Aether

Loïc Le Cunff

December 26, 2023

Contents

I Introduction **7**

1 About **9**
- 1.1 A bit of history . . . 9
- 1.2 Past Funding . . . 9
- 1.3 Aether's name . . . 9

2 Basics **11**
- 2.1 Conventions . . . 11
- 2.2 Lua . . . 11
- 2.3 Script based operation . . . 11
  - 2.3.1 Special Modes . . . 12
- 2.4 GUI Interface . . . 12
  - 2.4.1 Graphs . . . 12

3 The Materials Library **13**
- 3.1 Constant model . . . 13
- 3.2 Simple dispersive models . . . 13
  - 3.2.1 Cauchy . . . 13
  - 3.2.2 Sellmeier . . . 14
- 3.3 Absorbing dispersive models . . . 14
  - 3.3.1 Drude model . . . 14
  - 3.3.2 Lorentz model . . . 15
  - 3.3.3 Critical points model . . . 15
- 3.4 Data tables based materials . . . 15
- 3.5 Additional functions . . . 16
  - 3.5.1 description . . . 16
  - 3.5.2 validity_range . . . 16
- 3.6 Full example . . . 16

II Finite Differences **17**

4 Theoretical bases **19**
- 4.1 Finite differences . . . 19
- 4.2 FDTD . . . 19
- 4.3 Dispersive materials . . . 19
- 4.4 PMLs . . . 19

4.5 FDFD 19

5 Structures **21**
5.1 Main script and the structure object 21
5.1.1 Parameters 21
5.1.2 Other functions 21
5.2 Structure script 21
5.2.1 Parameters 22
5.2.2 Primitives 22
5.2.3 Non shape functions 24
5.2.4 Lua defined functions 24
5.2.5 Meshes 25
5.3 Full scripts example 25
5.3.1 Geometry 25
5.3.2 Main script 25

6 FDTD: scripting interface **27**
6.1 Common functions 27
6.2 Standard mode FDTD 31
6.2.1 Specific functions 31
6.3 Normal incidence FDTD 31
6.3.1 Specific functions 31
6.3.2 Full script example 31
6.4 Oblique incidence FDTD using the Aminian and Rahmat-Samii method 32
6.4.1 Specific functions 32
6.4.2 Real script example 32
6.5 Single particles 33
6.5.1 Specific functions 33
6.5.2 Real script example 33
6.6 Sensors 34
6.6.1 Common functions 34
6.6.2 Sensor types 34
6.7 Light sources 35
6.7.1 Common functions 35
6.7.2 Source types 35

7 FDTD: GUI **37**

III Ray optics: Selene **39**

8 Selene: scripting interface **41**
8.1 Principles 41
8.1.1 Organization 41
8.1.2 Inner workings 41
8.1.3 Probabilistic behavior 42
8.1.4 Simulation summary 43
8.2 Relative positioning 43

8.2.1 Orientation 43
8.2.2 Placement 44
8.3 Faces and Interface Response Functions 44
8.3.1 Face orientation 44
8.3.2 The Interface Response Function 46
8.4 Objects 47
8.4.1 Functions common to all objects 47
8.4.2 Volume primitives 51
8.4.3 Surface Primitives 57
8.4.4 Meshes 62
8.4.5 Composite objects 62
8.5 Sensors 63
8.5.1 Absorbing sensors 63
8.5.2 Transparent sensors 63
8.5.3 Recording options 63
8.5.4 File format 64
8.6 Sources 65
8.6.1 Functions common to all light sources 65
8.6.2 Source types 67
8.6.3 Spectra 68
8.6.4 Power normalization, spectra and rays 68

9 Selene: GUI **71**
9.1 Main window 71
9.1.1 3D display 71
9.1.2 Rays Display 71
9.1.3 Scene tree 71
9.1.4 Rendering 72
9.2 The object dialog 72
9.3 The light source dialog 72
9.4 The materials dialog 72

IV Multilayers **73**

10 Multilayers optical response: scripting interface **75**
10.1 Basic setup 75
10.2 Structure definition 75
10.2.1 Environment 75
10.2.2 Layers 75
10.3 Optical response computation 75
10.4 Guided modes computation 75

11 Multilayers optical response: GUI **77**
11.1 Computation modes 77
11.1.1 Spectral mode 77
11.1.2 Angular mode 77
11.2 Multilayer structure definition 77

11.2.1 Layer . . . 78
11.2.2 Bragg structure . . . 78
11.2.3 Statistical thickness distribution . . . 78
11.3 Results . . . 78
11.4 Exporting the result . . . 78

12 Guided modes calculation: GUI **79**

V Various tools **81**

13 Scripting tools **83**
13.1 dielec_planar_waveguide . . . 83
13.2 fieldblock_treat . . . 83
13.3 fieldmap_treat . . . 83
13.4 index_fit . . . 83
13.5 preview_fdtd_structure . . . 83

14 Diffraction orders **85**
14.1 Monochromatic mode . . . 85
14.2 Polychromatic mode . . . 86

15 Graphical functions fitter **87**
15.1 Exporting the fit . . . 87

16 Samples Explorer **89**
16.1 Working principle . . . 89
16.2 The description file . . . 90
16.3 Interface . . . 91
16.3.1 The sample panel . . . 91
16.3.2 Tags filtering . . . 91

Part I

Introduction

Chapter 1

About

1.1 A bit of history

The oldest roots of the software originate back in 2008 with personnal rendering projects.

This was followed two years later by an internship at the L2n on the ADI-FDTD method, which served as the basis for the FDTD module. Concurrent development happened through the OpenSource Gneiss project (now obsolete), dedicated to rendering space scenes.

The FDTD code was then developed during a PhD under the ANR MetaPhotonique project, together with some basic tools.

A few years later under the TurboPET project, the raytracing core was coded by integrating Gneiss' original raytracer into Aether.

Since then, many improvements have been added, like the GUI, and from 2015 onwards several upgrades were designed to accomodate the Limule project, a proprietary software coded for the In-Fine joint laboratory.

1.2 Past Funding

The following fundings have contributed directly and indirectly to the project:

- The University of Technology of Troyes
- The ANR MetaPhotonique project
- A Région Champagne-Ardenne PhD grant
- The Eurostars TurboPET project
- The BISONS RAPID project
- The ANR DECISIoN project
- The SURYS society through the In-Fine joint laboratory

1.3 Aether's name

The name is an obvious reference to the Luminiferous Aether, which was the theoretical medium that carried light, over a century ago. It has long been disproven and replaced with Relativity.

One might ask then: why is the name of this software the same as a flawed theory? Simply put, it is to always remember that simulations are flawed as well. They are nothing but approximations, always being crude representations of reality. One should always remain wary of them, and never hesitate to question their validity.

But they are by no mean useless, they are important tools that help guide us towards what we want to achieve. However, one should never forget that what matters is reality and experiments, and when experiments and simulations don't match, it's not because reality is wrong, but because we did not understand something.

Chapter 2

Basics

2.1 Conventions

In this user manual, some conventions will be followed:

- **var** is a function or a GUI control
- **var** is an integer
- **var** is a real number
- **"var"** is a string. The quotation marks are required in the corresponding code
- **var** is a string without quotation marks, usually a file name or a special variable
- **var** will be reserved for the less common types, and their actual type will be stated on a case by case basis
- function arguments are written in meters if they are lengths, and degrees if they are angles
- the chosen convention with respect to plane waves (and thus the fourier transform) is $e^{i(\vec{k} \cdot \vec{r} - \omega t)}$. This means that the imaginary parts of refractive indexes and permittivities must be positive.

2.2 Lua

The scripting language which was chosen is Lua. It is an interpreted language much like Python or MatLab.
Its manual is available on: `https://www.lua.org/manual/5.4/`

2.3 Script based operation

The software was originally written to be used without any graphical interface. Still, ease of use required some sort of interface, and it was chosen to bind the core of the software to a scripting language, so that scripts could be used to define the various parameters and functions. Moreover, a full scripting language allows running computations or operations within the script itself, expanding the capabilities of Aether.

The software is controlled through a script file which must be provided to it on launch. By default, it will look and use the **script.lua** in the working directory.

Because Aether is comprised of various tools, it is necessary to differentiate them in a somewhat unified manner. This is done through the **MODE** function, for which the sole argument is the name of these tools.

This function returns a pointer that can be used to modify the behavior of the related operation. The name of this pointer is entirely up to the user. For instance:

```
fdtd_1=MODE("fdtd_normal")
```

will return a object that specifies an FDTD computation. The object can then be modified. In Lua, operations on this pointer are done through the ":" operator which calls functions tied to this object, For instance:

```
fdtd_1:polarization("TE")
```

will modify the polarization property of the previously defined FDTD simulation.

2.3.1 Special Modes

Beyond the various modules, there are two special modes that can be called:

- Calling **MODE**(**"pause"**) pauses the software until the user presses the Enter key.
- Calling **MODE**(**"quit"**) discards the rest of the script and ends the software.

Those are mostly used for debugging or logging purposes. Still, their name is exposed here in case someone finds uses to them.

2.4 GUI Interface

2.4.1 Graphs

Chapter 3

The Materials Library

At their core, materials in Aether are defined through their permittivity. The idea is that the total permittivity is the sum of several dielectric model

$$\varepsilon = \sum_i \varepsilon_i \tag{3.1}$$

where each ε_i is a specific model with its own set of parameters. This is done in a dedicated Lua script, much like the simulation setup, with a specific function for each model.

3.1 Constant model

This is the most simple model and can be used to create constant materials materials. It leads to a real, constant permittivity for every wavelength and is often referred to as ε_∞.

This constant part is set either through the **epsilon_infinity**(**epsilon**) function, or, as a convenience, through the They are first defined through the **index_infinity**(**index**) function. In the latter case, the refractive index is turned into a permittivity internally. Example

```
index_infinity(1.5)
```

As explained in 6.1.8, in the case of a constant material the whole material script can also be replaced in the main script by a call to **const_material**.

3.2 Simple dispersive models

Those are the simplest models that are used to define materials like glasses in the spectral range where they are transparent. Those materials are not FDTD compatible.

3.2.1 Cauchy

The Cauchy dispersion formula can be thought of as an expansion on the refractive index in the form

$$n = A + \frac{B}{\lambda^2} + \frac{C}{\lambda^4} + \cdots \tag{3.2}$$

since the permittivity is built buy summing other permittivity, it was decided the one associated with the Cauchy model would then be

$$\varepsilon^C = \left(A + \frac{B}{\lambda^2} + \frac{C}{\lambda^4} + \cdots \right)^2 \tag{3.3}$$

This is instanced through the function **add_cauchy**(**A**,**B**,**C**,**...**) which takes as many arguments as there are in the expansion.

Example:

```
add_cauchy(1.5,0.01e-12,0.0002e-24)
```

Note that if your material was originally strictly defined by a Cauchy formula, it is important to set ε_∞ to 0.

3.2.2 Sellmeier

The Sellmeier formula is usually written as

$$n^2 = 1 + \sum_i \frac{B_i \lambda^2}{\lambda^2 - C_i} \tag{3.4}$$

but since $n^2 = \varepsilon$, it can be understood as

$$\varepsilon = 1 + \sum_i \epsilon_i^{\mathrm{SM}} \quad ; \quad \epsilon_i^{\mathrm{SM}} = \frac{B_i \lambda^2}{\lambda^2 - C_i} \tag{3.5}$$

where each ϵ_i^{SM} resembles a Lorentz model.

Since C_i obviously must have a dimension of m^2, it was decided it would be simpler for Aether to write ϵ^{SM} as

$$\epsilon_i^{\mathrm{SM}} = \frac{B_i \lambda^2}{\lambda^2 - C_i^2} \tag{3.6}$$

so that C_i and λ are homogeneous.

As such each Sellmeier term is added to the material dielectric model with the function **add_sellmeier(B,C)**, where **C** is the square-root of what is usually found in material databases. Note that the refractive index becomes undefined when $\lambda = C_i$.

Example:

```
add_sellmeier(2.0,100e-9)
```

3.3 Absorbing dispersive models

Those models are another kind of elementary models, mostly used to describe metals. They are FDTD compatible, and are implemented in it through the recursive convolution method [1].

3.3.1 Drude model

This basic model is very useful to describe metals in the infrared part of the spectrum. It is defined by

$$\varepsilon^D = -\frac{\omega_D^2}{\omega^2 + i\gamma\omega} \tag{3.7}$$

and is defined through **add_drude**(ω_D,γ).

3.3.2 Lorentz model

Another basic model useful for the description of metals, defined by

$$\varepsilon^{L} = A\frac{\Omega^2}{\Omega^2 - \omega^2 - i\Gamma\omega} \tag{3.8}$$

it is defined through **add_lorentz**(A,Ω,Γ)

3.3.3 Critical points model

This model was initially introduced to described the permittivity of gold in the blue part of the spectrum [2], and was then adapted to FDTD by [3]. Its dielectric function is

$$\varepsilon^{Cr} = \Omega\left(\frac{e^{i\varphi}}{\Omega - \omega - i\Gamma} + \frac{e^{-i\varphi}}{\Omega + \omega + i\Gamma}\right) \tag{3.9}$$

and this model is used through **add_crit_point**(A,Ω,φ,Γ).

3.4 Data tables based materials

Sometimes, the dielectric model of a particular material is not known, and only tabulated data is available. It is possible to use that data to define a material within Aether.

This data is to be defined through three Lua tables, that will then be fed to a function:

- the wavelength, in meters
- the real part of the input
- the imaginary part of the input

Those three tables must contain the same number of points. Then, they are passed as arguments to either

1. **add_data_epsilon**(**lambda**,**real**,**imag**): takes the permittivity as an input
2. **add_data_index**(**lambda**,**real**,**imag**): takes the refractive index as an input, which is internally turned into a permittivity before being memorized by the material

Example:

```
lambda={400e-9,500e-9,600e-9}
nr={2.3,2.04,1.96}
ni={2e-2,1e-2,1e-2}

add_data_index(lambda,nr,ni)
```

Of course, it is impossible to provide data for every wavelength. As such, **the permittivity between the data points is reconstructed through cubic splines interpolation**.

3.5 Additional functions

3.5.1 description("test")

This function is used to add information to the material. Its primary use is to document it in a way the GUI can read it.

3.5.2 validity_range(λ_{min},λ_{max})

Any dielectric model has a limited validity range. This function is there to document this to the software. It will define the validity range between

3.6 Full example

The following example defines a gold dielectric model from 400 to 1000 nm. The permittivity first is set to 1.03, and a drude model and two critical points models are then added. The validity range is then set based on the original data fit.

```
epsilon_infty(1.03)
add_drude(1.3064e16,1.1274e14)
add_crit_point(0.86822,4.0812e15,-0.60756,7.3277e14)
add_crit_point(1.3700,6.4269e15,-0.087341,6.7371e14)

validity_range(400e-9,1000e-9)
```

Part II

Finite Differences

Chapter 4

Theoretical bases

4.1 Finite differences

4.2 FDTD

4.3 Dispersive materials

4.4 PMLs

4.5 FDFD

Chapter 5

Structures

5.1 Main script and the structure object

Structures are not defined within the main script but in their own specific script. They still need to interact with the main script through **structure objects**. Those are defined by the user through the mean of the **Structure**(**"structure script"**) function, where **structure script** is the path to the relevant file. Example:

```
struct=Structure("structures/nanorods_grid.lua")
```

Structure objects have their own functions that can be used to interact with them after loading the file.

5.1.1 Parameters

Structure files can be written in a "generic" way, depending on named input parameters. These parameters have a default value, but can be set through the **parameter**(**"name"** , **value**) member function, where **name** is the related parameter name, and **value** what to assign to it. Example

```
struct:parameter("P",500e-9)
```

5.1.2 Other functions

print(**"directory"**,**Dx**,**Dy**,**Dz**)

This function will turn the structure into a sequence of images within **directory/grid**, using a color gradient to display the material index. It uses **Dx**, **Dy** and **Dz** as the spatial discretization to do so. Example

```
struct:print("test_directory",5e-9,5e-9,5e-9)
```

5.2 Structure script

Geometry files serve to define the structures that will be used in FDTD simulations. Like the main script, those files are written in Lua. The structures definition is done through functions that will assign an “index” to different parts of the discretized space. This index will then be linked to a specific material.

It is required that the variables **lx**, **ly** and **lz** are defined somewhere in the script file. Those variables define the size of the structure cell. For instance:

```
lx=300e-9
ly=300e-9
lz=150e-9
```

Except for very specific cases, the functions call order will matter. This allows to create more complex structures than if the functions call order were random. For instance:

```
add_layer("Z",0,100,1)
add_block(25,50,25,50,25,50,0)
```

defines a layer of index 1, then creates a hole of index 0 inside it, while

```
add_block(25,50,25,50,25,50,0)
add_layer("Z",0,100,1)
```

first creates a block of index 0, then overwrites it with the layer of index 1.

5.2.1 Parameters

As seen in 5.1.1, structure files can take named parameters into account. To do so, they need to be declared and given a default value. It is done with the **declare_parameter**(**"name"**,**value**) function. This creates a Lua variable with the related **name** and a default **value**. This variable can be used like any other variable afterwards. Example

```
declare_parameter("p",300e-9)
declare_parameter("h",300e-9)

lx=p
ly=p*math.sqrt(3)
lz=200e-9+h
```

Here, the Structure extent is defined through the two parameters first declared. Parameters must be declared before being used.

5.2.2 Primitives

add_block(**x1**,**x2**,**y1**,**y2**,**z1**,**z2**,**index**)

Defines the region between **x1**, **y1**, **z1** and **x2**, **y2**, **z2** as of index **index**, giving this region the shape of a block.
Example:

```
add_block(0,100e-9,0,150e-9,200e-9,300e-9,4)
```

add_cone(Ox,Oy,Oz,Ax,Ay,Az,r,index)

Defines a cone of radius **r** and index **index**. It's origin is given by the $\vec{O}$ vector, while its direction and length are defined through the $\vec{A}$ vector.
Example:

```
add_cone(50e-9,50e-9,0,0,0,150e-9,1000e-9,1)
```

add_cylinder(Ox,Oy,Oz,Ax,Ay,Az,r,index)

Defines a cylinder of radius **r** and index **index**. It's origin is given by the $\vec{O}$ vector, while its direction and length are defined through the $\vec{A}$ vector.
Example:

```
add_cylinder(50e-9,50e-9,0,0,0,150e-9,1000e-9,1)
```

add_layer("dir",x1,x2,index)

Defines a layer of index **index** along the **"dir"** direction between **x1** and **x2**. This layer fills space entirely in the two other directions. **"dir"** can take the following values: **"X"**, **"Y"** or **"Z"**.
Example:

```
add_layer("Z",0,100e-9,2)
```

add_sphere(x,y,z,r,index)

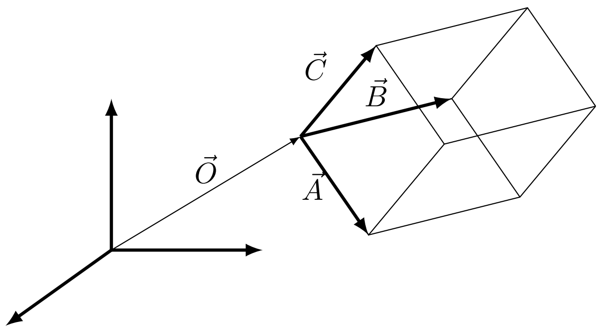
Sets a spherical region of space with the index **index**. This region is centered on **x**, **y**, **z** and is of radius **r**.
Example:

```
add_sphere(100e-9,100e-9,50e-9,25e-9,2)
```

add_vect_block(Ox,Oy,Oz,Ax,Ay,Az,Bx,By,Bz,Cx,Cy,Cz,index)

Defines a parallelepiped of index **index**. Its offset is given through the $\vec{O}$ vector, while the $\vec{A}$, $\vec{B}$ and $\vec{C}$ vectors define the edges. Those three vectors do not need to take the offset into account (c.f. figure below).
Example:

```
add_vect_block(100e-9,100e-9,50e-9,
               10e-9,0,0,
               0,10e-9,0,
               0,0,5e-9,5)
```



5.2.3 Non shape functions

default_material(index)

Fills the whole grid with the index **index**. This is mostly usefull to initialize the grid.
Example:

```
set_full(0)
```

flip(x,y,z)

loop(x,y,z)

5.2.4 Lua defined functions

Additionally to the preview basic shapes, users can define new shapes through Lua functions. To do this, the function definition shall follow a particular template.

add_lua_def("name",var_arg...,index)

Example

The following example illustrates the case of a torus defined through Lua, as this shape is not one of the basic shapes.

```
function torus(x,y,z,x0,y0,z0)
        r0=350e-9;
        a=40e-9
        b=20e-9

        r=math.abs(math.sqrt((x-x0)*(x-x0)+(y-y0)*(y-y0))-r0)
        h=math.abs(z-z0)

        rat=b*b*(1-r*r/a/a)

        if h*h<=rat
                then
                        return 1
        end
        return 0;
end

add_lua_def("torus",50e-9,100e-9,50e-9,2)
```

5.2.5 Meshes

5.3 Full scripts example

5.3.1 Geometry

The following script is a full geometry script example that defines a nanorods grid.

```
lx=300e-9
ly=300e-9
lz=150e-9

default_material(0)
add_layer("Z",0,50e-9,1)
add_block(105e-9,235e-9,0,35e-9,50e-9,70e-9,2)
add_block(0,35e-9,105e-9,235e-9,50e-9,70e-9,2)
```

First, **lx**, **ly** and **lz** are defined. Then the whole grid is intialized with the index 0 through the **set_full** function. A layer of index 1 is added at the bottom of the FDTD grid along the **"Z"** direction through **add_layer**, and will be the substrate in the simulation. Finally, two nanorods of index 2 are defined with the function **add_block**, and will later in the main script be associated to the gold material.

5.3.2 Main script

The following lines will instance a **structure object** based on the previous script, and pass it to the FDTD solver within the main script.

```
struct=Structure("../structures/nanorods_grid.lua")

fdtd=MODE("fdtd_normal")
...
fdtd:structure(struct)
...
```

Chapter 6

FDTD: scripting interface

6.1 Common functions

Some functions are common to some FDTD modes of the software, and are listed here.

6.1.1 auto_tsteps(Nt_max,Nt_check,lambda_min,lambda_max,coeff,Np,"layout")

This functions attemps to evaluate the advancement of a computation, and end if when results are unlikely to change much. To do so, it performs a check every **Nt_check** iterations in the following manner.

First **Np** points are randomly selected at the start of the simulation on the boundaries of the grid. Boundaries can be enabled or disabled through the **"layout"** argument which is a string of three characters, each of which being either **n**, **u**, **d** or **b**, which stand for **none**, **up**, **down** and **both**, in the x, y and z order. For instance **"dnb"** would enable the x_{min}, z_{min} and z_{max} boundaries, while **"nun"** would only enable the y_{max} boundary.

Once the points are chosen, they are assigned a wavelength chosen between **lambda_min** and **lambda_max**, and the related fourier transforms are performed at those locations.

Then, the algorithm evaluates the convergence of those fourier transforms through iterations, by comparing the variation to amplitude ratio of each transform to the **coeff** argument (typically around 10^{-4} or 10^{-5}). If the ratio for a point is less than said **coeff**, then the fourier transform at that point is assumed stable. Once that criterion is checked for every point, the simulation stops.

If this criterion is never met, the simulation will only end after **Nt_max** iterations.

```
fdtd:auto_tsteps(100000,500,400e-9,1000e-9,1e-5,200,"nnb")
```

6.1.2 compute()

Runs the FDTD simulation at the current state, that is with all the parameters set so far. Any following changes to the parameters will be ignored.

```
fdtd:compute()
```

6.1.3 Dx(dx)

Sets the x discretization to **dx**.

```
fdtd:Dx(5e-9)
```

6.1.4 Dy(dy)

Sets the y discretization to **dy**.

```
fdtd:Dy(5e-9)
```

6.1.5 Dz(dz)

Sets the z discretization to **dz**.

```
fdtd:Dz(5e-9)
```

6.1.6 Dxyz(d)

Sets the discretization along x, y and z to **d**.

```
fdtd:Dxyz(5e-9)
```

6.1.7 N_tsteps(Nt)

Defines the number of time steps **Nt** of a simulation.
Example

```
fdtd:set_N_tsteps(30000)
```

6.1.8 material(index,"material_file")

Links the material file **"material_file"** to the index **index** defined in the simulation geometry file.
Example:

```
fdtd:material(1,"mat_lib/Glass150.lua")
```

If the related material is supposed to be of constant real refractive index, **"material_file"** may be replaced with a call to **const_material**, of which the only argument is that refractive index. A material script will be generated on the fly.
Example:

```
fdtd:material(1,const_material(1.5))
```

6.1.9 padding(xm,xp,ym,yp,zm,zp)

- **xm**: PML padding along $-\vec{x}$
- **xp**: PML padding along $\vec{x}$
- **ym**: PML padding along $-\vec{y}$
- **yp**: PML padding along $\vec{y}$
- **zm**: PML padding along $-\vec{z}$
- **zp**: PML padding along $-\vec{z}$

6.1.10 pml_X(Npml,kappa,sigma,alpha)

Functions of the **pml_X** kind are used to define the PMLs (absorbing boundary conditions) of a computation. There are six of them:

- **pml_xm**: PML along $-\vec{x}$
- **pml_xp**: PML along $\vec{x}$
- **pml_ym**: PML along $-\vec{y}$
- **pml_yp**: PML along $\vec{y}$
- **pml_zm**: PML along $-\vec{z}$
- **pml_zp**: PML along $\vec{z}$

They take four arguments:

- **Npml** define the number of PML cells, usually around a dozen
- **kappa** defines what would roughly be ε_r at the PML end, and thus how it slows down waves propagating into it. A value between 15 and 25 is usually suitable.
- **sigma** corresponds to PMLs absorption. A value of $1/\sqrt{n}$ seems usually efficient.
- **alpha** is a shift in the complex plane of the PML spatial coordinates. It is mostly useful if PMLs encounter evanescent waves. Typically around 0.2.

The **kappa**, **sigma** and **alpha** coefficients are not constant along the PML axis. The first two grow together with the PML depth, while the latter decreases in value. Example:

```
fdtd:pml_zp(25,25,0.7,0.2)
fdtd:pml_zm(25,25,1,0.2);
```

6.1.11 prefix("name")

Names the simulation with the prefix **"name"**, which will be used in the name of several files written as computation results.

6.1.12 polarization("pol")

Defines the polarization of the incident field in a simulation. **"pol"** can be only one of those two cases: **"TE"** or **"TM"**.
Example:

```
fdtd:polarization("TE")
```

6.1.13 register_sensor(sensor)

Links a sensor, created through the **create_sensor** function (c.f. section 6.6), to the simulation.
Example:

```
fdtd:register_sensor(fieldmap_1550)
```

6.1.14 spectrum(lambda min , lambda max , Nl)

Defines the spectral range of analysis between **lambda min** and **lambda max** (in meters), with **Nl** points homogeneously distributed between those two wavelengths. Warning: requesting too many points can increase the computation time.
Example:

```
fdtd:spectrum(3000,1e-6,3e-6)
```

6.1.15 structure(structure object)

Links the simulation to the geometry file **"geometry_file"**. For further details, see 5.
Example:

```
structure_obj=Structure("structures/nanorods_grid.lua")

fdtd:structure(structure_obj)
```

6.1.16 struct_append("lua")

```
fdtd:struct_append("subdivision_modifier(2,2,2)")
```

6.1.17 struct_parameters("lua")

```
fdtd:struct_parameters("period=1")
```

6.1.18 time_mod(fact)

Multiplies the natural time step $\Delta t = \frac{\min(\Delta x, \Delta y, \Delta z)}{c\sqrt{3}}$ by a factor **fact** that must be set between 0 and 1.

6.2 Standard mode FDTD

```
fdtd=MODE("fdtd")
```

6.2.1 Specific functions

The **polarization** and **spectrum** functions are disabled in this mode.

register_source(source)

Links a source, created through the **create_source** function (c.f. section 6.7), to the simulation.
Example:

```
fdtd:register_source(oscillator)
```

6.3 Normal incidence FDTD

In this mode, the structure is an infinitely periodic array in the $\vec{x}$ and $\vec{y}$ directions. The incident field is a gaussian pulse propagating along $-\vec{z}$, for which the spectrum, and thus the analysis spectrum, is defined through the **spectrum** function. At the end of the computation several files are written onto the hard drive, each prefixed with the name given to the **prefix** function.

The **prefix_show_norm.m** file can be called by MatLab or GNU Octave. It reads **prefix_spectdata_norm** and plots the reflection and transmission coefficients of the structure, with no regard for the polarization, and then the transmittivity, reflectivity and absorption.

Finally, **prefix_show_norm2.m** reads the **prefix_spectdata_norm2** file and outputs a detailled plot of all the complex reflection and transmission coefficients for the three components of the electric field. The goal is to show any potential polarization rotation effect.

6.3.1 Specific functions

PMLs

In this mode, only the **pml_zm** and **pml_zp** functions are available. The other four similar functions will be disregarded if they happen to be called.

6.3.2 Full script example

```
structure=Structure("nanorods_grid.lua")

fdtd=MODE("fdtd_normal")
fdtd:prefix("fdtd_")
fdtd:polarization("TE")
fdtd:spectrum(1e-06,2e-06,481)
fdtd:N_tsteps(20000)
fdtd:structure(structure)
fdtd:Dxyz(5e-09)
```

```
fdtd:pml_zm(25,25,1,0.2)
fdtd:pml_zp(25,25,1,0.2)
fdtd:material(0,const_material(1))
fdtd:material(1,const_material(1.5))
fdtd:material(2,"mat_lib/Au_1m_5m_Vial.lua")

fdtd:compute()
```

6.4 Oblique incidence FDTD using the Aminian and Rahmat-Samii method

[4]

$$\lambda \sin \vartheta = \text{Constant} \tag{6.1}$$

6.4.1 Specific functions

kx_auto(ang_min,ang_max)

kx_fixed_angle(Nkx,lambda_min,lambda_max,angle)

Runs **Nkx** simulations for a specific angle **angle**, with the target wavelength varying between **lambda_min** and **lambda_max**.

kx_fixed_lambda(Nkx,lambda,ang_min,ang_max)

kx_target(lambda,angle)

PMLs

In this mode, only the **pml_zm** and **pml_zp** functions are available. The other four similar functions will be disregarded if they happen to be called.

6.4.2 Real script example

```
structure=Structure("structures_priv/hexa_cones_s150_h300_inv.lua")

fobl=MODE("fdtd_oblique_ARS")
fobl:prefix("fobl_");
fobl:polarization("TE");
fobl:kx_fixed_lambda(3,425e-9,10,60)
fobl:structure(structure);
fobl:N_tsteps(100000);
fobl:spectrum(3000,400e-9,450e-9);
fobl:pml_zp(25,25,0.7,0.2)
fobl:pml_zm(25,25,1,0.2);
fobl:material(0,"mat_lib/Ind181.lua")
fobl:material(1,"mat_lib/Fspace.lua")
```

```
fobl:compute()
```

6.5 Single particles

This is a specific mode dedicated to simulating single particles. It is called through

```
fdtd=MODE("fdtd_single_particle")
```

In this mode, the user can choose the periodicity of the structure. For instance, if the **pml_x** functions are not used the software will consider the structure is periodic along x. No analysis is performed, and that so this part is left to the user.

6.5.1 Specific functions

structure_aux("geometry_file")

This function is used to specify an auxiliary structure for the simulation. This structure is used for the padding around the main geometry, and is used to compute the incident field. If the function is not used the software will just use the main grid at $x = 0$, $y =$ as the auxiliary grid.

6.5.2 Real script example

```
structure=Structure("structures/donut_cavity_aux.lua")

fdonut=MODE("fdtd_single_particle")
fdonut:prefix(prefix);
fdonut:polarization("TE");
fdonut:structure(structure);
fdonut:structure_aux();
fdonut:auto_tsteps(100000,500,400e-9,1000e-9,1e-5,200);
fdonut:spectrum(3001,400e-9,1000e-9);
fdonut:pml_xp(25,25,1,0.2);
fdonut:pml_xm(25,25,1,0.2);
fdonut:pml_yp(25,25,1,0.2);
fdonut:pml_ym(25,25,1,0.2);
fdonut:pml_zp(25,25,1,0.2);
fdonut:pml_zm(25,25,0.8,0.2);
fdonut:material(0,"mat_lib/Fspace.lua")
fdonut:material(1,"mat_lib/Glass150.lua")
fdonut:material(2,const_material(index))
fdonut:material(3,"mat_lib/V_Au_400_1000_PCRC2.lua")

fdonut:compute()
```

6.6 Sensors

6.6.1 Common functions

location(x1,x2,y1,y2,z1,z2)

Sets the sensor location between **x1**, **y1**, **z1** (inclus) and **x2**, **y2**, **z2** (exclus). Those coordinates must be defined with respect to the computation geometry file, and this can be negative.

name("name")

Gives the sensor the name **"name"**, which will serve as a prefix to the names of the files written at the end of the computation.

orientation("orientation")

This defines the sensor orientation (its normal vector), through the **"orientation"** variable. Its value can be **"X"**, **"Y"**, **"Z"**, **"-X"**, **"-Y"** or **"-Z"**. This function has two main goals:

- setting a direction onto which the Poynting vector will be projected, for the sensors that need it.
- specifying the exact location of the sensor, which was previously defined through the **location** function. For instance, if the orientation is **"X"** or **"-X"** the **x2** variable will be ignored. Similarly, **"Y"** will affect **y2** and **"Z"** will affect **z2**.

spectrum(lambda)

Sets the sensor analysis wavelength **lambda**.

spectrum(lambda_min,lambda_max,Nl)

Sets the sensor analysis spectrum between **lambda_min** and **lambda_max**, with **Nl** sampling points.

6.6.2 Sensor types

fieldmap

This sensors is dedicated to creating 2D fieldmaps. The fieldmap is computed for a single wavelength, specified through the single argument variant of **spectrum**.

At the end of the simulation, several files are created:

- **name_fieldmap.bmp**: a quick fieldmap image using a non-linear normalization
- **name.m**: a MatLab script that will display the fieldmap
- **name_Ex_raw**: a file containing the sensor data for the x component of the field
- **name_Ey_raw**: a file containing the sensor data for the x component of the field
- **name_Ez_raw**: a file containing the sensor data for the x component of the field

Example:

```
map=create_sensor("fieldmap")
map:spectrum(405e-9)
map:name("map")
map:orientation("Y")
map:location(0,500,0,1,0,500)

fdtd:register_sensor(map)
```

fieldblock

planar_spectral_poynting

This sensor integrates the Poynting vector flux across a planar surface.

6.7 Light sources

6.7.1 Common functions

location(x1,y1,z1)

Set the source location at (**x1**, **y1**, **z1**). Those coordinates must be defined with respect to the computation geometry file, and this can be negative.

location(x1,x2,y1,y2,z1,z2)

Set the source location between **x1**, **y1**, **z1** (included) and **x2**, **y2**, **z2** (excluded). Those coordinates must be defined with respect to the computation geometry file, and this can be negative.

orientation("orientation")

Defines the source orientation, that is **"X"**, **"Y"** or **"Z"**. This function is only ûsed by point sources.

spectrum(lambda)

Defines the source wavelength as **lambda**.

spectrum(lambda_min,lambda_max)

Defines the source spectrum as gaussian, between **lambda_min** and **lambda_max**.

6.7.2 Source types

oscillator

This source adds field to the simulation grid at each time step. This done through a operation like:

$$E(t+1) = E(t) + sin(\omega t) \tag{6.2}$$

As such, this souce is not exactly a radiating electric dipole, but just shows a similar behavior. Furthermore, field injection method can potentially create a static electric and magnetic field at the end of the simulation, and so this source must be used with care.

This is a point source, and so its location is specified through the three arguments variant of the **location** function. The excited field component is set by the **orientation** function. The source has a wide spectrum, and so the two arguments variant of the **spectrum** function needs to be used.
Example:

```
src=create_source("oscillator")
src:location(250,0,410)
src:orientation("Z")
src:spectrum(300e-9,1000e-9)

fdtd:register_source(src)
```

Electric dipole

Todo.

Magnetic dipole

Todo.

Chapter 7

FDTD: GUI

Part III

Ray optics: Selene

Chapter 8

Selene: scripting interface

8.1 Principles

8.1.1 Organization

Raytracing calculations are done by setting a scene up and feeding it to a renderer. This is first done by calling **MODE**("selene") which returns an instance of the Selene renderer. It only has a few functions that can modify it.

The first one is **N_rays_total**(**N**) which will set the total number of rays that need to be computed.

Then the user has to define the optical elements, light sources, materials, etc, and feed them to the rendered. This is done by **add_object**(object) where object is a Selene object, and **add_light**(light) where light is a Selene light source.

Once this is done, the user needs to call **render**() to start the computation. This function can be called several times to run several simulations in a raw, after modifying an object's properties for instance.

Eventually, a typical script will look like

```
sln=MODE("selene")
sln:N_rays_total(2000000)

% Scene definition

sln:add_object(object_1)
sln:add_object(object_2)
sln:add_light(source)

sln:render()
```

8.1.2 Inner workings

This is a summary of the rendering algorithm:

1. At the beginning of the rendering the object and light sources locations are recomputed based on other objects. Various other properties can be computed, or memory or files allocated.

2. A ray is created from one of the registered light sources and is fetched by renderer
3. The intersection between this ray and every object is computed
    - First, by computing the intersection with the bounding box first to have a quick approximation
    - Then by computing the intersection with the various faces of the object, if the first step showed potential intersection
4. The intersections are looked through for the one with the smallest positive time, and the others are discarded
5. The object related to that intersection is asked to treat the ray based on the face intersected and the intersection location
6. It is determined from which side of the face (up or down, see 8.3) the ray is coming
7. The relevant Interface Response Function is given the ray and the refractive index of each side of the face, and determines how the ray must be redirected based or absorbed based on its optical properties
8. Intersections for the new ray are computed again until it is either absorbed or lost
9. The renderer fetches a new ray until the required number of rays has been computed

8.1.3 Probabilistic behavior

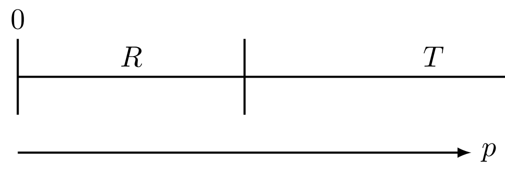
Raytracing usually follows a Monte-Carlo approach, that is to say one generates many random rays in a scene, and cast them with random directions and/or locations based on the properties of the sources, and have them intersect with objects.

How you handle those interactions, however, can be of two kinds: patch branching and probabilistic. To illustrate path branching, let's assume one computes the interaction of a ray with a glass surface. The transmission and reflection coefficients are not zero, so from one incident ray two would be generated. Those would then propagate again and generate new rays and so on. One can immediately notice that, in pathological cases, this could make the number of rays grow exponentially. Given enough computational time, it should be possible to address all of them, but then a lot of CPU time would have been spent computing the children of only one initial ray, giving much weight to that particular location.

With Selene, we have chosen to go with a full probabilistic approach. If an interface has a reflection, transmission, and absorption coefficient, energy conservation requires that

$$R + T + A = 1 \tag{8.1}$$

If one interprets this probabilistically, that means the sum of the probabilities for the ray to be reflected, transmitted or absorbed is 1. Another way to see it, is that one could split the numbers between 0 and 1 into three regions of lengths R, T and A as follows:



if one takes a random number p between 0 and 1, the region p falls in will determine the result of the calculation. This approach can be generalized to any number of effects the resulting ray could encounter. This ensures that one ray can only give rise to at most one ray.

While the probabilistic approach does not compute the exact behavior of a specific point, it will still statistically properly reconstruct the behavior of the whole neighborhood of it.

8.1.4 Simulation summary

After each call of **render** the program outputs a summary log under the file name **selene_render_X** within the **output_directory**, where **X** is the render number, starting from 0. This file is a mini-script that specifies

- the requested number of rays
- the power of each source
- the number of ray of each source
- the power carried by each ray

Its purpose is for the analysis of results after the render, for some of this information will be required.

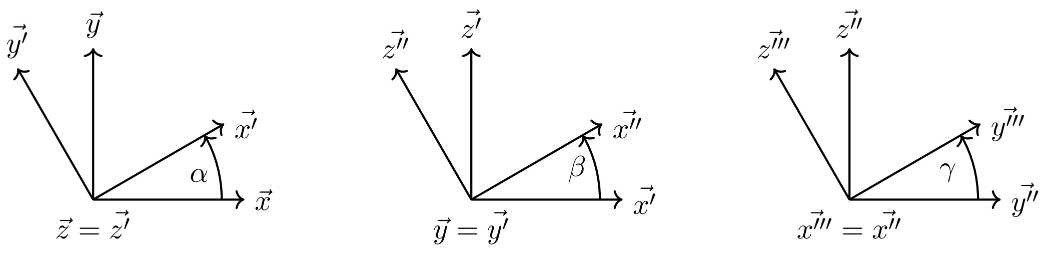
8.2 Relative positioning

Selene's positioning system can be relative. That means objects can be placed relatively to others, or the world's origin of course.

8.2.1 Orientation

The first part of the positioning system is defining the object's orientation. Each object comes with its own frame, and most object primitives are aligned with a particular axis within this frame. For instance, the axis of a cylinder is aligned with the $\vec{x}$ axis of the object's frame. To rotate the cylinder in the world, one thus needs to rotate the frame itself.

This rotation is specified through Euler angles. The convention chosen in this software is yaw→pitch→roll, or $zy'x''$. That is, first the frame is rotated by an angle α around $\vec{z}$, then by an angle β around the new $\vec{y}$ axis, and finally by an angle γ around the new $\vec{x}$ axis.

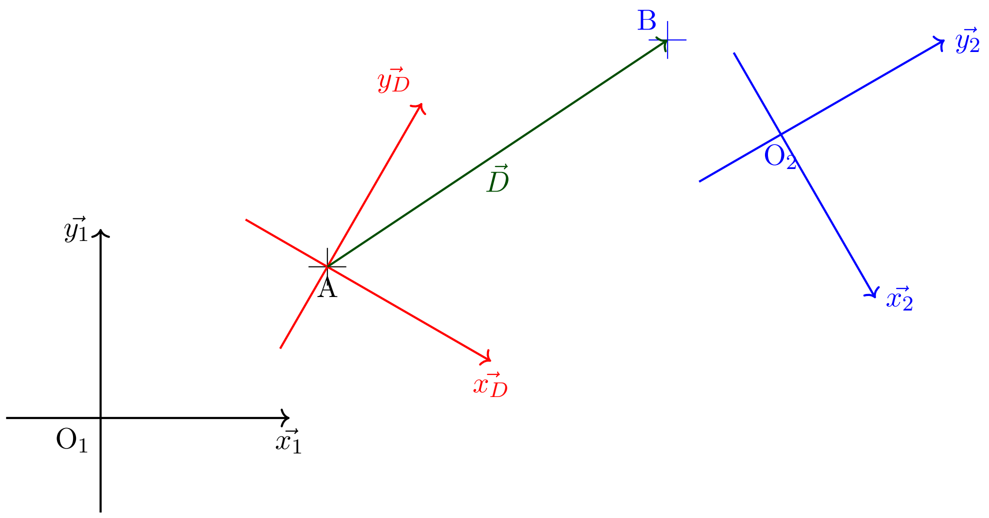


Now, this rotation can either be absolute, or relative to another object. In the latter case, the reference rotation frame will need to be set through the **rotation_frame**(**object**) function. The frame of the object we want to manipulate will be set identical to the frame of the reference, and then rotated.

8.2.2 Placement

The second part of the system is relative locations. Once an objet's orientation is set, it can be placed in the scene relatively to the world's origin or to another object. The purpose is to define object groups that can move with eachother. Now, this relative location does not necessarily need to be defined from an object's origin to another's. Indeed, some objects have particular points of interest, and it can be useful to place objects relatively to those points. For instance, two lenses can be aligned so that their focal points match. In this case, the relative locations of the lenses' centers is irrelevant. Those points will be called anchors, and each object type will have its own anchors depending on its parameters.

The process is illustrated with the figure below.



In this, we have two objects O_1 and O_2. First, O_2 had its orientation set, which is different from O_1's orientation. O_1 has an anchor A, and O_2 an anchor B, and we want both anchors to be displaced relatively to eachother through the vector $\vec{D}$. Moreover, $\vec{D}$ itself can be expressed in a frame that is independent from O_1 and O_2. We will then define the following properties for O_2:

- B will be its new origin
- A, belonging to O_1, will be set as its displacement reference.
- the frame D will be set based on any other frame in the scene, and $\vec{D}$ will be defined within it

We will then have

$$\vec{O_2} = \vec{O_1} + \vec{O_1A} + \vec{D} + \vec{BO_2} \tag{8.2}$$

8.3 Faces and Interface Response Functions

8.3.1 Face orientation

Up and Down

The core elements with which rays in Selene interact are the faces of the various objects of a scene, and this section aims to describes what happens.

Faces come in favious shapes. They can be a plane, a parabola, a triangle, etc. In every case, faces have two sides, and the software is made so that do not necessarily behave the same sway. We thus need to be able to differentiate both sides, and this is done by the orientation of the normal vector to a face.

The direction towards which the normal vector is pointing will be known as "up", or the outside of a closed surface, and the opposite direction will be known as "down", or the inside of a closed surface. The direction of the normal vector determines how an incident ray will

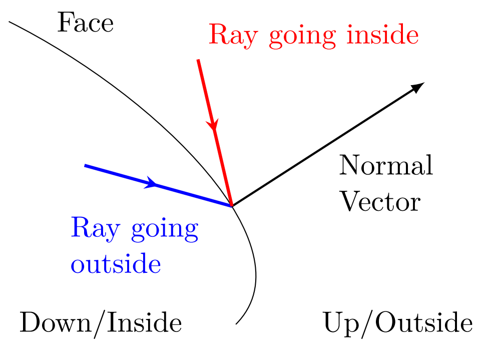


Figure 8.1: Conventions around faces in Selene

interact with the face. If the scalar product between the direction vector of a ray and the normal factor of the face is positive, then the ray is going outside, and will be treated using the "down" properties of the face. If that scalar product is negative, the ray is going inside, and will be treated using the "up" properties.

Properties of each side

The most obvious property difference between the up and down sides is the material they refer to. After all, faces are supposed to represent where there is a change in refractive index. As such, each side will point to a different material defined by the user. Note that in some cases, like a transparent sensor surface that is not supposed to modify rays, the materials of both sides can be the same.

But merely knowing the materials is not enough to define what will happen to an incident ray. Indeed, while one could just use them to compute refraction and reflection, this is not the only way a surface can behave. Photons can be scattered, diffracted, absorbed, etc. To handle those different behaviors, we use what we call an Interface Response Function, or IRF. The purpose of the IRF will be to redirect a ray or destroy it depending of the various properties of the face.

We've chosen to make it so that both sides of a face don't necessarily have the same IRF. It can be used to define non-physical, but convenient, behaviors such as a face being perfectly antireflective from one side, and perfectly reflective from the other, or to input simulation data for both sides (the behavior of nanostructures for instance will not be perfectly symmetrical). As such, both sides will refer to a potentially different IRFs.

Certains IRFs will require complete frames to determine the direction of the outgoing ray. In the case of Snell's law such a frame can be determined from the normal vector and the ray direction alone, but this would not work with a diffraction grating. As such, faces have an additional properties: a tangent vector. This vector is specified per face, per side, and can be fixed or have a special configuration that will adapt to the face. More details are to be found in the objects descriptions.

8.3.2 The Interface Response Function

As stated before, the purpose of the Interface Response Function is to turn an incident ray into a reflected or transmitted ray, or to absorb it, in the manner described in section 8.1.3. This this done in accordance to the refractive index of the related face and its tangent vector.

To each specific optical behavior corresponds an IRF. See the section 8.4.1 to known how to assign them to objects.

Default IRFs

There are several default, basic IRFs, than can be passed to objects. They are

SEL_IRF_FRESNEL: the interface's response obeys the Fresnel coefficients of the associated materials, while the

SEL_IRF_PERFECT_ABSORBER: all rays hitting the interface are absorbed and disappear

SEL_IRF_PERFECT_ANTIREFLECTOR: all rays hitting an interface go through it, but still obey Snell's law

SEL_IRF_PERFECT_MIRROR: rays are always reflected

User defined IRFs

8.4 Objects

Selene objects are added through the function **Selene_object**("type",**args**). Here, **args** is actually a variable sequence of argument depending on the object **type**. For instance, a sphere will only have one possible argument, its radius, while a cylinder will have two: its length and radius. This function returns an object which can then be manipulated in the script (see 8.4.1). Usage example:

```
obj=Selene_object("cylinder",0.5,5.0)
```

8.4.1 Functions common to all objects

contains(**x**,**y**,**z**)

Returns 1 if the object contains the point defined by (x, y, z), and zero otherwise. Only valid for volume objects. It can still be called with the others but will return an undefined value.

```
var=obj:contains(0.25,0,1.5)
```

default_in_IRF(**IRF**)

Sets the IRF for every face of the object, on the down side, to **IRF**.

```
obj:default_in_IRF(SEL_IRF_PERFECT_ABSORBER)
```

default_in_mat(**Material**)

Sets the material for every face of the object, on the down side, to **Material**.

```
obj:default_in_mat(Glass)
```

default_IRF(**IRF**)

Sets the IRF for every face of the object, on the both sides, to **IRF**.

```
obj:default_IRF(SEL_IRF_FRESNEL)
```

default_out_IRF(**IRF**)

Sets the IRF for every face of the object, on the up side, to **IRF**.

```
obj:default_out_IRF(SEL_IRF_PERFECT_MIRROR)
```

default_out_mat(**Material**)

Sets the material for every face of the object, on the up side, to **Material**.

```
obj:default_out_mat(Air)
```

face_down_IRF(**face number**,**IRF**)

Sets the down IRF of the face indexed by **face number** (starting at 0) to **IRF**.

```
obj:face_down_IRF(0,SEL_IRF_PERFECT_ANTIREFLECTOR)
```

face_down_mat(**face number**,**Material**)

Sets the down material of the face indexed by **face number** (starting at 0) to **Material**.

```
obj:face_down_mat(12,sapphire)
```

face_down_tangent(**face number**,**"type"**)

Sets type of the down tangent of the face indexed by **face number** (starting at 0) to **type**. See the objects properties for more details.

```
obj:face_down_tangent(4,"polar")
```

face_down_tangent(**face number**,**x**,**y**,**z**)

Sets =the down tangent of the face indexed by **face number** to the vector (**x**,**y**,**z**).

```
obj:face_down_tangent(0,1,0,0)
```

face_IRF(**face number**,**IRF**)

Sets the IRF of the face indexed by **face number** (starting at 0), on both sides, to **IRF**.

```
obj:face_IRF(0,SEL_IRF_FRESNEL)
```

face_up_IRF(**face number**,**IRF**)

Sets the up IRF of the face indexed by **face number** (starting at 0) to **IRF**.

```
obj:face_up_IRF(2,SEL_IRF_PERFECT_ANTIREFLECTOR)
```

face_up_mat(**face number**,**Material**)

Sets the up material of the face indexed by **face number** (starting at 0) to **Material**.

```
obj:face_up_mat(1,Air)
```

face_up_tangent(**face number**,**"type"**)

Sets type of the up tangent of the face indexed by **face number** (starting at 0) to **type**. See the objects properties for more details.

```
obj:face_up_tangent(2,"polar_neg")
```

face_up_tangent(face number,x,y,z)

Sets the up tangent of the face indexed by **face number** to the vector (**x**,**y**,**z**).

```
obj:face_up_tangent(4,0,1,0)
```

location(x,y,z)

Sets the object location to (**x**,**y**,**z**) within the translation frame. The true location will be defined according to the relative positioning system. See 8.2.1 for a detailled explanation.

```
obj:location(1,1,0)
```

name("object name")

Sets the object name to **name**.

```
obj:name("main_lens")
```

origin(anchor name)

Sets the origin of the object to its anchor defined by **anchor name**. See 8.2.1 for more details, and the objects descriptions for a list of the anchors.

```
obj:origin(SEL_OBJ_BOX_FACE_XP)
```

relative_origin(relative frame,anchor name)

Sets the relative origin of the object to anchor of the **relative frame** designed by **anchor name**. See 8.2.1 for more details, and the objects descriptions for a list of the anchors.

```
obj:relative_origin(object_0,SEL_OBJ_BOX_CORNER_XM_YM_ZM)
```

rotation(α,β,γ)

Set the angles of rotation of the object to α, β and γ, in degrees, relative to the **rotation frame**. See 8.2.1 for more details.

```
obj:rotation(45,20,0)
```

rotation_frame(frame)

Set the relative rotation frame of the object the to one held by **frame**. See 8.2.1 for more details.

```
obj:rotation_frame(object_1)
```

translation_frame(frame)

Set the relative translation frame of the object to the one held by **frame**. See 8.2.1 for more details.

```
obj:translation_frame(object_0)
```

sensor("transp"/"abs","options 1","options 2",...)

Will turn the object into a sensor. See the section 8.5 for a full description of the options and what they do.

```
obj:sensor("abs","wavelength","ID","obj_intersection")
```

8.4.2 Volume primitives

Volume objects are objects that are defined by a closed surface.

Box

As its name indicates, the **box** object is just a block with parallel faces, over variable width, heights and length. As such, adding one in the script will involve **Selene_object**(**"box"**,**lx**,**ly**,**lz**), where **lx** and so on will be the length along the x, y and z axes. By default, the box' origin is its center, meaning it extends from $-lx/2$ to $+lx/2$, and so on.
Example:

```
obj=Selene_object("box",2,3,4)
```

The default anchor is the center of the **box**: SEL_OBJ_BOX_CENTER

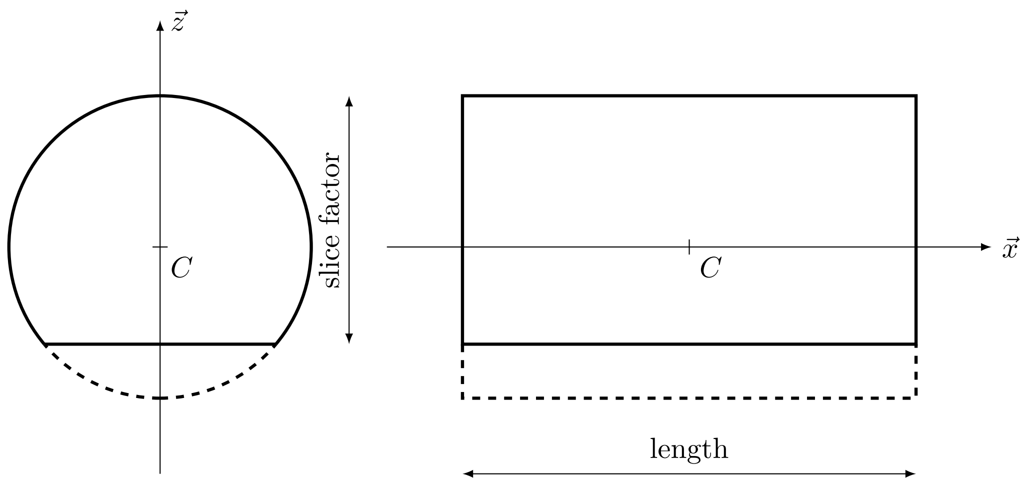
However, the following specific point serve as anchors for the **box** object. Here, XM refers to the first face on the x axis, and XP refers to the second face on the x axis. The same goes for the y and z axes. There are six face anchors corresponding to the center of each face, and eight corner anchors corresponding to each vertex of the box.

Faces	Corners
SEL_OBJ_BOX_FACE_XM	SEL_OBJ_BOX_CORNER_XM_YM_ZM
SEL_OBJ_BOX_FACE_XP	SEL_OBJ_BOX_CORNER_XP_YM_ZM
SEL_OBJ_BOX_FACE_YM	SEL_OBJ_BOX_CORNER_XM_YP_ZM
SEL_OBJ_BOX_FACE_YP	SEL_OBJ_BOX_CORNER_XP_YP_ZM
SEL_OBJ_BOX_FACE_ZM	SEL_OBJ_BOX_CORNER_XM_YM_ZP
SEL_OBJ_BOX_FACE_ZP	SEL_OBJ_BOX_CORNER_XP_YM_ZP
	SEL_OBJ_BOX_CORNER_XM_YP_ZP
	SEL_OBJ_BOX_CORNER_XP_YP_ZP

Cone

- **SEL_OBJ_CONE_CENTER**
- **SEL_OBJ_CONE_FACE**
- **SEL_OBJ_CONE_TIP**

Cylinder



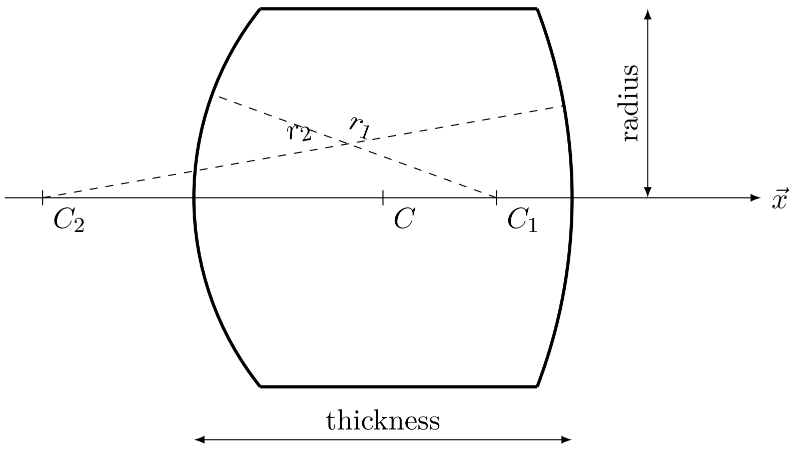
The anchor points are

- **SEL_OBJ_CYL_CENTER** : the cylinder center
- **SEL_OBJ_CYL_FACE_XM** : the center of the face first intersecting the x axis
- **SEL_OBJ_CYL_FACE_XP** : the center of the face last intersecting the x axis

Lens

While lenses could be defined from a sequence of spherical patches, a volume lens primitive is provided through the function **Selene_object**("**lens**",**thickness**,**radius**,r_1,r_2).

A lens is made of two spherical caps and one cylinder. **thickness** specifies the intersection points of each cap with the $\vec{x}$ axis of the lens, the first one being at -**thickness**/2 and the second at **thickness**/2. r_1 and r_2 are the radii of each, and **radius** is the maxiùam radius of the lens from its axis.



The faces of the lens are ordered as such:

0. the spherical cap that first intersects the x axis
1. the second spherical cap
2. the surrounding cylinder

The lens anchors are

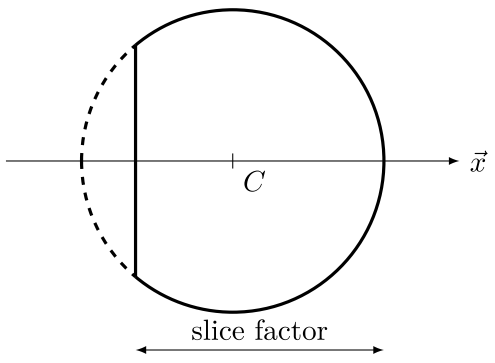
- **SEL_OBJ_LENS_CENTER**
- **SEL_OBJ_LENS_FRONT**
- **SEL_OBJ_LENS_REAR**

Prism

Sphere

Spheres are a primitive created through **Selene_object**("sphere",**radius**,**slice factor**), where **radius** is obviously the sphere radius.

Making hemispheres and every other similar shapes is the purpose of the **slice factor**. Just like for cylinders, a value of 1 will give a full sphere, and a value of 0.5 will result into an hemisphere.



There are two faces only to this object:

0. the spherical interface
1. the flat face

and two associated anchor points:

- **SEL_OBJ_SPHERE_CENTER**: the center of the sphere
- **SEL_OBJ_SPHERE_SLICE_CENTER**: the center of the flat face

8.4.3 Surface Primitives

Cylindrical surface

Conical surface

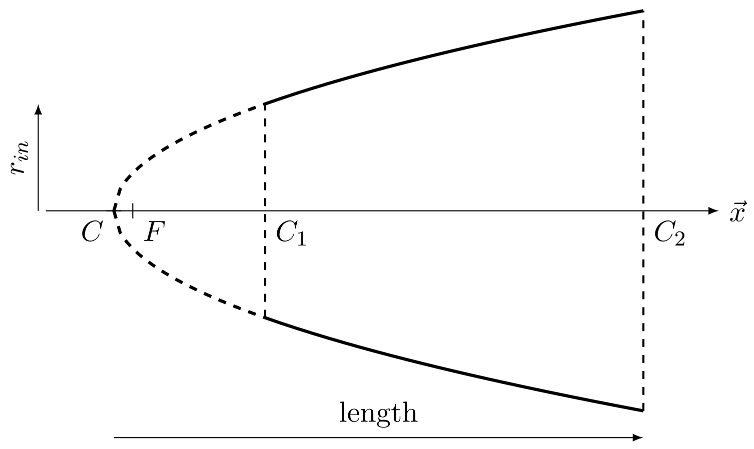
Disk

Disks are flat surfaces that can help make diaphragms, and are added through **Selene_object**(**"disk"**,r,r_{in}). The outer radius is defined by r, and inner radius by r_{in}. The disk extends in the (y, z) plane, and its normal vector is pointed towards $-\vec{x}$.

Its only notable point is **SEL_OBJ_CENTER**.

Parabola

Parabolas are basic optical surfaces that can be used to design aspheric lenses or parabolic mirrors. they can be added to a scene through **Selene_object**(**"parabola"**,**focal**,r_{in},**length**), with **focal** the location of the focal point with respect to the apex and **length** the total length of the parabola. The radius parameter r_{in} is there to provide an easy way to open an aperture around the parabola tip.



There is only one face to it, and its normal vector points outwards. There are, however, four notable points that can serve as anchors:

- **SEL_PARABOLA_CENTER**: the tip of the parabola if it were full, noted by C on the diagram above
- **SEL_PARABOLA_FOCUS**: the focal point of the parabola
- **SEL_PARABOLA_IN_CENTER**: the center of the disk formed by the small aperture, noted C_1 above
- **SEL_PARABOLA_OUT_CENTER**: the center of the disk formed by the end of the parabola, noted C_2 above

Rectangle

The rectangle is the flat equivalent of the box. Is this created through **Selene_object**(**"rectangle"**,**ly**,**lz**). It extends in the (y, z) plane from -**ly**/2 to +**ly**/2, and -**lz**/2 to +**lz**/2.

The normal vector of the face is directed towards $-\vec{x}$. The only admissible anchor point is **SEL_OBJ_CENTER**.

Spherical patch

- **SEL_OBJ_SPHERE_PATCH_CENTER**
- **SEL_OBJ_SPHERE_PATCH_SLICE_CENTER**

8.4.4 Meshes

This object type has access to specific functions

- **add_mesh**(**"file_name"**): appends the mesh designed by **file_name** to the object's mesh
- **auto_recalc_normals**(): recompute the face normals so that they all point outside
- **N_faces**(): returns the total number of faces of the mesh

Faces groups

define_faces_group(**group number**,**first face**,**last face**)

- **faces_group_down_IRF**(**group number**,**IRF**)
- **faces_group_down_mat**(**group number**,**Material**)
- **faces_group_down_tangent**(**group number**,**"type"**)
- **faces_group_down_tangent**(**group number**,**x**,**y**,**z**)
- **faces_group_IRF**(**group number**,**IRF**)
- **faces_group_up_IRF**(**group number**,**IRF**)
- **faces_group_up_mat**(**group number**,**Material**)
- **faces_group_up_tangent**(**group number**,**"type"**)
- **faces_group_up_tangent**(**group number**,**x**,**y**,**z**)

8.4.5 Composite objects

Booleans

Composites

8.5 Sensors

There would be not much point to a simulation software if one could do no analysis thanks to it. As such, some kind of sensors are necessary. In Selene, sensors are objects marked as such.

This is done through the **sensor** function, of which the first parameter specifies what kind of sensor. The following parameters specify what the sensor needs to record. Example:

```
obj:sensor("abs","wavelength","ID","obj_intersection")
```

8.5.1 Absorbing sensors

This kind of sensor is basically a screen. Rays interacting with it are absorbed and disappear from the simulation. Absorbing sensors are specified with the **"abs"** parameter.

8.5.2 Transparent sensors

Object defined as transparent sensors behave according to their geometry, materials, IRFs, etc, but record interactions with rays. They can be used to monitor flux through a surface, or evaluate the number of rays that will be absorbed through a specific region of space. They are specified with the **"transp"** parameter.

8.5.3 Recording options

- **"wavelength"**: records the ray wavelength
- **"source"**: records the ID of the source that generated the ray
- **"path"**: records the ray path ID. It is assigned by the source at generation, and can thus be common to several rays. It is unique for a single source.
- **"generation"**: records the ray number within a path, starting from 0, and increasing after each interaction
- **"length"**: records the ray optical path length at the intersection point
- **"phase"**: records the phase of the ray at the intersection point
- **"world_intersection"**: records the coordinates of the intersection with respect to the scene frame
- **"world_direction"**: records the directoin of the ray at the intersection with respect to the scene frame
- **"obj_intersection"**: records the coordinates of the intersection with respect to the object frame
- **"obj_direction"**: records the directoin of the ray at the intersection with respect to the object frame

8.5.4 File format

Objects marked as sensors will generate a file with a name like **object__name__ray__sensor**.

The first line contains in the following order: the coordinates of the object center (three numbers), the vectors of the object frame (nine numbers), and the bounding box in the object frame. All those follow the x, y, z order.

The second line is a string sequence that lists all the recorded options names.

Each subsequent line will correspond to an interaction containing the requested data in the order specified by the second line.

8.6 Sources

Sources are what generate rays for the renderer to compute, and nothing would happen without them. They share many common functions with objects, but have a few dedicated ones. Source are created using the **Selene_light**("type",**options...**) function. For instance

```
light_1=Selene_light("point_planar")
```

will create a planar point source.

8.6.1 Functions common to all light sources

discrete_spectrum({lambda 1,lambda 2,...},{weight 1,weight 2,...})

Gives the source a discrete spectrum from a list of wavelength in meters, with their associated wavelengths

```
light:discrete_spectrum({400e-9,500e-9,600e-9},{1,2,1.5})
```

material(Material)

Sets the ambient material of the source to **Material**.

```
light:material(Glass)
```

location(x,y,z)

Sets the source location to (**x**,**y**,**z**) within the translation frame. The true location will be defined according to the relative positioning system. See 8.2.1 for a detailled explanation.

```
light:location(1,1,0)
```

name("source name")

Sets the source name to **name**.

```
light:name("laser")
```

polar_along(x,y,z)

Defines the axis vector (**x**,**y**,**z**) along which the source will try to orient the polarization as close as possible. The sign of the whole vector is of no importance.

```
light:polar_along(1,1,0)
```

polar_not(x,y,z)

Defines the axis vector (**x**,**y**,**z**) that the source will try to avoid for rays polarization as much as possible. The sign of the whole vector is of no importance.

```
light:polar_not(0,0,1)
```

polar_unset()

Leaves to the source the choice of the polarization vector for each ray.

```
light:polar_unset()
```

power(**value**)

Sets the power of the source to anchor of the specified **value** in Watts.

```
light:power(55)
```

relative_origin(**relative frame**,**anchor name**)

Sets the relative origin of the source to anchor of the **relative frame** designed by **anchor name**. See 8.2.1 for more details, and the objects and sources descriptions for a list of the anchors.

```
light:relative_origin(object_0,SEL_OBJ_BOX_FACE_XM)
```

rotation(α,β,γ)

Set the angles of rotation of the source to α, β and γ, in degrees, relative to the **rotation frame**. See 8.2.1 for more details.

```
light:rotation(0,0,90)
```

rotation_frame(**frame**)

Set the relative rotation frame of the source the to one held by **frame**. See 8.2.1 for more details.

```
light:rotation_frame(object_1)
```

spectrum(**"spectrum shape"**,**option**)

Defines the source as a broadband source with the related **spectrum shape**. It can have three different values, with associated options.

- **spectrum**(**"flat"**,**lambda min**,**lambda max**) will give the source a flat spectral flux between both boundaries, in meters
- **spectrum**(**"file"**,**"file_name"**) will load the associated file and assign it as the spectral flux of the source
- **spectrum**(**"planck"**,**lambda min**,**lambda max**,**temperature**) will define the source as a black body with the associated temperature in Kelvin, that will be sampled between the boundaries in meters

translation_frame(frame)

Set the relative translation frame of the source to the one held by **frame**. See 8.2.1 for more details.

```
light:translation_frame(lens)
```

wavelength(lambda)

Defines the source as a monochromatic source of wavelength **lambda**, in meters.

```
light:wavelength(533e-9)
```

8.6.2 Source types

Cone

Lambertian

Perfect beam

This source aims to represent a source place at infinity. Its rays are strictly parallel, along the x axis of the source. It is instanced by the **"perfect_beam"** parameter.

The function **aperture(radius)** will set the size of the disk emitting the rays with the corresponding **radius**, in meters.
Example:

```
light=Selene_light("perfect_beam")
light:aperture(5e-3)
```

Planar point source

This source is similar to the point source, except it will emit in its (y, z) plane only. A rotation by $\gamma = 90°$ is thus necessary to target the (x, y) plane. It is created through the **"point_planar"** parameter.
Example:

```
light=Selene_light("point_planar")
```

Point source

This is the most basic source. It will emit in all directions homogeneously. It is created through the **"point"** parameter.
Example:

```
light=Selene_light("point")
```

User defined source

8.6.3 Spectra

To different types of analysis will correspond different input spectra. We provide a few ways of handling them.

In order to easily match the CIE illuminants, we assume all the spectra are actually spectral power distributions, in $W \cdot m^{-1}$, although this does not define the actual power of the source.

Monochromatic sources

Those are the most basic types of spectra. Every ray generated by those sources will be assigned the wavelength defined by **wavelength**.

Poly-monochromatic sources

They are the bridge between monochromatic sources and truly polychromatic sources, and can approximate lights with a small number of emission rays.

They are basically several monochromatic sources bundled into one, but with different weights associated with each wavelengths. Those wavelengths are then probabilistically given to the generated rays according to those weights.

Such spectra are assigned through the **discrete_spectrum** function.

Polychromatic sources

Those are the truly polychromatic sources. In this, the spectrum is a continuous function going from λ_{min} to λ_{max}. Wavelengths are then assigned to generated rays thanks to a rejection algorithm between those boundaries. We provide three types of polychromatic spectra

Flat : every wavelength between λ_{min} and λ_{max} is equiprobable

Planck : the probability distribution here is the black body distribution in wavelength form

$$P(\lambda, T) = \frac{2 * h * c^3}{\lambda^5} \frac{1}{e^{-\frac{hc}{\lambda k_B T}} - 1} \tag{8.3}$$

User defined : in this case, the user provides a two columns text file, the first being the wavelength, and the second the associated probability. Wavelengths are then drawn to match this probability distribution

8.6.4 Power normalization, spectra and rays

Time

Time is an undefined quantity with respect to raytracing. One can could assume all the paths are computed exactly at the same time, or spread across several seconds. For convenience, it was decided computations would occur within one second, so that the values of power and total energy emitted could be treated as the same.

Energy

Generally, rays can not represent real photons due to the sheer number of those cast by a real light source. For instance, a 1W source would emit in the order of 10^{20} photons per second, and computing them all would require incredible computational power.

The issue is then figuring out how to represent those photons with far fewer of them, in particular the energy they carry. Real photons energy is wavelength dependent, with $E = hc/\lambda$, but using this is not particularly convenient.

For instance, let us assume we want to simulate a light source with a flat power spectrum: using the previous definition would imply that for one blue ray emitted by the source, there would be two red ones. Since the computation spatial accuracy is a factor of the number of rays cast, it would be twice as good for red ones than blue ones, which is leads to an imbalance in simulation results.

As such, it was decided that, by default, all rays should carry the same number amount of energy.

Number of rays per source

The total number of photons is set by the user, and they need to be distributed among sources. This is done thanks to the **power** function which is serves both to relate sources to eachother, and to define how much energy rays will carry.

If each source i emits a power P_i, and the total number of photons is N, then the number of rays per source N_i will be

$$N_i = N \frac{P_i}{\sum\limits_i P_i} \tag{8.4}$$

and the power of each ray will be

$$P_r = \frac{1}{N} \sum_i P_i \tag{8.5}$$

Chapter 9

Selene: GUI

The purpose of this chapter is to explain the graphical interface used to manipulate Selene. In most aspects, it is a direct translation of the scripting interface into a GUI.

As such, it is highly advised to read the previous chapter containing all the fine details about the working principles, the constants related to the renderer or objects, etc.

9.1 Main window

Selene's GUI is initiated by clicking the Selene button on the startup window. The window that opens is separated into four main parts: the display panel, the rays display options, the scene tree and the rendering options.

9.1.1 3D display

9.1.2 Rays Display

9.1.3 Scene tree

The scene tree summarizes the objects and light sources included in the scene. It is also the control that will serve to add new elements, modify or delete them. Elements that have a relative origin will appear as leaves of the related element.

Adding a new element

At the top of the scene tree sits a drop-down list that contains all the elements that can be added. Those are both objects and sources. Adding a new element first requires selecting the relevant element, and then clicking the **Add** button.

The related element dialog with all its parameters will then open, but the element will only be added after clicking the **Ok** button. The addition will be cancelled otherwise.

Modifying an element

Elements belonging to the tree can be modified by a right click on them. This brings a popup menu up with two choices

- **Properties** shows the element dialog that allows one to modify the various properties

- **Delete** will remove the element from the scene. In the case it were used as a relative frame for any other element, the latter will have said frame invalidated

9.1.4 Rendering

Once a scene is complete and properly set up (and saved to a file to avoid any lost work), rendering it is the final step. The controls related to it will be found at the bottom left of the window. It contains three parts:

- The **N Disp** control will define how many paths to display at most in the 3D view
- The **N Tot** control defines how many initial rays to cast from the sources.
- The **Trace** button will render the scene.

After the computation is over, the 3D display will show a subset of the rays that have been computed for immediate feedback, and the various sensors will have their files written to the hard drive.

9.2 The object dialog

The object dialog is split into four parts, three of them being tabs:

- the controls panel on the left is always visible. It serves to control the object name, and the various properties of the positioning system. Those are then followed by the specific object properties.
- the **Geometry** tab shows a 3D view of the object geometry, according to the object properties
- the **Interfaces** tab serves to control the properties of each face: materials, IRFs and tangent vectors
- the **Sensor** tab helps defining if an object is a sensor, what kind and what it should record

9.3 The light source dialog

The source dialog is split into two parts

- the controls panel on the left is always visible. It serves to control the source name, and the various properties of the positioning system. Those are then followed by the specific source properties.
- the **Spectrum** tab serves to define the source spectral properties. Its layout changes depending on the kind of spectrum selected

9.4 The materials dialog

Part IV
Multilayers

Chapter 10

Multilayers optical response: scripting interface

10.1 Basic setup

spectrum
    output

10.2 Structure definition

10.2.1 Environment

substrate
    superstrate

10.2.2 Layers

add_layer

10.3 Optical response computation

compute_angle

10.4 Guided modes computation

compute_guided

Chapter 11

Multilayers optical response: GUI

The GUI version of Aether contains a tool dedicated to computing the optical response of multilayer structures. This is accessed from the startup window through the **Multilayers** button.

11.1 Computation modes

The tool is split into two modes: spectral and angular. Both requires setting up a few controls. First, the spectrum is set with **Lambda min** and **Lambda max**, containing **N Points**. The angles of incidence are assumed to go from 0deg to 90deg with **N Angles** points.

11.1.1 Spectral mode

In the spectral mode, the results will computed with respect to the spectrum, and the user will browse through the angles thanks to the slider below the graph. The slider will go from 0deg to 90deg with **N Angles**.

11.1.2 Angular mode

In the angular mode, the results will computed with respect to the angle of incidence, and the user will browse through the spectrum from **Lambda min** to **Lambda max** thanks to the slider with **N Points** increments.

11.2 Multilayer structure definition

First the materials of the substrate and superstrate need to be defined. This is done through the **Superstrate** and **Substrate** material selectors. Note that

- the incident light is assumed to be coming from the superstrate
- as such, the refractive index of the superstrate can only be real
- the angle of incidence is defined inside the superstrate

Once this is done, the various layers must be added. This is done by first selecting the layer type from the drop-down list, and then clicking **Add**, within the **Layers** control.

11.2.1 Layer

11.2.2 Bragg structure

For convenience, the possibility of creating Bragg structures has been added.

11.2.3 Statistical thickness distribution

If one wants

The total number of samples is selected from the **Statistical Sampling** control.

11.3 Results

The results are shown in real-time on the on the right of the window. There are nine curves in total:

- the reflection, transmission and absorption for the S polarization (TE)
- the reflection, transmission and absorption for the P polarization (TM)
- the averages of both sets of curves

Their display can be selectively enabled or disabled thanks to the checkboxes under **Display**.

11.4 Exporting the result

The data shown on the graph can be exported through the **Export Data** button.

Chapter 12

Guided modes calculation: GUI

Part V

Various tools

Chapter 13

Scripting tools

13.1 dielec_planar_waveguide

```
guide=MODE("dielec_planar_waveguide")
guide:guide(1.9)
guide:substrate(1.0)
guide:superstrate(1.1)
guide:thickness(200e-9)
guide:lambda(500e-9)
```

13.2 fieldblock_treat

13.3 fieldmap_treat

13.4 index_fit

13.5 preview_fdtd_structure

This mode allows to load and visualize a FDTD geometry file (c.f. chapter 5) without requiring a simulation to be launched. Only the **structure**("**geometry_file**") function is available in this mode. The result is displayed in the **grid** directory.
Example:

```
test=MODE("preview_fdtd_structure")
test:file("structures/nanorods_grid.lua")
```

Chapter 14

Diffraction orders

The purpose of this tool is to help understand the number of propagative diffraction orders that can exist in the optical response of a flat diffraction grating. It's split in two modes selected thanks to the **Mode** switch.

It is accessed through the **Diffraction Orders** button in the startup window.

14.1 Monochromatic mode

This is the most graphically interesting mode. The window is split in two parts: the controls on the left, and the 2D view on the right. The controls are as follows:

- the **Lambda** selector defines the wavelength of interest
- **Substrate** and **Superstrate** define the refractive index of either side of the grating. Note that only real refractive indexes must be used
- the **Incidence** of the incoming light is set through **Theta**, which is the angle from the normal vector, and **Phi** which is the azimuth
- the grating parameters are set by defining the vectors of the elementary cell. This is done through the controls inside **Grating Vector 1** and **Grating Vector 2**, for which the **Length** can be modified, as well as their **Angle** from the $\vec{x}$ axis.

The 3D view, on the right, shows the propagation direction of the various orders: red for those propagating into the superstrate, and blue for those propagating into the substrate. The green ray is the incident ray.

Note that the behavior of the 3D view can be modified through the **Display** controls:

- **Normalized** changes wether the rays have the same length, or are scaled like wave vectors.
- **Superstrate** enables or disables the superstrate rays.
- **Substrate** enables or disables the substrate rays.

14.2 Polychromatic mode

The polychromatic mode is similar to the monochromatic one, except for a few modifications.

The first one is, obviously, that the wavelength is replaced with a spectrum selector, that asks for a range between **Lambda min** and **Lambda max**, and a number of computed points **N Points**.

The second modification is that the 3D visualization is replaced with a graph. This graph counts the total number of propagative diffraction orders that are present either in the superstrate or in the substrate. Note that for the superstrate this number can only go down to 1, for specular reflection can always exist.

The final modification is the **Surface Modes** button. This brings a dialog up that summarizes the wavelengths for which new propagative orders appear. This computation is not exact, but based on the points of the defined spectrum. The purpose of this window is to have a guide to interpret experimental spectra, so perfect accuracy is not necessary, and can always be refined by increasing the number of points in the spectrum.

Chapter 15

Graphical functions fitter

15.1 Exporting the fit

Chapter 16

Samples Explorer

The aim of the samples explorer is to help navigate a large number of samples in a use friendly way, by dynamically generating an interface based on written rules for each sample.

16.1 Working principle

Let us assume we have a set of samples in a directory as shown in figure 16.1.

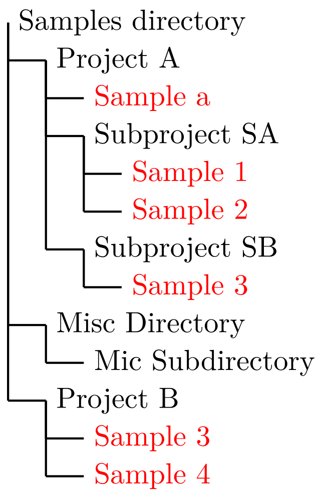


Figure 16.1: Example of a directory tree for the samples explorer module

In this, the directories containing actual samples data are highlighted in red. The aim of the SE is to create a GUI that will display the chosen samples information and data, while ignoring the directories that do not have any.

To do so, the user will chose the root directory that is to be analyzed, and the program will traverse all the subdirectories looking for samples. Those are indicated by the presence of a **desription.txt** file inside each of them. Without one, the content of a directory will be ignored, and so will be any text file that does not match that exact file name.

16.2 The description file

As stated before, each directory of interest must contain a **description.txt** file. This file contains the elementary instructions that will generate the related sample user interface. Strictly speaking, it is a Lua script and will be used to call several functions.

- **date**(**"date in any format"**) : will give a date to the sample. That date does not need to follow any particular format.
- **description**(**"description text"**) : adds a description to the sample. There is no size limit to it. New lines must be defined with a \n statement within the string.
- **tags**(**"tag 1"**,**"tag 2"**,**...**) : tags associated to the sample. They can be anything, usually project names, geometry, etc, and will be used for filtering (see 16.3.2).
- **add_image**(**"path"**) : path to the image to display, this can be called several times to display several pictures onto the interface
- **image_description**(**"description text"**) : adds a description to the last image defined with **add_image**
- **add_graph**(**"caption"**) : defines a new graph with its associated caption
- **add_data**(**"first file path"**,**column number of file 1**,**"second file path"**,**column number of file 2**) : this will add data to be displayed onto the last defined graph. The data must come from text files that contains several columns. The first two arguments will define the abscissa, while the last two define the data values. Columns counting start at 0.
- **axis**(**x1**,**x2**,**y1**,**y2**) : will set the default display axis of the last defined graph

Using those functions is not mandatory, and they can be omitted if irrelevant. Here is an example of what such a file could look like:

```
description("Sample 1")

tags("t1","t4")

add_image("im1.jpg")
image_description("First image")

add_image("im2.jpg")
image_description("Second image")

add_image("subdir/im3.jpg")

add_graph("Graph 1")
add_data("data1.txt",0,"data1.txt",1)
axis(0,10,-0.5,1.5)

add_graph("Graph 2")
```

```
add_data("data1.txt",0,"data1.txt",1)
add_data("data2.txt",0,"data2.txt",2)

add_graph("Graph 3")
add_data("data1.txt",0,"data1.txt",1)
add_data("data2.txt",0,"data2.txt",1)
add_data("data2.txt",0,"data2.txt",2)
```

16.3 Interface

Once the various samples have been put in their directories, and their description files written, using the module is fairly simple.

The very first step is to define the root directory of the samples. This is done through the **...** button within the **Directory** box. The dialog that appears will ask for the root directory.

Once set, the software browses all of the subdirectories looking for description files, and treats them. The processing can take some time depending on the number of samples. If all goes well, the interface should update in two ways:

- the list on the left of the window should now be populated by a list of all the samples and their directories.
- the panel on the right should now contain an aggregate of all the samples, each having a sample panel.

16.3.1 The sample panel

Each sample panel contains the information specified in the associated description file.

16.3.2 Tags filtering

Tags would be of little point if they could not be used to filter the results. This is the purpose of the **Filter Tags** button at the bottom left of the window. A dialog appears after clicking it, and it will help

Bibliography

[1] R. Luebbers, F.P. Hunsberger, K.S. Kunz, R.B. Standler, and M. Schneider. A frequency-dependent finite-difference time-domain formulation for dispersive materials. *Electromagnetic Compatibility, IEEE Transactions on*, 32(3):222 – 227, aug 1990.

[2] P. G. Etchegoin, E. C. Le Ru, and M. Meyer. An analytic model for the optical properties of gold. *The Journal of Chemical Physics*, 125(16):164705, 2006.

[3] Alexandre Vial. Implementation of the critical points model in the recursive convolution method for modelling dispersive media with the finite-difference time domain method. *Journal of Optics A: Pure and Applied Optics*, 9(7):745, 2007.

[4] A. Aminian and Y. Rahmat-Samii. Spectral fdtd: a novel technique for the analysis of oblique incident plane wave on periodic structures. *Antennas and Propagation, IEEE Transactions on*, 54(6):1818 – 1825, june 2006.