Angora User's Guide

A finite-difference time-domain (FDTD) electromagnetic simulation software
for version 0.10.1 and later, Feb 2012

İlker R. Çapoğlu

Table of Contents

Angora: A finite-difference time-domain simulation package

This is the user's guide for Angora, a software package that computes numerical solutions to electromagnetic radiation and scattering problems. It is based on the finite-difference time-domain (FDTD) method, which one of the most popular approaches for solving Maxwell's equations of electrodynamics.

1 Downloading

Angora is currently only available in source-code format for the **GNU/Linux operating system**. If you would like to port Angora to another operating system, please contact us at capoglu@angorafdtd.org. Contributions are always welcome.

The latest version of Angora can be found at http://angorafdtd.org.

2 Compilation and Installation

Angora is dependent on the following libraries: blitz++, libconfig, hdf5, and boost. These libraries should be installed on your system before Angora can be compiled. If possible, use the package manager for your specific GNU/Linux distribution (such as Synaptic in Ubuntu) to install the libraries directly from the package repository. Most major distributions provide these libraries in their package repositories. If you do not have root access to your system, you can install these libraries in your home directory. The installation instructions for the libraries usually provide detailed information on how to do this. For local installation, the usual trick is to set the installation path by specifying the `prefix` variable in the `Makefile`s. This is done either by using the '`--prefix=`*`local-path`*' option when calling the package's `configure` script, or customizing `make` at the final stage with the '`prefix=`*`local-path`*' command option.

Once the dependency libraries are installed, the Angora package is ready for compilation. Extract the package '`angora-`*`package-version`*`.tar.gz`' using `tar`, and enter the created directory:

```
johndoe@mysystem:~$ tar xvf angora-package-version.tar.gz
johndoe@mysystem:~$ cd angora-package-version
```

Run the `configure` script in this directory to create the `Makefile`s required to build the package:

```
johndoe@mysystem:~/angora-package-version$ ./configure
```

If any of the dependency libraries was installed in a local directory, then add the option '`--with-`*`library-name=local-path-to-library`*' to the above command line. For example, if the `blitz++` library was installed in '`/home/johndoe/blitz-0.9`', then the option to add is '`--with-blitz=/home/johndoe/blitz-0.9`'. Type '`./configure --help`' in the directory '`angora-`*`package-version`*' for information on specifying the paths to the other dependency libraries.

After the `configure` script finishes execution, compile and install Angora using the `make` command:

```
johndoe@mysystem:~/angora-package-version$ make
```

If your system has multiple cores, you can speed up the compilation by executing `make` in parallel. For example, you can use all 4 cores of your system by typing, instead of the above line,

```
johndoe@mysystem:~/angora-package-version$ make -j 4
```

This might take a couple of minutes, depending on your system. After `make` finishes, the executable `angora` will be located in the directory '`angora-`*`package-version`*'. If you wish to *install* the package globally so that it can be run from anywhere, type

```
johndoe@mysystem:~/angora-package-version$ sudo make install
```

Obviously, this requires super-user privileges on your system. By default, the package is installed in '`/usr/local`'; so the binary will reside in '`/usr/local/bin`'. If you don't have super-user privileges, you can install Angora in a local directory '`full-path-to-inst-dir`' by typing

```
johndoe@mysystem:~/angora-package-version$ make prefix=full-path-to-inst-
dir install
```

The location '`full-path-to-inst-dir`' should be an absolute path. After this, the binary `angora` will be located in the directory '`full-path-to-inst-dir/bin/`'.

2.1 Enabling MPI Support

Parallel execution on multiple processors or cores is supported by Angora, provided that the MPI (Message Passing Interface) libraries are installed on your system (e.g., OpenMPI or MPICH2 or other). This feature has to be enabled at compile time, and is *disabled* by default. You can enable MPI parallel execution in Angora by adding the option '`--with-mpi`' to the `configure` command line:

```
johndoe@mysystem:~/angora-package-version$ ./configure --with-mpi
```

For more information on launching Angora simulations on multiple processors or cores using MPI, see Section 3.1 [Parallel Execution], page 5.

2.2 Building the Documentation

The GNU `info` documentation for Angora is automatically built and installed by `make`. If you have the texi2html and latex2html utilities installed, you can create an HTML version of the Angora documentation by typing

```
johndoe@mysystem:~/angora-package-version$ make html
```

If you have the `texi2dvi` command available (provided as part of the GNU Texinfo package), you can also build a PDF version of the Angora documentation by typing

```
johndoe@mysystem:~/angora-package-version$ make pdf
```

Once built, both the HTML and PDF versions of the documentation will be located in the subdirectory '`doc/`'.

3 Execution

Angora operates by reading a text file, called the *configuration file*, that specifies the details of the simulation. Every aspect of the simulation is configured by a related *configuration variable* (or *variable* in short) in the configuration file; which comprises either a single line or a number of lines. In general, an Angora simulation is run by putting the name of the configuration file pertaining to the simulation as a command line option when calling the `angora` executable:

```
johndoe@mysystem:~/angora-package-version$ ./angora path-to-config-file
```

If the Angora executable is run without any command-line options, it looks for the configuration file named '`angora.cfg`' in the same directory from which the executable is run. See Chapter 5 [Configuration Variables], page 10, for details on configuration files.

3.1 Parallel Execution

If Angora is compiled with MPI support, then the standard MPI launcher (`mpirun`) can be used to execute the Angora binary `angora` in parallel:

```
johndoe@mysystem:~/angora-package-version$ mpirun -n num-of-processors
./angora path-to-config-file
```

For example, to run the simulation configured by '`mysimulation.cfg`' using Angora version `0.9` on 8 processors, one should type

```
johndoe@mysystem:~/angora-0.9$ mpirun -n 8 ./angora mysimulation.cfg
```

MPI support should be enabled in compile time in order to run simulations in parallel. For details, see Section 2.1 [Enabling MPI Support], page 4.

4 Configuration Format

Angora uses the libconfig library to read configuration variables regarding the simulation from a text file. The text file, called the *configuration file*, has to conform to the libconfig grammar; which is explained in greater detail at `http://www.hyperrealm.com/libconfig/libconfig_manual.html`. Here, we will provide the minimum information necessary to write configuration files for Angora simulations.

4.1 Variable Assignment

A variable in a configuration file is set using the following assignment:

```
name=value;
```

or:

```
name:value;
```

The trailing semicolon is required. Whitespace is not significant. Here, *name* is the name of the variable, and *value* is its value; which may be a scalar value, an array, a group, or a list. See Section 4.2 [Variable Types], page 6, for information on these value types.

The order in which variables are specified in the configuration file is insignificant, except within the `SimulationSpace` variable (see Section 5.4 [Simulation Space], page 16). The sub-variables of the `SimulationSpace` variable are processed in the order of appearance in the configuration file. This is necessary because the user needs to be able to control the order in which objects are placed in the grid, and predict the regions within an object that will be overwritten by another object.

4.2 Variable Types

Angora simulation variables can be assigned C++-type *scalar* values, as well as more complex values of type *group*, *array*, and *list*. The latter types are defined by the libconfig library. Some of the text in this section is copied verbatim from the libconfig manual. The `libconfig` library, along with its documentation, is distributed under the GNU Lesser Public License.

4.2.1 Integer Values

Integers can be represented in one of two ways: as a series of one or more decimal digits (‘`0`’ - ‘`9`’), with an optional leading sign character (‘`+`’ or ‘`-`’); or as a hexadecimal value consisting of the characters ‘`0x`’ followed by a series of one or more hexadecimal digits (‘`0`’ - ‘`9`’, ‘`A`’ - ‘`F`’, ‘`a`’ - ‘`f`’).

Examples:

```
n_sx = 3;
offset = -4;
address = 0xFFFF;
```

4.2.2 Floating-Point Values

Floating point values consist of a series of one or more digits, one decimal point, an optional leading sign character (‘`+`’ or ‘`-`’), and an optional exponent. An exponent consists of the letter ‘`E`’ or ‘`e`’, an optional sign character, and a series of one or more digits.

Except in special circumstances, floating-point values in Angora are read and processed in '`double`' precision, which corresponds to roughly 15 decimal digits.

Examples:

```
f = 1.0;
origin = -3e-6;
prefactor = 5E10;
```

4.2.3 Boolean Values

Boolean values may have one of the following values: '`true`', '`false`', or any mixed-case variation thereof.

Examples:

```
include_first_value = true;
include_last_value = FaLsE;
```

4.2.4 String Values

String values consist of arbitrary text delimited by double quotes. Literal double quotes can be escaped by preceding them with a backslash: '`\"`'. The escape sequences '`\\`', '`\f`', '`\n`', '`\r`', and '`\t`' are also recognized, and have the usual meaning.

In addition, the '`\x`' escape sequence is supported; this sequence must be followed by *exactly two* hexadecimal digits, which represent an 8-bit ASCII value. For example, '`\xFF`' represents the character with ASCII code 0xFF.

No other escape sequences are currently supported.

Adjacent strings are automatically concatenated, as in C/C++ source code. This is useful for formatting very long strings as sequences of shorter strings. For example, the following constructs are equivalent:

- `"The quick brown fox jumped over the lazy dog."`
- `"The quick brown fox"`
  `" jumped over the lazy dog."`
- `"The quick" /* comment */ " brown fox " // another comment`
  `"jumped over the lazy dog."`

4.2.5 Groups

A group has the form:

```
{
   name=value;
   other_name=other_value;
   ...
}
```

Notice the curly brackets '`{}`' around the variable assignments. Groups can contain any number of variable assignments (see Section 4.1 [Variable Assignment], page 6), but each variable must have a unique name within the group.

Example:

```
SphereProperties:
{
    position_x = 0;
    position_y = 0;
    position_z = 0;
    visible = true;
};
```

4.2.6 Arrays

An array has the form:

```
[ value, value, ... ]
```

Notice the square brackets '`[]`' delimiting the comma-separated values. An array may have zero or more elements, but the elements must all be **scalar** values of the **same type**.

Examples:

```
disabled_runs = [0,1,3];
output_variables = ["Ex","Ey"];
```

4.2.7 Lists

A list has the form:

```
( value, value, ... )
```

Notice the parantheses '`()`' delimiting the comma-separated values. A list may have zero or more elements, each of which can be a scalar value, an array, a group, or another list. The values in a list can be of *different types*; however, in Angora, the list type is exclusively used to contain a collection of *group* values. In Angora, the list type semantically represents a collection of objects, each with a collection of properties set within their respective group value. Here is an example:

```
Materials:
(
    {
        material_tag = "mat1";
        rel_permittivity = 2.0;
    },
    {
        material_tag = "mat2";
        rel_permittivity = 2.5;
    }
);
```

Here, the list structure named `Materials` contains two groups (each delimited by curly brackets '`{}`') separated by a comma. This defines two materials with different sets of properties.

4.2.8 Comments

Three types of comments are allowed within a configuration:

- Script-style comments. All text beginning with a '`#`' character to the end of the line is ignored.

- C-style comments. All text, including line breaks, between a starting '`/*`' sequence and an ending '`*/`' sequence is ignored.
- C++-style comments. All text beginning with a '`//`' sequence to the end of the line is ignored.

As expected, comment delimiters appearing within quoted strings are treated as literal text.

```
# Here's a comment
MyGroup:
(/* This is
    also a comment */
    {
        this_property = "myvalue";
        // Another comment
    }
);
```

4.2.9 Include Directives

A configuration file may "include" the contents of another file using an *include directive*. This directive has the effect of inlining the contents of the named file at the point of inclusion.

An include directive must appear on its own line in the input. It has the form:

`@include "`*`filename`*`"`

Any backslashes or double quotes in the file name must be escaped as '`\\`' and '`\"`', respectively.

For example, consider the following two configuration files:

```
# file: limits.cfg
back_x = -5e-6;
front_x = 6e-6;
left_y = -5e-6;
right_y = 6e-6;
lower_z = -3e-6;
upper_z = 4e-6;
```

```
# file: mysim.cfg
RectangularBoxes:
(
    {
        shape_tag = "mybox";
        @include "limits.cfg"
    }
);
```

Include files may be nested to a maximum of 10 levels; exceeding this limit results in a runtime error.

5 Configuration Variables

The variable assignments (or *settings* in libconfig terminology) in a configuration file reside either at the uppermost level (called the *Global* level) or within a group structure (see Section 4.2.5 [Groups], page 7). In the following, configuration variables will be characterized as either being a *Global variable*, or a *Sub-variable of ParentVariable*; where *ParentVariable* is the next parent variable upward in the hierarchy that has a name. The variable *ParentVariable* can either be a *group* or a *list* (see Section 4.2.5 [Groups], page 7 and Section 4.2.7 [Lists], page 8). Quite often, the immediate parent of a variable assignment is an unnamed group; therefore the *ParentVariable* of that assignment is the list that contains this unnamed group. For example, the *ParentVariable* of the variable `material_tag` in the example in Section 4.2.7 [Lists], page 8 is `Materials`, since its immediate parent is an unnamed group, but the list structure containing the unnamed group has a name (which is `Materials`). On the other hand, the variable `Materials` is a *Global variable*; since it is assigned at the uppermost level in a configuration file, outside any enclosing structure.

The configuration variable names are case sensitive; meaning that `Materials` and `materials` are not the same.

5.1 Grid Properties

Angora currently only supports a rectangular, Cartesian FDTD grid with equal grid spacing in the x, y, and z directions. Mesh refinement is not yet supported; therefore the grid spacing is uniform across the grid.

5.1.1 Courant Factor

`floating-point courant` [Global variable]

Angora adopts a slightly modified form for the Courant factor, defined as

$$\frac{1}{\sqrt{3}}\frac{c\Delta t}{\Delta x}$$

where c=299792458 m/s is the speed of light in vacuum, and Δt and Δx are the *temporal* and *spatial* step sizes (see Section 5.1.2 [Spatial Step Size], page 10). The Courant factor should be less than `1.0` for stability. A common value for `courant` is `0.98`.

5.1.2 Spatial Step Size

`floating-point dx` (*units: m*) [Global variable]

The spatial step size in the FDTD grid is specified by the `dx` variable. Currently only cubic FDTD cells are supported; therefore the spatial step sizes in the x, y, and z direction are all determined by `dx`.

5.1.3 Grid Dimensions

`integer NCELLS_X` [Global variable]

`integer NCELLS_Y` [Global variable]

`integer NCELLS_Z` [Global variable]

These variables determine the number of FDTD cells (or voxels) in the Cartesian FDTD grid in the x, y, and z directions. If no perfectly-matched layers are specified (see Section 5.1.4 [Perfectly-Matched Layer (PML)], page 11), the total number of FDTD cells in the three-dimensional FDTD grid is equal to (NCELLS_X) x (NCELLS_Y) x (NCELLS_Z).

5.1.4 Perfectly-Matched Layer (PML)

`integer NPML` [Global variable]

This variable sets the thickness (in grid cells) of the perfectly-matched layers (PMLs) around the grid. This sets the thickness of the PML in in all directions. Further customization of the PML thickness is not yet supported.

Typical PML thicknesses are `5` to `10`. If you do not want to place a PML layer around the grid, just assign `NPML=0`. Without a PML layer, the boundary of the FDTD grid acts as a perfect electric conductor (PEC). Other boundary conditions (perfect magnetic conductor, periodic, etc.) will also be supported in the future.

With a PML definition, the total number of FDTD cells in the three-dimensional FDTD grid becomes (NCELLS_X+2*NPML) x (NCELLS_Y+2*NPML) x (NCELLS_Z+2*NPML). The computational burden per FDTD cell associated with the PML layer is roughly twice that of the main grid.

Angora implements the convolution PML (CPML) formulation of the complex-frequency shifted (CFS) PML (see [Roden00], page 46; [Kuzuoglu96], page 46.)

`integer CPML_feature_size` (*default:* `max(NCELLS_X,NCELLS_Y,NCELLS_Z)`) [Global variable]

This variable specifies the maximum size (in grid cells) of the scattering or radiating structure in the FDTD grid. This information is used to determine the frequency-shifting parameter α in the CFS-PML formulation. Following Berenger's derivation (see [Berenger02], page 46), this parameter is defined as

$$\alpha = c\epsilon/w$$

where c is the velocity of propagation in the medium, ϵ is the absolute permittivity (in F/m) in the medium, and w is the maximum size of the structure.

The above relationship follows essentially from the low-frequency behavior of the CFS-PML. At low frequencies where the evanescent field around the structure dominates, the CFS-PML reduces to a real stretch of coordinates without any absorption. This helps the termination of evanescent fields, which are poorly handled by ordinary PMLs.

5.1.5 Number of Time Steps

`integer NSTEPS` [Global variable]

This variable determines the number of time steps in the FDTD simulation.

5.1.6 Coordinate Origin

`integer OriginX` (*default:* (*NCELLS_X+2*NPML*)/2+1) [Global variable]

`integer OriginY` (*default:* (*NCELLS_X+2*NPML*)/2+1) [Global variable]

`integer OriginZ` (*default:* (*NCELLS_X+2*NPML*)/2+1) [Global variable]

These variables set the origin of the coordinate system in the simulation. All other coordinates in a configuration file are taken as relative to this origin. The integer values correspond to Cartesian coordinates of the origin (in grid cells) from the back-left-lower corner of the grid. In Figure 5.1, the location of the coordinate origin in the FDTD grid is shown for (`OriginX`,`OriginY`,`OriginZ`)=(2,3,1). The FDTD grid is composed of (3x5x3) grids, and only the back (y=z=0), left (x=z=0), and lower (x=y=0) surfaces are shown in the figure.

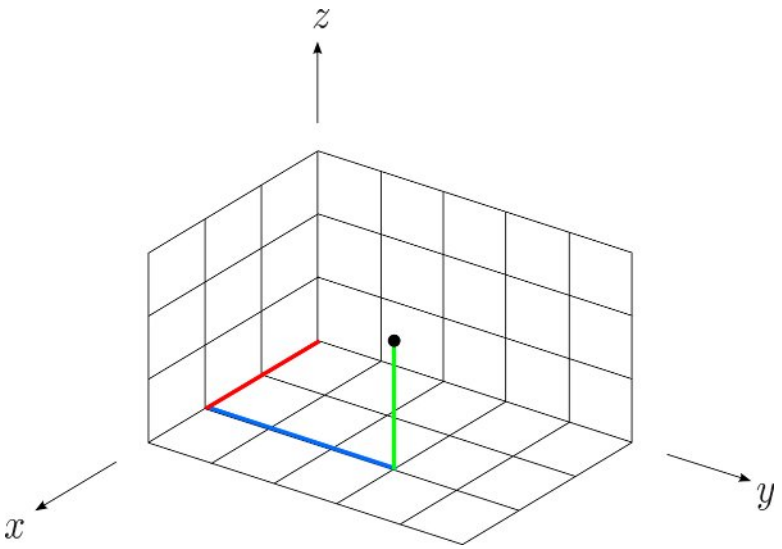

Figure 5.1: The location of the coordinate origin in the FDTD grid for (`OriginX`,`OriginY`,`OriginZ`)=(2,3,1).

5.1.7 Dynamic Range

The following two variables are only relevant in movie recording (see Section 5.10.1 [Movie Recording], page 32), wherein the floating-point field values on the movie frames are sometimes discretized to fit into 1 byte.

`floating-point max_field_value` (*default:* `1.0`) [Global variable]

This value specifies the maximum field value used in the discretization for 1-byte movie recording (see Section 5.10.1 [Movie Recording], page 32).

`floating-point dB_accuracy` (*default: automatic*) [Global variable]

This value specifies the dynamic range (in dB) to be used in the discretization for 1-byte movie recording (see Section 5.10.1 [Movie Recording], page 32). For example,

```
dB_accuracy = -60;
```

tells Angora to discretize the field values in a dynamic range between the maximum field value (specified by `max_field_value` above) and 60dB below that value. If `dB_`

`accuracy` is not specified, Angora tries to set this value automatically, based on its best guess on the useful accuracy range in the simulation. This value can also be read from the output of the movie recorder (see Section 5.10.1 [Movie Recording], page 32).

5.2 Shapes

`group Shapes` [Global variable]

In Angora, a geometrical shape and the material filling that shape are two distinct and independent elements of the definition of an object. The first of these elements is defined in the `Shapes` variable, which is a group (see Section 4.2.5 [Groups], page 7).

```
Shapes:
{
    RectangularBoxes:
    (
            ...
            ...
    );
    Spheres:
    (
            ...
            ...
    );
};
```

In this example, two sub-variables `RectangularBoxes` and `Spheres` of the `Shapes` group are shown. These are both list variables (see Section 4.2.7 [Lists], page 8).

Currently, rectangular boxes and spheres are the only basic shape classes defined in Angora. Unions, intersections, and geometrical transformations of shapes, as well as more basic shape classes will be added to Angora in the future. Please send any comments, suggestions, and requests to `help@angorafdtd.org`.

5.2.1 Rectangular Boxes

`group RectangularBoxes` [Sub-variable of Shapes]

Rectangular boxes are defined using the `RectangularBoxes` variable, which is a list structure under the `Shapes` group.

```
Shapes:
{
    RectangularBoxes:
    (
        {
            shape_tag = "mybox";
            back_x = -5e-6;
            front_x = 6e-6;
            left_y = -5e-6;
            right_y = 6e-6;
```

```
            lower_z = -3e-6;
            upper_z = 4e-6;
        },
        {
            shape_tag = "another_box";
            ...
            ...
        }
    );
};
```

In this example, two rectangular box shapes are defined in two respective unnamed groups; only the first being shown in complete detail.

`string shape_tag` [Sub-variable of RectangularBoxes]
: This string variable assigns a *name* to the particular shape, so it can be referred to later in the configuration file.

`floating-point back_x` (*units: m*) [Sub-variable of RectangularBoxes]

`floating-point front_x` (*units: m*) [Sub-variable of RectangularBoxes]

`floating-point left_y` (*units: m*) [Sub-variable of RectangularBoxes]

`floating-point right_y` (*units: m*) [Sub-variable of RectangularBoxes]

`floating-point lower_z` (*units: m*) [Sub-variable of RectangularBoxes]

`floating-point upper_z` (*units: m*) [Sub-variable of RectangularBoxes]
: These variables determine the minimum and maximum Cartesian coordinates of the box in the x, y, and z directions relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

5.2.2 Spheres

`group Spheres` [Sub-variable of Shapes]
: Spheres are defined using the `Spheres` variable, which is a list structure under the `Shapes` group.

```
Shapes:
{
    Spheres:
    (
        {
            shape_tag = "mysphere";
            center_x = 5e-6;
            center_y = 5e-6;
            center_z = 5e-6;
            radius = 4e-6;
        },
        {
            shape_tag = "another_sphere";
```

```
            ...
            ...
        }
    );
};
```

In this example, two spherical shapes are defined in two respective unnamed groups; only the first being shown in complete detail.

`string shape_tag` [Sub-variable of Spheres]
: This string variable assigns a *name* to the particular shape, so it can be referred to later in the configuration file.

`floating-point center_x` (*units: m*) [Sub-variable of Spheres]

`floating-point center_y` (*units: m*) [Sub-variable of Spheres]

`floating-point center_z` (*units: m*) [Sub-variable of Spheres]
: These variables determine the Cartesian coordinate of the center of the sphere relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`floating-point radius` (*units: m*) [Sub-variable of Spheres]
: This variable determines the radius of the sphere.

5.3 Materials

Currently, Angora only supports isotropic, non-dispersive materials. Anisotropic and dispersive materials will also be supported in the future. Please send any comments, suggestions, and requests to `help@angorafdtd.org`.

`list Materials` [Global variable]
: The properties of a certain material type are specified in the `Materials` list (see Section 4.2.7 [Lists], page 8).

```
Materials:
(
    {
        material_tag = "this_material";
        rel_permittivity = 2.0;
        rel_permeability = 1.0;
        electric_conductivity = 0.0;
        magnetic_conductivity = 0.0;
    },
    {
        material_tag = "another_material";
        ...
        ...
    }
);
```

In this example, two materials are defined in two respective unnamed groups; only the first being shown in complete detail.

`string material_tag` [Sub-variable of Materials]
This string variable assigns a *name* to the particular material, so it can be referred to later in the configuration file.

`floating-point rel_permittivity` [Sub-variable of Materials]
This (mandatory) variable specifies the relative permittivity (or the dielectric constant) of the material. In SI units, the absolute permittivity of the material is this variable multiplied by the permittivity of free space (8.85418782E-12 F/m).

`floating-point rel_permeability` (*default:* `1.0`) [Sub-variable of Materials]
This variable specifies the relative permeability (or the magnetic constant) of the material. In SI units, the absolute permeability of the material is this variable multiplied by the permeability of free space (4piE-7).

`floating-point electric_conductivity` (*units: S/m*) (*default:* `0`) [Sub-variable of Materials]
This variable specifies the electric conductivity (in Siemens/m or Mho/m) of the material.

`floating-point magnetic_conductivity` (*units: Ohm/m*) (*default:* `0`) [Sub-variable of Materials]
This variable specifies the magnetic conductivity (in Ohm/m) of the material.

5.4 Simulation Space

`group SimulationSpace` [Global variable]
The `SimulationSpace` group is where all the objects inside the simulation space are defined.

```
SimulationSpace:
{
    Objects:
    (
            ...
            ...
    );
    RandomMaterials:
    {
            ...
            ...
    };

    ...
    ...

};
```

In the above example, only two of the sub-variables of the `SimulationSpace` group, `Objects` and `RandomMaterials`, are shown. The sub-variable `Objects` is a list (see Section 4.2.7 [Lists], page 8), whereas `RandomMaterials` is a group (see Section 4.2.5 [Groups], page 7).

The definitions in the `SimulationSpace` group are processed **in the order of placement**. Thus, the user has complete control over which object is placed in the simulation space first. As a consequence of this first-come-first-serve policy, objects can overwrite regions of the simulation space occupied by other objects.

5.4.1 Objects

`list Objects` [Sub-variable of SimulationSpace]

The `Objects` list defines material objects to be placed in the simulation grid. An *object* in this context is defined as a combination of two abstract ingredients: A previously-defined shape (see Section 5.2 [Shapes], page 13), and a previously-defined material to fill that shape (see Section 5.3 [Materials], page 15). The shape and material are referred to using their shape and material tags, which are string variables assigned to them in their definitions.

Here is an example:

```
SimulationSpace:
{
    Objects:
    (
        {
            material_tag = "this_material";
            shape_tag = "mysphere";
        },
        {
            material_tag = "another_material";
            shape_tag = "mybox";
        }
    );
};
```

`string material_tag` [Sub-variable of Objects]

This string variable specifies the material that makes up the object. It should match a previously-defined tag in a `Materials` definition (see Section 5.3 [Materials], page 15).

`string shape_tag` [Sub-variable of Objects]

This string variable specifies the geometrical shape of the object. It should match a previously-defined tag in a `Shapes` definition (see Section 5.2 [Shapes], page 13).

5.4.2 Planar Layers

`list MaterialSlabs` [Sub-variable of SimulationSpace]

The purpose of the `MaterialSlab` list is to introduce **planar stratification** into the simulation grid. Currently, Angora only supports planar stratification along the z

direction. The handling of planar layers will be further improved in the future. Please send any comments, suggestions, and requests to help@angorafdtd.org.

Here is an example:

```
SimulationSpace:
{
    MaterialSlabs:
    (
        {
            tag = "material1";
            min_coord = 400.46;
            max_coord = 500.0;
        },
        {
            tag = "material2";
            min_coord = "min";
            max_coord = 300.0;
        }
    );
};
```

In the above example, two material slabs are placed in the grid. They are composed of `material1` and `material2`, respectively.

`string tag` [Sub-variable of MaterialSlabs]

This variable specifies the material that makes up the slab. It should match a previously-defined tag in a `Materials` definition (see Section 5.3 [Materials], page 15).

`floating-point/string min_coord` [Sub-variable of MaterialSlabs]

`floating-point/string max_coord` [Sub-variable of MaterialSlabs]

These two floating-point variables specify the lower and upper coordinates (in grid cells) of the material slab with respect to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12). These can be non-integer; in which case the layer boundaries are suitably interpolated. These variables can also be assigned the string values `"min"` or `"max"`; which correspond to the lower and upper boundaries of the simulation grid, respectively.

If the FDTD grid is terminated by absorbing PML boundaries (see Section 5.1.4 [Perfectly-Matched Layer (PML)], page 11), then the `MaterialSlab` definitions effectively create *infinite planar layers* that extend horizontally toward infinity. When the `"min"` or `"max"` strings are assigned as lower or upper coordinates of the slab, the `MaterialSlab` definition amounts to placing a *half space*. When the `MaterialSlab` variable is used, the incident beams (see Section 5.9 [Incident Wave Conditions], page 31) and the scattered far field (see Section 5.7 [Near-Field-to-Far-Field Transform], page 30) are both calculated as if the material slab horizontally extends toward infinity.

5.4.3 Random Materials

`group RandomMaterials` [Sub-variable of SimulationSpace]

Independent samples from a random distribution of material properties with a specified correlation function can be generated and placed into the simulation grid using the `RandomMaterials` group. It contains sub-variables in the form of lists (see Section 4.2.7 [Lists], page 8) that correspond to specific correlation functions. Currently, only the *Whittle-Matern* family of correlation functions is supported. More correlation functions can be added in the future. Please send any comments, suggestions, and requests to `help@angorafdtd.org`.

Although the spatial correlation of the generated random material blocks can vary, the joint probability density function of the material block is always a *multivariate normal (Gaussian) function.*

`list WhittleMaternCorrelated` [Sub-variable of RandomMaterials]

The Whittle-Matern family of correlations (see [Rogers09], page 46) is a three-parameter isotropic stochastic model that can represent a wide range of spatial correlations. The Whittle-Matern correlation function $B(r)$ for two points separated in space by a distance of r is given by the formula

$$B(r) = \sigma^2 \frac{2^{5/2-m}(r/l_c)^{m-3/2}}{\Gamma(m-3/2)} K_{m-3/2}(r/l_c)$$

where $K_{m-3/2}(\cdot)$ is the modified Bessel function of the second kind and order (m-3/2).

- m: The shape parameter that determines the overall behavior of the correlation function. As m->infinity, the function approaches a Gaussian distribution. If m=2, the function reduces to a decaying exponential. For m<3/2, the distribution acquires an inverse power law dependence near the origin; approximating a fractal distribution. For more details, see [Rogers09], page 46.
- l_c: (For m>3/2:) The correlation length. (For m<=3/2:) Loosely, the outer length scale where the fractal approximation no longer holds.
- σ: (For m>3/2:) The standard deviation of the distribution at a given point (r=0). (For m<=3/2:) In this range, the correlation function enters the fractal regime with an inverse-power-law dependence at the origin (see [Rogers09], page 46). The meaning of σ becomes more subtle in this regime. It can loosely be associated with the amplitude of the correlation between two points separated by l_c.

The `WhittleMaternCorrelated` list creates blocks with random material properties described by the Whittle-Matern correlation function above. Here is an example of its usage:

```
SimulationSpace:
{
    RandomMaterials:
    {
```

```
            WhittleMaternCorrelated:
            (
                {
                    constitutive_param_type = "rel_permittivity";
                    mean = 1.33;
                    std_dev = 0.05;
                    corr_len = 100e-9;
                    m = 2.0;
                    back_coord = -10.0;
                    front_coord = 10.0;
                    left_coord = -10.0;
                    right_coord = 10.0;
                    lower_coord = -10.0;
                    upper_coord = 10.0;
                    random_seed = 0;
                }
            );
        };

    };
```

`string` [Sub-variable of WhittleMaternCorrelated]
`constitutive_param_type`

The Whittle-Matern correlation function can describe the relative permittivity, relative permeability, or electric conductivity (in Siemens/m) of the material block. This is specified by assigning `"rel_permittivity"`, `"rel_permeability"`, or `"electric_conductivity"` to the `constitutive_param_type` string variable.

`floating-point mean` [Sub-variable of WhittleMaternCorrelated]
(*units: none or S/m*)

A baseline constant value equal to `mean` is added to the constitutive parameter described by the Whittle-Matern correlation function. If `mean=0`, then the generated random distribution will have zero mean. However, this will not necessarily be reflected to the actual constitutive parameter values in the grid; since Angora will automatically clip the constitutive parameters (permittivity, permeability, conductivity, etc.) from below to either unity or zero to avoid instabilities. For this reason, `mean` should be high enough to avoid this clipping as much as possible. As a rule of thumb, `mean` should be 5 to 6 times the standard deviation (`std_dev`) above unity or zero.

`floating-point std_dev` [Sub-variable of WhittleMaternCorrelated]
(*units: none or S/m*)

This variable specifies the σ parameter in the definition of the Whittle-Matern correlation function.

`floating-point corr_len` [Sub-variable of WhittleMaternCorrelated]
(*units: m*)

This variable specifies the l_c parameter in the definition of the Whittle-Matern correlation function.

`floating-point m` [Sub-variable of WhittleMaternCorrelated]

This variable specifies the m parameter in the definition of the Whittle-Matern correlation function.

`floating-point back_coord` [Sub-variable of WhittleMaternCorrelated]

`floating-point front_coord` [Sub-variable of WhittleMaternCorrelated]

`floating-point left_coord` [Sub-variable of WhittleMaternCorrelated]

`floating-point right_coord` [Sub-variable of WhittleMaternCorrelated]

`floating-point lower_coord` [Sub-variable of WhittleMaternCorrelated]

`floating-point upper_coord` [Sub-variable of WhittleMaternCorrelated]

These variables determine the bounding coordinates (in grid cells) of the rectangular prism containing the random material. The coordinates are with respect to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`integer/string random_seed` (*default: determined by system time*) [Sub-variable of WhittleMaternCorrelated]

If you would like to create *exactly the same random distribution* each time the simulation is run, you can assign an integer value to the `random_seed` variable. Otherwise, you **should not** define this variable. This value is used to initialize the random-number generator in Angora. If the same seed is used to initialize the random-number generator, the same sequence of random numbers will be generated each time, resulting in the same random distribution.

If multiple simulation runs are present (see Section 5.13 [Multiple Simulation Runs], page 43), you can create different random samples for each simulation run by assigning the string value `"run_index"` to `random_seed`. This will initialize the intenal random-number generator with the run index (ranging from `0` to `number_of_runs-1`) of each run. This way, a different random distribution will be obtained in each simulation run; but a distribution for a given simulation run will be *fixed* in subsequent executions of Angora.

In Figure 5.2, a 2D slice of an example zero-mean sample distribution generated by `WhittleMaternCorrelated` is depicted in grayscale.

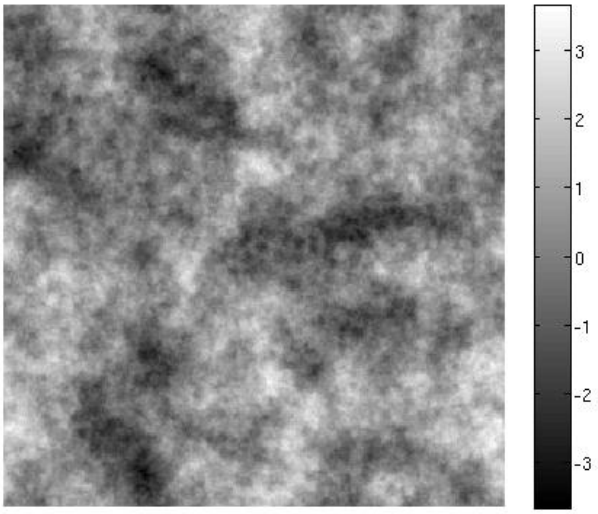

Figure 5.2: A 2D slice of an example zero-mean sample distribution. This distribution can be assigned to different constitutive parameters of a material.

5.4.4 File Input

`list MaterialsFromFiles` [Sub-variable of SimulationSpace]

Material information within rectangular regions of the FDTD simulation grid can be read from files using a `MaterialsFromFiles` list. Currently, only a single constitutive parameter can be read from a file; and dispersive or anisotropic materials are not supported. These issues will be handled more comprehensively in a future version. Please send any comments, suggestions, and requests to `help@angorafdtd.org`.

The material file should be in a simple custom binary format that Angora can recognize. The order and type of each variable in the file is explained below:

- '`x-extent`': The extent of the array in the x direction in grid cells (integer, 4 bytes)
- '`y-extent`': The extent of the array in the y direction in grid cells (integer, 4 bytes)
- '`z-extent`': The extent of the array in the z direction in grid cells (integer, 4 bytes)
- A floating-point array of length (`x-extent`) x (`y-extent`) x (`z-extent`). Each value in this array is either of type `double` (8 bytes) or `float` (4 bytes), depending on the `datatype` variable (see [datatype], page 25). The floating-point array should be laid out in the file in *column-major* order, meaning that the x dimension is looped over first, then the y dimension, and finally the z dimension. This ordering is depicted in Figure 5.3. The elements of the 2x2x2 array are numbered from 0 to 7. These elements should be laid out in the binary file in the same order:

```
..... 0  1  2  3  4  5  6  7  .....
```

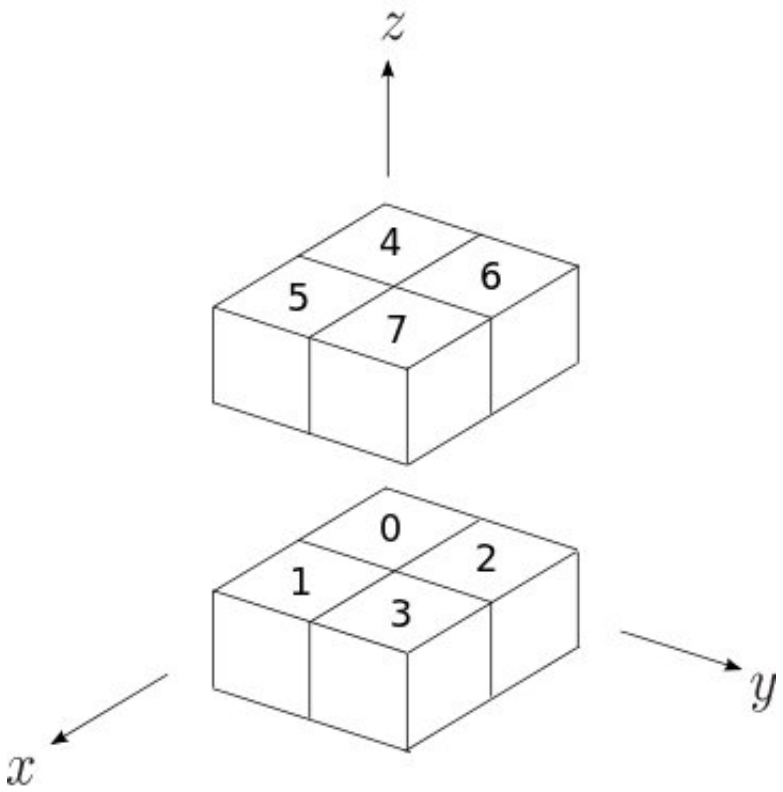

Figure 5.3: The illustration of the column-major ordering of a three-dimensional array. The values indicated by numbers should be laid out in the file in the same order.

Here is an example usage of `MaterialsFromFiles`:

```
SimulationSpace:
{
    MaterialsFromFiles:
    (
        {
            file_name = "path_to_file/materialfile";
            append_run_index_to_name = true;
            file_extension = "mat";
            constitutive_param_type = "rel_permittivity";
            anchor = "center";
            position_x = 131;
            position_y = 131;
            position_z = 131;
            datatype = "double";
            max_new_materials = 1000;
        }
```

```
    );
};
```

`string file_name` [Sub-variable of MaterialsFromFiles]
This string specifies the name of the binary file from which the material information will be read. Path information can be prepended to the file name, as shown in the example above. This path is interpreted as being relative to `input_dir` (see Section 5.11 [Paths], page 42), unless it is preceded by a slash '`/`'.

`string file_extension` (*default:* `""`) [Sub-variable of MaterialsFromFiles]
This is the extension of the material file to be read. In the above example, the file to be read is '`path_to_file/materialfile.mat`'.

`boolean append_run_index_to_name` [Sub-variable of MaterialsFromFiles]
This boolean flag becomes useful if there are multiple simulation runs (see Section 5.13 [Multiple Simulation Runs], page 43), and a different file needs to be read in each run. This can be accomplished by appending the run index (which ranges from `0` to `number_of_runs-1`) to the file name specified by `file_name`. For example, if there are 3 simulation runs (`number_of_runs` is 3) the above assignment will tell Angora to read the file '`path_to_file/materialfile0.mat`' in the first run, '`path_to_file/materialfile1.mat`' in the second, and '`path_to_file/materialfile2.mat`' in the third.

This variable is required for all simulations (hence no default value) to help the user prevent easy mistakes such as reading the same file for all simulation runs unintentionally, reading '`path_to_file/materialfile0.mat`' instead of '`path_to_file/materialfile.mat`', etc.

`string constitutive_param_type` [Sub-variable of MaterialsFromFiles]
The values read from the input file can be assigned to one of the following constitutive parameters: relative permittivity, relative permeability, or electric conductivity. This is determined by assigning `"rel_permittivity"`, `"rel_permeability"`, or `"electric_conductivity"` to the `constitutive_param_type` string variable. Electric conductivity is assumed to be in Siemens/m.

`string anchor` (*default:* `"center"`) [Sub-variable of MaterialsFromFiles]
This string defines an anchor point inside the rectangular-box-shaped region that is to be read from this file. This anchor is then assigned a coordinate in the FDTD grid, thereby determining the position of the rectangular box in the grid. Valid values for `anchor` are:

- `"center"`: center of the box
- `"BLL"`: back-left-lower corner of the box
- `"BLU"`: back-left-upper corner of the box
- `"BRL"`: back-right-lower corner of the box
- `"BRU"`: back-right-upper corner of the box
- `"FLL"`: front-left-lower corner of the box

- `"FLU"`: front-left-upper corner of the box
- `"FRL"`: front-right-lower corner of the box
- `"FRU"`: front-right-upper corner of the box

Here, as usual, "back"/"front" refers to the x coordinate, "left"/"right" refers to the y coordinate, and "lower"/"upper" refers to the z coordinate.

`integer position_x` [Sub-variable of MaterialsFromFiles]

`integer position_y` [Sub-variable of MaterialsFromFiles]

`integer position_z` [Sub-variable of MaterialsFromFiles]

These integer values determine the Cartesian x,y, and z coordinates of the anchor point (see above) assigned to the rectangular region to be read from the file. The coordinates are measured in grid cells from the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`string datatype` [Sub-variable of MaterialsFromFiles]

The datatype for the values read from the file is determined by this variable. It should be either `"double"` (8 bytes) or `"float"` (4 bytes).

`integer max_new_materials` (*default:* `1000`) [Sub-variable of MaterialsFromFiles]

Internally, Angora uses *material indexing* to reduce memory use for material arrays. Each material with a distinct set of constitutive parameters is assigned an index value of type `unsigned short` (2 bytes) that can range from `0` to `65,535`. Instead of storing a floating-point value (which is usually 4 or 8 bytes) for a permittivity value at a point, Angora stores an *index* that represents the material present at that point.

Each time a material region is read into the FDTD grid using `MaterialsFromFiles`, a fixed number of new materials are added to the material list. These materials cover a discrete range of constitutive parameters between the minimum and maximum values found in the file. Because of this discretization, some loss of information is inevitable. The number of new materials is determined by the variable `max_new_materials`; which is by equal to 1000 default. With the default value, the upper limit for the number of materials will be reached after about 65 material regions are inserted into the grid. If you wish to insert more material regions, and the dynamic ranges of constitutive parameters in your material files are not large, you can decrease `max_new_materials`.

5.4.5 Ground Planes

`list GroundPlanes` [Sub-variable of SimulationSpace]

Infinitely thin perfect-electric-conductor (PEC) sheets can be placed in the grid using a `GroundPlanes` list. Currently, only z-oriented (parallel to the xy plane) sheets at integer (full-cell) positions are supported.

```
SimulationSpace:
{
```

```
        GroundPlanes:
        (
            {
                coord = 0.0;
            }
        );
    };
```

`integer coord` [Sub-variable of GroundPlanes]

This integer variable specifies the z-coordinate (in grid cells) of the ground plane with respect to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

The `GroundPlanes` variable also updates the layering (stratification) information in the grid, much like `MaterialSlabs` (see Section 5.4.2 [Planar Layers], page 17).

5.5 Waveforms

`group Waveforms` [Global variable]

In Angora, a *time waveform* is defined as a self-contained structure that can be used by other structures; such as a Hertzian dipole source or a plane-wave injector. The library of available time waveforms will be expanded in the future. Please send any comments, suggestions, and requests to `help@angorafdtd.org`.

An example usage of `Waveforms`:

```
    Waveforms:
    {
        GaussianWaveforms:
        (
            {
                ...
            }
        );

        DifferentiatedGaussianWaveforms:
        (
            {
                ...
            }
        );
        ...
        ...
    };
```

5.5.1 Gaussian Waveforms

`list GaussianWaveforms` [Sub-variable of Waveforms]

This variable is used to define Gaussian time waveforms given by the formula

$$f(t) = A \ \exp\left(\frac{-(t - n_\tau \tau)^2}{2\tau^2}\right)$$

Gaussian waveforms are defined as follows:

```
Waveforms:
{
    GaussianWaveforms:
    (
        {
            tag = "my_waveform";
            amplitude = 1.0;
            tau = 2.1291e-15;
            delay = 3;
        }
    );
};
```

`string tag` [Sub-variable of GaussianWaveforms]
This is the string tag by which the waveform will later be referred to by another structure that requires a time waveform in its definition.

`floating-point amplitude` (*default:* `1.0`) [Sub-variable of GaussianWaveforms]
This specifies the variable A in [eq:gaussian], page 26 defining the Gaussian waveform.

`floating-point tau` (*units: sec*) [Sub-variable of GaussianWaveforms]
This specifies the variable τ in [eq:gaussian], page 26 defining the Gaussian waveform.

`floating-point delay` (*default:* `0.0`) [Sub-variable of GaussianWaveforms]
This specifies the variable n_τ in [eq:gaussian], page 26 defining the Gaussian waveform.

5.5.2 Differentiated-Gaussian Waveforms

`list DifferentiatedGaussianWaveforms` [Sub-variable of Waveforms]
This variable is used to define differentiated Gaussian time waveforms, given by the formula

$$f(t) = A\,\frac{d^n}{dt^n}\left[\exp\left(\frac{-(t-n_\tau\tau)^2}{2\tau^2}\right)\right] = A\,(\frac{-1}{\tau\sqrt{2}})^n H_n\left(\frac{t-n_\tau\tau}{\tau\sqrt{2}}\right)\exp\left(\frac{-(t-n_\tau\tau)^2}{2\tau^2}\right)$$

where $H_n(x)$ are the (physicists') Hermite polynomials.

Differentiated Gaussian waveforms are defined as follows:

```
Waveforms:
{
    DifferentiatedGaussianWaveforms:
    (
        {
            tag = "my_waveform";
```

```
                amplitude = 1.0;
                tau = 2.1291e-15;
                delay = 3;
                n = 3;
            }
        );
    };
```

`string tag` [Sub-variable of DifferentiatedGaussianWaveforms]
This is the string tag by which the waveform will later be referred to by another structure that requires a time waveform in its definition.

`floating-point` [Sub-variable of DifferentiatedGaussianWaveforms]
`amplitude` (*default:* `1.0`)
This specifies the variable A in [eq:diffgaussian], page 27 defining the differentiated Gaussian waveform.

`floating-point tau` [Sub-variable of DifferentiatedGaussianWaveforms]
(*units: sec*)
This specifies the variable τ in [eq:diffgaussian], page 27 defining the differentiated Gaussian waveform.

`floating-point delay` [Sub-variable of DifferentiatedGaussianWaveforms]
(*default:* `0.0`)
This specifies the variable n_τ in [eq:diffgaussian], page 27 defining the differentiated Gaussian waveform.

`integer n` [Sub-variable of DifferentiatedGaussianWaveforms]
This specifies the order of differentiation n in [eq:diffgaussian], page 27 defining the differentiated Gaussian waveform.

5.5.3 Modulated-Gaussian Waveforms

`list ModulatedGaussianWaveforms` [Sub-variable of Waveforms]
This variable is used to define sinusoidally-modulated Gaussian time waveforms, given by the formula

$$f(t) = A\ g(2\pi f_0(t - n_\tau\tau) + \phi) \exp\left(\frac{-(t - n_\tau\tau)^2}{2\tau^2}\right)$$

where the function $g(t)$ is a sinusoidal function, being either $\sin(t)$ or $\cos(t)$.

Modulated Gaussian waveforms are defined as follows:

```
Waveforms:
{
    ModulatedGaussianWaveforms:
    (
        {
            tag = "my_waveform";
            modulation_type = "sine";
```

```
                amplitude = 1.0;
                tau = 2.1291e-15;
                f_0 = 5.8929e14;
                delay = 3;
                phase = 90;
                differentiated = false;
            }
        );
    };
```

`string tag` [Sub-variable of ModulatedGaussianWaveforms]
This is the string tag by which the waveform will later be referred to by another structure that requires a time waveform in its definition.

`string modulation_type` [Sub-variable of ModulatedGaussianWaveforms]
If assigned `"sine"`, the modulation function $g(t)$ in [eq:modgaussian], page 28 becomes a sine. If assigned `"cosine"`, it becomes a cosine.

`floating-point amplitude` [Sub-variable of ModulatedGaussianWaveforms]
(*default:* `1.0`)
This specifies the variable A in [eq:modgaussian], page 28 defining the modulated Gaussian waveform.

`floating-point tau` (*units:* *sec*) [Sub-variable of ModulatedGaussianWaveforms]
This specifies the variable τ in [eq:modgaussian], page 28 defining the modulated Gaussian waveform.

`floating-point f_0` (*units:* *Hz*) [Sub-variable of ModulatedGaussianWaveforms]
This specifies the modulation frequency f_0 in [eq:modgaussian], page 28 defining the modulated Gaussian waveform.

`floating-point delay` [Sub-variable of ModulatedGaussianWaveforms]
(*default:* `0.0`)
This specifies the variable n_τ in [eq:modgaussian], page 28 defining the modulated Gaussian waveform.

`floating-point phase` [Sub-variable of ModulatedGaussianWaveforms]
(*units: degrees*, *default:* `0.0`)
This specifies the extra phase ϕ in [eq:modgaussian], page 28 defining the modulated Gaussian waveform. This phase should be specified in *degrees*, which is then converted internally to radians, which are the actual units of ϕ .

5.6 Point Sources

`list PointSources` [Global variable]
"Infinitesimal" electric dipole sources (also called *Hertzian* dipoles) can be simulated in Angora using the `PointSources` list.

A Hertzian dipole at position (x_0, y_0, z_0) is characterized by the following current distribution in space:

$$J(x, y, z; t) = \mathbf{a}\, j_0(t)\, \delta(x - x_0)\delta(y - y_0)\delta(z - z_0)$$

where $\delta(x)$ is the Dirac delta function. The vector **a** determines the orientation of the dipole, which can be along the x, y, or z directions. The prefactor $j_0(t)$ is called the *current moment* of the dipole, with the units (Ampere*m).

Here is an example usage of `PointSources`:

```
PointSources:
(
    {
        position_x = 0;
        position_y = 0;
        position_z = 0;
        source_orientation = "y_directed";
        waveform_tag = "moment_waveform";
        j0 = 1.0;
    }
);
```

`integer position_x` [Sub-variable of PointSources]

`integer position_y` [Sub-variable of PointSources]

`integer position_z` [Sub-variable of PointSources]

These integer variables specify (in grid cells) the Cartesian x, y, and z coordinates of the Hertzian dipole with respect to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`string source_orientation` [Sub-variable of PointSources]

This string specifies the spatial orientation of the Hertzian dipole. It should be `"x_directed"`, `"y_directed"`, or `"z_directed"`.

`string waveform_tag` [Sub-variable of PointSources]

This string variable specifies the waveform of the current moment $j_0(t)$. The waveform is interpreted in (Ampere*m) units. This should match a previously-defined tag in a `Waveforms` definition (see Section 5.5 [Waveforms], page 26).

`floating-point j0` (*units: Ampere/m, default:* `1.0`) [Sub-variable of PointSources]

This is an extra prefactor applied to the current moment waveform $j_0(t)$.

5.7 Near-Field-to-Far-Field Transform

Blah.

5.8 Optical Imaging

Blah.

5.9 Incident Wave Conditions

Blah.

5.10 Recording

Angora can record field values computed during a simulation into a file in a variety of ways. Field values can be recorded on a cross-section of the grid, along a line through the grid, or at a given point in the grid. Currently, Angora only supports the recording of the electric field. Recording of the magnetic field, as well as the energy, Poynting's vector, etc. will be implemented in the future. Please send any comments, suggestions, and requests to help@angorafdtd.org.

`string recorder_output_dir` (*default:* `"recorder"`) [Global variable]

This determines the subdirectory in which all the recording-related stuff will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `output_dir` (see Section 5.11 [Paths], page 42).

```
recorder_output_dir = "recorder";
Recorder:
{
        ...
        ...
};
```

`group Recorder` [Global variable]

The `Recorder` group contains the sub-variables related to different types of field recording. These are explained in the following subsections.

```
Recorder:
{
    MovieRecorders:
    (
        {
            ...
            ...
        }
    );
    LineRecorders:
    (
        {
            ...
            ...
        }
    );
    FieldValueRecorders:
    (
        {
            ...
            ...
```

```
            }
        );
    };
```

5.10.1 Movie Recording

Angora can record field components on a two-dimensional cross section of the grid into a custom movie file. The binary format used for movie recording is described in more detail in Section 5.10.1.1 [Movie File Format], page 34.

`string movie_recorder_output_dir` (*default:* `""`) [Sub-variable of Recorder]

This determines the subdirectory in which all the recorded movie files will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `recorder_output_dir` (see [recorder_output_dir], page 31).

```
Recorder:
{
    movie_recorder_output_dir = "movies";
    MovieRecorders:
    (
        ...
        ...
    );
};
```

`list MovieRecorders` [Sub-variable of Recorder]

Field values on a two-dimensional cross section of the FDTD grid can be recorded using the `MovieRecorders` list.

```
Recorder:
{
    MovieRecorders:
    (
        {
            recorded_section = "xz";
            recorded_position = 0;
            recorded_component = "Ex";
            recording_scale = "dB";
            recording_type = "uchar1";
            movie_dir = "this_movie_dir";
            movie_file_name = "MovieFile"
            movie_file_extension = "amv";
            only_records_material_info = false;
        }
    );
};
```

`string recorded_section` [Sub-variable of MovieRecorders]

This determines the cross section of the grid over which the field is recorded. Currently, only xz, yz, and xy cross sections are supported. These are represented by the string values `"xz"`, `"yz"`, and `"xy"`, respectively.

`integer recorded_position` [Sub-variable of MovieRecorders]
This integer value specifies (in grid cells) the coordinate of the recorded cross section along the perpendicular direction (e.g., the z direction if `recorded_section` is `"xy"`). The coordinate is relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`string recorded_component` [Sub-variable of MovieRecorders]
An individual movie recorder (in a group delineated by the curly brackets '`{}`') only records a single scalar value extracted from the vector-valued electric field. This could be one of the Cartesian components of the electric field, or its absolute value. These are represented by the string values `"Ex"`, `"Ey"`, `"Ez"`, and `"E"`, respectively. If you would like to record multiple Cartesian components of the electric field, simply add other movie recorders (i.e., other groups, see Section 4.2.5 [Groups], page 7) to the `MovieRecorders` list with the desired `recorded_component` values.

`string recording_scale` [Sub-variable of MovieRecorders]
If `"linear"`, the raw values are recorded. If `"absolute"`, the absolute value is taken before recording. If `"dB"`, the decibel value ($20\log_{10}(|\cdot|)$) is recorded.

`string recording_type` [Sub-variable of MovieRecorders]
Movies can either be recorded either in raw floating-point format, or in a single-byte compressed format. This is specified by assigning the string values `"dbl8"` or `"uchar1"` to the `recording_type` variable, respectively. Using the single-byte format reduces the file size considerably, but results in some data loss.

If `recording_type` is `"dbl8"`, then the field values are recorded in 8-byte `double` datatype, after processed in accordance with the `recording_scale` specification above. This provides practically lossless recording, albeit with increased computational burden and file size.

With the `"uchar1"` option, the field values are reduced to 256 discrete bins within a fixed *dynamic range*. This requires only a single byte per field value; reducing the storage requirement by a factor of 8.

- If `recording_type` is `"dB"`, the maximum and minimum values in this dynamic range are determined by the global variables `max_field_value` and `dB_accuracy` (see Section 5.1.7 [Dynamic Range], page 12):

max : $20\log_{10}(|\text{max\_field\_value}|)$ **min** : $20\log_{10}(|\text{max\_field\_value}|) + (\text{dB\_accuracy})$

The `dB_accuracy` variable should always be negative; therefore the minimum value in the dynamic range is lower than the maximum.

- If `recording_type` is `"linear"` or`"absolute"`, the maximum and minimum values are determined only by the global variable `max_field_value` (see Section 5.1.7 [Dynamic Range], page 12):

max : max_field_value **min** : $(-\text{max\_field\_value})$ or 0

`string movie_dir` (*default:* `""`) [Sub-variable of MovieRecorders]
This determines the subdirectory in which this individual movie file will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `movie_recorder_output_dir` (see [movie_recorder_output_dir],

page 32). By default, no subdirectory is created inside `movie_recorder_output_dir`.

`string movie_file_name` (*default:* `"MovieFile"`) [Sub-variable of MovieRecorders]
This determines the base string in the full name of the movie file. Other information is appended to the name of the file to differentiate individual movie files (see the example below).

`string movie_file_extension` (*default:* `"amv"`) [Sub-variable of MovieRecorders]
This is the extension of the movie file name. If assigned the value `""`, no extension is added to the file.

Here is an example movie file name:

```
MovieFile_Ex_0_1.amv
```

The base string in the name of the file (`"MovieFile"`) is specified by the `movie_file_name` variable. The second part of the file name, `"Ex"`, is determined by the recorded field component. The two integers that follow are the run index (see Section 5.13 [Multiple Simulation Runs], page 43) and the index of the movie inside the `MovieRecorders` list. For example, if there are two groups (two movies) in the `MovieRecorders` list, the first one will write into

```
MovieFile_Ex_0_0.amv
```

while the second will write into

```
MovieFile_Ex_0_1.amv
```

If there are two simulation runs (i.e., `number_of_runs` is equal to 2 – see Section 5.13 [Multiple Simulation Runs], page 43), then the files created in the second run will have `1` instead of `0` as the first integer in the above file names. Finally, the extension (`"amv"`) of the movie files is determined by the variable `movie_file_extension`.

`boolean only_records_material_info` (*default:* `false`) [Sub-variable of MovieRecorders]
If set to `true`, only the material information is recorded into the file, and no field recording is performed during the simulation.

5.10.1.1 Movie File Format

Angora records movies in a custom binary format for better speed and performance. Please be aware that this format is subject to modification. The changes in the format will be documented in this manual as necessary. You may refer to the '`ChangeLog`' file in the Angora distribution for recent changes in the movie recording format.

The MATLAB script '`angora_movie.m`', distributed as part of the Angora package, reads an Angora movie file and displays it as a MATLAB movie. It can also save the movie in AVI format. This script is installed in the directory '`$(prefix)/share/angora/`' (see Chapter 2 [Compilation and Installation], page 3). If Angora was installed without any `$(prefix)` configuration option, the default location is '`/usr/local/share/angora/`'.

The movie file is composed of chunks of data, ordered as follows. For each chunk, a short explanation (and maybe an alias) is given, followed by a description of the datatype in parantheses.

- major package version (integer, 4 bytes)
- minor package version (integer, 4 bytes)
- package revision number (integer, 4 bytes)
- number of bytes used to record each field component (integer, 4 bytes): This is either equal to `1` or `8`, depending on the `recording_type` variable.
- spatial step size (double, 8 bytes)
- temporal step size (double, 8 bytes)
- time value that corresponds to the beginning of the simulation (double, 8 bytes): This is usually a negative value, since time waveforms frequently begin before t=0.
- maximum value in the field discretization range (double, 8 bytes): This is the maximum value in the discretization dynamic range for single-byte recording (i.e., `recording_type` is `"uchar1"`). If `recording_type` is `"dbl8"`, this value is irrelevant. Same applies to the next value in the file.
- minimum value in the field discretization range (double, 8 bytes)
- '`length_1`': length along the first dimension of the recorded array (integer, 4 bytes): If the xy section were recorded, this would be the length of the array in the x dimension. This includes the thickness of the PML sections in both directions.
- '`length_2`': length along the second dimension of the recorded array (integer, 4 bytes): If the xy section were recorded, this would be the length of the array in the y dimension. This includes the thickness of the PML sections in both directions.
- '`length_time`': number of time steps in the simulation (integer, 4 bytes)
- thickness of the PML region, in grid cells (integer, 4 bytes): See Section 5.1.4 [Perfectly-Matched Layer (PML)], page 11 for more information on the PML. The PML sections are included in the recorded cross sectional area. They can easily be removed in post-processing.
- an array of length `length_1` with the actual physical coordinates (in m) along the first dimension of the recorded cross section (double, 8 bytes)
- an array of length `length_2` with the actual physical coordinates (in m) along the second dimension of the recorded cross section (double, 8 bytes)
- an array of length (`length_2` x `length_1`) holding the relative-permittivity values on the recorded cross section (double, 8 bytes): In Figure 5.4, it is assumed that the xy section is recorded, and the field positions are numbered from 0 to 11. It is seen that the **second dimension** (here, the y-dimension) is looped through first. These positions are laid out in the movie file in the same order:

```
0  1  2  3  4  5  6  7  8  9  10  11
```

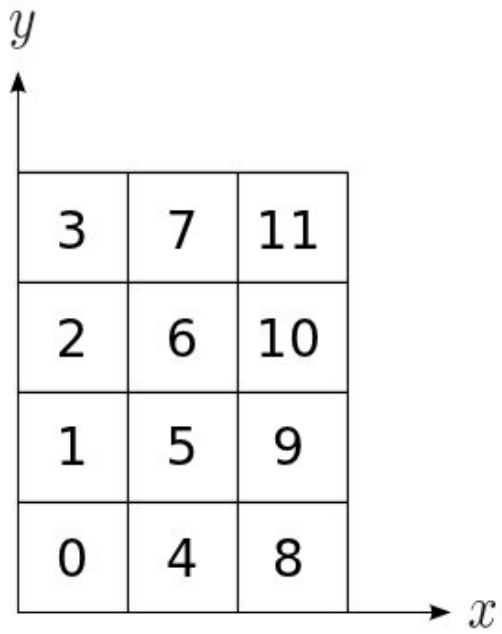

Figure 5.4: The layout of the field positions on the recorded cross section in the movie file.

- an array of length (`length_2` x `length_1`) holding the electric conductivity values (in Siemens/m) on the recorded cross section (double, 8 bytes): The 2D cross section is laid out in the movie file in the same way as the previous array.
- arrays (movie frames) of length (`length_2` x `length_1`) holding the field values on the recorded cross section (double, 8 bytes OR unsigned char, 1 byte – depending on `recording_type`): The total number of these movie frames is equal to `length_time`, read earlier from the binary file. Each of these frames is laid out in the movie file in the same way as the previous arrays.

5.10.2 Line Recording

Angora can record field components along a line into a file. The binary format used for line recording is described in more detail in Section 5.10.2.1 [Line File Format], page 38.

`string line_recorder_output_dir` (*default:* `""`) [Sub-variable of Recorder]

This determines the subdirectory in which all the recorded line files will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `recorder_output_dir` (see [recorder_output_dir], page 31).

```
Recorder:
{
    line_recorder_output_dir = "lines";
    LineRecorders:
    (
        ...
```

```
            ...
        );
    };
```

`list LineRecorders` [Sub-variable of Recorder]

```
    Recorder:
    {
        LineRecorders:
        (
            {
                line_orientation = "y_directed";
                line_position_x1 = 0;
                line_position_x2 = 0;
                recorded_component = "Ex";
                recording_scale = "linear";
                line_dir = "this_line_dir";
                line_file_name = "LineFile";
                line_file_extension = "aln";
            }
        );
    };
```

`string line_orientation` [Sub-variable of LineRecorders]

There are three possible orientations for the line over which the field values are recorded .These orientations are along the three principal axes of the grid; namely, the x,y, and z directions. These are specified by the strings `"x_directed"`, `"y_directed"`, and `"z_directed"`, respectively.

`integer line_position_x1` [Sub-variable of LineRecorders]

This is the first of the remaining two coordinates that specify the position of the recorded line (in grid cells). The coordinate is relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12). For example, if the line is oriented in the y direction (`line_orientation` is `"y_directed"`), then `line_position_x1` specifies the x coordinate of the line.

`integer line_position_x2` [Sub-variable of LineRecorders]

This is the second of the remaining two coordinates that specify the position of the recorded line (in grid cells). The coordinate is relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12). For example, if the line is oriented in the y direction (`line_orientation` is `"y_directed"`), then `line_position_x2` specifies the z coordinate of the line.

`string recorded_component` [Sub-variable of LineRecorders]

An individual line recorder (in a group delineated by the curly brackets '`{}`') only records a single scalar value extracted from the vector-valued electric field. This could be one of the Cartesian components of the electric field, or its absolute value. These are represented by the string values `"Ex"`, `"Ey"`, `"Ez"`, and `"E"`, respectively. If you would like to record multiple Cartesian components of the electric field, simply add other line recorders (i.e., other groups,

see Section 4.2.5 [Groups], page 7) to the `LineRecorders` list with the desired `recorded_component` values.

`string recording_scale` [Sub-variable of LineRecorders]
If `"linear"`, the raw values are recorded. If `"absolute"`, the absolute value is taken before recording. If `"dB"`, the decibel value ($20\log_{10}(|\cdot|)$) is recorded.

`string line_dir` (*default:* `""`) [Sub-variable of LineRecorders]
This determines the subdirectory in which this individual line file will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `line_recorder_output_dir` (see [line_recorder_output_dir], page 36). By default, no subdirectory is created inside `line_recorder_output_dir`.

`string line_file_name` (*default:* `"LineFile"`) [Sub-variable of LineRecorders]
This determines the base string in the full name of the line file. Other information is appended to the name of the file to differentiate individual line files (see the example below).

`string line_file_extension` (*default:* `"aln"`) [Sub-variable of LineRecorders]
This is the extension of the line file name. If assigned the value `""`, no extension is added to the file.

Here is an example line file name:

```
LineFile_Ey_Y_0_1.aln
```

The base string in the name of the file (`"LineFile"`) is specified by the `line_file_name` variable. The second part of the file name, `"Ey"`, is determined by the recorded field component. The following string `"Y"` indicates the orientation of the line, which is y-directed for this example. The two integers that follow are the run index (see Section 5.13 [Multiple Simulation Runs], page 43) and the index of the line recorder inside the `LineRecorders` list. For example, if there are two groups (two line recorders) in the `LineRecorders` list, the first one will write into

```
LineFile_Ey_Y_0_0.aln
```

while the second will write into

```
LineFile_Ey_Y_0_1.aln
```

If there are two simulation runs (i.e., `number_of_runs` is equal to 2 – see Section 5.13 [Multiple Simulation Runs], page 43), then the files created in the second run will have `1` instead of `0` as the first integer in the above file names. Finally, the extension (`"aln"`) of the line files is determined by the variable `line_file_extension`.

5.10.2.1 Line File Format

As with movies, Angora records the line files in a custom binary format for better speed and performance. Please be aware that this format is subject to modification. The changes in the format will be documented in this manual as necessary. You may refer to the '`ChangeLog`' file in the Angora distribution for recent changes in the line recording format.

The MATLAB script '`angora_line.m`', distributed as part of the Angora package, reads an Angora line file and displays it as a MATLAB movie. This script is installed in the directory '`$(prefix)/share/angora/`' (see Chapter 2 [Compilation and Installation], page 3). If Angora was installed without any `$(prefix)` configuration option, the default location is '`/usr/local/share/angora/`'.

The line file is composed of chunks of data, ordered as follows. For each chunk, a short explanation (and maybe an alias) is given, followed by a description of the datatype in parantheses.

- major package version (integer, 4 bytes)
- minor package version (integer, 4 bytes)
- package revision number (integer, 4 bytes)
- temporal step size (double, 8 bytes)
- time value that corresponds to the beginning of the simulation (double, 8 bytes): This is usually a negative value, since time waveforms frequently begin before t=0.
- '`total_length`': the number of recorded elements on each line snapshot (integer, 4 bytes)
- '`length_time`': number of time steps in the simulation (integer, 4 bytes)
- thickness of the PML region, in grid cells (integer, 4 bytes): See Section 5.1.4 [Perfectly-Matched Layer (PML)], page 11 for more information on the PML. The recorded line includes two PML sections on opposite ends, each with this length. These sections can easily be removed in post-processing.
- arrays (line snapshots) of length `total_length` holding the field values on the recorded line (double, 8 bytes): The total number of these line snapshots is equal to `length_time`, read earlier from the binary file.

5.10.3 Field-Value Recording

Angora can record the time history of the field at a given position in the simulation grid. The format used for this sort of recording is **HDF5** (Hierarchical Data Format) (`http://www.hdfgroup.org/HDF5/`). The HDF5 format was chosen for its standard interface, and the availability of free software tools for inspecting and modifying HDF5 output. The HDF5 output created by the field-value recorder is explained in more detail in Section 5.10.3.1 [Field-Value File HDF5 Content], page 41.

`string field_value_recorder_output_dir` (*default:* `""`) [Sub-variable of Recorder]

This determines the subdirectory in which all the recorded field-value files will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `recorder_output_dir` (see [recorder_output_dir], page 31).

```
Recorder:
{
    field_value_recorder_output_dir = "fieldvalues";
    FieldValueRecorders:
    (
        ...
        ...
```

```
        );
    };
```

`list FieldValueRecorders` [Sub-variable of Recorder]

```
    Recorder:
    {
        FieldValueRecorders:
        (
            {
                position_x = 0;
                position_y = 0;
                position_z = 0;
                recorded_component = "Ex";
                recording_scale = "linear";
                field_value_dir = "this_field_value_dir";
                field_value_file_name = "FieldValueFile";
                field_value_file_extension = "hd5";
            }
        );
    };
```

`integer position_x` [Sub-variable of FieldValueRecorders]

This is the x coordinate (in grid cells) of the recorded position in the simulation grid. It is relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`integer position_y` [Sub-variable of FieldValueRecorders]

This is the y coordinate (in grid cells) of the recorded position in the simulation grid. It is relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`integer position_z` [Sub-variable of FieldValueRecorders]

This is the z coordinate (in grid cells) of the recorded position in the simulation grid. It is relative to the grid origin (see Section 5.1.6 [Coordinate Origin], page 12).

`string recorded_component` [Sub-variable of FieldValueRecorders]

An individual field-value recorder (in a group delineated by the curly brackets '`{}`') only records a single scalar value extracted from the vector-valued electric field. This could be one of the Cartesian components of the electric field, or its absolute value. These are represented by the string values `"Ex"`, `"Ey"`, `"Ez"`, and `"E"`, respectively. If you would like to record multiple Cartesian components of the electric field, simply add other field-value recorders (i.e., other groups, see Section 4.2.5 [Groups], page 7) to the `FieldValueRecorders` list with the desired `recorded_component` values.

`string recording_scale` [Sub-variable of FieldValueRecorders]

If `"linear"`, the raw values are recorded. If `"absolute"`, the absolute value is taken before recording. If `"dB"`, the decibel value ($20 \log_{10}(|\cdot|)$) is recorded.

`string field_value_dir` (*default:* `""`) [Sub-variable of FieldValueRecorders]
This determines the subdirectory in which this individual field-value file will be placed. Unless it has a slash '`/`' up front; this path is interpreted as being relative to `field_value_recorder_output_dir` (see [field_value_recorder_output_dir], page 39). By default, no subdirectory is created inside `field_value_recorder_output_dir`.

`string field_value_file_name` [Sub-variable of FieldValueRecorders]
(*default:* `"FieldValueFile"`)
This determines the base string in the full name of the field-value file. Other information is appended to the name of the file to differentiate individual field-value files (see the example below).

`string` [Sub-variable of FieldValueRecorders]
`field_value_file_extension` (*default:* `"hd5"`)
This is the extension of the field-value file name. If assigned the value `""`, no extension is added to the file. The HDF5 extension `"hd5"` is applied by default.

Here is an example field-value file name:

```
FieldValueFile_Ex_0_1.hd5
```

The base string in the name of the file (`"FieldValueFile"`) is specified by the `field_value_file_name` variable. The second part of the file name, `"Ex"`, is determined by the recorded field component. The two integers that follow are the run index (see Section 5.13 [Multiple Simulation Runs], page 43) and the index of the field-value recorder inside the `FieldValueRecorders` list. For example, if there are two groups (two field-value recorders) in the `FieldValueRecorders` list, the first one will write into

```
FieldValueFile_Ex_0_0.hd5
```

while the second will write into

```
FieldValueFile_Ex_0_1.hd5
```

If there are two simulation runs (i.e., `number_of_runs` is equal to 2 – see Section 5.13 [Multiple Simulation Runs], page 43), then the files created in the second run will have `1` instead of `0` as the first integer in the above file names. Finally, the extension (`"hd5"`) of the line files is determined by the variable `field_value_file_extension`.

5.10.3.1 Field-Value File HDF5 Content

The HDF5 file created as the output of the field-value recorder can be viewed and modified using freely-available tools. One of these tools is HDFView, provided by the HDF Group. MATLAB also has built-in functions and tools that handle HDF5 files. For reference, a MATLAB script named '`angora_fieldvalue.m`' is distributed as part of the Angora package, which reads an Angora field-value file and plots the recorded waveform. This script is installed in the directory '`$(prefix)/share/angora/`' (see Chapter 2 [Compilation and Installation], page 3). If Angora was installed without any `$(prefix)` configuration option, the default location is '`/usr/local/share/angora/`'.

The HDF5 datasets in the field-value file are the following:

- '`angora_version`': An integer array of length 3 with the major version, minor version, and revision numbers of the Angora package used to create the file.
- '`num_time_steps`': An integer value specifying the number of time steps in the simulation.
- '`time_step`': A floating-point value specifying the temporal step in the simulation.
- '`initial_time_value`': A floating-point value specifying the time value corresponding to the beginning of the simulation. This is usually a negative value, since time waveforms frequently begin before t=0.
- '`field_values`': A floating-point array of length `num_time_steps` with the recorded field values.

5.11 Paths

`string angora_basepath` (*default:* `"."`) [Global variable]

This variable specifies the base directory for all the input-output operations in Angora. If there is no slash '`/`' in front of the path, it is interpreted as a relative path starting from the working directory (i.e, the one from which Angora is launched.)

Any other input or output directory will be assumed *relative* to `angora_basepath`. An overarching exception is when a directory is specified with a slash '`/`' up front; in which case that directory will be taken as an *absolute path*, and not relative to `angora_basepath`.

`string output_dir` (*default:* `"output"`) [Global variable]

This is the base directory for all the output that will result from Angora. It is interpreted as being relative to `angora_basepath`, unless it is preceded by a slash '`/`'. All other output directories are created as subdirectories of this directory.

Example:

```
angora_basepath = "angora_stuff";
output_dir = "data";
```

With these variable assignments, all the output will be written into subdirectories within '`./angora_stuff/data/`'.

`string input_dir` [Global variable]

This is the base directory for all the input that will be read by Angora. It is interpreted as being relative to `angora_basepath`, unless it is preceded by a slash '`/`'. Unless the path to an input file is absolute (i.e., preceded by a slash '`/`'), it is interpreted as being relative to `input_dir`.

Example:

```
angora_basepath = "angora_stuff";
input_dir = "input_data";
```

With these variable assignments, the input base directory becomes '`./angora_stuff/input_data/`'.

5.12 Logging

You can keep a log of the simulations run by Angora in a log file, which contains several lines of information for each simulation. First, an estimate of the finishing time and duration of the simulation is written into the log entry. The actual finishing time and duration is added to the log entry upon completion of the simulation.

Here is an example entry for a simulation in the log file:

```
johndoe started Angora on 02/22/12 11:54:36AM
    Estimated to finish on 02/22/12 11:54:42AM
    Estimated duration : 6 seconds.
    Simulation finished on 02/22/12 11:54:42AM
    Elapsed time : 6 seconds.
```

`boolean enable_logging` (*default:* `"true"`) [Global variable]
If set to '`true`', Angora will keep a record of the simulations that it runs in a log file. The name of this log file is specified by the `log_file_name` variable, and the directory in which this file resides is specified by the `log_output_dir` variable.

`string log_file_name` (*default:* `"angora.log"`) [Global variable]
This is the name of the Angora log file. It resides in the directory specified by the `log_output_dir` variable.

`string log_output_dir` (*default:* `"log"`) [Global variable]
This is the directory in which the Angora log file is kept. Unless it is preceded by a slash '`/`', it is taken as relative to the base output directory `output_dir` (see Section 5.11 [Paths], page 42).

5.13 Multiple Simulation Runs

A number of consecutive Angora simulations can be set up in a single configuration file.

`integer number_of_runs` (*default:* `1`) [Global variable]
The number of simulation runs is specified by the `number_of_runs` variable. The simulation runs (or *runs* for short) are indexed from `0` to `number_of_runs-1`. You can refer to these indices later in the configuration file for enabling or disabling certain configuration variables for certain runs.

`integer-array disabled_runs` (*default: none*) [Global variable]
This array of integers (see Section 4.2.6 [Arrays], page 8) lists the run indices for simulations that will be skipped.

Example:

```
disabled_runs = [1,2,3,4,5];
```

If `number_of_runs` was `7`, the above variable will cause only the simulations with indices `0` and `6` to be run.

`integer-array disabled_run_range` (*default: none*) [Global variable]
If you would like to disable simulations that correspond to a *range* of run indices, you can use this variable. This has to be an array of integers with only two elements (see

Section 4.2.6 [Arrays], page 8). Simulations with run indices between (and including) these two integers will be skipped.

The following variable assignment has the same effect as the one in the previous example:

```
disabled_run_range = [1,5];
```

Certain variables can be *enabled* or *disabled* for any of the simulation runs using the `enabled_for_runs` variable. This variable can be used inside any group structure (see Section 4.2.5 [Groups], page 7) to specify the run indices for which that group is enabled. For example,

```
number_of_runs = 4;
PointSources:
(
    {
        //point source #1
        enabled_for_runs = [0,1,2];
        position_x = 0;
        position_y = 0;
        position_z = 0;
        source_orientation = "x_directed";
        waveform_tag = "waveform1";
    },
    {
        //point source #2
        enabled_for_runs = [3];
        position_x = 0;
        position_y = 0;
        position_z = 0;
        source_orientation = "x_directed";
        waveform_tag = "waveform2";
    }
);
```

In this example, each group represents a collection of variable assignments that characterize an individual point source. With the `enabled_for_runs` variables set as shown, simulations `0`, `1`, and `2` will be run with the first point source; whereas simulation `3` will be run with the second point source.

5.14 Miscellaneous

5.14.1 Auto-Saving the Configuration

Angora can automatically save a record of every simulation configuration that it processes.

`boolean auto_save_cfg` (*default:* `"false"`) [Global variable]

If set to '`true`', Angora will automatically write every simulation configuration it runs into another configuration file, and save it in the directory specified by `cfg_output_dir`. A time/date string is appended to the name of the saved file to differentiate between subsequent executions of the same configuration file.

`string cfg_output_dir` (*default:* `"cfg"`) [Global variable]
: This is the directory in which the auto-saved configuration files are placed. Unless it is preceded by a slash '`/`', it is taken as relative to the base output directory `output_dir` (see Section 5.11 [Paths], page 42).

6 References

J. A. Roden and S. D. Gedney, "Convolution PML (CPML): an efficient FDTD implementation of the CFD-PML for arbitrary media," Microw. Opt. Technol. Lett., vol. 27, pp. 334-9, Dec. 2000.

M. Kuzuoglu and R. Mittra, "Frequency dependence of the constitutive parameters of causal perfectly matched absorbers," IEEE Microwave Guided Wave Lett., vol. 6, pp. 447-449, Dec. 1996.

J.-P. Berenger,"Numerical reflection from FDTD-PMLs: a comparison of the split PML with the unsplit and CFS PMLs," IEEE Trans. Antennas Propag, vol. 50, pp.258-265, Mar 2002.

J. D. Rogers, I. R. Capoglu, V. Backman, "Nonscalar elastic light scattering from continuous media in the Born approximation", Optics Letters, vol. 34, no. 12, pp. 1891-1893, 2009.

List of Figures

Indices

Configuration Variable Index

Concept Index