CAMFR manual

version 1.3
September 2007

Peter Bienstman

Table of Contents

Introduction

CAMFR (CAvity Modelling FRamework) is a fast, flexible, full-vectorial Maxwell solver, that is powerful and easy to use. Although it can tackle general electromagnetic problems, its main focus is the simulation of optical devices, e.g. wavelength-scale structures (like photonic crystals), lasers (like VCSELs) and light-emitting diodes (like RCLEDs).

As CAMFR is an ongoing active research project, it contains many attractive new algorithms and techniques which are currently not yet found in commercial software, most notably in the domain of advanced boundary conditions.

Contrary to other, more traditional approaches like FDTD, CAMFR is not based on spatial discretisation or finite differences, but rather on frequency-domain eigenmode expansion techniques. Instead of specifying the fields on a discrete set of grid points in space, the fields are described as a sum of local eigenmodes in each z-invariant layer of the structure. What this means concretely is that for a large variety of structures, CAMFR can be orders of magnitude faster than these traditional methods.

CAMFR can calculate:

- the scattering matrix of a structure
- the field inside a structure, for any given excitation
- band diagrams of an infinite periodic structure
- threshold material gain and resonance wavelength of laser modes
- the response to a current source in an arbitrary cavity
- structures containing a semi-infinite repetition of a basic period

This functionality is currently available for two types of geometries:

- 2D Cartesian structures
- 3D cylindrical symmetric structures

The rest of this manual is structured as follows:

- a tutorial introducing the basic concepts of CAMFR
- a chapter containing examples which illustrate other features, like modelling more complicated optical structures (VCSEls, spontaneous emission, photonic crystals,...)
- a reference guide

1 Tutorial

To run a simulation with CAMFR, you have to write a Python script describing the structure to be simulated and the calculations to be performed.

Python (`http://www.python.org`) is a general-purpose, clean and flexible scripting language, that is easily extendible and that really shines at integrating different pieces of software. Because Python is a full-blown programming language, it is e.g. very easy to write loops that sweep a certain simulation parameter. It is also trivial to use the CAMFR output in other Python-enabled applications, like e.g. a visualisation package.

In this tutorial, we will introduce the basic concepts of CAMFR, as well as a minimal subset of the Python language that is needed to perform simulations. Although a more in-depth knowledge of Python is not really necessary, it is reassuring to know that CAMFR is integrated in a powerful programming language, making it possible to perform sophisticated simulations and postprocessing.

1.1 Example 1: a simple waveguide example

The examples in this chapter can be found in the directory `examples/tutorial` in the main CAMFR directory. If you're on a Windows system, this directory could by something like `C:\python23\Lib\site-packages\camfr`, on a Unix system it is usually `/usr/lib/python2.5/site-packages/camfr`.

The first file `tutorial1.py` introduces some simple waveguide simulations and looks like this:

```
#!/usr/bin/env python

####################################################################
#
# Simple waveguide example
#
####################################################################

from camfr import *

set_lambda(1)
set_N(20)
set_polarisation(TE)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define waveguide.

slab = Slab(air(2) + GaAs(0.5) + air(2))

slab.calc()

# Print out some waveguide characteristics.

print slab.mode(0).kz()
print slab.mode(1).n_eff()
print slab.mode(2).field(Coord(2.25, 0, 0))
print slab.mode(3).field(Coord(2.25, 0, 0)).E2()
print slab.mode(4).field(Coord(2.25, 0, 0)).E2().real


# Do some interactive plotting.

slab.plot()
```

The first line `#!/usr/bin/env python` is an optional Unix incantation to tell the operating system that it should execute this script with the Python interpreter.

The next lines illustrate that comments in Python start with `#`.

The line `from camfr import *` informs Python that we want to start using the CAMFR library.

The following lines are pretty self-explanatory:

```
set_lambda(1)
set_N(20)
set_polarisation(TE)
```

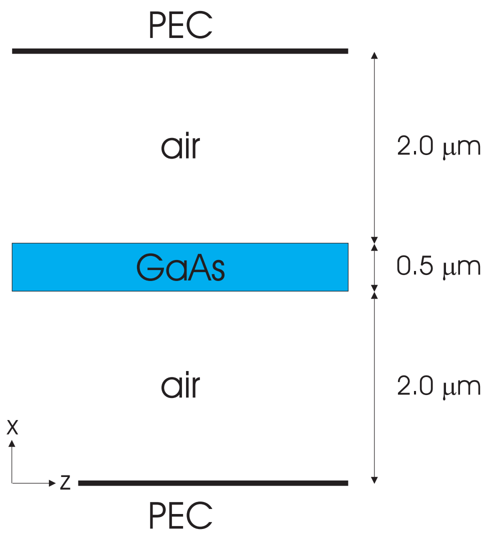
We set the wavelength to 1 micron, use 20 eigenmodes to expand the field, and deal with TE polarisation. Next, we define GaAs to be a material with refractive index 3.5 and air as a material with index 1:

```
GaAs = Material(3.5)
air  = Material(1.0)
```

We can now define a simple slab waveguide, with a GaAs core of half a micron thick, and air cladding layers of 2 micron thick.

```
slab = Slab(air(2) + GaAs(0.5) + air(2))
```

This structure is implicitly assumed to be sandwiched between two perfect electric conductors (PECs), as indicated in fig. 1:



This figure also indicates the conventions for the coordinate system. The `x`-axis lies along the cross-section of the slab waveguide and starts at the bottom wall. The waveguide is uniform in the `y`- and `z`- direction. The eigenmodes of the waveguide propagates along the `z`-direction.

Using the command `slab.calc()`, CAMFR will calculate the properties of the slab, some of which are then printed out in the following lines of the Python script.

`print slab.mode(0).kz()` prints out the propagation factor of the fundamental mode of the slab, while `slab.mode(1).n_eff()` displays the effective index of the first order mode.

`print slab.mode(2).field(Coord(2.25, 0, 0))` shows the field of the second order mode at `x=2.25`, which is in the center of the core. This field consists of six complex numbers. `E1`, `E2`, `Ez` are the phasors for the components of the electric field, and `H1`, `H2`, `Hz` represent the magnetic field. Here, 1 refers to the `x`-direction and 2 to the `y`-direction. (For cylindrical structures, 1 refers to the radial direction and 2 to the angular direction.)

`print slab.mode(4).field(Coord(2.25, 0, 0)).E2().real` illustrates how we can extract the real part from a complex number in Python. (`real` is a built-in Python attribute of a complex number, and as such requires no extra parentheses.) Similarly, the imaginary component can be extracted with `imag`.

Finally, `slab.plot()` shows a widget which can be used to interactively explore the properties of the waveguide modes, field profiles and effective index distribution. Note that you can zoom in these plots by drawing a rectangle with the left mouse button pressed. You can also bring up the effective index distribution in the complex plane and click on the modes in that plane to plot their field profiles.

The Python script can be executed in a number of ways. You can type `python tutorial1.py`, or under Unix you can just suffice by typing `tutorial1.py`, provided the first line of your script contains `#!/usr/bin/env python` and the script file's executable bit is set (with `chmod +x tutorial1.py`). (Note: all of these command need to be typed in on the system prompt, not on the Python prompt.)

Regardless of the invocation method, the output that will be printed out will look something like this:

```
CAMFR 1.3 - Copyright (C) 1998-2007 Peter Bienstman - Ghent University.

(21.3499038707+0j)
(3.07668661368+0j)
E1=(0,0), E2=(-16.0633,0), Ez=(0,0)
H1=(0.105632,0), H2=(0,0), Hz=(0,0.0966439)
(24.2902148016+0j)
0.649002451002
```

There is also a third way of starting the script, which is `python -i tutorial1.py`. This will print out the same output, but will afterwards present you with an interactive Python session. After Python's `>>>` prompt, you can type any Python command, like e.g. `print slab.mode(10).kz()`. This allows you to rapidly inspect any simulation results, without the need to adapt your script file. This Matlab-like interactivity can be very productive, because it can give you rapid feedback on your simulation results.

If you are on a Windows machine and use the Active Python distribution from `http://www.activestate.com` as your Python environment, you will have access to an interactive Python session semi-automatically through the PythonWin environment. Consult the Active Python documentation for more information.

1.2 Example 2: a simple stack example

The next example shows how to calculate the scattering matrix of a stack of waveguides and how to loop over a simulation parameter:

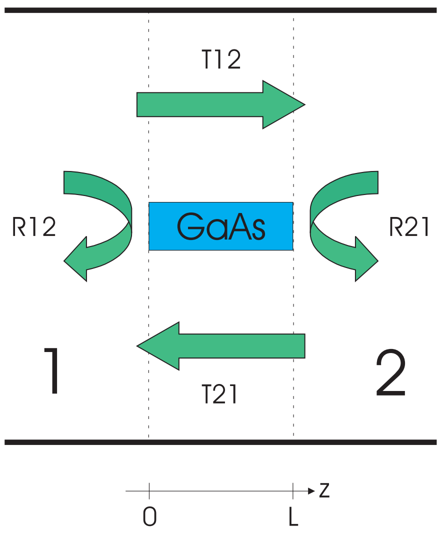
```
#!/usr/bin/env python

####################################################################
#
# Simple stack example
#
####################################################################

from camfr import *

set_lambda(1)
set_N(20)
set_polarisation(TE)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define slabs.

slab  = Slab(air(2) + GaAs(0.5) + air(2))
space = Slab(air(4.5))

# Print the reflectivity for different lengths.

for L in arange(0.005, 0.100, 0.005):
    stack = Stack(space(0) + slab(L) + space(0))
    stack.calc()
    print L, abs(stack.R12(0,0))
```

Apart from the same slab waveguide as in the previous example, we also define a second slab called `space`, a uniform air layer.

Let us skip ahead to this line:

```
    stack = Stack(space(0) + slab(L) + space(0))
```

This line defines a stack consisting of a sequence of waveguide sections, as shown in fig. 2:



Because of the way eigenmode expansion works, the incidence and exit waveguides (`space` in this case) are infinitely long. The thickness of the sections `space(0)` just serve to indicate the location of the reference input and output planes.

The command `stack.calc()` is used to calculate the scattering matrix of the stack named `stack`. Fig. 2 indicates the meaning of the four submatrices of this scattering matrix. There are two reflection and transmission matrices, for incidence from either medium 1 or medium 2.

More specifically, if you have e.g. an incident field from the left in medium 1 , described by a column vector `f` of expansion coefficients, the reflected field will be described by the product `R12 x f`.

So, `stack.R12(0,0)` is the reflection coefficient of the fundamental mode of the incident waveguide back to itself.

Also illustrated in this example is how to create loops in Python, e.g. to vary the length in central waveguide section from 5 to 100 nm (excluding 100 nm) in steps of 5 nm:

```
for L in arange(0.005, 0.100, 0.005):
    stack = Stack(space(0) + slab(L) + space(0))
    stack.calc()
    print L, abs(stack.R12(0,0))
```

The function `arange` is not part of the core Python language, but rather of the extension package NumPy.

Very important to notice is that Python uses indentation to distinguish statements which form part of the loop and which don't. So, writing

```
for L in arange(0.005, 0.100, 0.005):
    stack = Stack(space(0) + slab(L) + space(0))
    stack.calc()
print L, abs(stack.R12(0,0))
```

would only print out the results for the last value of `L`, which incidentally is 0.095. It does not really matter if you use tabs or spaces to indent, as long as you are consistent.

It is of course trivial to create nested loops:

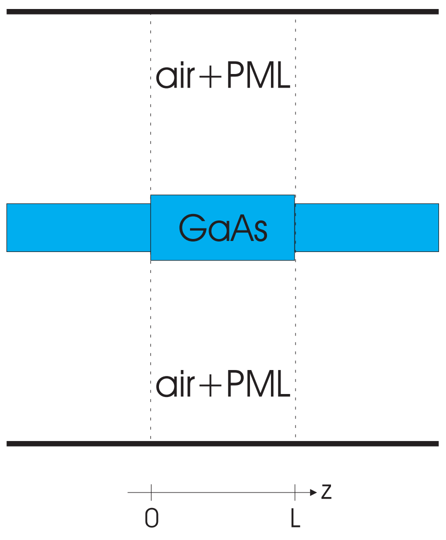
```
for x in arange(0.000, 0.100, 0.010):
    for y in arange(0.000, 0.200, 0.020):
        do_something()
```

1.3 Example 3: another stack example

In the next example, we will introduce PML boundary conditions and show how to work with files.

```
#!/usr/bin/env python

####################################################################
#
# Another stack example
#
####################################################################

from camfr import *

# Set constants.

set_lambda(1.5)
set_N(20)
set_polarisation(TE)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define waveguide sections.

set_lower_PML(-0.1)
set_upper_PML(-0.1)

normal = Slab(air(2.0) + GaAs(0.5) + air(2.0))
thick  = Slab(air(1.9) + GaAs(0.7) + air(1.9))

# Calculate reflection of the fundamental mode for different
# lengths of the central thick section.

outfile = file("tutorial3.out",'w')

for L in arange(0.000, 0.500, 0.010):
    stack = Stack(normal(0) + thick(L) + normal(0))
    stack.calc()
    print >> outfile, L, abs(stack.R12(0,0))

outfile.close()
```

This script is very similar in structure to the previous one, and describes a waveguide with a core thickness of 0.5 micron, which widens to 0.7 um in a central section with length `L` (fig. 3).



`set_lower_PML(-0.1)` gives all the lower claddings in subsequently defined slabs an imaginary component to its thickness of `-0.1j`. Similarly for `set_upper_PML`. `lower` refers to the layer at `x=0`, which is the first term in the expression used to define the slab.

This imaginary cladding thicknesses implement the so-called perfectly matched layer (PML) boundary conditions. PML can absorb radiation travelling toward the walls, without introducing any additional parasitic reflections, regardless of wavelength, incidence angle or polarisation of the incident light. The larger the imaginary component (in absolute value) of the thickness, the stronger the absorption.

The presence of advanced boundary conditions like PML is a very powerful feature of CAMFR. Without it, the PEC walls would reflect all the incoming radiation and send it back the structure under study, where it can disturb the simulation results.

The current example also illustrates how to write output to a file, rather than to the screen:

```
outfile = file("myfile",'w') # 'w' indicates writing.
print >> outfile, "Hello world"
outfile.close()
```

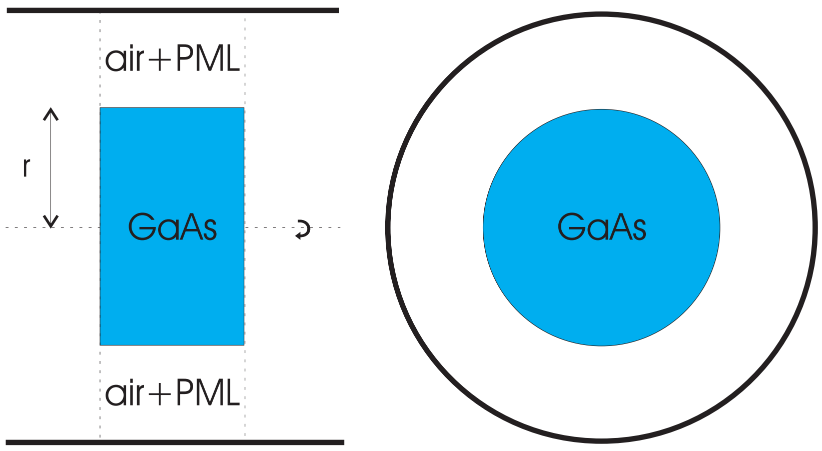
1.4 Example 4: a circular stack

Not only can CAMFR deal with 2D Cartesian structures, it can also handle circular waveguides with an arbitrary number of index steps in the radial direction.

This is illustrated in the following Python file:

```
#!/usr/bin/env python

#####################################################################
#
# A circular stack
#
#####################################################################

from camfr import *

set_lambda(1)
set_N(20)
set_circ_order(0)
set_polarisation(TE)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define uniform circular waveguide.

set_circ_PML(-0.1)
space = Circ(air(1))

# Calculate the reflectivity for different widths
# of the central core.

for r in arange(0.100, 0.500, 0.050):
    circ = Circ(GaAs(r) + air(1 - r))
    stack = Stack(space(0) + circ(0.5) + space(0))
    stack.calc()
    print r, abs(stack.R12(0,0))
    free_tmps()
```

The structure defined by this script is shown in fig. 4.



A new command specifically related to circular structures is `set_circ_order(0)`, which tells CAMFR to look for eigenmodes with Bessel order 0. For this Bessel order, we can have both TE and TM modes, and in this case we choose for TE modes by setting `set_polarisation(TE)`. For Bessel orders other than 0, the modes are hybrid TE/TM, and any use of `set_polarisation` will be ignored.

Circular waveguides are defined in pretty much the same way as slab waveguides, except that the argument to Circ describes the structure from `r=0` to the wall, rather than from wall to wall as was the case with slabs. PML in circular structures is set by using `set_circ_PML`, which adds a complex thickness to the cladding layer.

Another new command that is introduced in this example is `free_temps()`, which is called at the end of each loop iteration to free all the data structures (eigenmodes, scattering matrices, ...) that were allocated so far. The default mode of operation in CAMFR is to keep this data around, so that it might be reused in future calculations. However, in this particular example, this would only waste a lot of memory, since every iteration through the loop creates a completely new structure with a different core width, whose results cannot reused in subsequent calculations.

This is different from `example2.py`, where we did not create new waveguides, but only varied the length of the waveguide sections. In this case, it made sense to keep the previous calculation results around, because eigenmode expansion can easily update the scattering matrices of a structure if only the length of the individual waveguide sections changes.

For a more complex example dealing with Omniguide fibres, see `omniguide.py` in the `examples/contrib` directory.

1.5 Example 5: field profiles in a stack

In this example, we will excite the stack from example 2 with a certain incident field distribution and calculate the resulting field in the stack.

```
#!/usr/bin/env python

####################################################################
#
# Calculate field profiles in a stack.
#
####################################################################

from camfr import *

set_lambda(1)
set_N(20)
set_polarisation(TE)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define stack.

set_lower_PML(-0.1)
set_upper_PML(-0.1)

slab  = Slab(air(2) + GaAs(.5) + air(2))
space = Slab(air(4.5))

stack = Stack(space(0) + slab(0.5) + space(0))

# Set incident field and calculate stack.

inc = zeros(N())
inc[0] = 1
stack.set_inc_field(inc)

stack.calc()


# Save the field to a file.
# Do some interactive plotting.

stack.plot()
```

The new code is in the following lines:

```
inc = zeros(N())
inc[0] = 1
stack.set_inc_field(inc)
```

These commands prepare a column vector named `inc` describing the incident field. This vector consists of `N` elements, one for each eigenmode. Initially, all elements are zero, but afterwards, we set the element corresponding to the fundamental mode equal to one. This vector is then used as the incident field.

`stack.plot()` brings up a widget which can show and animate the field profile in a stack, write the pictures to files, ... Once again, you can zoom in these widgets by drawing a rectangle with the left mouse button pressed.

If you want to do some post-processing of the field data yourself, you can access it by commands like `stack.field(Coord(x,0,z))`.

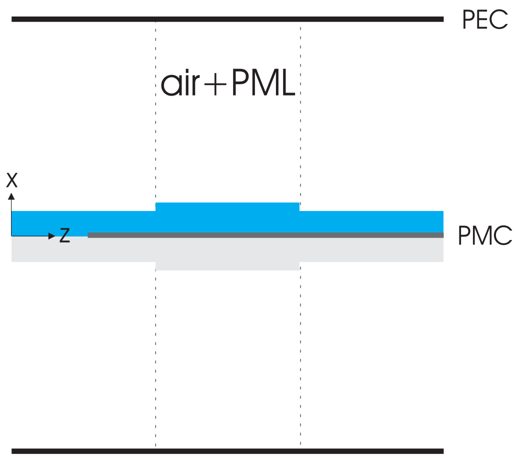
1.6 Example 6: exploiting symmetry

In this example, we will simulate the same structure as in the previous example, but this time we will exploit the symmetry in order to speed up the calculation.

```
#!/usr/bin/env python

####################################################################
#
# Calculate field profiles in a stack, but exploit symmetry
#
####################################################################

from camfr import *

set_lambda(1)
set_N(10)
set_polarisation(TE)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define stack.

set_upper_PML(-0.1)

set_lower_wall(slab_H_wall)

slab  = Slab(GaAs(.25) + air(2))
space = Slab(air(2.25))

stack = Stack(space(0) + slab(0.5) + space(0))

# Set incident field and calculate stack.

inc = zeros(N())
inc[0] = 1
stack.set_inc_field(inc)
```

```
# Save the field to a file.

outfile = file("tutorial6.out",'w')

for x in arange(0.000, 2.250, 0.100):
    for z in arange(0.000, 0.500, 0.010):
        print >> outfile, abs(stack.field(Coord(x,0,z)).E2()),
    print >> outfile

outfile.close()
```

The structure we are studying is symmetric along the horizontal axis. If we want to excite it with a symmetric excitation as well, we can speed up the simulation by introducing a symmetry wall and only calculating half of the structure (fig. 5).



For excitation with the fundamental TE mode, we need to introduce a magnetic wall at the center. This is done with the command `set_lower_wall(slab_H_wall)` before defining any slabs. `lower` refers to the bottom side of the slabs (`x=0`). Obviously, there exists a command `set_upper_wall(...)` as well. It is also possible to specify a `slab_E_wall` as boundary condition, but this not necessary as it is the default.

Because the structure now has half the size as the original structure, we can suffice with only half as many modes, which typically buys us a factor eight in computation time.

Another thing to pay attention to is that in this case there is only PML at the upper wall, since the lower wall is effectively the inside of the structure we want to model, so we don't want any absorbing boundary conditions there.

This time, instead of interactively plotting the field, we choose to write it to a file for further analysis. When printing the field, note the trailing command at the end of the `print` statement. This will prevent the `print` statement from going to the next line. After we have iterated over all `z`-values for a given `x`-value, we force a line break with the command `print >> outfile`. In this way, the field data is nicely arranged in a matrix format.

1.7 Example 7: using functions to define complex structures

In this final example we will illustrate how we can simplify the definition of non-trivial structures by using Python functions, rather than explicitly specifying the refractive index profile.

The following code shows how we automatically generate a staircase approximation of a parabolic refractive index profile.

```
#!/usr/bin/env python

####################################################################
#
# Parabolic refractive index profile.
#
####################################################################

from camfr import *

set_lambda(1)
set_N(20)
set_polarisation(TE)

w = 5.0  # width of waveguide

set_upper_PML(-0.1)
set_lower_PML(-0.1)

# Define parabolic refractive index profile.

def index(x):
    max_n = 3.5  # max refractive index
    a = 0.1      # slope
    n = max_n - a * pow(w / 2.0 - x, 2)
    if n < 1:
        return 1
    else:
        return n
# Construct a staircase approximation.

expr = Expression()
materials = []
steps = 10
for i in range(steps):
    x = i * w / steps
    m = Material(index(x + 0.5 * w / steps))
    materials.append(m)
    d = w / steps
    expr.add(m(d))

slab = Slab(expr)

# Compare continuous and staircase profile.

outfile = file("tutorial7.out",'w')

steps2 = 100
for i in range(steps2):
    x = i * w / steps2
    print >> outfile, x, index(x), slab.n(Coord(x, 0, 0)).real

outfile.close()
```

The lines after `def index(x):` define a function which takes as argument the `x` coordinate and returns the refractive index at this position in a certain parabolic refractive index profile. This code also illustrates the use of conditionals, and stresses once again the importance of indentation in Python.

The next block of code creates the expression `expr`, which will describe the staircase approximation of the index profile. The width of the waveguide will be divided in `steeps` piecewise constant parts.

First, an empty expression is created: `expr = Expression()`. This expression is gradually filled in a loop while `i` runs from 0 to `steps-1`.

A material `m` is created with the refractive index from the middle of the current step of the index profile. Finally, a term is added to the expression which describes the current step, consisting of material `m` with thickness `d`: `expr.add(m(d))`.

At the end of the loop, this expression is used to define a slab, which can be used in subsequent calculations : `slab = Slab(expr)`.

Very important to notice is that we keep hold of the materials `m` we create in the loop, by first defining an empty list `materials = []`, to which we add each material: `materials.append(m)`.

The reason for this lies in Python's garbage collection. Without the list `materials` to hold on to the materials, Python will see `m` as just a temporary variable, which it will deallocate and garbage collect after the loop has finished. Since we still need these materials in the subsequent calculations, we have to tell Python not to free them by storing them in a list at the top level in the source code.

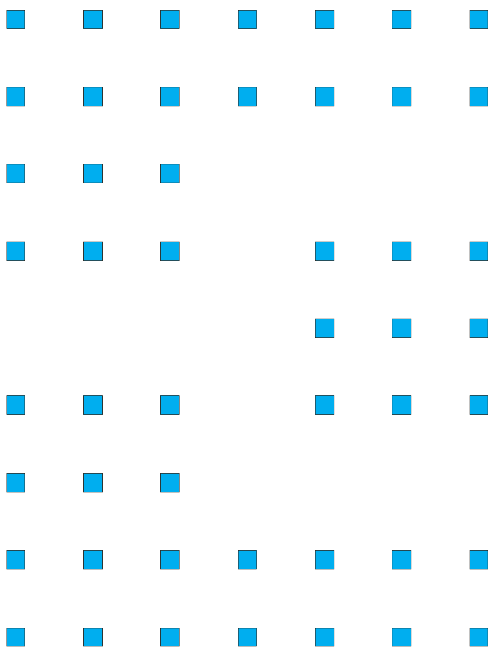
2 Modelling optical devices

In this chapter, we will build upon the basic concepts introduced in the tutorial chapter to model more complicated optical devices. More specifically, we will see how to calculate band diagrams for infinite periodic stacks, how to locate laser modes in arbitrary cavities and how to calculate the response to a current source inside a cavity. We will also explain how to model purely 1D planar structures, how to model semi-infinite stacks, how to input more complicated structures using geometric primitives, and how to use more complex excitations for slabs.

All of this functionality is available for both geometries (2D Cartesian, cylindrical), although the examples given are only for a single geometry.

2.1 Modelling photonic crystal devices

Let us model the photonic crystal splitter from fig 2.1:



This device consists of a 2D periodic arrangement of GaAs rods in air, which is impenetrable for certain wavelengths. By omitting some rods, we can create channels which guide the light. In the example of the splitter, light enters the structure from the left, is guided by the air channel in the photonic crystal, and finally split into two equal parts.

Although we can already analyse such a structure with the knowledge we have from the tutorial, there are a number of techniques that can be applied to speed up the simulation. This is illustrated in the following code, which can be found in `examples/other`:

```
#! /usr/bin/env python

###################################################################
#
# 90 deg 3dB splitter in rectangular lattice of rectangular
# GaAs rods in air
#
###################################################################

from camfr import *

set_polarisation(TE)
set_lambda(1.5)
set_N(50)

# Set geometry parameters

GaAs = Material(3.4)
air  = Material(1.0)

a = .600     # period
r = .150/2.0 # rod radius

set_lower_wall(slab_H_wall)

cl = 0 # air cladding

periods = 3  # periods above outer waveguide
sections = 1 # intermediate 90 deg sections

# Define slabs.

no_rods = Slab(air(a-r+(sections+1+periods)*a+cl))

# Central waveguide.

cen = Slab(  air(a-r)                                          \
           + (sections+1+periods)*(GaAs(2*r) + air(a-2*r))     \
           + air(cl) )
```

```
# Vertical section.

ver = Slab(  air(a-r + (sections+1)*a)                              \
           + periods*(GaAs(2*r) + air(a-2*r) )                      \
           + air(cl) )

# Outer arms.

arm = Slab(  GaAs(r) + air(a-2*r)                                   \
           + sections*(GaAs(2*r) + air(a-2*r))                      \
           + air(a)                                                 \
           + periods*(GaAs(2*r) + air(a-2*r))                       \
           + air(cl) )

# Find lowest order waveguide mode.

wg = BlochStack(cen(2*r) + no_rods(a-2*r))
wg.calc()

print wg

guided = 0
for i in range(2*N()):
    if (abs(wg.mode(i).kz().imag) < abs(wg.mode(guided).kz().imag)):
        if wg.mode(i).kz().real > 0:
            guided = i

# Calculate splitter.

splitter = Stack(  5*(cen(2*r) + no_rods(a-2*r))                   \
                 +    ver(2*r) + no_rods(a-2*r)                     \
                 + 5*(arm(2*r) + no_rods(a-2*r)) )

splitter.set_inc_field(wg.mode(guided).fw_field())
splitter.calc()

print "R", splitter.R12(0,0)
```

```
# Calculate field.

outfile = open("splitter.out", 'w')

for x in arange(0.000, no_rods.width() - cl - a, a/20.):
    for z in arange(0.000, splitter.length(), a/20.):
        print >> outfile, abs(splitter.field(Coord(x, 0, z)).E2()),
    print >> outfile

outfile.close()
```

Note in this example the use of \ to split up long lines. This is necessary because of the way Python relies on whitespace.

An important note is that CAMFR always uses the polarisation definitions as they are conventional in waveguide theory, i.e. TE polarisation corresponds here to the electric field along the rods, out of the plane of the picture. Unfortunately in photonic crystal literature this is called TM polarisation.

Now, we need to find a suitable excitation in order to calculate the field profiles in this structure. In other methods like FDTD, we would do this by placing a current source in a very long stretch of the photonic crystal waveguide in the left of fig. 2.1. After this long stretch of waveguide, an equilibrium field distribution with only the fundamental mode of the waveguide would appear, which can be used to excite the splitter.

This is rather slow, since we need a long stretch of waveguide to obtain this equilibrium field distribution. However, because of the frequency domain nature of CAMFR, we can directly excite the structure with a quasi-equilibrium field distribution. In order to do that, we first perform a sub-calculation which gives us the Bloch modes of the photonic crystal waveguide without the splitter:

```
wg = BlochStack(cen(2*r) + no_rods(a-2*r))
wg.calc()
```

Here, we define `wg` to be an infinite repetition of the sections `cen` and `no_rods`. After a call to `calc`, we can examine the properties of this waveguide and its Bloch modes just like any other waveguide mode (with calls to `kz()`, `field()`, ...). The only difference is that a BlochStack has `2N` modes rather than `N`, because it contains both forward and backward Bloch modes.

(Note that if we wanted to calculate the full band diagram of this periodic structure in the direction of `z`, we simply have to place this calculation inside a loop over all frequencies of interest, and look at the BlochModes where the imaginary part of the propagation constant is sufficiently small.)

After we inspect the propagation factors to determine to locate the fundamental mode, we excite the splitter with the forward field of this mode:

```
splitter.set_inc_field(wg.mode(guided).fw_field())
```

This gives us a quasi-equilibrium field distribution almost immediately, and we can very efficiently calculate the properties of the splitter using only a small computational domain.

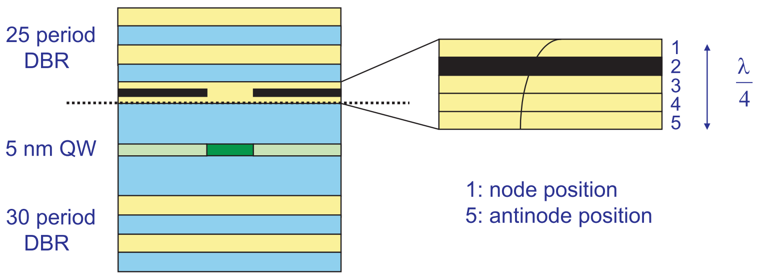
2.2 Locating laser modes

The following code illustrates how to use CAMFR to find laser modes in a cavity, in this case a cylindrical oxidised VCSEL (vertical-cavity surface-emitting laser):

```
#!/usr/bin/env python

####################################################################
#
# Finds a laser mode in a VCSEL (from COST 268 modelling exercise.)
#
####################################################################

from camfr import *

set_lambda(.980)
set_N(100)
set_circ_order(1)

# Define materials.

GaAs_m   = Material(3.53)
AlGaAs_m = Material(3.08)
AlAs_m   = Material(2.95)
AlOx_m   = Material(1.60)
air_m    = Material(1.00)

gain_m = Material(3.53)
loss_m = Material(3.53 - 0.01j)

set_gain_material(gain_m)

# Define geometry parameters

set_circ_PML(-0.1)

r = 4.0
d_cladding = 4.0

d_GaAs   = .06949
d_AlGaAs = .07963
```

```
# Define cross sections

GaAs   = Circ(  GaAs_m(r+d_cladding))
AlGaAs = Circ(AlGaAs_m(r+d_cladding))
air    = Circ(   air_m(r+d_cladding))

ox = Circ(AlAs_m(r) + AlOx_m(d_cladding))
QW = Circ(gain_m(r) + loss_m(d_cladding))

# Set oxide window position.

position = 4 # 1: node, 5: antinode
x = (5. - position) * d_AlGaAs/5.

# Define top half of cavity.

top = Stack( (GaAs(0) + AlGaAs(x)) + ox(.2*d_AlGaAs)           \
    + (AlGaAs(.8*d_AlGaAs - x) + GaAs(d_GaAs)                   \
    + 24*(AlGaAs(d_AlGaAs) + GaAs(d_GaAs)) + air(0)) )

# Define bottom half of cavity.

bottom = Stack(GaAs(.13659) + QW(.00500)                        \
    + (GaAs(.13659) + 30*(AlGaAs(d_AlGaAs) + GaAs(d_GaAs) + GaAs(0))) )

# Define cavity and find laser mode.

cavity = Cavity(bottom, top)

cavity.find_mode(.980, .981)
```

First, we need to divide the cavity in an arbitrary location in a top and a bottom part (the dashed line in fig. 2.2):



We then define two stacks describing these top and bottom part, starting from the dashed line. Defining the cavity is as simple as writing

```
cavity = Cavity(bottom, top)
```

In order to locate a laser mode, we need to vary the wavelength to achieve phase resonance and to vary the material gain in the active region to achieve amplitude resonance. For that, we first had to inform CAMFR which material would provide the gain:

```
    set_gain_material(gain_m)
```

From the definition of the waveguide `QW`, we can see that this device provides gain in the central part of the active region, while there is a small constant loss outside of the oxide aperture.

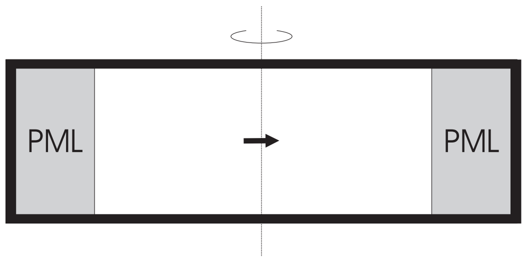
Finally, the command `cavity.find_mode(.980, .981)` will locate a laser mode in the wavelength range between 980 and 981 nm. Once it has found such a mode, it will print out its wavelength and threshold material gain. There are a few other arguments that can be passed to `find_mode()` to control the search process, but for that we refer to the reference guide.

A trick worth pointing out is the use of parentheses to group all the transverse uniform waveguide sections in the definitions of `top` and `bottom`. These substacks have diagonal reflection and transmission matrices and can therefore be calculated more efficiently. By explicitly grouping these diagonal substacks, CAMFR will recognise them as such, rather than treating the whole stack as completely non-diagonal.

2.3 Putting a current source inside a cavity

This section illustrates how we can put a current source inside a cavity. This can be useful either to calculate the Green's function of a cavity, or the calculate the modification of spontaneous emission in resonant-cavity LEDs.

As a first example, we calculate the modification of spontaneous emission of a horizontal current dipole placed between two parallel metallic plates. This 3D planar open geometry is converted to a 3D closed cylindrical geometry by using PMLs (fig 2.3). Since this configuration can also be calculated analytically, we can easily verify the results.



```
#! /usr/bin/env python

####################################################################
#
# Calculates modification of spontaneous emission of a dipole
# between two metal plates.
#
####################################################################

from camfr import *

set_N(60)
set_lambda(1)
set_circ_order(1)
set_circ_field_type(cos_type)

# Define waveguide and wall.

set_circ_PML(-0.5)

air_m = Material(1.0)
air = Circ(air_m(10))
air.calc()

wall = E_Wall(air)
```

```
    # Calculate change in spontaneous emission rate.

    for d in arange(0.01, 3.0, 0.05):

        # Define cavities.

        half      = Stack(air(d/2.) + wall)
        half_open = Stack(air(d/2.))

        source_pos  = Coord(0,0,0)
        orientation = Coord(1,0,0)

        cav = Cavity(half, half)
        cav.set_source(source_pos, orientation)

        cav_open = Cavity(half_open, half_open)
        cav_open.set_source(source_pos, orientation)

        # Analytic formula for spontaneous emission rate.

        x = floor(d+0.5)
        exact = 3.*x/4./d + pow(x/d,3)/4. - x/16./d/d/d

        # Numerical formula as ratio of total emitted powers.

        numeric =   half.     field(Coord(0,0,0)).E1().real \
                  / half_open.field(Coord(0,0,0)).E1().real

        print d, exact, numeric
```

This script introduces a couple of new features. First of all `set_circ_field_type()`, which is only relevant for these kind of source problems, indicates whether the generated fields have `cos` or `sin` type angular dependence. For calculating scattering matrices, these are completely decoupled, but for source problems, which one of these field types is excited is determined by the orientation of the current source.

`wall = E_Wall(air)` defines a perfectly conducting metal plate with air on the incidence side. This structure can be used in an expression to describe the horizontal metal walls surrounding the dipole.

In order to place a source inside a cavity at a given position and with a given vectorial orientation, we use the following command:

```
cav.set_source(source_pos, orientation)
```

From Poynting's theorem, we can calculate the total power radiated by this dipole from the real part of the electric field at the dipole itself.

This is much more accurate than explicitly calculating power flows using e.g. the commands `stack.inc_S_flux` and `stack.ext_S_flux` to calculate the real power flow at the incidence and at the exit side of a stack (see reference manual).

2.4 1D planar structures

The following Python code shows how to model a purely 1D Bragg stack, where all the layers extend infinitely in the transverse direction.

```
#!/usr/bin/env python

####################################################################
#
# planar 1D DBR
#
####################################################################

from camfr import *

# Define structure.

set_lambda(1)

GaAs_m = Material(3.5)
AlAs_m = Material(2.9)

GaAs = Planar(GaAs_m)
AlAs = Planar(AlAs_m)

d_GaAs = get_lambda()/4./GaAs_m.n().real
d_AlAs = get_lambda()/4./AlAs_m.n().real

s = Stack(GaAs(0) + 10*(GaAs(d_GaAs) + AlAs(d_AlAs)) + GaAs(0))

# Loop over incidence angles.

for theta in arange(0, 90, 0.5):

  GaAs.set_theta(theta * pi / 180.)
  print theta,

  set_polarisation(TE)
  s.calc()
  print abs(s.R12(0,0))**2 ,

  set_polarisation(TM)
  s.calc()
  print abs(s.R12(0,0))**2
```

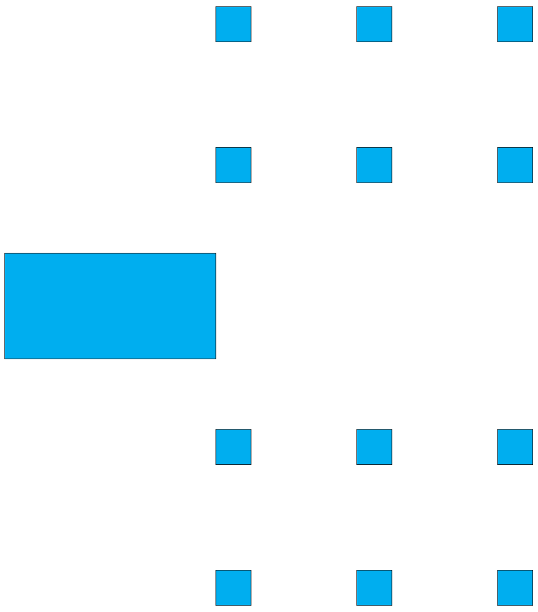
`Planar` takes a material as argument to create an infinitely extended planar layer, which can be used in `Stack` to create a layered 1D structure.

The stack is calculated for one angle at a time, which can be set in radians using `set_theta` on a given material. Note that you should set the wavelength before setting the angle, as `set_theta` effectively sets the transverse part of the wavevector, which is dependent on the wavelength.

Because only one mode (or plane wave in this case) is needed at any time for a calculation, the reflection and transmission matrices have only one element.

2.5 Semi-infinite stacks

Say you wanted to model the incoupling from a dielectric waveguide to a photonic crystal waveguide (fig. 2.4).



Using other modelling tools, you would have to study a finite structure, i.e. at some point you would have to terminate the photonic crystal and introduce a photonic crystal/air interface. Reflections from this interface could travel back to the dielectric waveguide and disturb the simulation results due to the Fabry-Perot effect.

In CAMFR we can solve this problem by using an `InfStack`, which allows us to model the structure of fig. 2.4 where the crystal extends all the way towards infinity in the propagation direction. In this way, we can study the incoupling problem in isolation of edge effects from the other side of the crystal.

The following code illustrates how to achieve this:

```
#! /usr/bin/env python

####################################################################
#
# A semi-infinite photonic crystal.
#
####################################################################

from camfr import *

set_lambda(1.5)
set_N(50)

# Set geometry parameters

GaAs = Material(3.4)
air  = Material(1.0)

a = .600     # period
r = .150/2.0 # rod radius

set_lower_wall(slab_H_wall)

cl = 0       # air cladding
periods = 4  # lateral periods

# Define slabs.

inc_wg = Slab(GaAs(1.5*r) + air(a-2.5*r+periods*a+cl))

no_rods = Slab(air(a-r+periods*a+cl))

cen = Slab(  air(a-r)                                          \
           + periods*(GaAs(2*r) + air(a-2*r))                  \
           + air(cl) )

# Calculate semi-infinite stack.

s_inf = InfStack(cen(2*r) + no_rods(a-2*r))
s = Stack(inc_wg(a) + s_inf)

s.calc()

print abs(s.R12(0,0))**2
```

If you are only interested in the reflection coefficient of the fundamental mode of the incoupling waveguide, this works fine. However, when using `InfStack`, the transmission to the Bloch modes of the semi-infinite stack is not calculated to save time.

If you are interested in these quantanties, you have to make some small modifications:

```
s_inf = BlochStack(cen(2*r) + no_rods(a-2*r))
s = Stack(inc_wg(a) + s_inf(0))
```

When you are using a `BlochStack` instead of an `InfStack`, all the scattering matrices are calculated. Moreover, `s.T12(0,0)` refers to the transmission to the Bloch mode of the semi-infinite stack, rather than to the transmission to some waveguide mode of one of the slabs making up the semi-infinite structure, which is usually not of interest.

Note that `BlochStack` can be used in the beginning of a `Stack` too.

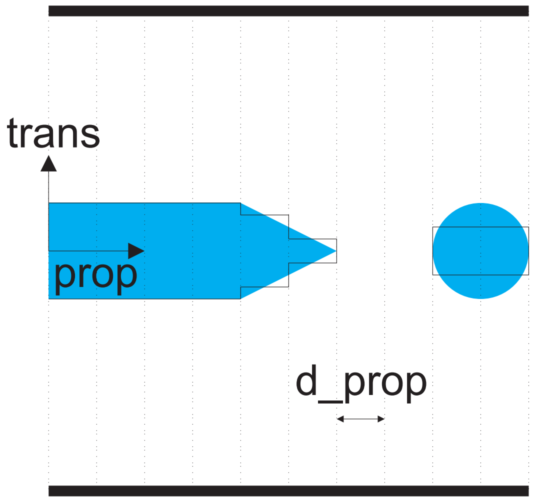
2.6 Defining complicated structures

CAMFR has a facility which makes it easier to define complicated non-trivial structures. Rather than manually specifying the geometry slice by slice, we can define a 2D structure in terms of higher level geometric shapes, like circles, triangles or rectangles. This definition can then be automatically discretised and converted to an expression which can be used to initialise a stack.

To illustrate this, we model the structure from Fig. 2.5.

```
#!/usr/bin/env python

####################################################################
#
# Illustrates how to define complicated structures using Geometry.
#
####################################################################

from camfr import *

set_lambda(1)
set_N(40)

air = Material(1)
mat = Material(3)

g = Geometry(air)

g += Rectangle(Point(0.0,-0.5), Point(2.0, 0.5), mat)
g += Triangle (Point(2.0, 0.5), Point(2.0,-0.5), Point(3.0, 0.0), mat)
g += Circle   (Point(4.5, 0.0), 0.5, mat)
```

```
set_lower_PML(-0.1)
set_upper_PML(-0.1)

prop0,  prop1,  d_prop  =  0.0, 5.0, 0.50
trans0, trans1, d_trans = -2.5, 2.5, 0.01

exp = g.to_expression(prop0,  prop1,  d_prop,
                      trans0, trans1, d_trans)

s = Stack(exp)

s.calc()

print s.R12(0,0)
```



The command `g = Geometry(air)` creates a `Geometry` object with air as background material. We can subsequently add shapes to this object, e.g. `g += Rectangle(Point(0.0,-0.5), Point(2.0, 0.5), mat)` adds a rectangle consisting of material `mat` with given lower left and top right vertices. Note that the first coordinate refers the propagation direction (horizontal in the case of Fig. 2.5).

Other shapes we can add are `Circle`, `Triangle` or `Square`. If two shapes overlap, the shape that was added last takes precedence.

Finally, we can convert the geometry object to an expression which can be used to create a stack. The arguments to this `to_expression` function are the range and precision in both the propagation and transverse direction (see Fig 2.5).

The increments `d_prop` and `d_trans` deserve some additional explanation, because they don't correspond to a uniform rectangular grid. The way the discretisation in performed is as follows. First, the structure is split up into a number of slices in the propagation direction, each `d_prop` long. Then, subsequent slices are combined if the material discontinuities in the transverse direction are no more than `d_trans` apart.. This creates some kind of dynamic resolution, because in regions where the shapes vary slower, we will have fewer (but thicker) slices. Also, it makes sure that subsequent identical slices are always combined into one slab. All of this makes for a more efficient discretisation than a traditional uniform grid.

Finally, we want to point out that it's quite easy for the user to add custom shapes by defining a Python object which overloads `intersection_at_x`. See `camfr/geometry.py` for more details.

2.7 General excitations for slabs

So far, we've mostly excited the structure with a single mode, often the fundamental one. However, by taking the right linear combination of modes, we can approximate any arbitrary incident field shape, like a gaussian or a plane wave.

To make this easier for the case of cartesian waveguides, CAMFR provides a number of variants to `Stack.set_incident_field`, which are illustrated in the following piece of code:

```
#! /usr/bin/env python

###################################################################
#
# Illustrates different excitations for cartesian stacks.
#
###################################################################

from camfr import *

set_N(50)
set_lambda(1.55)

GaAs = Material(3.5)
air  = Material(1.0)

set_lower_PML(-0.1)
set_upper_PML(-0.1)

slab = Slab(air(2)+ GaAs(1)+ air(2))
s = Stack(slab(1))

eps = 1e-3 # Precision for calculating overlap integrals.

# General excitation using a Python function.

A     = 1.0
x0    = slab.width()/2.
sigma = 0.5

def f(x):
  return A*exp(-0.5*((x-x0)/sigma)**2)

s.set_inc_field_function(f, eps)
```

```
# Faster variant using a built-in Gaussian excitation.

s.set_inc_field_gaussian(A, sigma, x0, eps)

# Plane wave with amplitude A and angle theta (radians).

A = 1.0
theta = 0.0
s.set_inc_field_plane_wave(A, theta, eps)
```

First, we define a Python function `f`, which describes a Gaussian. This field profile is then used as an excitation by calling `s.set_inc_field_function(f, eps)`, where `eps` is the precision with which the overlap integrals are calculated. `f` describes the `Ey` field in the case of TE polarisation and the `Hy` field for TM.

By using Python's `lambda` expressions, this code could also have been written as

```
s.set_inc_field_function(lambda x : exp(-0.5*((x-2.5)/0.5)**2), eps)
```

This is slightly faster, because Python doesn't have to reference the global variables `A`, `x0` and `sigma`.

For even greater speed, there exists a predefined Gaussian excitation:

```
s.set_inc_field_gaussian(A, sigma, x0, eps)
```

Finally, there is also a predefined excitation for a plane wave with amplitude `A` making an angle `theta` (in radians) with the propagation direction:

```
s.set_inc_field_plane_wave(A, theta, eps)
```

The accuracy which can be achieved to represent an arbitrary field is of course dependent on the number of modes used. Similarly, Gibbs phenomena can occur when modelling discontinuous functions, just like e.g. in Fourier analysis.

2.8 Working with dispersive materials

Since CAMFR is a frequency domain method, working with dispersive materials is in a sense trivial: just do a different simulation for each wavelength with adjusted material parameters.

There are however a few convenience functions to help you work with materials whose refractive index is specified in an external file. A popular format among material scientists is a text file consisting of three columns: the wavelength in nm, the real part of the index, the imaginary part of the index where a positive value means loss. The following code shows how to use such a file, automatically adapting it to the CAMFR conventions (i.e. wavelength in micron and loss corresponding to a negative imaginary index):

```
SiO2_factory = Dispersive_Material_Factory('SiO2.txt')

for wavelength in arange(1,2,0.1):
  set_lambda(wavelength)
  m = SiO2_factory()
  print m()
```

This assumes that a file `SiO2.txt` is present in your current location. Note how you first need to set the wavelength, and then let the factory object create a new material object for you to use in the rest of your calculations. Changing the wavelength after the structure is created will NOT change the refractive index.

In case your refractive index data is tabulated in a different way, have a look at `material.py` in the CAMFR source code at the implementation of `Dispersive_Material_Factory` to see how to adapt it.

Sometimes, the refractive index is given by a formula. In this case, you can write your own factory object like this:

```
class ZnS_Factory:

  def __call__(self):

    eps = 5.164 + 0.1208 / (get_lambda().real **2 - 0.27055**2)
    return Material(sqrt(eps))
```

2.9 Modelling light emission in planar LEDs

There is an extra set of functions written on top of CAMFR which allows you to study spontaneous emission and light extraction in planar stratified resonant cavity light emitting diodes. The sponteneous emission is modelled as a dipole source with a certain orientation which is then expanded in plane waves.

These functions can be imported by `from RCLED import *`. An example of using this module is given by the following code (available as `OLED.py` in the `examples/other` directory):

```
#!/usr/bin/env python

####################################################################
#
# Extraction efficiency in an organic LED
#
####################################################################

from RCLED import *

# Set parameters.

set_lambda(0.565)
# Create materials.

Al    = Material(1.031-6.861j)
Alq3  = Material(1.655)
NPD   = Material(1.807)
ITO   = Material(1.806-0.012j)
glass = Material(1.528)
air   = Material(1)
```

```
# Define layer structure.

top = Uniform(Alq3,  0.050) + \
      Uniform(Al,    0.150) + \
      Uniform(air,   0.000)

bot = Uniform(Alq3,  0.010) + \
      Uniform(NPD,   0.045) + \
      Uniform(ITO,   0.050) + \
      Uniform(glass, 0.000)

sub = Uniform(glass, 0.000) + \
      Uniform(air,   0.000)

ref = Uniform(10)

cav = RCLED(top, bot, sub, ref)

# Calculate.

cav.calc()

cav.radiation_profile(horizontal)
```

This defines a structure where the sequence of layers from top to bottom is air, 150 nm Al, 60 nm Alq3 with the dipole 50 nm from the Al, 45 nm NPD, 50 nm ITO, an optically thick glass substrate and finally air below.

Just as in other cavity calculations, the `top` and `bot` stacks are defined from the inside out, i.e. from the location of the emitter (in this case the Alq3 layer) to the outside world.

`Uniform` is a convenience function of the `RCLED` module, which relieves the user of having to create the `Planar` objects (See Section 2.4 [1D planar structures], page 29).

The distinction between the `bot` and `sub` stacks is needed because the substrate is considered to be optically thick, in the sense that reflections from the substrate-air interface do not contribute to the microcavity effect. In this way, it is also possible to separately calculate how much light is coupled into the glass substrate and how much is coupled into the air, which is sometimes useful information. Note that in order to calculate the extraction to air, the correct Fresnel coefficients for the substrate-air interface are taken into account.

The `ref` stack also deserves additional explanation. For numerical reasons, the emission is actually taking place in a zero-thickness reference layer which in this example is set to have an index of 10. The index of the reference layer is particularly important when the dipole emits close to a metal layer and you want to model the coupling to surface plasmons accurately. In this case, you might want to experiment with the reference index to make sure it is high enough.

The `cav.calc()` command prints out the following summary of where the light goes for different dipole orientations:

```
Source orientation: vertical (weight=1)
Emitted power                                : 2.393
Extraction efficiency to substrate           : 0.094
Extraction efficiency to bottom outside world : 0.009

Source orientation: horizontal (weight=2)
Emitted power                                : 0.909
Extraction efficiency to substrate           : 0.758
Extraction efficiency to bottom outside world : 0.312

Source orientation: average
Emitted power                                : 1.404
Extraction efficiency to substrate           : 0.380
Extraction efficiency to bottom outside world : 0.139
```

The vertical dipole is oriented perpendicular to the layer interfaces, and the horizontal one has random azimuth in the plane of the layers. Also note that the convention we adopt here is that the useful emission is in the bottom direction. If your device is actually top emitting, you have to invert the layer stack.

If you want to do some post-processing on these numbers, or plug them in an optimisation routine, the function `cav.calc()` actually returns a data structure which can be used to access these numbers. E.g., the following code will print out the same sequence of nine numbers as above:

```
res=cav.calc()

print res[vertical].P_source
print res[vertical].eta_sub
print res[vertical].eta_out

print res[horizontal].P_source
print res[horizontal].eta_sub
print res[horizontal].eta_out

print res.P_source
print res.eta_sub
print res.eta_out
```

If you are only interested in e.g. the horizontal dipole, you can specify this in the code as follows: `cav.calc(sources=[horizontal])`. (By default, the following calculation is performed: `calc(sources = [vertical,horizontal], weights = [1,2]`, indicating that the horizontal dipole needs to be counted twice to get the correct average over source orientation.)

Finally, the command `cav.radiation_profile(horizontal)` will plot the TE and the TM radiation profile of a horizontal dipole. To look at the radiation profile of a vertical dipole, simply replace `horizontal` by `vertical`.

To do your own post-processing, the function returns the following values:

```
theta, P_TE, P_TM = cav.radiation_profile(horizontal)
```

By default, the `radiation_profile` command plots the radiation profile in the bottom output medium, in the case of our example air. To look at the field in other locations, there is an optional function argument `location`, which can be any of the following strings: `'source_top'` or `'source_bot'` (to look at the upward and downward radition profile at the dipole location), `'top'` (in the top output medium, in this case above the metal), `'sub'` (in the substrate, in this case glass) and `'out'` (the default value). Note that the angles always refer to angles in the particular layer indicated by the `location` parameter, e.g. for `'sub'` the angles are angles in the glass substrate.

The values which are plotted by the function are densities per unit solid angle, i.e. the total power is given by the integral of the `P` values times an infinitesimal solid angle. For densities per other units, use the following optional argument: `density='per_kt'` or `density='per_kx_ky'`.

To turn off the interactive plotting (e.g. for use in batch calculations) use the optional argument `show = False`. To write the figures to files, specify the base filename by the optional argument `filename='my_file_name'`, which will create `my_file_name_TE.png` and `my_file_name_TM.png`.

2.10 Modelling light emission in LEDs with gratings

The presence of a grating (or a photonic crystal if you prefer), can have a big influence on optical outcoupling. Therefore, CAMFR also contains a library `GARCLED` for modelling these kinds of grating-assisted resonant-cavity LEDs. The gratings can be placed at the interface between the cavity and the substrate, and/or at the interface between the substrate and the air. Multiple incoherent reflections in the optically thick substrate can also be taken into account.

The following is a simple example:

```
#!/usr/bin/env python

####################################################################
#
# Extraction efficiency in an organic LED with gratings
#
####################################################################

from GARCLED import *

# Set parameters.

set_lambda(0.565)

orders = 2
set_fourier_orders(orders,orders)

L = 0.400
set_period(L, L)

# Create materials.

Al    = Material(1.031-6.861j)
Alq3  = Material(1.655)
NPD   = Material(1.807)
SiO2  = Material(1.48)
SiN   = Material(1.95)
ITO   = Material(1.806-0.012j)
glass = Material(1.528)
air   = Material(1)
```

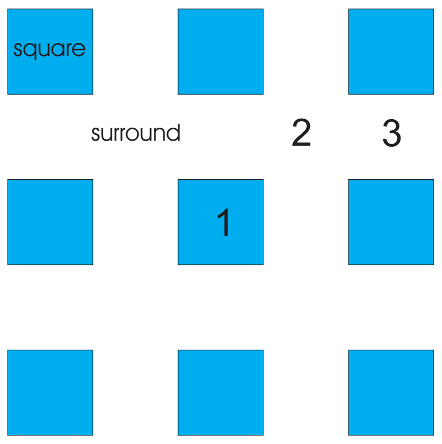
```
# Define layer structure.

top = Uniform(Alq3,  0.050) + \
      Uniform(Al,    0.150) + \
      Uniform(air,   0.000)

bot = Uniform(Alq3,  0.010) + \
      Uniform(NPD,   0.045) + \
      Uniform(ITO,   0.050) + \
      SquareGrating(L/4, SiO2, SiN, "1", 0.100) + \
      Uniform(glass, 0.000)

sub = Uniform(glass, 0.000) + \
      Uniform(air,   0.000)

ref = Uniform(3) # Don't choose too high: otherwise takes too long

cav = RCLED(top, bot, sub, ref)

# Calculate.

cav.calc(sources=[vertical, horizontal_x], weights=[1, 2],
         steps=30, symmetric=True)

cav.radiation_profile(horizontal_x, steps=30)
```

The main structure of the code should look very similar as compared to the planar case from the previous section.

The command `set_period(Lx,Ly)` allows you to set the period in the x and y transverse directions. `set_fourier_orders(Nx,Ny)` sets the number of diffraction orders that are taken into account when simulating the grating with the rigorous coupled wave analysis model (RCWA). This number has a big influence on the calculation time.

Also for reasons of calculation time, the index of the reference material should not be excessively high (but at least as high as the index of the material where the emission takes place).

`SquareGrating(D, square_m, surround_m, pos, thickness)` is a convenience function to create a square grating of squares of material `square_m` in a background material `surround_m`. The side of the squares has length `D` and the period of the grating has already been set by `set_period(Lx,Ly)`. The thickness or height of the grating layer is given by the `thickness` parameter. The position of the emitting dipole with respect the grating pattern has an influence on the radiation profile. With the `pos` argument you can specify three different high-symmetry positions '1', '2' and '3'. These are illustrated in the following figure.



For this particular example, the extraction efficiency for the `horizontal_x` and `horizontal_y` dipoles will be the same due to symmetry. Therefore, we only calculate the `horizontal_x` dipole, but give it a weight coefficient of two, so that the average over dipole orientation gives the correct result.

The `steps` parameter determines in how many equidistant steps the basic integration interval along one `k`-axis is subdivided. Together with `set_fourier_orders(Nx,Ny)`, these are the main parameters that determine accuracy and calculation time.

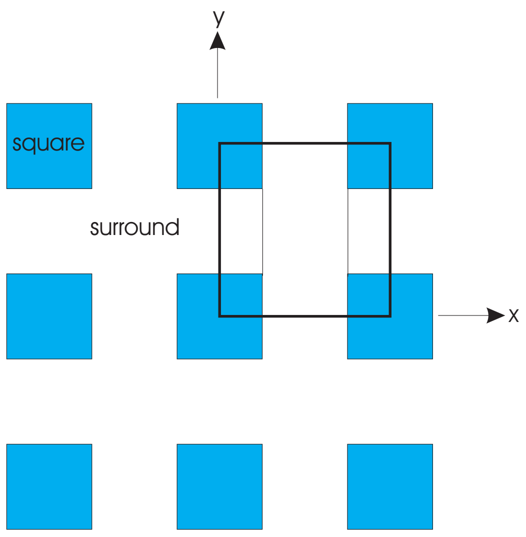
`symmetric=True` tells the software that the grating together with the dipole orientation possess mirror symmetry along the `x` and `y` axis. This allows the integration to be restricted to positive values of `kx` and `ky`, resulting in a factor 4 speedup.

`cav.calc()` also returns an object containing all the results, which can be queried in exactly the same way as for the planar case.

Finally, `cav.radiation_profile(horizontal_x, steps=30)` calculates the TE and TM polarised radiation profile for a single source orientation. This functions in exactly the same way as in the planar case, e.g. it also supports the same optional arguments: `radiation_profile(source, steps=30, location='out', density='per_solid_angle', show=True, filename=None)`. The return values of this function are similarly `k, P_TE, P_TM`, where `k` is a 1D vector containing the k-values which are used both for kx and ky. `P_TE` and `P_TM` are matrices.

There is also a function to create a square grating with round holes: `SquareGratingRoundHoles(D, dx, dy, core_m, surround_m, pos, thickness)`.
Here, `dx` and `dy` are the same discretisation parameters used for `Geometry` (See Section 2.6 [Defining complicated structures], page 34).

To show how you can manually create different unit cells which are not covered by the convenience functions, we will now explain the code behind the `SquareGrating` at position 1. As shown in the following figure, the unit cell is built up from three vertical slices, which are represented in the following code by the `Slabs` `s1`, `s2` and `s1`. These are then combined in a `BlochSection`.



```
s1 = Slab(square_m(D/2.) + surround_m(_Ly-D) + square_m(D/2.))
s2 = Slab(surround_m(_Ly))

grating = BlochSection(s1(D/2.) + s2(_Lx-D) + s1(D/2.))
```

To use this `grating` object in your stack definition, just do the following:

```
bot = Uniform(Alq3,  0.010) + \
      Uniform(NPD,   0.045) + \
      Uniform(ITO,   0.050) + \
      grating(0.010) + \
      Uniform(glass, 0.000)
```

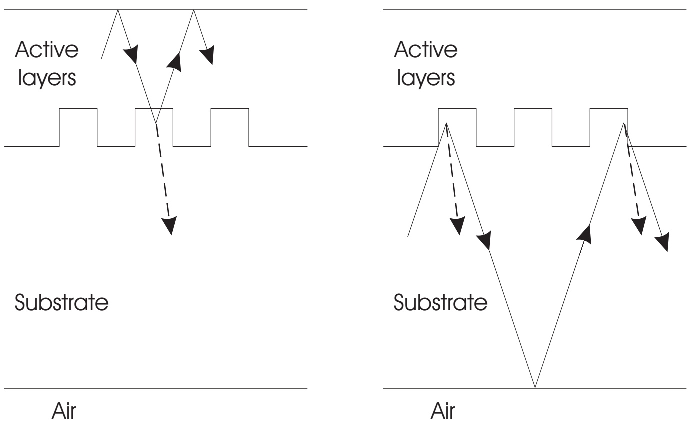
In reality, the dipole position will be random with respect to the grating. The following code illustrates a more elaborate example, where the emission is averaged over the three high-symmetry dipole positions. Also, the structure is calculated for several grating parameters.

```
#!/usr/bin/env python

######################################################################
#
# Extraction efficiency in an organic LED with gratings
# averaged over dipole position.
#
######################################################################

from GARCLED import *

def calc(L, h, D, orders, wavelength=0.565):

  # Set parameters.

  set_period(L, L)

  set_lambda(wavelength)

  set_fourier_orders(orders, orders)

  print
  print "wavelength, orders L h:", wavelength, orders, L, h

  # Create materials.

  Al    = Material(1.031-6.861j)
  Alq3  = Material(1.655)
  NPD   = Material(1.807)
  ITO   = Material(1.806-0.012j)
  SiO2  = Material(1.48)
  SiN   = Material(1.95)
  glass = Material(1.528)
  air   = Material(1)

  eta_sub, eta_out = [], []
```

```
# Define layer structure.

for pos in ['1', '2', '3']:

  print
  print "*** Position", pos, "***"
  print

  top = Uniform(Alq3,  0.050) + \
        Uniform(Al,    0.150) + \
        Uniform(air,   0.000)

  bot = Uniform(Alq3,  0.010) + \
        Uniform(NPD,   0.045) + \
        Uniform(ITO,   0.050) + \
        SquareGrating(D, SiO2, SiN, pos, h) + \
        Uniform(glass, 0.000)

  sub = Uniform(glass, 0.000) + \
        Uniform(air,   0.000)

  ref = Uniform(3) # Don't choose too high: otherwise takes too long

  cav = RCLED(top, bot, sub, ref)

  # Calculate average over source orientation for a given position.

  res = cav.calc(sources = [vertical, horizontal_x, horizontal_y],
                 steps=100, symmetric=True)

  eta_sub.append(res.eta_sub)
  eta_out.append(res.eta_out)

  sys.stdout.flush()

  free_tmps()
```

```
      # Print average over position and source.

      print
      print "*** Averaged over dipole position and source orientation ***"
      print
      print "Averaged extraction efficiency to substrate            :", \
                                                           average(eta_sub)
      print "Averaged extraction efficiency to bottom outside world :", \
                                                           average(eta_out)

    # Loop over period and thickness of the grating.

    for L in [.300, .500, .700]:
      for h in [.100, .200, .400]:
        calc(L, h, L/2, orders=3)
```

A grating close to the light emitting layer will have an influence on the extraction of the light trapped inside the active layers. However, as the figure below illustrates, this grating can also help to couple out light trapped inside the substrate, provided the lateral extent of the device is sufficienly large.



These multipass effects in the substrate happen incoherently because of the large thickness of the substrate. Normally, they are not taken into account, i.e. the substrate is considered to be single-pass. You can calculate the multipass effects by specifying the optional parameter `single_pass_substrate=False` to `cav.calc()`.

Finally, instead of placing the grating at the cavity-substrate interface, you can place it at the substrate-air interface by including a grating in the `sub` stack. In this case, the grating will only have an influence on the substrate modes, and you need to specify `single_pass_substrate=False`.

2.11 CAMFR 3D

The following pages illustrate how to use CAMFR to find modes of waveguides with an arbitrary 2D cross-section. These 2D waveguides can subsequently be stacked together in `Stack`, `BlochStack`, or `Cavity` objects. Since this code is still in development, some of the things mentioned here are subject to change.

The following code looks for the ground mode of a (quarter) of a square waveguide, exploiting symmetry:

```
from camfr import *

set_lambda(1.0)
set_N(1)

core = Material(1.5)
clad = Material(1.0)

set_section_solver(L)
set_mode_correction(full)

set_left_wall(H_wall)
set_right_wall(H_wall)
set_right_PML(-0.05)

set_lower_wall(slab_E_wall)
set_upper_wall(slab_E_wall)
set_upper_PML(-0.05)

wg = Slab(core(0.25) + clad(1))
air = Slab(clad(wg.width()))

all = wg(0.25) + air(1)

s = Section(all, 10, 40)
s.calc()

print s
```

In order to explain all the options, some insight into the nature of the 3D model is needed.

As a first step of the model, a coarse estimate of all the modes of a `Section` (a waveguide with a 2D cross section) is created using a plane-wave model. In a second step, these estimates (or a subset of them) are refined using an expansion based on modes of the 1D waveguides making up the `Section`. Since these can better satisfy the boundary conditions, more accurate results can be obtained.

The line `s = Section(all, 10, 40)` means we use (2*10+1)^2 plane waves in the first stage, and refine it using 40 1D modes in each Slab during the second stage. `all` can be any 2D stack, including those fabricated with the `Geometry` function. Note: currently the 100 is a rough estimate: the software will print out how many modes were actually used.

The user has the option of choosing which modes get refined during the second stage:

- `set_mode_correction(full)`: all modes.
- `set_mode_correction(guided_only)`: only guided modes
- `set_mode_correction(snap)`: don't refine any modes, but set numerical noise on the imaginary part of `kz` for guided modes to zero (obsolete)..
- `set_mode_correction(none)`: don't refine any modes (default).

When you're only interested in the modes of a `Section`, and not putting `Section`s together is a `Stack`, `set_mode_correction(full)` is the way to go, as it is more accurate. For a full 3D model with different `Section`s however, such an approach can be very time-consuming, especially in the calculation of the overlap integrals, which are much quicker in a plane wave basis. In that case, it's better to use `guided_only` or `none`. `none` is the fastest, but can sometimes require a lot of plane waves to get accurate results. `guided_only` is a hybrid approach, which is more accurate, but can sometimes miss a mode. When using `guided_only`, it's adviseable to use `set_orthogonal(False)` as well, as using a mixture of two expansion sets somewhat distorts the mode orthogonality.

Some loose remarks:

- If you have estimates on the location of the modes, e.g. from a previous simulation for a slightly different structure, you can skip stage 1 and use user-provided estimates instead: `s.set_estimate(n_eff1)`, `s.set_estimate(n_eff2)`, ...
- When you're only interested in the fundamental mode, it's tempting to use `set_N(1)`. However, sometimes this yields a higher-order mode, because the noise on the coarse estimate during stage 1 can be such that the order of mode 0 and mode 1 gets swapped. One option is to use more palne waves, but often it works just as well to increase the value of `set_N()`. Taking more estimates to stage 2 will improve the chances of finding all modes you are interested in.

Some interface quirks which are due to the development nature of the model and will probably change in the final version :

- For left and right walls, the syntax is `E_wall` and `H_wall`, while for upper and lower walls it's `slab_E_wall` and `slab_H_wall`. For the left and right side, you can also use `no_wall`, which is e.g. very accurately to study leakage losses provided you don't use PML.
- For the `NT` and `Li` solver, currently opposite walls have to be of the same type (i.e. no electric wall on the left with a magnetic wall on the right).

2.12 Low level field plotting

In you want more control over the field plotting than is provided by the `plot` widgets, you can use a set of lower level plotting functions:

```
#####################################################################
#
# Illustrates low level plotting.
#
#####################################################################

from camfr import *

set_N(40)
set_lambda(1.55)

# Define materials.

GaAs = Material(3.5)
air  = Material(1.0)

# Define geometry.

set_lower_PML(-0.1)
set_upper_PML(-0.1)

wg  = Slab(air(2.0) + GaAs(0.2) + air(2.0))
gap = Slab(air(4.2))

s = Stack(wg(0) + gap(1) + wg(0))

inc = zeros(N())
inc[0] = 1
s.set_inc_field(inc)

s.calc()
```

```
# Do some plotting.

r_x = arange( 1.5, 2.8, 0.05)
r_z = arange(-1.0, 2.0, 0.05)

print "Doing some plots. Close window to continue to the next one."

wg.plot_n(r_x)

wg.mode(0).plot_field(lambda f : f.E2().real, r_x)

s.plot_field(lambda f : f.E2().real, r_x, r_z)

s.animate_field(lambda f : f.E2(), r_x, r_z)
```

This code is pretty self-explanatory. Ranges are used to specify the area to be plotted and the resolutions.

Note the use of Python's lambda functions to indicate which field component to plot. Lambda functions are little anonymous functions which in this case will be called on a `Field` object. This greatly increases the flexibility of the `plot_field` function: e.g. to plot the absolute value one would write `lambda f : abs(f.E2()**2)`.

In all these plots, you can zoom by dragging a rectangle in the window. Also, left clicking and dragging under the horizontal axis will print the corresponding coordinates. Note that these functions are also very handy in interactive mode, to quickly inspect modes and their field profiles.

Important to note is that `animate_field` requires a complex number to work correctly, so the full complex field component needs to be used rather than e.g. the real part.

`plot_n`, `plot_field` and `animate_field` can take an additional named argument like `filename="out.gif"` which indicates the filename to write the picture to. For pictures, a variety of formats are supported (gif, jpg, png, eps, ...), where the suffix of the filename will determine the format to be used. For animations only uncompressed animated gif is supported.

Note that these plotting functions also work for `BlochStack`, `BlochMode` and `Cavity`.

For additional functionality of these functions, see the reference guide.

2.13 Tips for troubleshooting problems

When CAMFR produces results that are clearly nonphysical, here are number of things you can do to improve convergence:

- If CAMFR is missing modes, increase the precision with `set_precision(p)`, where p is an integer number that defaults to 100. If you expect missing radiation modes, increase `set_precision_rad()` beyond the default of 100.
- Increase or decrease the absorption in the PML, the real distance between the metal walls and the number of modes. Using too few modes will cause convergence problems. A PML absorption that is too low will not damp out all parasitic reflections. On the other hand, if the absorption is excessively high, some modes can be missed. Increasing the real distance between the walls and lowering the PML absorption is a viable option in this case, but also requires that you use more modes.
- If the PML absorption is high, try `set_chunk_tracing(0)` at the expense of longer run times.
- If there are problems due to inverting numerically unstable matrices, you can change the way these situations are handled. Use `set_stability(extra)` to use extra row and column equilibration. With `set_stability(SVD)` a pseudo inverse is calculated using singular value decomposition.
- If your waveguides don't have multiple cores, try using `set_degenerate(0)`.
- If the field profiles seem to be calculated incorrectly, try `set_orthogonal(0)`. This will cause CAMFR to treat modes which are not exactly orthogonal due to rounding errors as non-orthogonal.
- If you are dealing with very lossy materials, using a different solver might help: `set_solver(series)` and `set_mode_surplus(n)`. This will first construct an initial estimate of the modes based on a plane wave expansion. The number of plane waves used in this expansion is `n` times `get_N()`. These estimates are subsequently refined using the full dispersion relation.
- For trouble with uncoupled waveguides, try experimenting with `set_unstable_exp_threshold`. E.g. setting it to 1e-6 will mean waveguides will be treated sooner as decoupled than when the parameter was 1e-12.
- If you're working with circular structures in the regime of backward and complex modes (e.g. when the metal wall is close to the last interface), you can use `set_backward_modes(1)` for extra stability.

If CAMFR crashes, make sure you haven't called `free_tmps()` too soon, i.e. before you're done with the objects. Another common cause of crashes to look out for is defining a `wall` or an `InfStack` inside an expression. Also, if you're upgrading from a pre 1.0 version of CAMFR, make sure all your coordinates are purely real.

Finally, we want to remind the user that in Python `1/3 = 0` rather than `.3333`. This will change in Python 3.0, but if you already want such behaviour now, include `from __future__ import division` at the top of your scripts.

3 Reference guide

This chapter contains a listing all classes and functions contained in CAMFR.

set_lambda

`set_lambda(Real)`: sets the wavelength.

See [get_lambda], page 56.

get_lambda

`get_lambda()`: returns the wavelength.

Note: this is the only CAMFR function which uses a `get_` suffix, as `lambda` is a reserved Python keyword.

See [set_lambda], page 56.

set_N

`set_N(int)`: sets the number of modes used in the series expansion.

See [N], page 56.

N

`N()`: returns number of modes used in the series expansion.

See [set_N], page 56.

set_polarisation

`set_polarisation(Pol)`: sets the polarisation. `Pol` can be either `TE` (default) or `TM`. Only relevant when the polarisations are indeed decoupled, i.e. for 2D Cartesian structures and cylindrical ones with Bessel order 0. Ignored otherwise.

set_gain_material

`set_gain_material(Material)`: sets the gain material, i.e. the material whose imaginary part of its refractive index will be adjusted to find a lasing mode in a `Cavity`.

See [Cavity], page 63.

set_stability

`set_stability(Stability)`: determines how to handle matrices that are close to singular (rare case). `Stability` can be either

- `normal`: default, no special measures
- `extra`: uses row and column equilibration
- `SVD`: uses singular value decomposition to calculate a pseudo inverse

set_precision

`set_precision(int)`: sets the precision used when scanning for guided modes (defaults to 100, higher values have less chances of missing modes, but are slower).

See [set_precision_rad], page 57

set_precision_rad

`set_precision_rad(int)`: sets the precision used when scanning for radiation modes (defaults to 100, higher values have less chances of missing modes, but are slower).

See [set_precision], page 57.

set_sweep_from_previous

`set_sweep_from_previous(bool)`: determines whether or not previously calculated eigenmodes (e.g. for a slightly different wavelength) are used as a starting point for finding the new modes. Can speed up the procedure, but is sometimes less stable. Note: a Python bool is 1 for true, and 0 for false.

set_chunk_tracing

`set_chunk_tracing(bool)`: determines whether or not all modes are located at the same time or in chunks. Defaults to true, which is faster but can sometimes lose modes, especially for high PML absorption. Note: a Python bool is 1 for true, and 0 for false.

set_degenerate

`set_degenerate(bool)`: determines whether or not special precautions are taken to locate degenerate modes. This is generally a good idea and therefore this option defaults to true, but it can sometimes cause trouble. Note: a Python bool is 1 for true, and 0 for false.

set_orthogonal

`set_orthogonal(bool)`: sometimes modes that should be orthogonal are not exactly orthogonal due to rounding errors. Setting this option to false will treat these modes as non-orthogonal, and can sometimes drastically improve convergence. Note: a Python bool is 1 for true, and 0 for false.

set_solver

`set_solver(series)`: activates a different solver to calculate the modes in a slab. It will first construct an estimate of the modes based on a plane wave expansion. These estimates will subsequently be refined using the full dispersion relation. This is especially useful for lossy structures. The number of plane waves used can be set through `set_mode_surplus()`. See [set_mode_surplus], page 57.

set_mode_surplus

`set_mode_surplus(n)`: sets the number of plane waves used in the series expansion to `n*get_N()`. Only affects the series solver for Slabs. See [set_solver], page 57.

set_unstable_exp_threshold

`set_unstable_exp_threshold(eps)`: sets numerical parameter dealing with how soon waveguides will be treated as uncoupled. Setting this parameter to 1e-6 will result in a quicker decoupling as compared to e.g. 1e-12.

set_backward_modes

`set_backward_modes(1)`: use this for extra stability if you're working with circular structures in the regime of backward and complex modes (e.g. when the metal wall is close to the last interface). Involves a performance penalty, so it is off be default.

set_circ_order

`set_circ_order(int)`: set the order of the Bessel modes and the angular dependence in cylindrical structures.

set_circ_field_type

`set_circ_field_type(Fieldtype)`: determines which kind of angular field dependence is used when calculating field due to a current source. Can be either `cos_type` (default) or `sin_type`.

set_lower_PML

`set_lower_PML(p)`: gives the lower cladding of all subsequently defined `Slab`s an imaginary thickness of `p*1j`, with `p` usually negative for absorption. Implements PML boundary conditions. The lower cladding is at `x=0`, at the first term in the expression to create a `Slab`. By default, `p` is zero. Needs to be set BEFORE structures are defined.

See [set_upper_PML], page 58, [set_circ_PML], page 58.

set_upper_PML

`set_upper_PML(p)`: gives the upper cladding of all subsequently defined `Slab`s an imaginary thickness of `p*1j`, with `p` usually negative for absorption. Implements PML boundary conditions. The upper cladding is at `x=slab.width()`, at the last term in the expression to create a `Slab`. By default, `p` is zero. Needs to be set BEFORE structures are defined.

See [set_lower_PML], page 58, [set_circ_PML], page 58.

set_circ_PML

`set_circ_PML(p)`: gives the cladding of all subsequently defined circular waveguides an imaginary thickness of `p*1j`, with `p` usually negative for absorption. Implements PML boundary conditions. By default, `p` is zero. Needs to be set BEFORE structures are defined.

See [set_lower_PML], page 58, [set_upper_PML], page 58.

set_lower_wall

`set_lower_wall(SlabWall)`: sets the lower wall (i.e. at `x=0`) to the given wall. Has an effect on all `Slabs` that are subsequently defined.

Note that walls can also be set on a slab-by-slab basis, with `slab.set_lower_wall()`.

See [set_upper_wall], page 59.

set_upper_wall

`set_upper_wall(SlabWall)`: sets the upper wall (i.e. at `x = slab.width()`) to the given wall. Has an effect on all `Slabs` that are subsequently defined.

Note that walls can also be set on a slab-by-slab basis, with `slab.set_upper_wall()`.

See [set_lower_wall], page 59.

free_tmps

`free_tmps()`: frees all the temporary scattering matrices that were calculated so far. Useful at the end of an inner loop, in order to save memory.

Drawback is that when the same structures are needed in a subsequent loop iteration, they have to be recalculated. Whether or not this is an issue depends on the actual simulation.

Coord

A three-dimensional coordinate.

Constructors:

- `Coord(Complex c1,Complex c2,Complex z)`
- `Coord(Complex c1,Complex c2,Complex z, Limit c1_l, Limit c2_l, Limit z_l)`

`c1` and `c2` stand for `x` and `y` in Cartesian coordinates, and for `rho` and `phi` in cylindrical coordinates.

Optionally, `Limit` can be specified as either `Plus` or `Min`, e.g. to indicated on which side of an index discontinuity the field has to be calculated.

Field

A data structure containing an electromagnetic field at a given location.

Member functions:

- `E1()`: first component (`x` in Cartesian, `rho` in cylindrical systems) of `E` field.
- `E2()`: second component (`y` in Cartesian, `phi` in cylindrical systems) of `E` field.
- `Ez()`: `z` component of `E` field.
- `H1()`: first component (`x` in Cartesian, `rho` in cylindrical systems) of `H` field.
- `H2()`: second component (`y` in Cartesian, `phi` in cylindrical systems) of `H` field.
- `Hz()`: `z` component of `H` field.
- `S1()`: first component (`x` in Cartesian, `rho` in cylindrical systems) of Poynting vector `E x H*`.

- `S2()`: second component (`y` in Cartesian, `phi` in cylindrical systems) of Poynting vector `E x H*`.
- `Sz()`: `z` component of Poynting vector `E x H*`.
- `abs_E()`: magnitude of `E` field.
- `abs_H()`: magnitude of `H` field.
- `abs_S()`: magnitude of Poynting vector.

Material

An isotropic material.

Constructors:

- `Material(Complex n)`
- `Material(Complex n, Complex mur)`

Member functions:

- `n()`: returns refractive index.
- `set_n()`: sets refractive index.
- `epsr()`: returns relative permittivity.
- `mur()`: returns relative permeability.
- `set_mur()`: sets relative permeability.
- `eps()`: returns permittivity.
- `mu()`: returns permeability.
- `gain()`: returns material gain at current wavelength in 1/cm.

Waveguide

A general waveguide. Serves a base class for e.g. `Slab` or `Circ`.

Member functions:

- `core()`: returns core material
- `eps(Coord)`: returns permittivity at a certain coordinate.
- `mu(Coord)`: return permeability at a certain coordinate.
- `n(Coord)`: return refractive index at a certain coordinate.
- `N()`: returns number of modes in this waveguide. Is the number set by `set_N()`.
- `mode(int i)`: returns Mode with index `i`. Zero is the fundamental mode.
- `calc()`: calculates the modes in this waveguide.
- `plot()`: interactively plots the modes in this waveguide.
- `plot_n(r_x))`: lower level routine to plot the refractive index. `r_x` is range objects (e.g `arange(x0, x1, dx)`) specifying which interval to plot and with which resolution. `x` is the transverse direction. An extra named argument `filename='outputname.formatextension'` is supported. For pictures, a variety of formats are supported (gif, jpg, png, bmp, eps, ps, tiff, pdf, xbm, dib), where the suffix of the filename will determine the format to be used.

- `plot_field(component, r_x))`: lower level field plot routine. `component` is a function operating on a `Field` object specifying which field component to plot, e.g. `lambda f : f.E2().real`. `r_x` is a range object (e.g `arange(x0, x1, dx)`) specifying which interval to plot and with which resolution. `x` is the transverse direction. Note that the function also supports an additional named argument `filename='outputname.formatextension'`. For pictures, a variety of formats are supported (gif, jpg, png, bmp, eps, ps, tiff, pdf, xbm, dib), where the suffix of the filename will determine the format to be used. Other arguments are `overlay_n` (defaults to true) which overlays the index profile on the field plot, and `contour` (defaults to true) which uses contours to show the index profile rather than gray scale.

See [Slab], page 68, [Circ], page 68.

Mode

Member functions:

- `kz()`: returns propagation factor.
- `n_eff()`: returns effective index.
- `pol()`: returns polarisation.
- `field(Coord)`: returns field at a given coordinate.

E_Wall

Constructor:

- `E_Wall(Waveguide)`

An electric wall with a Waveguide in front of it. Used a a boundary condition in the `z`-direction, not to be confused with boundaries in the transverse direction.

Note: the default boundary condition in the `z`-direction is an open boundary.

See [H_Wall], page 61.

H_Wall

Constructor:

- `H_Wall(Waveguide)`

An magnetic wall with a Waveguide in front of it. Used a a boundary condition in the `z`-direction, not to be confused with boundaries in the transverse direction.

Note: the default boundary condition in the `z`-direction is an open boundary.

See [E_Wall], page 61.

Expression

This class only has to be used explicitly when assembling expressions term-by-term:

`e = Expression()` creates an empty expression, and `e.add(Term)` adds a term to it.

See [Term], page 62.

Term

Basic building block of expressions. Some examples of terms are

```
material(d)
waveguide(d)
2*expression
(expression)
```

See [Expression], page 61.

Stack

A stack of waveguides.

Constructor:

- `Stack(Expression)`

Member functions:

- `calc()`: calculates the scattering matrices of the stack.
- `free()`: frees the memory allocated for the scattering matrices.
- `inc()`: returns incidence waveguide.
- `ext()`: return exit waveguide.
- `length()`: returns length of the stack along the `z`-axis.
- `width()`: returns the `c1`-length of the stack.
- `n(Coord)`: returns refraction index at a given coordinate.
- `eps(Coord)`: returns permittivity at a given coordinate.
- `mu(Coord)`: returns permeability at a given coordinate.
- `R12()`: returns reflection matrix for fields incident from the left (`z=0`).
- `R12(i,j)`: returns element of the reflection matrix R12, index starting at zero. Because of the definition of this matrix, this is the reflection from mode `j` to mode `i`. Similar functions exist for the transmission matrix `T12`, and for `R21` and `T21` describing incidence from the right side. (`z=stack.length()`)
- `set_inc_field(vector)`: sets incident field from left side (`z=0`).
- `set_inc_field(vector, vector)`: sets incident field from left and right side.
- `set_inc_field_gaussian(A, sigma, x0, eps)`: set a gaussian incident field `A*exp(-0.5*((x-x0)/sigma)**2` and use precision `eps` to calculate the overlap integrals.
- `set_inc_field_plane_wave(A, theta, eps)`: set a plane wave incident field with amplitude `A`, angle `theta` in radians, and use precision `eps` to calculate the overlap integrals.
- `set_inc_field_function(f, eps)`: set an incident field described by a function `f` and use precision `eps` to calculate the overlap integrals. `f` describes the `E2` for TE polarisation and the `H2` field for TM polarisation.
- `plot_n(stack, r_x, r_z)`: lower level routine to plot the refractive index. `r_x` and `r_z` are range objects (e.g `arange(x0, x1, dx)`) specifying which interval to plot and with which resolution. `x` is the transverse direction, `z` the propagation direction. An extra named argument `filename='outputname.formatextension'` is supported. For

pictures, a variety of formats are supported (gif, jpg, png, bmp, eps, ps, tiff, pdf, xbm, dib), where the suffix of the filename will determine the format to be used.

The following functions are only meaningful when an incident field has been set:

- `plot()`: interactively plots the fields in the stack.
- `plot_field(component, r_x, r_z))`: lower level field plotting routine. `component` is a function operating on a `Field` object specifying which field component to plot, e.g. `lambda f : f.E2().real`. `r_x` and `r_z` are range objects (e.g `arange(x0, x1, dx)`) specifying which interval to plot and with which resolution. `x` is the transverse direction, `z` the propagation direction. Note that the function also supports an additional named argument `filename='outputname.formatextension'`. For pictures, a variety of formats are supported (gif, jpg, png, bmp, eps, ps, tiff, pdf, xbm, dib), where the suffix of the filename will determine the format to be used. Other arguments are `overlay_n` (defaults to true) which overlays the index profile on the field plot, and `contour` (defaults to true) which uses contours to show the index profile rather than gray scale.
- `animate_field(stack, component, r_x, r_z)`: animates the 2D field profile. `component` is a function operating on a `Field` object specifying which (complex) field component to plot, e.g. `lambda f : f.E2()`. `r_x` and `r_z` are range objects (e.g `arange(x0, x1, dx)`) specifying which interval to plot and with which resolution. `x` is the transverse direction, `z` the propagation direction. Note that the function also supports an additional named argument `filename='outputname'`, which writes the animation to an uncompressed gif. Other arguments are `overlay_n` (defaults to true) which overlays the index profile on the field plot, and `contour` (defaults to true) which uses contours to show the index profile rather than gray scale.
- `field(Coord)`: returns field at a given coordinate.
- `inc_field()`: returns incident field from left side.
- `refl_field()`: returns reflected field from left side.
- `trans_field()`: returns transmitted field to right side.
- `fw_bw(z, limit=Min)`: returns a tuple with two vectors containing the forward and backward field expansions at position `z, limit`. `limit` is optional and can be either `Min` or `Plus`.
- `inc_S_flux(Complex x0, Complex x1, Real eps)`: returns power flux in the z-direction at `z=0` between `x0` and `x1`, calculated with relative precision `eps`.
- `ext_S_flux(Complex x0, Complex x1, Real eps)`: returns power flux in the z-direction at `z=stack.length()` between `x0` and `x1`, calculated with relative precision `eps`.
- `lateral_S_flux(Complex x0)`: returns power flux in the `x-direction` at `x=x0` between `z=0` and `z=stack.length()`.

Cavity

A cavity containing a cavity cut, dividing it into a top and a bottom stack. These stacks are defined as seen from the cavity cut. A gain material has to be set with `set_gain_material(Material)`.

Constructor:

- `Cavity(Stack bottom, Stack top)`

Member functions:

- `find_mode(Real lambda0, Real lambda1, Real n_imag0=0, Real n_imag1=0.015, int passes=1)`: finds a laser mode in a given wavelength and gain interval. Wavelength and gain are optimised around the resonance for `passes` times. For the gain interval, a default range is assumed for the imaginary part of the gain index between 0 and 0.015. The parameters of the laser mode are printed out upon completion, and the cavity field is automatically set to the lasing field.
- `find_all_modes(lambda0, lambda1, deltalambda, n_imag0=0.0, n_imag1=0.015, passes=1)`: similar to `find_mode()`, but prints out all laser modes found in this range. Modes that are spaced closer than `deltalambda` will not be resolved.
- `sigma()`: returns the singular value of the cavity at the current wavelength and material gain. (The singular value is minimised when looking for laser modes.)
- `set_source(Coord pos, Coord orientation)`: places a dipole current source at the cavity cut at a position `pos` and with a vectorial orientation given by `orientation`.
- `set_source(forward, backward)`: places a source at the cavity cut described by the expansion vectors `forward` and `backward`.
- `field(Coord)`: returns field at given coordinate. Only meaningful if a laser mode has been located, or if a source has been explicitly set.
- `plot()`: interactively plots the fields in the cavity. Only meaningful ifa laser mode has been located, or if a source has been explicitly set.
- `plot_n()`: lower level routine to plot the refractive index. See `Stack` for more details.
- `plot_field()`: lower level field plot routine. See `Stack` for more details.
- `animate_field()`: lower level field plot routine. See `Stack` for more details.
- `n(Coord)`: returns refractive index at given coordinate.
- `bot_stack()`: returns bottom stack.
- `top_stack()`: returns top stack.
- `length()`: returns the `z`-length of the cavity.
- `width()`: returns the `c1`-length of the cavity.

BlochStack

An infinite periodic repetition of a given Stack, supporting Bloch modes.

Constructor:

- `BlochStack(Expression)`

Member functions

- `N()`: returns number of modes in this waveguide. Is twice the number set by `set_N()`, as it includes both forward and backward Bloch waves.
- `mode(int i)`: returns BlochMode with index `i`. Index starts at zero.
- `calc()`: calculates the BlochModes in this waveguide.
- `plot()`: plots the BlochModes.
- `plot_n()`: lower level routine to plot the refractive index. See `Stack` for more details.

- `length()`: returns the `z`-length of the basic period.
- `width()`: returns the `c1`-length of the basic period.
- `beta_vector()`: returns a vector containing all the propagation constants of the BlochModes.

See [BlochMode], page 65.

BlochMode

Mode inside a BlochStack.

Member functions:

- `kz()`: returns propagation factor.
- `n_eff()`: returns effective index.
- `pol()`: returns polarisation.
- `field(Coord)`: returns field at a given coordinate.
- `fw_field()`: returns expansion coefficients of forward field component of this mode.
- `bw_field()`: returns expansion coefficients of backward field component of this mode.
- `fw_bw(z, limit=Min)`: returns a tuple with two vectors containing the forward and backward field expansions at position `z`. `limit` is optional and can be either `Min` or `Plus`.
- `S_flux(Complex x0, Complex x1, Real eps)`: returns power flux in the `z`-direction at `z=0` between `x0` and `x1`, calculated with relative precision `eps`.
- `plot_field()`: lower level field plot routine. See `Stack` for more details.
- `animate_field()`: lower level field plot routine. See `Stack` for more details.

See [BlochStack], page 64.

InfStack

An infinite periodic repetition of a given period, which can be used to terminate another stack.

Constructor:

- `InfStack(Expression)`

Geometry

Creates an expression from a collection of geometric shapes.

Constructor:

- `Geometry(Mat)`: creates a geometry with a given background material.

Member functions:

- `+= Shape`: add a shape, which can be a `Circle`, `Square`. `Rectangle`, `Triangle` or a `Picture`. In case of overlap, shapes that are added later take precedence. When using Picture, only the last one added will be used.

- `to_expression(prop0,prop1,d_prop,trans0,trans1,d_trans)`: see the explanation in See Section 2.6 [Defining complicated structures], page 34 for a description of these arguments. When working with Picture, see See [Picture], page 67 for more details.

See [Circle], page 66, See [Square], page 66, See [Rectangle], page 66, See [Triangle], page 66, See [Picture], page 67.

`Circle`

A circle shape for use in a `Geometry`.

Constructor:

- `Circle(Point(x,y), rad, Material)`: creates a circle with a given center, radius and material.

See [Geometry], page 65

`Square`

A square shape for use in a `Geometry`.

Constructor:

- `Square(Point(x,y), a, Material)`: creates a square with a given center, side and material.

See [Geometry], page 65

`Rectangle`

A rectangular shape for use in a `Geometry`.

Constructor:

- `Rectangle(Point(x,y), Point(x,y), Material)`: creates a rectangle with a given bottom left and top right point and material.

See [Geometry], page 65

`Triangle`

A triangular shape for use in a `Geometry`.

Constructor:

- `Triangle(Point(x,y), Point(x,y), Point(x,y), Material)`: creates a triangle out of three vertices and a material.

See [Geometry], page 65

Picture

A picture (in png, jpg, ... format) for use in a `Geometry`, to be converted to an expression. It will not be combined with the other shapes in `Geometry`, and only the picture that is added last to the `Geometry` will be used.

Constructor:

- `Picture(picturename, n_min=x, n_max=y)`: creates a picture object from filename `picturename` to be used in a `Geometry`.

The picture will be converted to grayscale, with white being mapped to a material with refractive index `n_min` and black to `n_max`. The indices for grayscales are obtained through linear interpolation.

When using the `Geometry` function

`to_expression(prop0, prop1, d_prop, verbose=True,rescaling='ANTIALIAS')`, the following will happen. The values of `prop0` and `prop1` will set the conversion from pixel units to physical length units. The parameter `d_prop` determines the resolution: when set to 0 (the default), no information is lost in the picture and every pixel is used as is. In case the thickness of the slice `d_prop` is set to such a value that results in a coarser picture than the original one, the following down-sample mechanisms can be used:

- `NEAREST`: 'Nearest' algorithm from the PIL library. As this method takes the nearest point, no new gray colours will be created.
- `ANTIALIAS`: 'Average' algorithm from the PIL library. This will create a grayscale image.
- `INV_AVERAGE`: Similar to `ANTIALIAS`, but use geometric averaging.

See [Geometry], page 65

Planar

A type of waveguide, that can be used e.g. to construct Stacks. `Planar` is an infinitely stratified medium, and is different from other waveguides (like `Slab` and `Circ`) in the sense that all modes (or propagation angles) are decoupled. Therefore, only one propagation angle is considered at a time.

Constructor:

- `Planar(Material)`

Member functions: same as `Waveguide`, in addition to

- `set_theta(Complex)`: set propagation angle in radians in this layer. This automatically fixes the propagation angle in the other Planars because of Snell's Law. Note that you should set the wavelength before setting the angle, as `set_theta` effectively sets the transverse part of the wavevector, which is dependent on the wavelength.

See [Waveguide], page 60.

Circ

A type of waveguide, that can be used e.g. to construct Stacks, BlochStacks and Cavities. `Circ` is an cylindrical waveguide bounded by a perfectly conducting metal wall. Currently, at most on radial refractive index step is supported in the waveguide.

Constructor:

- `Circ(Expression)`

Member functions: same as `Waveguide`.

See [Waveguide], page 60.

Slab

A type of waveguide, that can be used e.g. to construct Stacks, BlochStacks and Cavities. `Slab` is a 1D slab waveguide consisting of an arbitrary number of layers. Lateral boundary conditions default to electric walls, but can be set at will on a slab-by-slab basis or globally.

Constructor:

- `Slab(Expression)`

Member functions: same as `Waveguide` in addition to

- `width()`: returns the width along `x` of the slab.
- `set_lower_wall(SlabWall)`: sets the wall at `x=0` for this slab. Possible choices for SlabWall include `slab_E_wall` and `slab_H_wall`.
- `set_upper_wall(SlabWall)`: sets the wall at `x=slab.width()` for this slab. Possible choices for SlabWall include `slab_E_wall` and `slab_H_wall`.

See [Waveguide], page 60.

Index