→copy(srcFab, srcComp, destComp, numComp);
}
```

¶ This version of *copy*( ) lets a programmer copy data between two component intervals and two subregions of the **EdgeFab**s' domains. The copy operation is carried out using **BaseFab**⟨**T**⟩::*copy*( ) and for all directions. The order of arguments is the same as in a similar **BaseFab**⟨**T**⟩::*copy*( ) method.

The source and destination boxes are resized for edge mounted data. There is also a change in the semantics of this code. The original **FluxBox** code had destination and source intervals swapped around both in the argument list and in the call to **BaseFab**⟨**T**⟩::*copy*( ), which was a bug–currently fixed in the latest release of Chombo. This code adheres to the original **BaseFab**⟨**T**⟩::*copy*( ) order, as stated above. There are no new *assert*s since **BaseFab**⟨**T**⟩::*copy*( ) does its own checks here. ◀

The checks carried out by **BaseFab**⟨**T**⟩::*copy*( ) are several and important to fully understand the semantics of this operation:

1. If *srcFab* is the same as **this** and both boxes are the same, the operation returns without doing anything, because nothing needs to be done.
2. The size of both **Interval**s must be the same, and they must both fit in the source and destination component range respectively.
3. *RedgeFrom* and *RedgeTo* must be of the same size.
4. *RedgeFrom* must be contained in the source domain, and *RedgeTo* must be contained in the destination domain.

240 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const Box &RegionFrom, const Interval &Cdest, const
        Box &RegionTo, const EdgeFab⟨T⟩ &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box RedgeFrom(surroundingNodes(RegionFrom));
    RedgeFrom.enclosedCells(dir);
    Box RedgeTo(surroundingNodes(RegionTo));
    RedgeTo.enclosedCells(dir);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    RedgeTo &= (m_edgeData[dir]↦box( ));
    m_edgeData[dir]↦copy(RedgeFrom, Cdest, RedgeTo, srcFab, Csrc);
  }
}
```

¶ This is another variant of a *copy*( ) method that copies data within a specified region, which must be a shared sub-box of both *src.box*( ) and *m_box*. The copy operation is carried out using **BaseFab**⟨**T**⟩::*copy*( ) for a range of components specified by **Interval**s.

Within the loop over all directions we check that the fields of *this* **EdgeFab**⟨**T**⟩ are fully defined, then grab the *Region* box and stretch it for the edge-mounted field in this direction. ◀

Having done so, we take its intersection with the source field box and then with the destination field box. If there is a non-empty overlap of the three, we perform the **BaseFab**⟨**T**⟩::*copy*( ) operation on this overlap.

241 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::copy(const Box &Region, const Interval &Cdest, const
        EdgeFab⟨T⟩ &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box edgeRegion(surroundingNodes(Region));
    edgeRegion.enclosedCells(dir);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    edgeRegion &= srcFab.box( );
    edgeRegion &= m_edgeData[dir]↦box( );
    if (¬edgeRegion.isEmpty( )) {
      m_edgeData[dir]↦copy(edgeRegion, Cdest, edgeRegion, srcFab, Csrc);
    }
  }
}
```

Shift

This function just shifts *m_box* and the boxes of the **BaseFab**⟨**T**⟩ fields by an **IntVect** *iv*. Although it does not touch the data arrays, it modifies the domain of the **EdgeFab**.

242 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ EdgeFab⟨T⟩ &EdgeFab⟨T⟩::shift(const IntVect &iv)
{
  m_box.shift(iv);
  for (int dir = 0; dir < SpaceDim; dir++) {
    m_edgeData[dir]↦shift(iv);
```

```
  }
  return *this;
}
```

Transfer Helpers

These methods are used in transferring **EdgeFab**⟨**T**⟩ data between MPI processes. The first one, *size*, calculates the size of the buffer that will be needed for the transfer. Chombo functions use it to allocate the buffer space. Then the buffer is filled by calling *linearOut*, the data is transferred to other processes, which call *linearIn* to unpack it and put it back into their local **EdgeFab**⟨**T**⟩s.

As with everything else here, these methods are basically wrappers around **BaseFab**⟨**T**⟩ functions with identical names.

Size

This function calls **BaseFab**⟨**T**⟩::*size*(**Box** &*box*, **Interval** *comps*), which returns the size of the linear representation of the **BaseFab**⟨**T**⟩ data for all components in *comps* and built on this *box*. The size does *not* include the size of the box itself or of the components structure.

The function creates a temporary and undefined **BaseFab**⟨**T**⟩, just so as to provide data typing to **BaseFab**⟨**T**⟩::*size*( ), then enters the loop over directions. For each direction we stretch the original *box* for the edge data placement, as was first done in *define*( ), then call **BaseFab**⟨**T**⟩::*size*( ) on this stretched box and with the components in the **Interval** specified. The returns are then accumulated on *totalSize*, which is returned when the function completes. ◀

244 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ int EdgeFab⟨T⟩::size(const Box &box, const Interval &comps) const
{
  int totalSize = 0;
  BaseFab⟨T⟩ tempFab;
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box edgeBox(surroundingNodes(box));
    edgeBox.enclosedCells(dir);
    const int dirSize = tempFab.size(edgeBox, comps);
    totalSize += dirSize;
  }
  return totalSize;
}
```

linearOut

This function writes out onto buffer *buf* the content of all components within the **Interval** *comps* of *all three* fields that constitute an **EdgeFab**⟨**T**⟩ for all cells contained by the *box*. The *box* in question does not have to be the same as *m_box*. Data transfer functions will usually build their own boxes around data that needs to be transferred into the ghost regions of adjacent boxes.

We begin our peregrinations by casting *buf* onto a pointer to **T**. This then becomes a pointer to the **T** buffer, called *buffer*. As we push each **BaseFab**⟨**T**⟩ onto the buffer in turn we are going to advance the pointer *buffer*, to prepare the buffer for the next write.

And so, we enter the loop over directions and, the first thing we check is if there is anything in the data portion of the **EdgeFab**⟨**T**⟩ with an *assert*. Then we take the input **Box** *box* and we stretch it into an edge mounted box for this direction. This new stretched box is called *dirBox*. ◀

Now we are ready to use **BaseFab**⟨**T**⟩::*linearOut*( ) method to just write the data for this **BaseFab**⟨**T**⟩ and this *dirBox*. But before we do this, we find how much data is going to be written, in bytes, and store this number on *dirSize*. We do this, because immediately after the write we are going to advance the **T** **buffer* pointer by *dirSize*/**sizeof**(**T**).

245 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::linearOut(void *buf, const Box &box, const Interval &comps)
        const
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearOut(buffer, dirBox, comps);
    buffer += dirSize/sizeof(T);
  }
}
```

linearIn

And here we do exactly as above, but we replace *linearOut* with *linearIn*. All other comments made in the *linearOut* section pertain to this function as well.

246 ⟨ **EdgeFab** template implementation 229 ⟩ +≡

```
template⟨class T⟩ void EdgeFab⟨T⟩::linearIn(void *buf, const Box &box, const Interval &comps)
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearIn(buffer, dirBox, comps);
    buffer += dirSize/sizeof(T);
  }
}
```

12.3 FaceFab⟨T⟩ Class Template

The **FaceFab**⟨**T**⟩ template is even more like **FluxBox**. It is **FluxBox** made into a template, so that we can mount fields of various types on its faces.

The source listed below was produced by running automatic renames on `"EdgeFab.H"` and `"EdgeFabImplem.H"` files, and replacing constructs such as

Box *dirBox*(*surroundingNodes*(*box*)); *dirBox*.*enclosedCells*(*dir*);

with

Box *dirBox*(*surroundingNodes*(*box*, *dir*));

This means that this code inherits all little mods (mostly *assert* constructions) from **EdgeFab**⟨**T**⟩.

Otherwise it is presented here without additional comments.

12.3.1 FaceFab⟨T⟩ Headers

248 ⟨ `FaceFab.hw` 248 ⟩ ≡

```
#ifndef CH_FACEFAB_H
# define CH_FACEFAB_H
#include <Box.H>
#include <Vector.H>
#include <BaseFab.H>
#include <REAL.H>
  template⟨class T⟩ class FaceFab {
  public: FaceFab( );
    FaceFab(const Box &box, int nComp = 1);
    ∼FaceFab( );
    void resize(const Box &box, int nComp = 1);
    void define(const Box &box, int nComp = 1);
    void clear( );
    int nComp( ) const;
    const Box &box( ) const;
    BaseFab⟨T⟩ &getFaceData(const int dir);
    const BaseFab⟨T⟩ &getFaceData(const int dir) const;
    BaseFab⟨T⟩ &operator[](const int dir);
    const BaseFab⟨T⟩ &operator[](const int dir) const;
    void setVal(const T val);
    void setVal(const T val, const int dir);
    void setVal(const T val, const int dir, const int startComp, const int nComp);
    void setVal(const T val, const Box &box);
    void setVal(const T val, const Box &box, const int dir, const int startComp, const int nComp);
    void copy(const FaceFab⟨T⟩ &src);
    void copy(const FaceFab⟨T⟩ &src, const int srcComp, const int destComp, const int numComp);
    void copy(const FaceFab⟨T⟩ &src, const int dir, const int srcComp, const int destComp, const int
        numComp);
    void copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const FaceFab⟨T⟩
        &src, const Interval &Csrc);
    void copy(const Box &Region, const Interval &Cdest, const FaceFab⟨T⟩ &src, const Interval
        &Csrc);
    FaceFab⟨T⟩ &shift(const IntVect &v);
    int size(const Box &box, const Interval &comps) const;
    void linearOut(void *buf, const Box &R, const Interval &comps) const;
    void linearIn(void *buf, const Box &R, const Interval &comps);
```

```
    static int preAllocatable( )
    {
      return 0;
    }
  protected: Box m_box;
    int m_nComp;
    Vector⟨BaseFab⟨T⟩ *⟩ m_faceData;
  private: FaceFab(const FaceFab⟨T⟩ &);
    FaceFab⟨T⟩ &operator=(const FaceFab⟨T⟩ &);
  };
#include <FaceFabImplem.H>
#endif
```

12.3.2 FaceFab⟨T⟩ Implementation

Extractors

250 ⟨ FaceFabImplem.hw 250 ⟩ ≡

```
#ifndef CH_FACEFABIMPLEM_H
# define CH_FACEFABIMPLEM_H
  template⟨class T⟩ int FaceFab⟨T⟩::nComp( ) const
  {
    return m_nComp;
  }
  template⟨class T⟩ const Box &FaceFab⟨T⟩::box( ) const
  {
    return m_box;
  }
  template⟨class T⟩ BaseFab⟨T⟩ &FaceFab⟨T⟩::getFaceData(const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
  template⟨class T⟩ const BaseFab⟨T⟩ &FaceFab⟨T⟩::getFaceData(const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
  template⟨class T⟩ BaseFab⟨T⟩ &FaceFab⟨T⟩::operator[](const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
  template⟨class T⟩ const BaseFab⟨T⟩ &FaceFab⟨T⟩::operator[](const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
```

```
    assert(m_faceData[dir] ≠ Λ);
    return *m_faceData[dir];
  }
```
See also chunks 251, 252, and 253.

Constructors and Destructors

251 ⟨ FaceFabImplem.hw 250 ⟩ +≡
```
  template⟨class T⟩ FaceFab⟨T⟩::FaceFab( )
  : m_faceData(SpaceDim, Λ) {
    m_nComp = −1;
  }
  template⟨class T⟩ FaceFab⟨T⟩::FaceFab(const Box &a_box, int a_nComp)
  : m_faceData(SpaceDim, Λ) {
    define(a_box, a_nComp);
  }
  template⟨class T⟩ FaceFab⟨T⟩::~FaceFab( )
  {
    clear( );
  }
  template⟨class T⟩ void FaceFab⟨T⟩::clear( )
  {
    for (int dir = 0; dir < SpaceDim; dir++) {
      if (m_faceData[dir] ≠ Λ) {
        delete m_faceData[dir];
        m_faceData[dir] = Λ;
      }
    }
    m_nComp = −1;
    m_box = Box( );
  }
  template⟨class T⟩ void FaceFab⟨T⟩::define(const Box &a_box, int a_nComp)
  {
    assert(a_nComp > 0);
    m_box = a_box;
    m_nComp = a_nComp;
    m_faceData.resize(SpaceDim);
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box faceBox(surroundingNodes(m_box, dir));
      BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(faceBox, m_nComp);
      m_faceData[dir] = newFabPtr;
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::resize(const Box &a_box, int a_nComp)
  {
    if (m_nComp < 0) {
      define(a_box, a_nComp);
    }
    else {
      assert(a_nComp > 0);
      m_box = a_box;
      m_nComp = a_nComp;
      for (int dir = 0; dir < SpaceDim; dir++) {
        Box faceBox(surroundingNodes(m_box, dir));
```

```
        if (m_faceData[dir] ≠ Λ) {
          m_faceData[dir]→resize(faceBox, m_nComp);
        }
        else {
          BaseFab⟨T⟩ *newFabPtr = new BaseFab⟨T⟩(faceBox, m_nComp);
          m_faceData[dir] = newFabPtr;
        }
      }
    }
  }
```

Modifiers

252 ⟨ FaceFabImplem.hw 250 ⟩ +≡

```
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val)
  {
    assert(m_nComp > 0);
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_faceData[dir] ≠ Λ);
      m_faceData[dir]→setVal(val);
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, int dir)
  {
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    m_faceData[dir]→setVal(val);
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, const int dir, const int startComp, const
        int nComp)
  {
    assert(startComp > −1);
    assert(startComp + nComp ≤ m_nComp);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_faceData[dir] ≠ Λ);
    for (int comp = startComp; comp < startComp + nComp; comp++) {
      m_faceData[dir]→setVal(val, comp);
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, const Box &box)
  {
    assert(m_box.contains(box));
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_faceData[dir] ≠ Λ);
      Box faceBox(surroundingNodes(box, dir));
      m_faceData[dir]→setVal(val, faceBox, 0, m_nComp);
    }
  }
  template⟨class T⟩ void FaceFab⟨T⟩::setVal(const T val, const Box &box, const int dir, const int
        startComp, const int nComp)
  {
    assert(m_box.contains(box));
```

```
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_faceData[dir] ≠ Λ);
  assert(startComp > −1);
  assert(startComp + nComp ≤ m_nComp);
  Box faceBox(surroundingNodes(box, dir));
  m_faceData[dir]→setVal(val, faceBox, startComp, nComp);
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const FaceFab⟨T⟩ &src)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0) ∧ (src.nComp( ) ≡ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    m_faceData[dir]→copy(src[dir]);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const FaceFab⟨T⟩ &src, const int srcComp, const int
        destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    const BaseFab⟨T⟩ &srcFab = src[dir];
    m_faceData[dir]→copy(srcFab, srcComp, destComp, numComp);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const FaceFab⟨T⟩ &src, const int dir, const int
        srcComp, const int destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_faceData[dir] ≠ Λ);
  const BaseFab⟨T⟩ &srcFab = src[dir];
  m_faceData[dir]→copy(srcFab, srcComp, destComp, numComp);
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const Box &RegionFrom, const Interval &Cdest, const
        Box &RegionTo, const FaceFab⟨T⟩ &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    Box RfaceFrom(surroundingNodes(RegionFrom, dir));
    Box RfaceTo(surroundingNodes(RegionTo, dir));
    const BaseFab⟨T⟩ &srcFab = src[dir];
    RfaceTo &= (m_faceData[dir]→box( ));
    m_faceData[dir]→copy(RfaceFrom, Cdest, RfaceTo, srcFab, Csrc);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::copy(const Box &Region, const Interval &Cdest, const
        FaceFab⟨T⟩ &src, const Interval &Csrc)
```

```
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    Box faceRegion(surroundingNodes(Region, dir));
    const BaseFab⟨T⟩ &srcFab = src[dir];
    faceRegion &= srcFab.box( );
    faceRegion &= m_faceData[dir]→box( );
    if (¬faceRegion.isEmpty( )) {
      m_faceData[dir]→copy(faceRegion, Cdest, faceRegion, srcFab, Csrc);
    }
  }
}
template⟨class T⟩ FaceFab⟨T⟩ &FaceFab⟨T⟩::shift(const IntVect &iv)
{
  m_box.shift(iv);
  for (int dir = 0; dir < SpaceDim; dir++) {
    m_faceData[dir]→shift(iv);
  }
  return *this;
}
```

Transfer Helpers

253 ⟨ FaceFabImplem.hw 250 ⟩ +≡

```
template⟨class T⟩ int FaceFab⟨T⟩::size(const Box &box, const Interval &comps) const
{
  int totalSize = 0;
  BaseFab⟨T⟩ tempFab;
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box faceBox(surroundingNodes(box, dir));
    const int dirSize = tempFab.size(faceBox, comps);
    totalSize += dirSize;
  }
  return totalSize;
}
template⟨class T⟩ void FaceFab⟨T⟩::linearOut(void *buf, const Box &box, const Interval &comps)
        const
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box, dir));
    int dirSize = m_faceData[dir]→size(dirBox, comps);
    m_faceData[dir]→linearOut(buffer, dirBox, comps);
    buffer += dirSize/sizeof(T);
  }
}
template⟨class T⟩ void FaceFab⟨T⟩::linearIn(void *buf, const Box &box, const Interval &comps)
{
  T *buffer = (T *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_faceData[dir] ≠ Λ);
```

```
        Box dirBox(surroundingNodes(box, dir));
        int dirSize = m_faceData[dir]->size(dirBox, comps);
        m_faceData[dir]->linearIn(buffer, dirBox, comps);
        buffer += dirSize/sizeof(T);
      }
    }
#endif
```

12.4 CurlBox Class

The **CurlBox** class is basically the **EdgeFab**⟨**T**⟩ template specified for **T** = **Real**. But we do this on the **BaseFab**⟨**T**⟩ level, replacing it with **FArrayBox**, which gives us **FArrayBox** arithmetic and direct access to Chombo HDF5/IO routines.

12.4.1 CurlBox Class Headers

255 ⟨ CurlBox.hw 255 ⟩ ≡

```
#ifndef CH_CURLBOX_H
# define CH_CURLBOX_H
#include <Box.H>
#include <Vector.H>
#include <FArrayBox.H>
#include <REAL.H>
  class CurlBox {
  public: CurlBox( );
    CurlBox(const Box &box, int nComp = 1);
    ~CurlBox( );
    void resize(const Box &box, int nComp = 1);
    void define(const Box &box, int nComp = 1);
    void clear( );
    int nComp( ) const;
    const Box &box( ) const;
    FArrayBox &getEdgeData(const int dir);
    const FArrayBox &getEdgeData(const int dir) const;
    FArrayBox &operator[](const int dir);
    const FArrayBox &operator[](const int dir) const;
    void setVal(const Real val);
    void setVal(const Real val, const int dir);
    void setVal(const Real val, const int dir, const int startComp, const int nComp);
    void setVal(const Real val, const Box &box);
    void setVal(const Real val, const Box &box, const int dir, const int startComp, const int nComp);
    void copy(const CurlBox &src);
    void copy(const CurlBox &src, const int srcComp, const int destComp, const int numComp);
    void copy(const CurlBox &src, const int dir, const int srcComp, const int destComp, const int
        numComp);
    void copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const CurlBox
        &src, const Interval &Csrc);
    void copy(const Box &Region, const Interval &Cdest, const CurlBox &src, const Interval &Csrc);
    CurlBox &shift(const IntVect &v);
    int size(const Box &box, const Interval &comps) const;
    void linearOut(void *buf, const Box &R, const Interval &comps) const;
    void linearIn(void *buf, const Box &R, const Interval &comps);
    static int preAllocatable( )
    {
      return 0;
    }
  protected: Box m_box;
    int m_nComp;
    Vector⟨FArrayBox *⟩ m_edgeData;
  private: CurlBox(const CurlBox &);
```

```
    CurlBox &operator=(const CurlBox &);
  };
#endif
```

12.4.2 CurlBox Class Implementation

Extractors

257 ⟨ CurlBox.c 257 ⟩ ≡

```
#include <CurlBox.H>
  int CurlBox::nComp( ) const
  {
    return m_nComp;
  }
  const Box &CurlBox::box( ) const
  {
    return m_box;
  }
  FArrayBox &CurlBox::getEdgeData(const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
  const FArrayBox &CurlBox::getEdgeData(const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
  FArrayBox &CurlBox::operator[](const int dir)
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
  const FArrayBox &CurlBox::operator[](const int dir) const
  {
    assert(m_nComp > 0);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    return *m_edgeData[dir];
  }
```

See also chunks 258, 259, and 260.

Constructors and Destructors

258 ⟨ CurlBox.c 257 ⟩ +≡

```
CurlBox::CurlBox( )
: m_edgeData(SpaceDim, Λ) {
  m_nComp = −1;
}
CurlBox::CurlBox(const Box &a_box, int a_nComp)
: m_edgeData(SpaceDim, Λ) {
  define(a_box, a_nComp);
}
CurlBox::~CurlBox( )
{
  clear( );
}
void CurlBox::clear( )
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    if (m_edgeData[dir] ≠ Λ) {
      delete m_edgeData[dir];
      m_edgeData[dir] = Λ;
    }
  }
  m_nComp = −1;
  m_box = Box( );
}
void CurlBox::define(const Box &a_box, int a_nComp)
{
  assert(a_nComp > 0);
  m_box = a_box;
  m_nComp = a_nComp;
  m_edgeData.resize(SpaceDim);
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box edgeBox(surroundingNodes(m_box));
    edgeBox.enclosedCells(dir);
    FArrayBox *newFabPtr = new FArrayBox(edgeBox, m_nComp);
    m_edgeData[dir] = newFabPtr;
  }
}
void CurlBox::resize(const Box &a_box, int a_nComp)
{
  if (m_nComp < 0) {
    define(a_box, a_nComp);
  }
  else {
    assert(a_nComp > 0);
    m_box = a_box;
    m_nComp = a_nComp;
    for (int dir = 0; dir < SpaceDim; dir++) {
      Box edgeBox(surroundingNodes(m_box));
      edgeBox.enclosedCells(dir);
      if (m_edgeData[dir] ≠ Λ) {
        m_edgeData[dir]→resize(edgeBox, m_nComp);
      }
      else {
```

```
            FArrayBox *newFabPtr = new FArrayBox(edgeBox, m_nComp);
            m_edgeData[dir] = newFabPtr;
          }
        }
      }
    }
```

Modifiers

259 ⟨ CurlBox.c 257 ⟩ +≡

```
  void CurlBox::setVal(const Real val)
  {
    assert(m_nComp > 0);
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_edgeData[dir] ≠ Λ);
      m_edgeData[dir]→setVal(val);
    }
  }
  void CurlBox::setVal(const Real val, int dir)
  {
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]→setVal(val);
  }
  void CurlBox::setVal(const Real val, const int dir, const int startComp, const int nComp)
  {
    assert(startComp > −1);
    assert(startComp + nComp ≤ m_nComp);
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    for (int comp = startComp; comp < startComp + nComp; comp++) {
      m_edgeData[dir]→setVal(val, comp);
    }
  }
  void CurlBox::setVal(const Real val, const Box &box)
  {
    assert(m_box.contains(box));
    for (int dir = 0; dir < SpaceDim; dir++) {
      assert(m_edgeData[dir] ≠ Λ);
      Box edgeBox(surroundingNodes(box));
      edgeBox.enclosedCells(dir);
      m_edgeData[dir]→setVal(val, edgeBox, 0, m_nComp);
    }
  }
  void CurlBox::setVal(const Real val, const Box &box, const int dir, const int startComp, const int
      nComp)
  {
    assert(m_box.contains(box));
    assert((0 ≤ dir) ∧ (dir < SpaceDim));
    assert(m_edgeData[dir] ≠ Λ);
    assert(startComp > −1);
    assert(startComp + nComp ≤ m_nComp);
```

```
  Box edgeBox(surroundingNodes(box));
  edgeBox.enclosedCells(dir);
  m_edgeData[dir]->setVal(val, edgeBox, startComp, nComp);
}
void CurlBox::copy(const CurlBox &src)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0) ∧ (src.nComp( ) ≡ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    m_edgeData[dir]->copy(src[dir]);
  }
}
void CurlBox::copy(const CurlBox &src, const int srcComp, const int destComp, const int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    const FArrayBox &srcFab = src[dir];
    m_edgeData[dir]->copy(srcFab, srcComp, destComp, numComp);
  }
}
void CurlBox::copy(const CurlBox &src, const int dir, const int srcComp, const int destComp, const
        int numComp)
{
  assert((src.nComp( ) > 0) ∧ (m_nComp > 0));
  assert((srcComp ≥ 0) ∧ (destComp ≥ 0));
  assert((srcComp + numComp ≤ src.nComp( )) ∧ (destComp + numComp ≤ m_nComp));
  assert((0 ≤ dir) ∧ (dir < SpaceDim));
  assert(m_edgeData[dir] ≠ Λ);
  const FArrayBox &srcFab = src[dir];
  m_edgeData[dir]->copy(srcFab, srcComp, destComp, numComp);
}
void CurlBox::copy(const Box &RegionFrom, const Interval &Cdest, const Box &RegionTo, const
        CurlBox &src, const Interval &Csrc)
{
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box RedgeFrom(surroundingNodes(RegionFrom));
    RedgeFrom.enclosedCells(dir);
    Box RedgeTo(surroundingNodes(RegionTo));
    RedgeTo.enclosedCells(dir);
    const FArrayBox &srcFab = src[dir];
    RedgeTo &= (m_edgeData[dir]->box( ));
    m_edgeData[dir]->copy(RedgeFrom, Cdest, RedgeTo, srcFab, Csrc);
  }
}
void CurlBox::copy(const Box &Region, const Interval &Cdest, const CurlBox &src, const Interval
        &Csrc)
{
```

```
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box edgeRegion(surroundingNodes(Region));
    edgeRegion.enclosedCells(dir);
    const FArrayBox &srcFab = src[dir];
    edgeRegion &= srcFab.box( );
    edgeRegion &= m_edgeData[dir]→box( );
    if (¬edgeRegion.isEmpty( )) {
      m_edgeData[dir]→copy(edgeRegion, Cdest, edgeRegion, srcFab, Csrc);
    }
  }
}
CurlBox &CurlBox::shift(const IntVect &iv)
{
  m_box.shift(iv);
  for (int dir = 0; dir < SpaceDim; dir++) {
    m_edgeData[dir]→shift(iv);
  }
  return *this;
}
```

Transfer Helpers

260 ⟨ CurlBox.c 257 ⟩ +≡

```
int CurlBox::size(const Box &box, const Interval &comps) const
{
  int totalSize = 0;
  FArrayBox tempFab;
  for (int dir = 0; dir < SpaceDim; dir++) {
    Box edgeBox(surroundingNodes(box));
    edgeBox.enclosedCells(dir);
    const int dirSize = tempFab.size(edgeBox, comps);
    totalSize += dirSize;
  }
  return totalSize;
}
void CurlBox::linearOut(void *buf, const Box &box, const Interval &comps) const
{
  Real *buffer = (Real *) buf;
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearOut(buffer, dirBox, comps);
    buffer += dirSize/sizeof(Real);
  }
}
void CurlBox::linearIn(void *buf, const Box &box, const Interval &comps)
{
  Real *buffer = (Real *) buf;
```

```
  for (int dir = 0; dir < SpaceDim; dir++) {
    assert(m_edgeData[dir] ≠ Λ);
    Box dirBox(surroundingNodes(box));
    dirBox.enclosedCells(dir);
    int dirSize = m_edgeData[dir]→size(dirBox, comps);
    m_edgeData[dir]→linearIn(buffer, dirBox, comps);
    buffer += dirSize/sizeof(Real);
  }
}
```

13 Fortran Subroutines

The Fortran subroutines presented in this section come with little comments, as they are not part of the Web system. Instead they have been included by the `\lstinputlisting` command of the *listings* package.

13.1 Subroutines `update_d` and `update_d_upml`

Update the $\boldsymbol{D}$ field by applying Maxwell equations. Both UPML and normal versions are provided on this file.

```
C Fortran subroutines update_d and update_d_upml
C Update the D field by taking curl H - corrected for UPML
C if needed.
C
C $Id: update_d.f,v 1.8 2007/08/31 16:00:53 gustav Exp $
C
      subroutine update_d(
     & SpaceDim, margin, D_index, H_index, dt_by_dg,
     & imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx, Dx,
     & imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy, Dy,
     & imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz, Dz,
     & imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx, Hx,
     & imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy, Hy,
     & imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz, Hz,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, D_index(0:1), H_index(0:1),
     & fortran_verbose, return_status
C
      double precision dt_by_dg
C
      integer imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx
      double precision
     & Dx(imin_Dx:imax_Dx, jmin_Dx:jmax_Dx, kmin_Dx:kmax_Dx, 0:1)
C
      integer imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy
      double precision
     & Dy(imin_Dy:imax_Dy, jmin_Dy:jmax_Dy, kmin_Dy:kmax_Dy, 0:1)
C
      integer imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz
      double precision
     & Dz(imin_Dz:imax_Dz, jmin_Dz:jmax_Dz, kmin_Dz:kmax_Dz, 0:1)
C
      integer imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx
      double precision
     & Hx(imin_Hx:imax_Hx, jmin_Hx:jmax_Hx, kmin_Hx:kmax_Hx, 0:1)
C
      integer imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy
      double precision
     & Hy(imin_Hy:imax_Hy, jmin_Hy:jmax_Hy, kmin_Hy:kmax_Hy, 0:1)
C
      integer imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz
      double precision
     & Hz(imin_Hz:imax_Hz, jmin_Hz:jmax_Hz, kmin_Hz:kmax_Hz, 0:1)
C
      integer i, j, k, new_D, old_D, new_H
      integer ERR_ADVANCE_E
      parameter (ERR_ADVANCE_E = 5)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_d_): bad SpaceDim'
         return_status = ERR_ADVANCE_E
         goto 100
```

```
      end if

      new_D = D_index(0)
      old_D = D_index(1)
      new_H = H_index(0)

      do k = kmin_Dx + margin, kmax_Dx - margin
         do j = jmin_Dx + margin, jmax_Dx - margin
            do i = imin_Dx + margin, imax_Dx - margin
               Dx(i, j, k, new_D) = Dx(i, j, k, old_D) + dt_by_dg *
     & (Hz(i, j, k, new_H) - Hz(i, j - 1, k, new_H)
     & - Hy(i, j, k, new_H) + Hy(i, j, k - 1, new_H))
            end do
         end do
      end do

      do k = kmin_Dy + margin, kmax_Dy - margin
         do j = jmin_Dy + margin, jmax_Dy - margin
            do i = imin_Dy + margin, imax_Dy - margin
              Dy(i, j, k, new_D) = Dy(i, j, k, old_D) + dt_by_dg *
     & (Hx(i, j, k, new_H) - Hx(i, j, k - 1, new_H)
     & - Hz(i, j, k, new_H) + Hz(i - 1, j, k, new_H))
            end do
         end do
      end do

      do k = kmin_Dz + margin, kmax_Dz - margin
         do j = jmin_Dz + margin, jmax_Dz - margin
            do i = imin_Dz + margin, imax_Dz - margin
               Dz(i, j, k, new_D) = Dz(i, j, k, old_D) + dt_by_dg *
     & (Hy(i, j, k, new_H) - Hy(i - 1, j, k, new_H)
     & - Hx(i, j, k, new_H) + Hx(i, j - 1, k, new_H))
            end do
         end do
      end do

  100 continue

      return
      end
C
C
C
      subroutine update_d_upml(
     & SpaceDim, margin, D_index, H_index,
     & imin_Cx, imax_Cx, C1x, C2x,
     & jmin_Cy, jmax_Cy, C1y, C2y,
     & kmin_Cz, kmax_Cz, C1z, C2z,
     & imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx, Dx,
     & imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy, Dy,
     & imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz, Dz,
     & imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx, Hx,
     & imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy, Hy,
     & imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz, Hz,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, D_index(0:1), H_index(0:1),
     & fortran_verbose, return_status
```

```
C
      integer imin_Cx, imax_Cx
      double precision C1x(imin_Cx:imax_Cx), C2x(imin_Cx:imax_Cx)
C
      integer jmin_Cy, jmax_Cy
      double precision C1y(jmin_Cy:jmax_Cy), C2y(jmin_Cy:jmax_Cy)
C
      integer kmin_Cz, kmax_Cz
      double precision C1z(kmin_Cz:kmax_Cz), C2z(kmin_Cz:kmax_Cz)
C
      integer imin_Dx, jmin_Dx, kmin_Dx, imax_Dx, jmax_Dx, kmax_Dx
      double precision
     & Dx(imin_Dx:imax_Dx, jmin_Dx:jmax_Dx, kmin_Dx:kmax_Dx, 0:1)
C
      integer imin_Dy, jmin_Dy, kmin_Dy, imax_Dy, jmax_Dy, kmax_Dy
      double precision
     & Dy(imin_Dy:imax_Dy, jmin_Dy:jmax_Dy, kmin_Dy:kmax_Dy, 0:1)
C
      integer imin_Dz, jmin_Dz, kmin_Dz, imax_Dz, jmax_Dz, kmax_Dz
      double precision
     & Dz(imin_Dz:imax_Dz, jmin_Dz:jmax_Dz, kmin_Dz:kmax_Dz, 0:1)
C
      integer imin_Hx, jmin_Hx, kmin_Hx, imax_Hx, jmax_Hx, kmax_Hx
      double precision
     & Hx(imin_Hx:imax_Hx, jmin_Hx:jmax_Hx, kmin_Hx:kmax_Hx, 0:1)
C
      integer imin_Hy, jmin_Hy, kmin_Hy, imax_Hy, jmax_Hy, kmax_Hy
      double precision
     & Hy(imin_Hy:imax_Hy, jmin_Hy:jmax_Hy, kmin_Hy:kmax_Hy, 0:1)
C
      integer imin_Hz, jmin_Hz, kmin_Hz, imax_Hz, jmax_Hz, kmax_Hz
      double precision
     & Hz(imin_Hz:imax_Hz, jmin_Hz:jmax_Hz, kmin_Hz:kmax_Hz, 0:1)
C
      integer i, j, k, new_D, old_D, new_H
      integer ERR_ADVANCE_E
      parameter (ERR_ADVANCE_E = 5)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_d_upml_): Bad SpaceDim'
         return_status = ERR_ADVANCE_E
         goto 100
      end if

      new_D = D_index(0)
      old_D = D_index(1)
      new_H = H_index(0)

      do k = kmin_Dx + margin, kmax_Dx - margin
         do j = jmin_Dx + margin, jmax_Dx - margin
            do i = imin_Dx + margin, imax_Dx - margin
               Dx(i, j, k, new_D) = C1y(j) * Dx(i, j, k, old_D)
     & + C2y(j) * (Hz(i, j, k, new_H)
     & - Hz(i, j - 1, k, new_H)
     & - Hy(i, j, k, new_H)
     & + Hy(i, j, k - 1, new_H))
            end do
```

```
          end do
        end do

        do k = kmin_Dy + margin, kmax_Dy − margin
          do j = jmin_Dy + margin, jmax_Dy − margin
            do i = imin_Dy + margin, imax_Dy − margin
              Dy(i, j, k, new_D) = C1z(k) ∗ Dy(i, j, k, old_D)
     & + C2z(k) ∗ (Hx(i, j, k, new_H)
     & − Hx(i, j, k − 1, new_H)
     & − Hz(i, j, k, new_H) + Hz(i − 1, j, k, new_H))
            end do
          end do
        end do

        do k = kmin_Dz + margin, kmax_Dz − margin
          do j = jmin_Dz + margin, jmax_Dz − margin
            do i = imin_Dz + margin, imax_Dz − margin
              Dz(i, j, k, new_D) = C1x(i) ∗ Dz(i, j, k, old_D)
     & + C2x(i) ∗ (Hy(i, j, k, new_H)
     & − Hy(i − 1, j, k, new_H)
     & − Hx(i, j, k, new_H) + Hx(i, j − 1, k, new_H))
            end do
          end do
        end do

100     continue

        return
        end

C
C $Log: update_d.f,v $
C Revision 1.8 2007/08/31 16:00:53 gustav
C Combined update_d_ and update_d_upml_ on this single file.
C Added handling of the new/old−field index to both.
C
C
```

13.2 Subroutines `update_b` and `update_b_upml`

Update the $\boldsymbol{B}$ field by applying Maxwell equations. Both UPML and normal versions are provided on this file.

```
C Fortran subroutine update_b
C
C Update the B field, both pure and UPML version.
C
C $Id: update_b.f,v 1.5 2007/09/03 19:04:02 gustav Exp $
C
C Pure Maxwell first.
C
      subroutine update_b(
     & SpaceDim, margin, B_index, E_index, dt_by_dg,
     & imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx, Bx,
     & imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By, By,
     & imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz, Bz,
     & imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex, Ex,
     & imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey, Ey,
     & imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez, Ez,
     & fortran_verbose, return_status)
      implicit none
C
      integer SpaceDim, margin, B_index(0:1), E_index(0:1),
     & fortran_verbose, return_status
C
      double precision dt_by_dg
C
      integer imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx
      double precision
     & Bx(imin_Bx:imax_Bx, jmin_Bx:jmax_Bx, kmin_Bx:kmax_Bx, 0:1)
C
      integer imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By
      double precision
     & By(imin_By:imax_By, jmin_By:jmax_By, kmin_By:kmax_By, 0:1)
C
      integer imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz
      double precision
     & Bz(imin_Bz:imax_Bz, jmin_Bz:jmax_Bz, kmin_Bz:kmax_Bz, 0:1)
C
      integer imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex
      double precision
     & Ex(imin_Ex:imax_Ex, jmin_Ex:jmax_Ex, kmin_Ex:kmax_Ex, 0:1)
C
      integer imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey
      double precision
     & Ey(imin_Ey:imax_Ey, jmin_Ey:jmax_Ey, kmin_Ey:kmax_Ey, 0:1)
C
      integer imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez
      double precision
     & Ez(imin_Ez:imax_Ez, jmin_Ez:jmax_Ez, kmin_Ez:kmax_Ez, 0:1)
C
      integer i, j, k, new_B, old_B, new_E
      integer ERR_ADVANCE_H
      parameter (ERR_ADVANCE_H = 6)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_b_): bad SapceDim'
```

```
         return_status = ERR_ADVANCE_H
         goto 100
      end if

      new_B = B_index(0)
      old_B = B_index(1)
      new_E = E_index(0)

      do k = kmin_Bx + margin, kmax_Bx - margin
         do j = jmin_Bx + margin, jmax_Bx - margin
            do i = imin_Bx + margin, imax_Bx - margin
               Bx(i, j, k, new_B) = Bx(i, j, k, old_B) + dt_by_dg *
     & (Ey(i, j, k + 1, new_E) - Ey(i, j, k, new_E)
     & - Ez(i, j + 1, k, new_E) + Ez(i, j, k, new_E))
            end do
         end do
      end do

      do k = kmin_By + margin, kmax_By - margin
         do j = jmin_By + margin, jmax_By - margin
            do i = imin_By + margin, imax_By - margin
               By(i, j, k, new_B) = By(i, j, k, old_B) + dt_by_dg *
     & (Ez(i + 1, j, k, new_E) - Ez(i, j, k, new_E)
     & - Ex(i, j, k + 1, new_E) + Ex(i, j, k, new_E))
            end do
         end do
      end do

      do k = kmin_Bz + margin, kmax_Bz - margin
         do j = jmin_Bz + margin, jmax_Bz - margin
            do i = imin_Bz + margin, imax_Bz - margin
               Bz(i, j, k, new_B) = Bz(i, j, k, old_B) + dt_by_dg *
     & (Ex(i, j + 1, k, new_E) - Ex(i, j, k, new_E)
     & - Ey(i + 1, j, k, new_E) + Ey(i, j, k, new_E))
            end do
         end do
      end do

 100  continue

      return
      end
C
C The UPML version.
C
      subroutine update_b_upml(
     & SpaceDim, margin, B_index, E_index,
     & imin_Cx, imax_Cx, C1x, C2x,
     & jmin_Cy, jmax_Cy, C1y, C2y,
     & kmin_Cz, kmax_Cz, C1z, C2z,
     & imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx, Bx,
     & imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By, By,
     & imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz, Bz,
     & imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex, Ex,
     & imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey, Ey,
     & imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez, Ez,
     & fortran_verbose, return_status)
      implicit none
C
```

```
      integer SpaceDim, margin, B_index(0:1), E_index(0:1),
     & fortran_verbose, return_status
C
      integer imin_Cx, imax_Cx
      double precision C1x(imin_Cx:imax_Cx), C2x(imin_Cx:imax_Cx)
C
      integer jmin_Cy, jmax_Cy
      double precision C1y(jmin_Cy:jmax_Cy), C2y(jmin_Cy:jmax_Cy)
C
      integer kmin_Cz, kmax_Cz
      double precision C1z(kmin_Cz:kmax_Cz), C2z(kmin_Cz:kmax_Cz)
C
      integer imin_Bx, jmin_Bx, kmin_Bx, imax_Bx, jmax_Bx, kmax_Bx
      double precision
     & Bx(imin_Bx:imax_Bx, jmin_Bx:jmax_Bx, kmin_Bx:kmax_Bx, 0:1)
C
      integer imin_By, jmin_By, kmin_By, imax_By, jmax_By, kmax_By
      double precision
     & By(imin_By:imax_By, jmin_By:jmax_By, kmin_By:kmax_By, 0:1)
C
      integer imin_Bz, jmin_Bz, kmin_Bz, imax_Bz, jmax_Bz, kmax_Bz
      double precision
     & Bz(imin_Bz:imax_Bz, jmin_Bz:jmax_Bz, kmin_Bz:kmax_Bz, 0:1)
C
      integer imin_Ex, jmin_Ex, kmin_Ex, imax_Ex, jmax_Ex, kmax_Ex
      double precision
     & Ex(imin_Ex:imax_Ex, jmin_Ex:jmax_Ex, kmin_Ex:kmax_Ex, 0:1)
C
      integer imin_Ey, jmin_Ey, kmin_Ey, imax_Ey, jmax_Ey, kmax_Ey
      double precision
     & Ey(imin_Ey:imax_Ey, jmin_Ey:jmax_Ey, kmin_Ey:kmax_Ey, 0:1)
C
      integer imin_Ez, jmin_Ez, kmin_Ez, imax_Ez, jmax_Ez, kmax_Ez
      double precision
     & Ez(imin_Ez:imax_Ez, jmin_Ez:jmax_Ez, kmin_Ez:kmax_Ez, 0:1)
C
      integer i, j, k, new_B, old_B, new_E
      integer ERR_ADVANCE_H
      parameter (ERR_ADVANCE_H = 6)
C
C Action
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(update_b_upml_): Bad SpaceDim'
         return_status = ERR_ADVANCE_H
         goto 100
      end if

      new_B = B_index(0)
      old_B = B_index(1)
      new_E = E_index(0)

      do k = kmin_Bx + margin, kmax_Bx - margin
         do j = jmin_Bx + margin, jmax_Bx - margin
            do i = imin_Bx + margin, imax_Bx - margin
               Bx(i, j, k, new_B) = C1y(j) * Bx(i, j, k, old_B)
     & + C2y(j) * (Ey(i, j, k + 1, new_E)
     & - Ey(i, j, k, new_E)
     & - Ez(i, j + 1, k, new_E) + Ez(i, j, k, new_E))
```

```
            end do
         end do
      end do

      do k = kmin_By + margin, kmax_By − margin
         do j = jmin_By + margin, jmax_By − margin
            do i = imin_By + margin, imax_By − margin
               By(i, j, k, new_B) = C1z(k) ∗ By(i, j, k, old_B)
     & + C2z(k) ∗ (Ez(i + 1, j, k, new_E)
     & − Ez(i, j, k, new_E)
     & − Ex(i, j, k + 1, new_E) + Ex(i, j, k, new_E))
            end do
         end do
      end do

      do k = kmin_Bz + margin, kmax_Bz − margin
         do j = jmin_Bz + margin, jmax_Bz − margin
            do i = imin_Bz + margin, imax_Bz − margin
               Bz(i, j, k, new_B) = C1x(i) ∗ Bz(i, j, k, old_B)
     & + C2x(i) ∗ (Ex(i, j + 1, k, new_E)
     & − Ex(i, j, k, new_E)
     & − Ey(i + 1, j, k, new_E) + Ey(i, j, k, new_E))
            end do
         end do
      end do

 100 continue

      return
      end
C
C $Log: update_b.f,v $
C Revision 1.5 2007/09/03 19:04:02 gustav
C Forgot to initialize new_B, old_B and new_E in update_b_upml_.
C
C Revision 1.4 2007/09/03 17:45:44 gustav
C Combined non−UPML and UPML versions on a single file.
C Added handling of the new/old−index switch.
C
C
```

13.3 *D*-to-*E* Conversion Subroutines

```
C Convert D to E within the region provided, taking care of UPMLs,
C for level 0 and otherwise for levels other than zero.
C
C $Id: d_to_e.f,v 1.10 2007/09/28 00:41:26 gustav Exp $
C
C Level zero case
C
C C++ declaration
C
C This first function assumes no auxiliary fields.
C
C extern"C"
C {
C void d_to_e_0_0_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const E_idx,
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_Cx,
C const int*const imax_Cx,
C const Real*const C0x,
C const Real*const C1x,
C const Real*const C3x,
C const Real*const C4x,
C const int*const imin_Cy,
C const int*const imax_Cy,
C const Real*const C0y,
C const Real*const C1y,
C const Real*const C3y,
C const Real*const C4y,
C const int*const imin_Cz,
C const int*const imax_Cz,
C const Real*const C0z,
C const Real*const C1z,
C const Real*const C3z,
C const Real*const C4z,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_0_0(
     & SpaceDim, dir, margin,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_Cx, imax_Cx, C0x, C1x, C3x, C4x,
```

```
      & imin_Cy, imax_Cy, C0y, C1y, C3y, C4y,
      & imin_Cz, imax_Cz, C0z, C1z, C3z, C4z,
      & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
      & D, E, medium_E,
      & watch_d_to_e, return_status)
       implicit none
C
C passed variables
C
       integer SpaceDim, dir, margin, E_idx(0:1), D_idx(0:1)
       double precision time_e, dt
       integer imin_Cx, imax_Cx
       double precision
      & C0x(imin_Cx:imax_Cx),
      & C1x(imin_Cx:imax_Cx),
      & C3x(imin_Cx:imax_Cx),
      & C4x(imin_Cx:imax_Cx)
       integer imin_Cy, imax_Cy
       double precision
      & C0y(imin_Cy:imax_Cy),
      & C1y(imin_Cy:imax_Cy),
      & C3y(imin_Cy:imax_Cy),
      & C4y(imin_Cy:imax_Cy)
       integer imin_Cz, imax_Cz
       double precision
      & C0z(imin_Cz:imax_Cz),
      & C1z(imin_Cz:imax_Cz),
      & C3z(imin_Cz:imax_Cz),
      & C4z(imin_Cz:imax_Cz)
       integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
       double precision
      & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
      & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1)
       integer
      & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
       integer watch_d_to_e, return_status
C
C internal variables
C
       integer i, j, k, new_E, new_D, old_E, old_D
       double precision C0, C1, C3, C4
       integer ERROR_IN_D_TO_E, NO_ERROR
       parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external function
C
       double precision scm_d_to_e_0
C
C action
C
       return_status = NO_ERROR
C
       if (SpaceDim .ne. 3) then
          write(*,*) 'ERROR(d_to_e_0_0_): bad SpaceDim'
          return_status = ERROR_IN_D_TO_E
          goto 100
       end if
C
       if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
```

```
          write(*,*) 'ERROR(d_to_e_0_0_): bad dir'
          return_status = ERROR_IN_D_TO_E
          goto 100
        end if
C
        new_E = E_idx(0)
        old_E = E_idx(1)
        new_D = D_idx(0)
        old_D = D_idx(1)
C
        do k = kmin_D + margin, kmax_D - margin
          do j = jmin_D + margin, jmax_D - margin
            do i = imin_D + margin, imax_D - margin
              select case (medium_E(i, j, k))
                case (-2)
                  select case (dir)
                    case (0)
                      C0 = C0z(k)
                      C1 = C1z(k)
                      C3 = C3x(i)
                      C4 = C4x(i)
                    case (1)
                      C0 = C0x(i)
                      C1 = C1x(i)
                      C3 = C3y(j)
                      C4 = C4y(j)
                    case (2)
                      C0 = C0y(j)
                      C1 = C1y(j)
                      C3 = C3z(k)
                      C4 = C4z(k)
                  end select
                  E(i, j, k, new_E) = C1 * E(i, j, k, old_E)
     & + C3 * C0 * D(i, j, k, new_D)
     & - C4 * C0 * D(i, j, k, old_D)
                case (-1:0)
                  E(i,j,k, new_E) = D(i,j,k, new_D)
                case default
                  E(i,j,k, new_E) = scm_d_to_e_0(
     & dir, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & time_e, dt)
              end select
            end do
          end do
        end do
C
 100  continue
        return
        end
C
C C++ declaration
C
C This function is similar to the one above, but it makes
C provision for passing of auxiliary fields.
C
C extern"C"
C {
```

```
C void d_to_e_0_n_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const number_of_auxiliary_fields,
C const int*const E_idx,
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_Cx,
C const int*const imax_Cx,
C const Real*const C0x,
C const Real*const C1x,
C const Real*const C3x,
C const Real*const C4x,
C const int*const imin_Cy,
C const int*const imax_Cy,
C const Real*const C0y,
C const Real*const C1y,
C const Real*const C3y,
C const Real*const C4y,
C const int*const imin_Cz,
C const int*const imax_Cz,
C const Real*const C0z,
C const Real*const C1z,
C const Real*const C3z,
C const Real*const C4z,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const Real*const S,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_0_n(
     & SpaceDim, dir, margin, n_aux_fields,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_Cx, imax_Cx, C0x, C1x, C3x, C4x,
     & imin_Cy, imax_Cy, C0y, C1y, C3y, C4y,
     & imin_Cz, imax_Cz, C0z, C1z, C3z, C4z,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, S, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, n_aux_fields,
     & E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
```

```
      integer imin_Cx, imax_Cx
      double precision
     & C0x(imin_Cx:imax_Cx),
     & C1x(imin_Cx:imax_Cx),
     & C3x(imin_Cx:imax_Cx),
     & C4x(imin_Cx:imax_Cx)
      integer imin_Cy, imax_Cy
      double precision
     & C0y(imin_Cy:imax_Cy),
     & C1y(imin_Cy:imax_Cy),
     & C3y(imin_Cy:imax_Cy),
     & C4y(imin_Cy:imax_Cy)
      integer imin_Cz, imax_Cz
      double precision
     & C0z(imin_Cz:imax_Cz),
     & C1z(imin_Cz:imax_Cz),
     & C3z(imin_Cz:imax_Cz),
     & C4z(imin_Cz:imax_Cz)
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & S(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, n_aux_fields)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, n, n_max, new_E, old_E, new_D, old_D
      parameter (n_max = 100)
      double precision C0, C1, C3, C4
      double precision S_vector(n_max)
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external functions
C
      double precision scm_d_to_e_n
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_0_n_): bad SpaceDim'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(d_to_e_0_n_): bad dir'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if (n_aux_fields .gt. n_max) then
         write(*,*) 'ERROR(d_to_e_0_n_): too many fields'
         return_status = ERROR_IN_D_TO_E
```

```
          goto 100
        end if
C
        new_E = E_idx(0)
        old_E = E_idx(1)
        new_D = D_idx(0)
        old_D = D_idx(1)
C
        do k = kmin_D + margin, kmax_D - margin
          do j = jmin_D + margin, jmax_D - margin
            do i = imin_D + margin, imax_D - margin
              select case (medium_E(i, j, k))
                case (-2)
                  select case (dir)
                    case (0)
                      C0 = C0z(k)
                      C1 = C1z(k)
                      C3 = C3x(i)
                      C4 = C4x(i)
                    case (1)
                      C0 = C0x(i)
                      C1 = C1x(i)
                      C3 = C3y(j)
                      C4 = C4y(j)
                    case (2)
                      C0 = C0y(j)
                      C1 = C1y(j)
                      C3 = C3z(k)
                      C4 = C4z(k)
                  end select
                  E(i, j, k, new_E) = C1 * E(i, j, k, old_E)
     & + C3 * C0 * D(i, j, k, new_D)
     & - C4 * C0 * D(i, j, k, old_D)
                case (-1:0)
                  E(i,j,k,new_E) = D(i,j,k,new_D)
                case default
                  do n = 1, n_aux_fields
                    S_vector(n) = S(i,j,k,n)
                  end do
                  E(i,j,k,new_E) = scm_d_to_e_n(
     & dir, n_aux_fields, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & S_vector, time_e, dt)
                  do n = 1, n_aux_fields
                    S(i,j,k,n) = S_vector(n)
                  end do
              end select
            end do
          end do
        end do
C
 100    continue
        return
        end
C
C Higher levels case
C
C C++ declaration
```

```
C
C This first function for higher levels assumes no auxiliary fields.
C
C extern"C"
C {
C void d_to_e_n_0_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const E_idx,
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_n_0(
     & SpaceDim, dir, margin,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, new_E, old_E, new_D, old_D
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external function
C
      double precision scm_d_to_e_0
C
C action
```

```
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_n_0_): bad SpaceDim'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(d_to_e_n_0_): bad dir'
         return_status = ERROR_IN_D_TO_E
         goto 100
      end if
C
      new_E = E_idx(0)
      old_E = E_idx(1)
      new_D = D_idx(0)
      old_D = D_idx(1)
C
      do k = kmin_D + margin, kmax_D - margin
         do j = jmin_D + margin, jmax_D - margin
            do i = imin_D + margin, imax_D - margin
               select case (medium_E(i, j, k))
                  case (-2)
                     write(*,*)
     & 'ERROR(d_to_e_n_0_): no media handler'
                     return_status = ERROR_IN_D_TO_E
                     goto 100
                  case (-1:0)
                     E(i,j,k,new_E) = D(i,j,k,new_D)
                  case default
                     E(i,j,k,new_E) = scm_d_to_e_0(
     & dir, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & time_e, dt)
               end select
            end do
         end do
      end do
C
 100  continue
      return
      end
C
C C++ declaration
C
C This function is similar to the one above, but it makes
C provision for passing of auxiliary fields.
C
C extern"C"
C {
C void d_to_e_n_n_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const number_of_auxiliary_fields,
C const int*const E_idx,
```

```
C const int*const D_idx,
C const Real*const time_e,
C const Real*const dt,
C const int*const imin_D,
C const int*const jmin_D,
C const int*const kmin_D,
C const int*const imax_D,
C const int*const jmax_D,
C const int*const kmax_D,
C const Real*const D,
C const Real*const E,
C const Real*const S,
C const int*const medium_E,
C const int*const watch_d_to_e,
C const int*const return_status
C );
C }
C
      subroutine d_to_e_n_n(
     & SpaceDim, dir, margin, n_aux_fields,
     & E_idx, D_idx,
     & time_e, dt,
     & imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D,
     & D, E, S, medium_E,
     & watch_d_to_e, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, n_aux_fields,
     & E_idx(0:1), D_idx(0:1)
      double precision time_e, dt
      integer imin_D, jmin_D, kmin_D, imax_D, jmax_D, kmax_D
      double precision
     & D(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, 0:1),
     & S(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D, n_aux_fields)
      integer
     & medium_E(imin_D:imax_D, jmin_D:jmax_D, kmin_D:kmax_D)
      integer watch_d_to_e, return_status
C
C internal variables
C
      integer i, j, k, n, n_max, new_E, old_E, new_D, old_D
      parameter (n_max = 100)
      double precision S_vector(n_max)
      integer ERROR_IN_D_TO_E, NO_ERROR
      parameter (ERROR_IN_D_TO_E = 33, NO_ERROR = 0)
C
C external functions
C
      double precision scm_d_to_e_n
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(d_to_e_n_n_): bad SpaceDim'
```

```
          return_status = ERROR_IN_D_TO_E
          goto 100
        end if
C
        if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
          write(*,*) 'ERROR(d_to_e_n_n_): bad dir'
          return_status = ERROR_IN_D_TO_E
          goto 100
        end if
C
        if (n_aux_fields .gt. n_max) then
          write(*,*) 'ERROR(d_to_e_n_n_): too many fields'
          return_status = ERROR_IN_D_TO_E
          goto 100
        end if
C
        new_E = E_idx(0)
        old_E = E_idx(1)
        new_D = D_idx(0)
        old_D = D_idx(1)
C
        do k = kmin_D + margin, kmax_D - margin
          do j = jmin_D + margin, jmax_D - margin
            do i = imin_D + margin, imax_D - margin
              select case (medium_E(i, j, k))
                case (-2)
                  write(*,*)
     & 'ERROR(d_to_e_n_n_): no media handler'
                  return_status = ERROR_IN_D_TO_E
                  goto 100
                case (-1:0)
                  E(i,j,k,new_E) = D(i,j,k,new_D)
                case default
                  do n = 1, n_aux_fields
                    S_vector(n) = S(i,j,k,n)
                  end do
                  E(i,j,k,new_E) = scm_d_to_e_n(
     & dir, n_aux_fields, medium_E(i,j,k),
     & E(i,j,k,old_E),
     & D(i,j,k,new_D), D(i,j,k,old_D),
     & S_vector, time_e, dt)
                  do n = 1, n_aux_fields
                    S(i,j,k,n) = S_vector(n)
                  end do
              end select
            end do
          end do
        end do
C
 100  continue
        return
        end
C
C $Log: d_to_e.f,v $
C Revision 1.10 2007/09/28 00:41:26 gustav
C + C4 -> - C4.
C
C Revision 1.9 2007/08/31 20:56:08 gustav
C Changed the order of parameters as passed to the functions with
```

```
C Auxiliary fields. There is now D, E, S, medium_E.
C
C Revision 1.8 2007/08/31 20:46:04 gustav
C Added handling of E_idx and D_idx and deleted any references to
C D_old and E_old.
C
C Revision 1.7 2007/08/28 20:32:27 gustav
C Fixed a bug in the declaration of scm_d_to_e_0. It was declared as
C integer, which caused a significant slow-down of the code. Probably
C generating NANs.
C
C Revision 1.6 2007/08/28 18:08:13 gustav
C Added handlers for functions for media with auxiliary fields.
C changed error exit to 33 everywhere.
C Added error messages on errors encountered and caught.
C Simplified logic, because errors are handled up front.
C
C Revision 1.5 2007/08/27 21:45:26 gustav
C added dir to the call to scm_d_to_e_0.
C
C Revision 1.4 2007/08/27 21:35:02 gustav
C Function scm_d_to_e_0 now returns E, not return_status.
C
C Revision 1.3 2007/08/27 18:10:32 gustav
C Added calls to an external function scm_d_to_e_0, which is going to
C handle Scheme driven conversion of D to E at this point. Only for the
C case with no auxiliary fields for the time being.
C
C Revision 1.2 2007/08/23 21:02:39 gustav
C Added subroutines with auxiliary fields.
C
C Revision 1.1 2007/08/23 20:32:32 gustav
C Initial revision
C
C
```

13.4 *B*-to-*H* Conversion Subroutines

```
C Convert B to H within the region provided, taking care of UPMLs
C for level 0 and otherwise for levels other than zero.
C
C $Id: b_to_h.f,v 1.3 2007/09/28 00:41:39 gustav Exp $
C
C Level zero case
C
C C++ declaration
C
C extern"C"
C {
C void b_to_h_0_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const H_idx,
C const int*const B_idx,
C const Real*const time_h,
C const Real*const dt,
C const int*const imin_Cx,
C const int*const imax_Cx,
C const Real*const C0x,
C const Real*const C1x,
C const Real*const C3x,
C const Real*const C4x,
C const int*const imin_Cy,
C const int*const imax_Cy,
C const Real*const C0y,
C const Real*const C1y,
C const Real*const C3y,
C const Real*const C4y,
C const int*const imin_Cz,
C const int*const imax_Cz,
C const Real*const C0z,
C const Real*const C1z,
C const Real*const C3z,
C const Real*const C4z,
C const int*const imin_B,
C const int*const jmin_B,
C const int*const kmin_B,
C const int*const imax_B,
C const int*const jmax_B,
C const int*const kmax_B,
C const Real*const B,
C const Real*const H,
C const int*const medium_H,
C const int*const watch_b_to_h,
C const int*const return_status
C );
C }
C
      subroutine b_to_h_0(
     & SpaceDim, dir, margin, H_idx, B_idx,
     & time_h, dt,
     & imin_Cx, imax_Cx, C0x, C1x, C3x, C4x,
     & imin_Cy, imax_Cy, C0y, C1y, C3y, C4y,
     & imin_Cz, imax_Cz, C0z, C1z, C3z, C4z,
     & imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B,
```

```
      & B, H, medium_H,
      & watch_b_to_h, return_status)
       implicit none
C
C passed variables
C
       integer SpaceDim, dir, margin, H_idx(0:1), B_idx(0:1)
       double precision time_h, dt
       integer imin_Cx, imax_Cx
       double precision
      & C0x(imin_Cx:imax_Cx),
      & C1x(imin_Cx:imax_Cx),
      & C3x(imin_Cx:imax_Cx),
      & C4x(imin_Cx:imax_Cx)
       integer imin_Cy, imax_Cy
       double precision
      & C0y(imin_Cy:imax_Cy),
      & C1y(imin_Cy:imax_Cy),
      & C3y(imin_Cy:imax_Cy),
      & C4y(imin_Cy:imax_Cy)
       integer imin_Cz, imax_Cz
       double precision
      & C0z(imin_Cz:imax_Cz),
      & C1z(imin_Cz:imax_Cz),
      & C3z(imin_Cz:imax_Cz),
      & C4z(imin_Cz:imax_Cz)
       integer imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B
       double precision
      & B(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1),
      & H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1)
       integer
      & medium_H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B)
       integer watch_b_to_h, return_status
C
C internal variables
C
       integer i, j, k, new_H, old_H, new_B, old_B
       double precision C0, C1, C3, C4
       integer ERROR_IN_B_TO_H, NO_ERROR
       parameter (ERROR_IN_B_TO_H = 9, NO_ERROR = 0)
C
C action
C
       return_status = NO_ERROR
C
       if (SpaceDim .ne. 3) then
          write(*,*) 'ERROR(b_to_h_0_): bad SpaceDim'
          return_status = ERROR_IN_B_TO_H
          goto 100
       end if
C
       if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
          write(*,*) 'ERROR(b_to_h_0_): bad dir'
          return_status = ERROR_IN_B_TO_H
          goto 100
       end if
C
       new_H = H_idx(0)
       old_H = H_idx(1)
```

```
      new_B = B_idx(0)
      old_B = B_idx(1)
C
      do k = kmin_B + margin, kmax_B − margin
         do j = jmin_B + margin, jmax_B − margin
            do i = imin_B + margin, imax_B − margin
               select case (medium_H(i, j, k))
                  case (−2)
                     select case (dir)
                        case (0)
                           C0 = C0z(k)
                           C1 = C1z(k)
                           C3 = C3x(i)
                           C4 = C4x(i)
                        case (1)
                           C0 = C0x(i)
                           C1 = C1x(i)
                           C3 = C3y(j)
                           C4 = C4y(j)
                        case (2)
                           C0 = C0y(j)
                           C1 = C1y(j)
                           C3 = C3z(k)
                           C4 = C4z(k)
                     end select
                     H(i, j, k, new_H) = C1 ∗ H(i, j, k, old_H)
     & + C3 ∗ C0 ∗ B(i, j, k, new_B)
     & − C4 ∗ C0 ∗ B(i, j, k, old_B)
                  case default
                     H(i, j, k, new_H) = B(i, j, k, new_B)
               end select
            end do
         end do
      end do
C
 100 continue
      return
      end
C
C Higher levels case
C
C C++ declaration
C
C extern"C"
C {
C void b_to_h_n_(
C const int*const space_dim,
C const int*const dir,
C const int*const ghost_margin,
C const int*const H_idx,
C const int*const B_idx,
C const Real*const time_h,
C const Real*const dt,
C const int*const imin_B,
C const int*const jmin_B,
C const int*const kmin_B,
C const int*const imax_B,
C const int*const jmax_B,
C const int*const kmax_B,
```

```
C const Real*const B,
C const Real*const H,
C const int*const medium_H,
C const int*const watch_b_to_h,
C const int*const return_status
C );
C }
C
      subroutine b_to_h_n(
     & SpaceDim, dir, margin, H_idx, B_idx,
     & time_h, dt,
     & imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B,
     & B, H, medium_H,
     & watch_b_to_h, return_status)
      implicit none
C
C passed variables
C
      integer SpaceDim, dir, margin, H_idx(0:1), B_idx(0:1)
      double precision time_h, dt
      integer imin_B, jmin_B, kmin_B, imax_B, jmax_B, kmax_B
      double precision
     & B(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1),
     & H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B, 0:1)
      integer
     & medium_H(imin_B:imax_B, jmin_B:jmax_B, kmin_B:kmax_B)
      integer watch_b_to_h, return_status
C
C internal variables
C
      integer i, j, k, new_H, new_B
      integer ERROR_IN_B_TO_H, NO_ERROR
      parameter (ERROR_IN_B_TO_H = 9, NO_ERROR = 0)
C
C action
C
      return_status = NO_ERROR
C
      if (SpaceDim .ne. 3) then
         write(*,*) 'ERROR(b_to_h_n_): bad SpaceDim'
         return_status = ERROR_IN_B_TO_H
         goto 100
      end if
C
      if ((dir .lt. 0) .or. (SpaceDim .le. dir)) then
         write(*,*) 'ERROR(b_to_h_n_): bad dir'
         return_status = ERROR_IN_B_TO_H
         goto 100
      end if
C
      new_H = H_idx(0)
      new_B = B_idx(0)
C
      do k = kmin_B + margin, kmax_B - margin
         do j = jmin_B + margin, jmax_B - margin
            do i = imin_B + margin, imax_B - margin
               select case (medium_H(i, j, k))
                  case(-2)
                     write(*,*)
```

```
      & 'ERROR(b_to_h_n_): no media handler'
                        return_status = ERROR_IN_B_TO_H
                        goto 100
                     case default
                        H(i,j,k, new_H) = B(i,j,k, new_B)
                  end select
               end do
            end do
         end do
C
 100 continue
         return
         end
C
C $Log: b_to_h.f,v $
C Revision 1.3 2007/09/28 00:41:39 gustav
C + C4 -> - C4
C
C Revision 1.2 2007/09/03 18:22:19 gustav
C Implemented the new/old index switch.
C
C Revision 1.1 2007/08/23 19:07:04 gustav
C Initial revision
C
C
```

13.5 Fortran Subroutines for Drawing on Chombo Data Structures

```
C
C $Id: draw.f,v 1.6 2007/10/10 17:15:44 gustav Exp $
C
C Draw a box on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_box_(
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
C const Real*const delta,
C const int*const iv_lower_corner,
C const int*const iv_upper_corner,
C const Real*const lower_corner,
C const Real*const upper_corner,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
      subroutine f_draw_box(
     & spacedim, margin,
     & origin, delta,
     & iv_low_corner, iv_high_corner,
     & low_corner, high_corner, color,
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
     & medium,
     & verbosity, return_status)
      implicit none
C
      integer spacedim, margin
      double precision origin(0:spacedim - 1), delta
      integer
     & iv_low_corner(0:spacedim - 1),
     & iv_high_corner(0:spacedim - 1)
      double precision
     & low_corner(0:spacedim - 1),
     & high_corner(0:spacedim - 1)
      integer color
      integer
     & box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
```

```
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x, y, z
      integer count
C
C Action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_box_): ',
     & 'bad dimension'
         return_status = 0
         goto 100
      end if
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_box_:'
         write(*,*) ' spacedim = ', spacedim
         write(*,*) ' margin = ', margin
         write(*,*) ' delta = ', delta
         write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(*,*) ' iv_low = ',
     & iv_low_corner(0), ', ',
     & iv_low_corner(1), ', ',
     & iv_low_corner(2)
         write(*,*) ' iv_hi = ',
     & iv_high_corner(0), ', ',
     & iv_high_corner(1), ', ',
     & iv_high_corner(2)
         write(*,*) ' low = ',
     & low_corner(0), ', ',
     & low_corner(1), ', ',
     & low_corner(2)
         write(*,*) ' hi = ',
     & high_corner(0), ', ',
     & high_corner(1), ', ',
     & high_corner(2)
         write(*,*) ' type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      i_min = max(i_lo + margin, iv_low_corner(0))
      j_min = max(j_lo + margin, iv_low_corner(1))
      k_min = max(k_lo + margin, iv_low_corner(2))
C
      i_max = min(i_hi - margin, iv_high_corner(0))
      j_max = min(j_hi - margin, iv_high_corner(1))
      k_max = min(k_hi - margin, iv_high_corner(2))
C
      count = 0
C
      do k = k_min, k_max
```

```
          z = origin(2) + (k - 0.5 * box_type(2)) * delta
          if ((low_corner(2) .le. z) .and.
     & (z .le. high_corner(2))) then
             do j = j_min, j_max
                y = origin(1) + (j - 0.5 * box_type(1)) * delta
                if ((low_corner(1) .le. y) .and.
     & (y .le. high_corner(1))) then
                   do i = i_min, i_max
                      x = origin(0) + (i - 0.5 * box_type(0)) * delta
                      if ((low_corner(0) .le. x) .and.
     & (x .le. high_corner(0))) then
                         medium(i,j,k) = color
                         count = count + 1
                      end if
                   end do
                end if
             end do
          end if
       end do
C
       if (verbosity .gt. 0) then
          write(*,*) ' marked ', count, ' cells'
       end if
C
 100 continue
C
       end
C
C Draw a ball on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_ball_(
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
C const Real*const delta,
C const int*const iv_lower_corner,
C const int*const iv_upper_corner,
C const Real*const center,
C const Real*const radius,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
       subroutine f_draw_ball(
     & spacedim, margin,
```

```
      & origin, delta,
      & iv_low_corner, iv_high_corner,
      & center, radius, color,
      & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
      & medium,
      & verbosity, return_status)
       implicit none
C
       integer spacedim, margin
       double precision origin(0:spacedim − 1), delta
       integer
      & iv_low_corner(0:spacedim − 1),
      & iv_high_corner(0:spacedim − 1)
       double precision
      & center(0:spacedim − 1)
       double precision radius
       integer color
       integer
      & box_type(0:spacedim − 1)
       integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
       integer
      & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
       integer verbosity, return_status
C
C internal variables
C
       integer i_min, j_min, k_min
       integer i_max, j_max, k_max
       integer i, j, k
       double precision x, y, z
       double precision dx2, dy2, dz2, d2, r2
       integer count
C
C Action
C
       if (spacedim .ne. 3) then
           write(∗,∗) 'ERROR(f_draw_ball_): ',
      & 'bad dimension'
           return_status = 0
           goto 100
       end if
C
       if (verbosity .gt. 0) then
           write(∗,∗) ' f_draw_ball_:'
           write(∗,∗) ' spacedim = ', spacedim
           write(∗,∗) ' margin = ', margin
           write(∗,∗) ' delta = ', delta
           write(∗,∗) ' origin = ',
      & origin(0), ', ',
      & origin(1), ', ',
      & origin(2)
           write(∗,∗) ' iv_low = ',
      & iv_low_corner(0), ', ',
      & iv_low_corner(1), ', ',
      & iv_low_corner(2)
           write(∗,∗) ' iv_hi = ',
      & iv_high_corner(0), ', ',
      & iv_high_corner(1), ', ',
      & iv_high_corner(2)
```

```
          write(*,*) ' center = ',
     & center(0), ', ',
     & center(1), ', ',
     & center(2)
          write(*,*) ' radius = ',
     & radius
          write(*,*) ' type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
       end if
C
       i_min = max(i_lo + margin, iv_low_corner(0))
       j_min = max(j_lo + margin, iv_low_corner(1))
       k_min = max(k_lo + margin, iv_low_corner(2))
C
       i_max = min(i_hi - margin, iv_high_corner(0))
       j_max = min(j_hi - margin, iv_high_corner(1))
       k_max = min(k_hi - margin, iv_high_corner(2))
C
       count = 0
       r2 = radius**2
C
       do k = k_min, k_max
          z = origin(2) + (k - 0.5 * box_type(2)) * delta
          dz2 = (center(2) - z)**2
          do j = j_min, j_max
             y = origin(1) + (j - 0.5 * box_type(1)) * delta
             dy2 = (center(1) - y)**2
             do i = i_min, i_max
                x = origin(0) + (i - 0.5 * box_type(0)) * delta
                dx2 = (center(0) - x)**2
                d2 = (dx2 + dy2 + dz2)
                if (d2 .le. r2) then
                   medium(i,j,k) = color
                   count = count + 1
                end if
             end do
          end do
       end do
C
       if (verbosity .gt. 0) then
          write(*,*) ' marked ', count, ' cells'
       end if
C
 100 continue
C
       end
C
C Draw an ellipsoid on Chombo arrays
C
C C++ declaration on Forms.H
C
C extern"C"
C {
C void f_draw_ellipsoid_(
C const int*const SpaceDim,
C const int*const margin,
C const Real*const origin,
```

```
C const Real*const delta,
C const int*const iv_lower_corner,
C const int*const iv_upper_corner,
C const Real*const focus_1,
C const Real*const focus_2,
C const Real*const sum,
C const int*const color,
C const int*const box_type,
C const int*const i_lo,
C const int*const j_lo,
C const int*const k_lo,
C const int*const i_hi,
C const int*const j_hi,
C const int*const k_hi,
C const int*const medium,
C const int*const verbosity,
C const int*const return_status
C );
C }
C
      subroutine f_draw_ellipsoid(
     & spacedim, margin,
     & origin, delta,
     & iv_low_corner, iv_high_corner,
     & focus_1, focus_2, sum, color,
     & box_type, i_lo, j_lo, k_lo, i_hi, j_hi, k_hi,
     & medium,
     & verbosity, return_status)
      implicit none
C
      integer spacedim, margin
      double precision origin(0:spacedim - 1), delta
      integer
     & iv_low_corner(0:spacedim - 1),
     & iv_high_corner(0:spacedim - 1)
      double precision
     & focus_1(0:spacedim - 1),
     & focus_2(0:spacedim - 1)
      double precision sum
      integer color
      integer
     & box_type(0:spacedim - 1)
      integer i_lo, j_lo, k_lo, i_hi, j_hi, k_hi
      integer
     & medium(i_lo:i_hi, j_lo:j_hi, k_lo:k_hi)
      integer verbosity, return_status
C
C internal variables
C
      integer i_min, j_min, k_min
      integer i_max, j_max, k_max
      integer i, j, k
      double precision x, y, z
      double precision dx1sqr, dy1sqr, dz1sqr
      double precision dx2sqr, dy2sqr, dz2sqr
      double precision dd
      double precision interfocal, eccentricity
      integer count
C
```

```
C Action
C
      if (spacedim .ne. 3) then
         write(*,*) 'ERROR(f_draw_ellipsoid_): ',
     & 'bad dimension'
         return_status = 0
         goto 100
      end if
C
      interfocal =
     & sqrt((focus_2(0) - focus_1(0))**2
     & + (focus_2(1) - focus_1(1))**2
     & + (focus_2(2) - focus_1(2))**2)
      eccentricity = interfocal/sum
C
      if (verbosity .gt. 0) then
         write(*,*) ' f_draw_ellipsoid_:'
         write(*,*) ' spacedim = ', spacedim
         write(*,*) ' margin = ', margin
         write(*,*) ' delta = ', delta
         write(*,*) ' origin = ',
     & origin(0), ', ',
     & origin(1), ', ',
     & origin(2)
         write(*,*) ' iv_low = ',
     & iv_low_corner(0), ', ',
     & iv_low_corner(1), ', ',
     & iv_low_corner(2)
         write(*,*) ' iv_hi = ',
     & iv_high_corner(0), ', ',
     & iv_high_corner(1), ', ',
     & iv_high_corner(2)
         write(*,*) ' focus_1 = ',
     & focus_1(0), ', ',
     & focus_1(1), ', ',
     & focus_1(2)
         write(*,*) ' focus_2 = ',
     & focus_2(0), ', ',
     & focus_2(1), ', ',
     & focus_2(2)
         write(*,*) ' sum = ',
     & sum
         write(*,*) ' interfcl = ',
     & interfocal
         write(*,*) ' eccentr = ',
     & eccentricity
         write(*,*) ' type = ',
     & box_type(0), ', ',
     & box_type(1), ', ',
     & box_type(2)
      end if
C
      if (eccentricity .ge. 1) then
         write(*,*) 'ERROR(f_draw_ellipsoid_): ',
     & 'bad eccentricity'
         return_status = 0
         goto 100
      end if
C
```

```
      i_min = max(i_lo + margin, iv_low_corner(0))
      j_min = max(j_lo + margin, iv_low_corner(1))
      k_min = max(k_lo + margin, iv_low_corner(2))
C
      i_max = min(i_hi − margin, iv_high_corner(0))
      j_max = min(j_hi − margin, iv_high_corner(1))
      k_max = min(k_hi − margin, iv_high_corner(2))
C
      count = 0
C
      do k = k_min, k_max
         z = origin(2) + (k − 0.5 ∗ box_type(2)) ∗ delta
         dz1sqr = (focus_1(2) − z)**2
         dz2sqr = (focus_2(2) − z)**2
         do j = j_min, j_max
            y = origin(1) + (j − 0.5 ∗ box_type(1)) ∗ delta
            dy1sqr = (focus_1(1) − y)**2
            dy2sqr = (focus_2(1) − y)**2
            do i = i_min, i_max
               x = origin(0) + (i − 0.5 ∗ box_type(0)) ∗ delta
               dx1sqr = (focus_1(0) − x)**2
               dx2sqr = (focus_2(0) − x)**2
               dd = sqrt(dz1sqr + dy1sqr + dx1sqr)
     & + sqrt(dz2sqr + dy2sqr + dx2sqr)
               if (dd .le. sum) then
                  medium(i,j,k) = color
                  count = count + 1
               end if
            end do
         end do
      end do
C
      if (verbosity .gt. 0) then
         write(∗,∗) ' marked ', count, ' cells'
      end if
C
  100 continue
C
      end
```

13.6 Subroutine `inject_d`

Inject the $\boldsymbol{D}$ field into the total field region, on a specified face.

```
C
C Fortran subroutine inject_d
C
C $Id: inject_d.f,v 1.5 2007/10/02 21:04:24 gustav Exp $
C
      subroutine inject_d(
     & dir, side, margin, new_component,
     & x_0, y_0, z_0, delta,
     & dt_by_dg, time_h,
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi,
     & i_D1_lo, j_D1_lo, k_D1_lo,
     & i_D1_hi, j_D1_hi, k_D1_hi, D1,
     & i_D2_lo, j_D2_lo, k_D2_lo,
     & i_D2_hi, j_D2_hi, k_D2_hi, D2,
     & watch_inject_d, return_status)
      implicit none
C
C passed variables
C
      integer dir, side, margin, new_component
      double precision x_0, y_0, z_0, delta
      double precision dt_by_dg, time_h
      integer
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi
      integer
     & i_D1_lo, j_D1_lo, k_D1_lo,
     & i_D1_hi, j_D1_hi, k_D1_hi
      double precision
     & D1(i_D1_lo:i_D1_hi, j_D1_lo:j_D1_hi, k_D1_lo:k_D1_hi, 0:1)
      integer
     & i_D2_lo, j_D2_lo, k_D2_lo,
     & i_D2_hi, j_D2_hi, k_D2_hi
      double precision
     & D2(i_D2_lo:i_D2_hi, j_D2_lo:j_D2_hi, k_D2_lo:k_D2_hi, 0:1)
      integer watch_inject_d, return_status
C
C local variables
C
      integer
     & i1_min, j1_min, k1_min,
     & i1_max, j1_max, k1_max,
     & i2_min, j2_min, k2_min,
     & i2_max, j2_max, k2_max
      integer i_min, i_max, j_min, j_max, k_min, k_max
      integer i, j, k, i_D, j_D, k_D, factor
      integer i_H, j_H, k_H
      double precision x_H, y_H, z_H
      integer lo, hi
      parameter (lo = 0, hi = 1)
      integer ERROR_IN_INJECT_D
      parameter (ERROR_IN_INJECT_D = 10)
      integer Hx_box_type(3), Hy_box_type(3), Hz_box_type(3)
      data (Hx_box_type(i), i = 1,3) /1, 0, 0/
      data (Hy_box_type(i), i = 1,3) /0, 1, 0/
```

```
      data (Hz_box_type(i), i = 1,3) /0, 0, 1/
C
C external function
C
      double precision Hx_inc, Hy_inc, Hz_inc
C
C action
C
C 1. Find the real index ranges for the two fields provided.
C
      i1_min = i_D1_lo + margin;
      j1_min = j_D1_lo + margin;
      k1_min = k_D1_lo + margin;
      i1_max = i_D1_hi − margin;
      j1_max = j_D1_hi − margin;
      k1_max = k_D1_hi − margin;

      i2_min = i_D2_lo + margin;
      j2_min = j_D2_lo + margin;
      k2_min = k_D2_lo + margin;
      i2_max = i_D2_hi − margin;
      j2_max = j_D2_hi − margin;
      k2_max = k_D2_hi − margin;
C
C 2. Switch the action depending on dir
C a. Select the fixed index and afix it
C b. Select the sign of the correction
C c. Select the loop ranges for this field
C d. Loop over all cells of the wall and tweak the field
C
      select case (dir)
C
C West−−East
C
      case (0)

         select case (side)
         case (lo)
            i_D = i_sgnl_lo
            i_H = i_sgnl_lo
            factor = 1
         case (hi)
            i_D = i_sgnl_hi + 1
            i_H = i_sgnl_hi
            factor = −1
         case default
            write(∗,∗) "Bad side: ", side
            return_status = ERROR_IN_INJECT_D
            goto 100
         end select

         j_min = max(j1_min, j_sgnl_lo)
         j_max = min(j1_max, j_sgnl_hi)
         k_min = max(k1_min, k_sgnl_lo + 1)
         k_max = min(k1_max, k_sgnl_hi)
         x_H = x_0 + (i_H − 0.5 ∗ Hz_box_type(1)) ∗ delta
         do k = k_min, k_max
            z_H = z_0 + (k − 0.5 ∗ Hz_box_type(3)) ∗ delta
            do j = j_min, j_max
```

```
                y_H = y_0 + (j − 0.5 ∗ Hz_box_type(2)) ∗ delta
                D1(i_D, j, k, new_component)
     & = D1(i_D, j, k, new_component)
     & + factor ∗ dt_by_dg ∗ Hz_inc(x_H, y_H, z_H, time_h)
          end do
        end do

        j_min = max(j2_min, j_sgnl_lo + 1)
        j_max = min(j2_max, j_sgnl_hi)
        k_min = max(k2_min, k_sgnl_lo)
        k_max = min(k2_max, k_sgnl_hi)
        x_H = x_0 + (i_H − 0.5 ∗ Hy_box_type(1)) ∗ delta
        do k = k_min, k_max
          z_H = z_0 + (k − 0.5 ∗ Hy_box_type(3)) ∗ delta
          do j = j_min, j_max
            y_H = y_0 + (j − 0.5 ∗ Hy_box_type(2)) ∗ delta
            D2(i_D, j, k, new_component)
     & = D2(i_D, j, k, new_component)
     & − factor ∗ dt_by_dg ∗ Hy_inc(x_H, y_H, z_H, time_h)
          end do
        end do
C
C Front−−Back
C
      case (1)

        select case (side)
        case (lo)
          j_D = j_sgnl_lo
          j_H = j_sgnl_lo
          factor = −1
        case (hi)
          j_D = j_sgnl_hi + 1
          j_H = j_sgnl_hi
          factor = 1
        case default
          write(∗,∗) "Bad side: ", side
          return_status = ERROR_IN_INJECT_D
          goto 100
        end select

        i_min = max(i1_min, i_sgnl_lo)
        i_max = min(i1_max, i_sgnl_hi)
        k_min = max(k1_min, k_sgnl_lo + 1)
        k_max = min(k1_max, k_sgnl_hi)
        y_H = y_0 + (j_H − 0.5 ∗ Hz_box_type(2)) ∗ delta
        do k = k_min, k_max
          z_H = z_0 + (k − 0.5 ∗ Hz_box_type(3)) ∗ delta
          do i = i_min, i_max
            x_H = x_0 + (i − 0.5 ∗ Hz_box_type(1)) ∗ delta
            D1(i, j_D, k, new_component)
     & = D1(i, j_D, k, new_component)
     & + factor ∗ dt_by_dg ∗ Hz_inc(x_H, y_H, z_H, time_h)
          end do
        end do

        i_min = max(i2_min, i_sgnl_lo + 1)
        i_max = min(i2_max, i_sgnl_hi)
        k_min = max(k2_min, k_sgnl_lo)
```

```
          k_max = min(k2_max, k_sgnl_hi)
          y_H = y_0 + (j_H − 0.5 ∗ Hx_box_type(2)) ∗ delta
          do k = k_min, k_max
             z_H = z_0 + (k − 0.5 ∗ Hx_box_type(3)) ∗ delta
             do i = i_min, i_max
                x_H = x_0 + (i − 0.5 ∗ Hx_box_type(1)) ∗ delta
                D2(i, j_D, k, new_component)
     & = D2(i, j_D, k, new_component)
     & − factor ∗ dt_by_dg ∗ Hx_inc(x_H, y_H, z_H, time_h)
             end do
          end do
C
C Bottom−−Top
C
       case (2)

          select case (side)
          case (lo)
             k_D = k_sgnl_lo
             k_H = k_sgnl_lo
             factor = 1
          case (hi)
             k_D = k_sgnl_hi + 1
             k_H = k_sgnl_hi
             factor = −1
          case default
             write(∗,∗) "Bad side: ", side
             return_status = ERROR_IN_INJECT_D
             goto 100
          end select

          i_min = max(i1_min, i_sgnl_lo)
          i_max = min(i1_max, i_sgnl_hi)
          j_min = max(j1_min, j_sgnl_lo + 1)
          j_max = min(j1_max, j_sgnl_hi)
          z_H = z_0 + (k_H − 0.5 ∗ Hy_box_type(3)) ∗ delta
          do j = j_min, j_max
             y_H = y_0 + (j − 0.5 ∗ Hy_box_type(2)) ∗ delta
             do i = i_min, i_max
                x_H = x_0 + (i − 0.5 ∗ Hy_box_type(1)) ∗ delta
                D1(i, j, k_D, new_component)
     & = D1(i, j, k_D, new_component)
     & + factor ∗ dt_by_dg ∗ Hy_inc(x_H, y_H, z_H, time_h)
             end do
          end do

          i_min = max(i2_min, i_sgnl_lo + 1)
          i_max = min(i2_max, i_sgnl_hi)
          j_min = max(j2_min, j_sgnl_lo)
          j_max = min(j2_max, j_sgnl_hi)
          z_H = z_0 + (k_H − 0.5 ∗ Hx_box_type(3)) ∗ delta
          do j = j_min, j_max
             y_H = y_0 + (j − 0.5 ∗ Hx_box_type(2)) ∗ delta
             do i = i_min, i_max
                x_H = x_0 + (i − 0.5 ∗ Hx_box_type(1)) ∗ delta
                D2(i, j, k_D, new_component)
     & = D2(i, j, k_D, new_component)
     & − factor ∗ dt_by_dg ∗ Hx_inc(x_H, y_H, z_H, time_h)
             end do
```

```
        end do
C
C Error: no such direction
C
      case default
        write(*,*) "Bad direction: ", dir
        return_status = ERROR_IN_INJECT_D
        goto 100
      end select

100 continue

      end
```

13.7 Subroutine `inject_b`

Inject the $\boldsymbol{B}$ field into the total field region, on a specified face.

```
C
C Fortran subroutine inject_b
C
C $Id: inject_b.f,v 1.5 2007/10/02 21:13:08 gustav Exp $
C
      subroutine inject_b(
     & dir, side, margin, new_component,
     & x_0, y_0, z_0, delta,
     & dt_by_dg, time_e,
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi,
     & i_B1_lo, j_B1_lo, k_B1_lo,
     & i_B1_hi, j_B1_hi, k_B1_hi, B1,
     & i_B2_lo, j_B2_lo, k_B2_lo,
     & i_B2_hi, j_B2_hi, k_B2_hi, B2,
     & watch_inject_b, return_status)
      implicit none
C
C passed variables
C
      integer dir, side, margin, new_component
      double precision x_0, y_0, z_0, delta
      double precision dt_by_dg, time_e
      integer
     & i_sgnl_lo, j_sgnl_lo, k_sgnl_lo,
     & i_sgnl_hi, j_sgnl_hi, k_sgnl_hi
      integer
     & i_B1_lo, j_B1_lo, k_B1_lo,
     & i_B1_hi, j_B1_hi, k_B1_hi
      double precision
     & B1(i_B1_lo:i_B1_hi, j_B1_lo:j_B1_hi, k_B1_lo:k_B1_hi, 0:1)
      integer
     & i_B2_lo, j_B2_lo, k_B2_lo,
     & i_B2_hi, j_B2_hi, k_B2_hi
      double precision
     & B2(i_B2_lo:i_B2_hi, j_B2_lo:j_B2_hi, k_B2_lo:k_B2_hi, 0:1)
      integer watch_inject_b, return_status
C
C local variables
C
      integer
     & i1_min, j1_min, k1_min,
     & i1_max, j1_max, k1_max,
     & i2_min, j2_min, k2_min,
     & i2_max, j2_max, k2_max
      integer i_min, i_max, j_min, j_max, k_min, k_max
      integer i, j, k, i_B, j_B, k_B, factor
      integer i_E, j_E, k_E
      double precision x_E, y_E, z_E
      integer lo, hi
      parameter (lo = 0, hi = 1)
      integer ERROR_IN_INJECT_B
      parameter (ERROR_IN_INJECT_B = 10)
      integer Ex_box_type(3), Ey_box_type(3), Ez_box_type(3)
      data (Ex_box_type(i), i = 1,3) /0, 1, 1/
      data (Ey_box_type(i), i = 1,3) /1, 0, 1/
```

```
      data (Ez_box_type(i), i = 1,3) /1, 1, 0/
C
C external function
C
      double precision Ex_inc, Ey_inc, Ez_inc
C
C action
C
C 1. Find the real index ranges for the two fields provided.
C
      i1_min = i_B1_lo + margin;
      j1_min = j_B1_lo + margin;
      k1_min = k_B1_lo + margin;
      i1_max = i_B1_hi − margin;
      j1_max = j_B1_hi − margin;
      k1_max = k_B1_hi − margin;

      i2_min = i_B2_lo + margin;
      j2_min = j_B2_lo + margin;
      k2_min = k_B2_lo + margin;
      i2_max = i_B2_hi − margin;
      j2_max = j_B2_hi − margin;
      k2_max = k_B2_hi − margin;
      select case (dir)
C
C West−−East
C
      case (0)

         select case (side)
         case (lo)
            i_B = i_sgnl_lo
            i_E = i_sgnl_lo
            factor = −1
         case (hi)
            i_B = i_sgnl_hi
            i_E = i_sgnl_hi + 1
            factor = 1
         case default
            write(∗,∗) "Bad side: ", side
            return_status = ERROR_IN_INJECT_B
            goto 100
         end select

         j_min = max(j1_min, j_sgnl_lo + 1)
         j_max = min(j1_max, j_sgnl_hi)
         k_min = max(k1_min, k_sgnl_lo)
         k_max = min(k1_max, k_sgnl_hi)
         x_E = x_0 + (i_E − 0.5 ∗ Ez_box_type(1)) ∗ delta
         do k = k_min, k_max
            z_E = z_0 + (k − 0.5 ∗ Ez_box_type(3)) ∗ delta
            do j = j_min, j_max
               y_E = y_0 + (j − 0.5 ∗ Ez_box_type(2)) ∗ delta
               B1(i_B, j, k, new_component)
     & = B1(i_B, j, k, new_component)
     & + factor ∗ dt_by_dg ∗ Ez_inc(x_E, y_E, z_E, time_e)
            end do
         end do
```

```
          j_min = max(j2_min, j_sgnl_lo)
          j_max = min(j2_max, j_sgnl_hi)
          k_min = max(k2_min, k_sgnl_lo + 1)
          k_max = min(k2_max, k_sgnl_hi)
          x_E = x_0 + (i_E − 0.5 ∗ Ey_box_type(1)) ∗ delta
          do k = k_min, k_max
             z_E = z_0 + (k − 0.5 ∗ Ey_box_type(3)) ∗ delta
             do j = j_min, j_max
                y_E = y_0 + (j − 0.5 ∗ Ey_box_type(2)) ∗ delta
                B2(i_B, j, k, new_component)
     & = B2(i_B, j, k, new_component)
     & − factor ∗ dt_by_dg ∗ Ey_inc(x_E, y_E, z_E, time_e)
             end do
          end do
C
C Front−Back
C
       case (1)

          select case (side)
          case (lo)
             j_B = j_sgnl_lo
             j_E = j_sgnl_lo
             factor = 1
          case (hi)
             j_B = j_sgnl_hi
             j_E = j_sgnl_hi + 1
             factor = −1
          case default
             write(∗,∗) "Bad side: ", side
             return_status = ERROR_IN_INJECT_B
             goto 100
          end select

          i_min = max(i1_min, i_sgnl_lo + 1)
          i_max = min(i1_max, i_sgnl_hi)
          k_min = max(k1_min, k_sgnl_lo)
          k_max = min(k1_max, k_sgnl_hi)
          y_E = y_0 + (j_E − 0.5 ∗ Ez_box_type(2)) ∗ delta
          do k = k_min, k_max
             z_E = z_0 + (k − 0.5 ∗ Ez_box_type(3)) ∗ delta
             do i = i_min, i_max
                x_E = x_0 + (i − 0.5 ∗ Ez_box_type(1)) ∗ delta
                B1(i, j_B, k, new_component)
     & = B1(i, j_B, k, new_component)
     & + factor ∗ dt_by_dg ∗ Ez_inc(x_E, y_E, z_E, time_e)
             end do
          end do

          i_min = max(i2_min, i_sgnl_lo)
          i_max = min(i2_max, i_sgnl_hi)
          k_min = max(k2_min, k_sgnl_lo + 1)
          k_max = min(k2_max, k_sgnl_hi)
          y_E = y_0 + (j_E − 0.5 ∗ Ex_box_type(2)) ∗ delta
          do k = k_min, k_max
             z_E = z_0 + (k − 0.5 ∗ Ex_box_type(3)) ∗ delta
             do i = i_min, i_max
                x_E = x_0 + (i − 0.5 ∗ Ex_box_type(1)) ∗ delta
                B2(i, j_B, k, new_component)
```

```
     & = B2(i, j_B, k, new_component)
     & - factor * dt_by_dg * Ex_inc(x_E, y_E, z_E, time_e)
           end do
         end do
C
C Bottom--Top
C
      case (2)

         select case (side)
         case (lo)
            k_B = k_sgnl_lo
            k_E = k_sgnl_lo
            factor = -1
         case (hi)
            k_B = k_sgnl_hi
            k_E = k_sgnl_hi + 1
            factor = 1
         case default
            write(*,*) "Bad side: ", side
            return_status = ERROR_IN_INJECT_B
            goto 100
         end select

         i_min = max(i1_min, i_sgnl_lo + 1)
         i_max = min(i1_max, i_sgnl_hi)
         j_min = max(j1_min, j_sgnl_lo)
         j_max = min(j1_max, j_sgnl_hi)
         z_E = z_0 + (k_E - 0.5 * Ey_box_type(3)) * delta
         do j = j_min, j_max
            y_E = y_0 + (j - 0.5 * Ey_box_type(2)) * delta
            do i = i_min, i_max
               x_E = x_0 + (i - 0.5 * Ey_box_type(1)) * delta
               B1(i, j, k_B, new_component)
     & = B1(i, j, k_B, new_component)
     & + factor * dt_by_dg * Ey_inc(x_E, y_E, z_E, time_e)
           end do
         end do

         i_min = max(i2_min, i_sgnl_lo)
         i_max = min(i2_max, i_sgnl_hi)
         j_min = max(j2_min, j_sgnl_lo + 1)
         j_max = min(j2_max, j_sgnl_hi)
         z_E = z_0 + (k_E - 0.5 * Ex_box_type(3)) * delta
         do j = j_min, j_max
            y_E = y_0 + (j - 0.5 * Ex_box_type(2)) * delta
            do i = i_min, i_max
               x_E = x_0 + (i - 0.5 * Ex_box_type(1)) * delta
               B2(i, j, k_B, new_component)
     & = B2(i, j, k_B, new_component)
     & - factor * dt_by_dg * Ex_inc(x_E, y_E, z_E, time_e)
           end do
         end do
C
C Error: no such direction
C
      case default
         write(*,*) "Bad direction: ", dir
         return_status = ERROR_IN_INJECT_B
```

```
        goto 100
      end select

100 continue

      end
```

13.8 Output Subroutines

Two subroutines live on this file so far. More will be added. They are *from_edge_to_center_* and *from_face_to_center_*.

```
C Fortran subroutines used in output operations.
C
C $Id: output.f,v 1.4 2008/04/06 00:05:09 gustav Exp $
C
      subroutine from_edge_to_center(
     & spacedim, dir, n_out, count, idx,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi, fld,
     & watch, status)
      implicit none
C
      integer spacedim, dir, n_out, count, idx, watch, status
C
      integer
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
      double precision out(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_out - 1)
C
      integer
     & i_fld_lo, j_fld_lo, k_fld_lo,
     & i_fld_hi, j_fld_hi, k_fld_hi
      double precision fld(
     & i_fld_lo:i_fld_hi,
     & j_fld_lo:j_fld_hi,
     & k_fld_lo:k_fld_hi, 0:1)
C
      integer i, j, k
      integer ERROR_IN_EDGE
      parameter (ERROR_IN_EDGE = 83)
C
      if ((count .gt. n_out - 1) .or. (count .lt. 0)) then
         write(*,*) 'Bad output field count'
         status = ERROR_IN_EDGE
         goto 100
      end if
      if ((idx .lt. 0) .or. (idx .gt. 1)) then
         write(*,*) 'Bad field index'
         status = ERROR_IN_EDGE
         goto 100
      end if
      if ((i_out_lo .lt. i_fld_lo)
     & .or. (j_out_lo .lt. j_fld_lo)
     & .or. (k_out_lo .lt. k_fld_lo)) then
         write(*,*) 'Bad low index on field'
         status = ERROR_IN_EDGE
         goto 100
      end if
C
      select case (dir)
      case (0)
```

```
            if ((j_out_hi + 1 .gt. j_fld_hi)
     & .or. (k_out_hi + 1 .gt. k_fld_hi)) then
               write(*,*) 'Bad high index on field'
               status = ERROR_IN_EDGE
               goto 100
            end if
            do k = k_out_lo, k_out_hi
               do j = j_out_lo, j_out_hi
                  do i = i_out_lo, i_out_hi
                     out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,idx) + fld(i,j+1,k,idx)
     & + fld(i,j,k+1,idx) + fld(i,j+1,k+1,idx))
                  end do
               end do
            end do
          case (1)
            if ((i_out_hi + 1 .gt. i_fld_hi)
     & .or. (k_out_hi + 1 .gt. k_fld_hi)) then
               write(*,*) 'Bad high index on field'
               status = ERROR_IN_EDGE
               goto 100
            end if
            do k = k_out_lo, k_out_hi
               do j = j_out_lo, j_out_hi
                  do i = i_out_lo, i_out_hi
                     out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,idx) + fld(i+1,j,k,idx)
     & + fld(i,j,k+1,idx) + fld(i+1,j,k+1,idx))
                  end do
               end do
            end do
          case (2)
            if ((i_out_hi + 1 .gt. i_fld_hi)
     & .or. (j_out_hi + 1 .gt. j_fld_hi)) then
               write(*,*) 'Bad high index on field'
               status = ERROR_IN_EDGE
               goto 100
            end if
            do k = k_out_lo, k_out_hi
               do j = j_out_lo, j_out_hi
                  do i = i_out_lo, i_out_hi
                     out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,idx) + fld(i+1,j,k,idx)
     & + fld(i,j+1,k,idx) + fld(i+1,j+1,k,idx))
                  end do
               end do
            end do
          end select

  100 continue

          end
C
C
C
          subroutine from_face_to_center(
     & spacedim, dir, n_out, count, idx,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
```

```
      & i_fld_lo, j_fld_lo, k_fld_lo,
      & i_fld_hi, j_fld_hi, k_fld_hi, fld,
      & watch, status)
       implicit none
C
       integer spacedim, dir, n_out, count, idx, watch, status
C
       integer
      & i_out_lo, j_out_lo, k_out_lo,
      & i_out_hi, j_out_hi, k_out_hi
       double precision out(
      & i_out_lo:i_out_hi,
      & j_out_lo:j_out_hi,
      & k_out_lo:k_out_hi, 0:n_out - 1)
C
       integer
      & i_fld_lo, j_fld_lo, k_fld_lo,
      & i_fld_hi, j_fld_hi, k_fld_hi
       double precision fld(
      & i_fld_lo:i_fld_hi,
      & j_fld_lo:j_fld_hi,
      & k_fld_lo:k_fld_hi, 0:1)
C
       integer i, j, k
       integer ERROR_IN_FACE
       parameter (ERROR_IN_FACE = 84)
C
       if ((count .gt. n_out - 1) .or. (count .lt. 0)) then
          write(*,*) 'Bad output field count'
          status = ERROR_IN_FACE
          goto 100
       end if
       if ((idx .lt. 0) .or. (idx .gt. 1)) then
          write(*,*) 'Bad field index'
          status = ERROR_IN_FACE
          goto 100
       end if
       if ((i_out_lo .lt. i_fld_lo)
      & .or. (j_out_lo .lt. j_fld_lo)
      & .or. (k_out_lo .lt. k_fld_lo)) then
          write(*,*) 'Bad low index on field'
          status = ERROR_IN_FACE
          goto 100
       end if
C
C Face centered fields are magnetic and we interpolate
C them both in space and time. So we do not actually make
C any use in computations of the idx index. But we keep it
C here, so as not to change the calling interface.
C
       select case (dir)
       case (0)
          if (i_out_hi + 1 .gt. i_fld_hi) then
             write(*,*) 'Bad high index on field'
             status = ERROR_IN_FACE
             goto 100
          end if
          do k = k_out_lo, k_out_hi
             do j = j_out_lo, j_out_hi
```

```
                do i = i_out_lo, i_out_hi
                   out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,0) + fld(i+1,j,k,0)
     & + fld(i,j,k,1) + fld(i+1,j,k,1))
                end do
             end do
          end do
       case (1)
          if (j_out_hi + 1 .gt. j_fld_hi) then
             write(*,*) 'Bad high index on field'
             status = ERROR_IN_FACE
             goto 100
          end if
          do k = k_out_lo, k_out_hi
             do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                   out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,0) + fld(i,j+1,k,0)
     & + fld(i,j,k,1) + fld(i,j+1,k,1))
                end do
             end do
          end do
       case (2)
          if (k_out_hi + 1 .gt. k_fld_hi) then
             write(*,*) 'Bad high index on field'
             status = ERROR_IN_FACE
             goto 100
          end if
          do k = k_out_lo, k_out_hi
             do j = j_out_lo, j_out_hi
                do i = i_out_lo, i_out_hi
                   out(i,j,k,count) =
     & 0.25 * (fld(i,j,k,0) + fld(i,j,k+1,0)
     & + fld(i,j,k,1) + fld(i,j,k+1,1))
                end do
             end do
          end do
       end select

 100 continue

       end
C
C Evaluate energy density
C
       subroutine evaluate_energy(
     & spacedim, n_out, count,
     & Ex, Ey, Ez, Bx, By, Bz,
     & i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi, out,
     & watch, status)
       implicit none
C
       integer spacedim, n_out, count
       integer Ex, Ey, Ez, Bx, By, Bz
       integer i_out_lo, j_out_lo, k_out_lo
       integer i_out_hi, j_out_hi, k_out_hi
       double precision out(
     & i_out_lo:i_out_hi,
```

```
      & j_out_lo:j_out_hi,
      & k_out_lo:k_out_hi, 0:n_out − 1)
       integer watch, status
C
       integer i, j, k, ERROR_IN_ENERGY
       parameter (ERROR_IN_ENERGY = 85)
C
       if ((Ex .lt. 0) .or. (Ex .gt. n_out − 1)
      & .or. (Ey .lt. 0) .or. (Ey .gt. n_out − 1)
      & .or. (Ez .lt. 0) .or. (Ez .gt. n_out − 1)
      & .or. (Bx .lt. 0) .or. (Bx .gt. n_out − 1)
      & .or. (By .lt. 0) .or. (By .gt. n_out − 1)
      & .or. (Bz .lt. 0) .or. (Bz .gt. n_out − 1)) then
           write(∗,∗) 'Bad field index in evaluate_energy_'
           status = ERROR_IN_ENERGY
           goto 100
       end if
       do k = k_out_lo, k_out_hi
           do j = j_out_lo, j_out_hi
               do i = i_out_lo, i_out_hi
                   out(i,j,k,count) =
      & 0.5 ∗ (out(i,j,k,Ex)∗∗2
      & + out(i,j,k,Ey)∗∗2
      & + out(i,j,k,Ez)∗∗2
      & + out(i,j,k,Bx)∗∗2
      & + out(i,j,k,By)∗∗2
      & + out(i,j,k,Bz)∗∗2)
               end do
           end do
       end do

 100 continue

       end
C
C Evaluate energy flux
C
       subroutine evaluate_flow(
      & spacedim, n_out, count,
      & Ey, Bz, Ez, By,
      & i_out_lo, j_out_lo, k_out_lo,
      & i_out_hi, j_out_hi, k_out_hi, out,
      & watch, status)
       implicit none
C
       integer spacedim, n_out, count
       integer Ey, Bz, Ez, By
       integer i_out_lo, j_out_lo, k_out_lo
       integer i_out_hi, j_out_hi, k_out_hi
       double precision out(
      & i_out_lo:i_out_hi,
      & j_out_lo:j_out_hi,
      & k_out_lo:k_out_hi, 0:n_out − 1)
       integer watch, status
C
       integer i, j, k, ERROR_IN_FLOW
       parameter (ERROR_IN_FLOW = 86)
C
       if ((Ey .lt. 0) .or. (Ey .gt. n_out − 1)
```

```
     & .or. (Bz .lt. 0) .or. (Bz .gt. n_out − 1)
     & .or. (Ez .lt. 0) .or. (Ez .gt. n_out − 1)
     & .or. (By .lt. 0) .or. (By .gt. n_out − 1)) then
         write(∗,∗) 'Bad field index in evaluate_flow_'
         status = ERROR_IN_FLOW
         goto 100
       end if
       do k = k_out_lo, k_out_hi
         do j = j_out_lo, j_out_hi
           do i = i_out_lo, i_out_hi
             out(i,j,k,count) =
     & out(i,j,k,Ey) ∗ out(i,j,k,Bz)
     & − out(i,j,k,Ez) ∗ out(i,j,k,By)
           end do
         end do
       end do

 100 continue

       end
C
C
C
       subroutine output_field_copy(
     & spacedim, n_pcmp, idx_pcmp, n_outp, idx_outp,
     & i_out_lo, j_out_lo, k_out_lo, i_out_hi, j_out_hi, k_out_hi,
     & pcmp, outp,
     & watch, status)
       implicit none
C
       integer spacedim, n_pcmp, idx_pcmp, n_outp, idx_outp
       integer i_out_lo, j_out_lo, k_out_lo,
     & i_out_hi, j_out_hi, k_out_hi
       double precision
     & pcmp(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_pcmp − 1),
     & outp(
     & i_out_lo:i_out_hi,
     & j_out_lo:j_out_hi,
     & k_out_lo:k_out_hi, 0:n_outp − 1)
       integer watch, status
C
       integer i, j, k, ERROR_IN_COPY
       parameter (ERROR_IN_COPY = 87)
C
       if ((idx_pcmp .lt. 0) .or. (idx_pcmp .gt. n_pcmp − 1)) then
         write(∗,∗) 'Bad source index in output_field_copy_'
         status = ERROR_IN_COPY
         goto 100
       end if
C
       if ((idx_outp .lt. 0) .or. (idx_outp .gt. n_outp −1)) then
         write(∗,∗) 'Bad destination index in output_field_copy_'
         status = ERROR_IN_COPY
         goto 100
       end if
C
```

```
      do k = k_out_lo, k_out_hi
         do j = j_out_lo, j_out_hi
            do i = i_out_lo, i_out_hi
               outp(i, j, k, idx_outp) = pcmp(i, j, k, idx_pcmp)
            end do
         end do
      end do

100 continue

      end
```

14 Makefile and srcMakefile

The `Makefile` is simplified now to make Chombo happy.

```
# Makefile for a 3D FDTD Chombo Program "Forms".
#
# $Id: Makefile,v 1.60 2007/09/21 02:03:22 gustav Exp $
#
# My Cygwin locations
#
CHOMBO_HOME = /home/gustav/Chombo-2.0-Aug2007/lib3d.CYGWIN.g++.g77.OPT
#
# My Linux locations
#
# CHOMBO_HOME = /home/gustav/Chombo-2.0-Aug2007/lib3d.Linux.64.g++.g77.OPT
#
ebase = Forms
#
LibNames = AMRTools BoxTools
#
base_dir = ./
src_dirs = ./src
XTRALIBFLAGS = `guile-config link`
XTRACXXFLAGS = `guile-config compile`
INPUT = Input.scm

include $(CHOMBO_HOME)/mk/Make.example
```

I have offloaded all source management and documentation generation to `srcMakefile` listed below.

```
# Makefile for the Forms source.
#
# $Id: srcMakefile,v 1.51 2009/01/13 22:53:41 gustav Exp $
#
SRC_DEST = ./src
DOC_DEST = ./doc
CXX_SOURCES = Forms.cpp Initialize.cpp Iteration.cpp Scheme.cpp Auxiliary.cpp \
              SCMParmParse.cpp IO.cpp CurlBox.cpp
C_SOURCES = Injection.c Conversion.c
F_SOURCES = update_d.f update_b.f d_to_e.f b_to_h.f inject_d.f inject_b.f output.f draw.f
PY_SOURCES = script.py
SH_SOURCES = make_movie.sh
EXAMPLES = Examples/dielectric.scm Examples/distribution.scm Examples/signal.scm \
           Examples/waveform.gnplt Examples/dielectric.c Examples/include_graphic.tex \
           Examples/auxiliary-fields.cpp Examples/d-to-e-conversion.f \
           Examples/d-to-e-conversion.cpp Examples/d-to-e-conversion.c \
           Examples/create-s-array.cpp Examples/drude.c Examples/drude.scm \
           Examples/TestBall.scm Examples/inject.c Examples/TestBallC.scm \
           Examples/jazz_submit.sh Examples/Dust.scm Examples/domain_partition.cpp \
           Examples/Flux_1.scm Examples/FourierFlux.scm Examples/interpolate.scm \
           Examples/TestBallD.scm Examples/TestBallE.scm Examples/vinterpolate.scm \
           Examples/PlaneWave.scm Examples/Accuracy.txt Examples/PlaneWaveB.scm \
           Examples/Job-128.txt Examples/Job-256.txt Examples/WaveFlux.scm \
           Examples/FluxOutput.txt Examples/PowerFlux.scm Examples/PowerFlux-out-128.txt \
           Examples/FourierFlux.scm Examples/signal.eps Examples/FourierOutput.txt \
           Examples/FourierOutput-256.txt
LOCAL_HEADERS = Forms.H EdgeFab.H EdgeFabImplem.H FaceFab.H \
           FaceFabImplem.H CurlBox.H SCMParmParse.H Injection.h Conversion.h
SOURCES = $(CXX_SOURCES) $(C_SOURCES) $(F_SOURCES) $(LOCAL_HEADERS)
WEB_FILES = Forms.w Introduction.w Numerics.w Mainlines.w Initialize.w \
```

```
                Iteration.w Scheme.w IO.w Auxiliary.w Includes.w Extensions.w Fortran.w \
                Makefile.w Closing.w Bibliography.w
EPS_FILES = signal.eps exp-hahn.eps linear-gauss.eps linear-hann.eps shrink-exp-hahn.eps \
                shrink-linear-hahn.eps
WEB_OUTPUTS = Forms.c Initialize.c Iteration.c Scheme.c Auxiliary.c IO.c \
                CurlBox.c SCMParmParse.c \
                Forms.hw Initialize.hw Iteration.hw Scheme.hw Auxiliary.hw IO.hw \
                CurlBox.hw SCMParmParse.hw \
                Conversion.c_std Injection.c_std \
                Conversion.h_std Injection.h_std \
                EdgeFab.hw EdgeFabImplem.hw FaceFab.hw FaceFabImplem.hw
TEX_OUTPUTS = Forms.aux Forms.idx Forms.log Forms.scn Forms.toc Forms.tex Forms.dvi

INPUT = Input.scm

ASTYLE = astyle --style=ansi --break-blocks --pad=oper

all: raw doc src clean

raw: $(WEB_FILES) $(F_SOURCES)
        ctangle Forms.w

src: $(SOURCES)
        [ \! -d $(SRC_DEST) ] && mkdir $(SRC_DEST)
        mv $(SOURCES) $(SRC_DEST)
        mv ./src/Forms.cpp .
        mv ./src/Forms.H .

doc: $(EXAMPLES) $(F_SOURCES) $(EPS_FILES) $(PY_SOURCES) $(SH_SOURCES) Forms.pdf
        [ \! -d $(DOC_DEST) ] && mkdir $(DOC_DEST)
        mv Forms.pdf $(DOC_DEST)

Forms.pdf: $(WEB_FILES) $(EXAMPLES) $(F_SOURCES) $(PY_SOURCES) $(SH_SOURCES) Makefile

%.tex: %.w
        cweave $<

%.dvi: %.tex
        latex $<
        latex $<
        latex $<

%.pdf: %.dvi
        dvipdfm -p letter $<

%.cpp: %.c
        sed -e '/^#line/d' \
            -e '/^\/\*/d' \
            -e '/^$$/d' $< > $@
        $(ASTYLE) $@
        rm $@.orig

%.H: %.hw
        sed -e '/^#line/d' \
            -e '/^\/\*/d' \
            -e '/^$$/d' $< > $@
        $(ASTYLE) $@
        rm $@.orig
```

```
%.h: %.h_std
	sed -e '/^#line/d' \
	    -e '/^\/\*/d' \
	    -e '/^$$/d' $< > $@
	$(ASTYLE) $@
	rm $@.orig

%.c: %.c_std
	sed -e '/^#line/d' \
	    -e '/^\/\*/d' \
	    -e '/^$$/d' $< > $@
	$(ASTYLE) $@
	rm $@.orig

clean:
	rcsclean
	rm -f $(WEB_OUTPUTS)
	rm -f $(TEX_OUTPUTS)
	rm -f *.orig
	( cd Examples ; rcsclean )
	co Makefile

clobber: clean
	rcsclean
	rm -rf $(SRC_DEST)
	rm -rf $(DOC_DEST)
	rm -f Forms.pdf
	rm -f Forms.cpp Forms.H

#
# Special dependencies
#
SCMParmParse.hw: Extensions.w

SCMParmParse.c: Extensions.w

EdgeFab.hw: Extensions.w

EdgeFabImplem.hw: Extensions.w

FaceFab.hw: Extensions.w

FaceFabImplem.hw: Extensions.w

CurlBox.hw: Extensions.w

CurlBox.c: Extensions.w

Injection.c_std: Iteration.w

Injection.h_std: Iteration.w

Conversion.c_std: Iteration.w

Conversion.h_std: Iteration.w

Forms.hw: Includes.w

#
```

```
# $Log: srcMakefile,v $
# Revision 1.51 2009/01/13 22:53:41 gustav
# add more eps files.
#
# Revision 1.50 2008/06/09 01:38:40 gustav
# Added FourierOutput-256.txt to Examples.
#
# Revision 1.49 2008/06/08 21:28:12 gustav
# signal.scm -> signal.eps
#
# Revision 1.48 2008/06/08 21:23:21 gustav
# Added signal.eps to examples and FourierOutput, too.
#
# Revision 1.47 2008/06/08 20:34:08 gustav
# Added FourierFlux.scm to Examples.
#
# Revision 1.46 2008/06/07 19:23:36 gustav
# Added PowerFlux-out-128.txt to the examples.
#
# Revision 1.45 2008/06/07 19:01:22 gustav
# Added PowerFlux.scm to examples.
#
# Revision 1.44 2008/06/06 16:13:37 gustav
# Added WaveFlux.scm and FluxOutput.scm to Examples.
#
# Revision 1.43 2008/06/03 00:53:29 gustav
# Added Job-128.txt and Job-256.txt to Examples.
#
# Revision 1.42 2008/06/03 00:24:22 gustav
# Added PlaneWaveB.scm to Examples.
#
# Revision 1.41 2008/05/30 20:16:10 gustav
# Added PlaneWave.scm to Examples.
#
# Revision 1.40 2008/05/30 20:08:10 gustav
# Added accuracy.txt to examples.
#
# Revision 1.39 2008/05/29 20:36:43 gustav
# Added vinterpolate.scm to examples.
#
# Revision 1.38 2008/05/24 17:07:50 gustav
# added TestBallE.scm to examples.
#
# Revision 1.37 2008/05/16 20:33:04 gustav
# Added TestBallD.scm to examples.
#
# Revision 1.36 2008/05/09 22:23:32 gustav
# Added interpolate.scm to examples.
#
# Revision 1.35 2008/04/30 21:37:17 gustav
# Added FourierFlux.scm to Examples
#
# Revision 1.34 2008/04/30 17:44:26 gustav
# Added Flux_1.scm to Examples.
#
# Revision 1.33 2008/04/22 17:40:57 gustav
# Added domain_partition.cpp to examples.
#
# Revision 1.32 2008/04/21 18:45:43 gustav
```

```
# Added jazz_submit.sh and Dust.scm to Examples.
#
# Revision 1.31 2008/04/04 14:50:09 gustav
# Added rcsclean op to clobber: this should remove Makefile that's been put
# in the top directory by "clean".
#
# Revision 1.30 2008/04/04 14:48:50 gustav
# Moved extraction of Makefile to the clean target.
#
# Revision 1.29 2008/04/04 14:46:09 gustav
# Added the rule for getting Makefile out.
#
# Revision 1.28 2008/04/04 14:44:21 gustav
# Added Makefile to all target.
#
# Revision 1.27 2008/04/03 19:28:31 gustav
# Added inject.c and TestBallC.scm to Examples.
#
# Revision 1.26 2008/04/03 14:15:08 gustav
# Added TestBall.scm to the Examples list.
#
# Revision 1.25 2008/04/02 21:12:56 gustav
# Added drude.scm to Examples.
#
# Revision 1.24 2008/04/02 19:37:31 gustav
# Added drude.c to the Examples list.
#
# Revision 1.23 2008/04/02 17:32:04 gustav
# Added an rcsclean in the Examples directory.
#
# Revision 1.22 2008/04/02 16:48:39 gustav
# Added d-to-e-conversion.c and create-s-array.cpp to Examples.
#
# Revision 1.21 2008/04/02 15:45:22 gustav
# Added d-to-e-conversion.cpp to Examples.
#
# Revision 1.20 2008/04/02 15:31:03 gustav
# Added d-to-e-conversion.f to Examples.
#
# Revision 1.19 2008/04/02 14:55:41 gustav
# Added auxiliary-fields.cpp to Examples.
#
# Revision 1.18 2008/03/27 19:55:29 gustav
# Expanded the list of examples.
#
# Revision 1.17 2008/03/26 20:09:16 gustav
# Defined EXAMPLES.
#
# Revision 1.16 2008/03/20 18:54:25 gustav
# Added PY_SOURCES and SH_SOURCES.
#
# Revision 1.15 2008/03/14 21:05:50 gustav
# EPS_SOURCES had to be moved to the doc target together with F_FILES.
#
# Revision 1.14 2008/03/14 21:02:07 gustav
# Added EPS_FILES and made .tex depend on them and F_SOURCES.
#
# Revision 1.13 2008/02/28 19:28:40 gustav
# Changed the order of targets in all.
```

```
#
# Revision 1.12 2008/02/28 19:26:01 gustav
# Added a doc target to all.
#
# Revision 1.11 2008/02/14 20:12:59 gustav
# Added dependencies for Injection.h_std, Conversion.h_std, and Forms.hw.
#
# Revision 1.10 2008/02/14 20:03:59 gustav
# Added dependencies for Injection.c and Conversion.c.
#
# Revision 1.9 2008/02/14 19:57:10 gustav
# Added specific dependencies for files defined on Extensions.w
#
# Revision 1.8 2007/10/04 22:56:57 gustav
# Added draw.f to Fortran sources.
#
# Revision 1.7 2007/10/04 15:17:44 gustav
# Added a clause for making the doc directory and cleaning it away.
# Cleaned up the "move" clause.
#
# Revision 1.6 2007/09/21 14:17:01 gustav
# Replaced cweave with ctangle in the "raw" clause.
#
# Revision 1.5 2007/09/21 14:09:32 gustav
# added target "move" to all
# added "move" clause that moves Forms.cpp and Form.H back to the top directory,
# because this is how Chombo Makefiles want it.
# added clean up of Forms.cpp and Forms.H from the top directory.
#
# Revision 1.4 2007/09/20 18:46:35 gustav
# more minor changes.
#
# Revision 1.3 2007/09/20 18:36:44 gustav
# Fixed clean up problems.
#
# Revision 1.2 2007/09/20 18:34:14 gustav
# Had to add dependencies for making a doc.
#
# Revision 1.1 2007/09/20 18:25:42 gustav
# Initial revision
#
#
```

15 Closing Comments

Corea Septentrionalis nuntiavit se primum experimentum nucleare fecisse. Ita illa dictatura Stalinistica, pauper et inclusa, futura est nona in orbe terrarum potestas, cui arma nuclearia esse constat.

Experimentum Coreæ Septentrionalis a moderatoribus orbis terrarum et a Consilio Securitatis Nationum Unitarum severissime condemnatum est.

Americani novas contra Coream Septentrionalem sanctiones a Nationibus Unitis flagitaverunt, Coreani Meridiani nuntiaverunt se potestatem nuclearem sibi propinquam pati non posse.

Etiam Sinenses, qui fœderati fideles Coreanorum Septentrionalium esse solent, eos reprobaverunt.

Minister primarius Iaponiæ dixit illo experimento nucleari stabilitatem Asiæ Orientalis mutari.

Sinenses anno vergente plus miliardum trecenti miliones numerabantur.

Ex eis quadraginta quattuor centesimæ in urbibus vivebant. Politica unius infantis æquilibrium sexuum perturbavit, nam plures quam dimidium Sinensium (51.5%) sunt viri.

Magistratus censent post quindecim annos numerum virorum cælibum triginta milionibus maiorem quam feminarum apud Sinenses futurum esse.

References

[1] Dinshaw S. Balsara, "Divergence-Free Adaptive Mesh Refinement for Magnetohydrodynamics," Journal of Computational Physics, vol. 174, pp. 614–648, 2001.

[2] F. I. Baida and D. Van Labeke, "Three-dimensional structures for enhanced transmission through a metallic film: Annular aperture arrays," Physical Review B, Vol. 67, No. 15, p. 155314(7), 25 April 2003

[3] 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.

[4] R. Courant, K. Friedrichs and H. Lewy, "Über die partiellen Differenzengleichungen der mathematischen Physik," Mathematische Annalen, vol. 100, no. 1, pp. 32–74, 1928.

[5] David Drysdale, "Tutorial Introduction to Guile," `http://www.gnu.org/software/guile/docs/guile-tut/tutorial.html`, 2000.

[6] Richard P. Feynman, Robert B. Leighton and Matthew Sands, "The Feynman Lectures on Physics–Mainly Electromagnetism and Matter," Volume II, Addison-Wesley Publishing Company, Fourteenth Printing, 1981

[7] Stephen D. Gedney, "An Anisotropic Perfectly Matched Layer—Absorbing Medium for the Truncation of FDTD Lattices," IEEE Transactions on Antennas and Propagation, Vol. 44, No. 12, pp. 1630–1639, December 1996.

[8] Michael Gran, "How to extend C programs with Guile," `http://lonelycactus.com/guilebook/book1.html`, 2004

[9] Stephen K. Gray and Teobald Kupka, "Propagation of light in metallic nanowire arrays: Finite-difference time-domain studies of silver cylinders," Physical Review B, Vol. 68, No. 4, p. 045415(11), 16 July 2003

[10] Guile: "Project GNU's extension language," `http://www.gnu.org/software/guile/guile.html`

[11] P. B. Johnson and R. W. Christy, "Optical Constants of the Noble Metals," Physical Review B, Vol. 6, No. 12, pp. 4370–4379, 15 December 1972

[12] Tae-Woo Lee and Stephen K. Gray, "Subwavelength light bending by metal slit structures," Optics Express, Vol. 13, No. 24, p. 9652(8), 28 November 2005

[13] D. W. Lynch and W. R. Hunter, in "Handbook of Optical Constants of Solids," ed. E. D. Palic, pp. 350–357, Academic, Orlando 1985

[14] Dan Martin and Gustav Meglicki, "Chombo Class PiecewiseLinearFillPatchFace," Code Weave Report, Argonne National Laboratory, October 13, 2006, `http://www-unix.mcs.anl.gov/~meglicki/Shapes/Prolongate.pdf`

[15] Dan Martin and Gustav Meglicki, "Chombo Class LevelFluxRegisterEdge," Code Weave Report, Argonne National Laboratory, November 29, 2006, `http://www-unix.mcs.anl.gov/~meglicki/Shapes/Reflux.pdf`

[16] Jeffrey M. McMahon, Joel Henzie, Teri W. Odom, George C. Schatz, and Stephen K. Gray, "Tailoring the sensing capabilities of nanohole arrays in gold films with Rayleigh anomaly-surface plasmon polaritons," Optics Express, Vol. 15, No. 26, pp. 18119–18129, 24 December 2007

[17] Zdzisław Meglicki, Stephen K. Gray and Boyana Norris, "Multigrid FDTD with Chombo," Computer Physics Communications, vol. 176, no. 2, pp. 109–120, January 2007.

[18] Dennis M. Sullivan, "Electromagnetic Simulation Using the FDTD Method," IEEE Press Series on RF and Microwave Technology, IEEE Press, 2000.

[19] Allen Taflove and Susan C. Hagness, "Computational Electrodynamics: The Finite-Difference Time-Domain Method," Second Edition, Artech House, 2000.

[20] A. R. Zakharian, M. Brio, C. Dineen, and J. V. Moloney, "Second-Order Accurate FDTD Space and Time Grid Refinement Method in Three Space Dimensions," IEEE Photonics Technology Letters, Vol. 18, No. 11, pp. 1237–1239, June 1, 2006

[21] A. R. Zakharian, M. Brio, and J. V. Moloney, "FDTD based local mesh refinement method for Maxwell's equations in two space dimensions," Communications in Mathematical Sciences, vol. 2, no. 3, pp. 497–513, September 2004

[22] A. R. Zakharian, M. Brio, C. Dineen, and J. V. Moloney, "Stability of 2-D FDTD algorithms with local mesh refinement for Maxwell's equations," Communications in Mathematical Sciences, vol. 4, no. 2, pp. 345–374, June 2006

Index

Here is a list of the identifiers used, and the chunks where they appear. Underlined entries indicate the place of definition.

List of Refinements

⟨ Auxiliary Scheme Procedures 134 ⟩ Used in chunk 114.
⟨ `Auxiliary.c` 147 ⟩
⟨ Build a Multigrid 58 ⟩ Used in chunk 53.
⟨ `Conversion.c_std` 102, 103 ⟩ Cited in chunk 18.
⟨ `Conversion.h_std` 104 ⟩
⟨ `CurlBox.c` 257, 258, 259, 260 ⟩
⟨ `CurlBox.hw` 255 ⟩
⟨ `EdgeFab.hw` 219 ⟩ Cited in chunk 228.
⟨ `EdgeFabImplem.hw` 228 ⟩
⟨ `FaceFab.hw` 248 ⟩
⟨ `FaceFabImplem.hw` 250, 251, 252, 253 ⟩
⟨ Fill the Multigrid 59 ⟩ Used in chunk 53.
⟨ `Forms.hw` 173, 174, 175, 176, 177, 179, 181, 182, 184, 185, 186, 188, 189, 190 ⟩
⟨ Function *advance_b_0* 106 ⟩ Used in chunk 84.
⟨ Function *advance_b_n* 107 ⟩ Used in chunk 84.
⟨ Function *advance_b* 105 ⟩ Used in chunk 84.
⟨ Function *advance_d_0* 86 ⟩ Used in chunk 84.
⟨ Function *advance_d_n* 87 ⟩ Used in chunk 84.
⟨ Function *advance_d* 85 ⟩ Used in chunk 84.
⟨ Function *b_to_h_0* 112 ⟩ Used in chunk 84.
⟨ Function *b_to_h_n* 113 ⟩ Used in chunk 84.
⟨ Function *b_to_h* 111 ⟩ Used in chunk 84.
⟨ Function *build_levels* 62 ⟩ Cited in chunk 53. Used in chunk 61.
⟨ Function *complement* 168 ⟩ Used in chunk 147.
⟨ Function *d_to_e_0_0* 97 ⟩ Used in chunk 84.
⟨ Function *d_to_e_0_n* 98 ⟩ Cited in chunk 18. Used in chunk 84.
⟨ Function *d_to_e_0* 96 ⟩ Cited in chunk 18. Used in chunk 84.
⟨ Function *d_to_e_n_0* 100 ⟩ Used in chunk 84.
⟨ Function *d_to_e_n_n* 101 ⟩ Used in chunk 84.
⟨ Function *d_to_e_n* 99 ⟩ Used in chunk 84.
⟨ Function *d_to_e* 95 ⟩ Used in chunk 84.
⟨ Function *draw_ball* 121 ⟩ Used in chunk 114.
⟨ Function *draw_box* 115 ⟩ Cited in chunk 236. Used in chunk 114.
⟨ Function *draw_ellipsoid* 127 ⟩ Used in chunk 114.
⟨ Function *draw_media_ball* 125 ⟩ Used in chunk 121.
⟨ Function *draw_media_box* 119 ⟩ Used in chunk 115.
⟨ Function *draw_media_ellipsoid* 131 ⟩ Used in chunk 127.
⟨ Function *draw_tags_ball* 126 ⟩ Used in chunk 121.
⟨ Function *draw_tags_box* 120 ⟩ Used in chunk 115.
⟨ Function *draw_tags_ellipsoid* 132 ⟩ Used in chunk 127.
⟨ Function *dump_data* 139 ⟩ Cited in chunk 53. Used in chunk 138.
⟨ Function *effective_grid_bounds* 150, 151, 152, 153, 154, 155, 156, 157 ⟩ Cited in chunks 70 and 158. Used in chunk 147.
⟨ Function *fill_PML_arrays* 171 ⟩ Used in chunk 147.
⟨ Function *fill_levels* 74, 75, 76, 77, 78, 82, 83 ⟩ Cited in chunk 18. Used in chunk 61.
⟨ Function *inject_b* 108, 109 ⟩ Used in chunk 84.
⟨ Function *inject_d* 88, 89 ⟩ Used in chunk 84.
⟨ Function *is_a_pwr_of_two* 148 ⟩ Used in chunk 147.
⟨ Function *is_file_readable* 149 ⟩ Used in chunk 147.
⟨ Function *make_tag_set* 158, 159, 160, 161, 162, 163, 164, 165 ⟩ Cited in chunk 73. Used in chunk 147.
⟨ Function *mark_TFR_faces* 169, 170 ⟩ Used in chunk 147.
⟨ Function *mark_regions* 167 ⟩ Used in chunk 147.
⟨ Function *swap_idx* 172 ⟩ Used in chunk 147.
⟨ Function *upml_sigma* 166 ⟩ Used in chunk 147.

⟨ `IO.c` 138 ⟩
⟨ Initialize Scheme 137 ⟩ Cited in chunks 18 and 46. Used in chunk 114.
⟨ `Initialize.c` 61 ⟩
⟨ `Injection.c_std` 91, 92 ⟩
⟨ `Injection.h_std` 93 ⟩
⟨ Inner Main 53 ⟩ Cited in chunks 64 and 137. Used in chunk 52.
⟨ Interpolation 136 ⟩ Cited in chunks 13 and 33. Used in chunk 114.
⟨ Iterate 60 ⟩ Cited in chunk 49. Used in chunk 53.
⟨ `Iteration.c` 84 ⟩
⟨ MPI Functions 135 ⟩ Cited in chunks 13 and 46. Used in chunk 114.
⟨ Process an Input File 54, 55, 56 ⟩ Used in chunk 53.
⟨ `SCMParmParse.c` 198 ⟩
⟨ `SCMParmParse.hw` 194 ⟩
⟨ `Scheme.c` 114 ⟩
⟨ Structure level 191 ⟩ Cited in chunk 18. Used in chunk 176.
⟨ print *Ckx* arrays 79 ⟩ Used in chunk 78.
⟨ print *Cky* arrays 80 ⟩ Used in chunk 78.
⟨ print *Ckz* arrays 81 ⟩ Used in chunk 78.
⟨ **EdgeFab** template implementation 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 244, 245, 246 ⟩ Used in chunk 228.
⟨ **EdgeFab**: class template declaration 220 ⟩ Used in chunk 219.
⟨ **EdgeFab**: private members 227 ⟩ Used in chunk 220.
⟨ **EdgeFab**: protected members 226 ⟩ Used in chunk 220.
⟨ **EdgeFab**: public members 221, 222, 223, 224, 225 ⟩ Used in chunk 220.
⟨ **SCMParmParse** headers: constructors and destructors 195 ⟩ Used in chunk 194.
⟨ **SCMParmParse** headers: protected members 197 ⟩ Used in chunk 194.
⟨ **SCMParmParse** headers: queries and extractors 196 ⟩ Used in chunk 194.
⟨ **SCMParmParse** implementation: constructors and destructors 199 ⟩ Cited in chunk 6. Used in chunk 198.
⟨ **SCMParmParse** implementation: protected members 216, 217 ⟩ Used in chunk 198.
⟨ **SCMParmParse** implementation: queries and extractors 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215 ⟩ Used in chunk 198.
⟨ *build_levels*: build higher levels 71 ⟩ Used in chunk 62.
⟨ *build_levels*: build level 0 70 ⟩ Cited in chunks 19 and 69. Used in chunk 62.
⟨ *build_levels*: build next level 72, 73 ⟩ Cited in chunk 71. Used in chunk 71.
⟨ *build_levels*: read grid specifications 68, 69 ⟩ Used in chunk 62.
⟨ *build_levels*: variable definitions 63, 64, 65, 66, 67 ⟩ Cited in chunk 71. Used in chunk 62.
⟨ *draw_ball*: call appropriate utility 124 ⟩ Used in chunk 121.
⟨ *draw_ball*: introduce yourself 122 ⟩ Used in chunk 121.
⟨ *draw_ball*: read, check, and translate arguments 123 ⟩ Used in chunk 121.
⟨ *draw_box*: call appropriate utility 118 ⟩ Used in chunk 115.
⟨ *draw_box*: introduce yourself 116 ⟩ Used in chunk 115.
⟨ *draw_box*: read, check, and translate arguments 117 ⟩ Used in chunk 115.
⟨ *draw_ellipsoid*: call appropriate utility 130 ⟩ Used in chunk 127.
⟨ *draw_ellipsoid*: introduce yourself 128 ⟩ Used in chunk 127.
⟨ *draw_ellipsoid*: read, check, and translate arguments 129 ⟩ Used in chunk 127.
⟨ *dump_data*: extract field from precomputes 146 ⟩ Used in chunk 139.
⟨ *dump_data*: precompute field 141, 142, 143, 144, 145 ⟩ Used in chunk 139.