# Meep

*MIT Electromagnetic Equation Propogation*

(this is the old and somewhat obsolete LaTeX manual)

—See http://ab-initio.mit.edu/meep/doc for the latest documentation—

David Roundy, Mihai Ibanescu, Peter Bermel,
Steven G. Johnson, and Ardavan Farjadpour

August 21, 2006

# Contents


**1 Discretizing Maxwell's equations** **5**
- 1.0.1 Frequency-dependent epsilon . . . 5
- 1.0.2 Nonlinear dielectrics . . . 6
- 1.0.3 Anisotropic dielectrics . . . 6
- 1.0.4 Putting it all together . . . 6

1.1 The Yee lattice . . . 6
1.2 Maxwell's equations in cylindrical coordinates . . . 7

**2 PML** **9**

**3 Of polaritons and plasmons** **11**

**4 Hints for writing finite difference time domain code** **13**

**5 Tutorial** **15**
5.1 A simple 2D system. . . . 15
5.2 A considerably more complicated 2D example. . . . 17
5.3 Baby's First Bandstructure . . . 21
5.4 Computing the band structure of an omniguide . . . 22
5.5 Band structure of a polariton . . . 24
5.6 Energy conservation in cylindrical coordinates . . . 25
5.7 Energy conservation in one dimension . . . 26
5.8 Epsilon of a polaritonic material in one dimension . . . 28
5.9 Dielectric function of a material with loss and gain . . . 29
5.10 Nonlinear materials . . . 30

**A GNU General Public License** **33**

# Chapter 1

# Discretizing Maxwell's equations

Maxwell's equations in the absence of sources are:

$$\frac{d\mathbf{H}}{dt} = -c\nabla \times \mathbf{E} \tag{1.1}$$

$$\frac{d\mathbf{D}}{dt} = c\nabla \times \mathbf{H} \tag{1.2}$$

If the material is a simple isotropic dielectric, we can simply write $\mathbf{D} = \epsilon\mathbf{E}$ and get on with our lives. Alas, all too often this is not the case! We need to be able to deal with anisotropic dielectrics in which $\epsilon$ is a tensor quantity, nonlinear materials in which $\epsilon$ is a function of $\mathbf{E}$ itself, and polaritonic and polaronic materials in which $\epsilon$ is a function of frequency.

### 1.0.1 Frequency-dependent epsilon

In the case of a frequency-dependent $\epsilon$, we write

$$\mathbf{D} = \epsilon_\infty \mathbf{E} + \mathbf{P} \tag{1.3}$$

where $\mathbf{P}$ is the polarization as a function of time associated with the frequency dependence of $\epsilon$. Actually, in general there will be a set of polarizations, and we'll need a summation here. For simplicity we'll only describe the case of a single polarization in this section. The time dependence of a single polarization is given by

$$\frac{d^2\mathbf{P}}{dt^2} + \gamma\frac{d\mathbf{P}}{dt} + \omega_0^2\mathbf{P} = \Delta\epsilon\ \omega_0^2\mathbf{E} \tag{1.4}$$

where $\gamma$, $\omega_0$ and $\Delta\epsilon$ are material parameters. The energy lost due to the absorption by this resonance is simply

$$\Delta U = \mathbf{P}\frac{d\mathbf{E}}{dt} \tag{1.5}$$

In fact, if one sets $\Delta\epsilon$ to be negative, we can model gain effectively in this way, and in this case keeping track of the energy allows us to model a situation in which there is a depleteable population inversion which is causing the gain—this is the situation of gain with saturation.

### 1.0.2 Nonlinear dielectrics

In nonlinear dielectrics $\mathbf{D}$ is typically given by a cubic function of $\mathbf{E}$.

$$\mathbf{D} = \left(\epsilon + \xi \left|\mathbf{E}\right|^2\right) \mathbf{E} \tag{1.6}$$

### 1.0.3 Anisotropic dielectrics

In anisotropic dielectrics the dielectric constant is a tensor quantity rather than a scalar quantity. In this case we write (FIXME: how to do a tensor in latex?)

$$\mathbf{D} = \bar{\bar{\epsilon}}\mathbf{E} \tag{1.7}$$

$$\mathbf{E} = \bar{\bar{\epsilon}}^{-1}\mathbf{D} \tag{1.8}$$

$$\tag{1.9}$$

### 1.0.4 Putting it all together

Putting it all together, we get a simplified time stepping of something like

$$\frac{d\mathbf{H}}{dt} = -c\nabla \times \mathbf{E} \tag{1.10}$$

$$\frac{d\mathbf{E}}{dt} = \left(\bar{\bar{\epsilon}}_\infty + \mathbf{E} \cdot \bar{\bar{\xi}} \cdot \mathbf{E}\right)^{-1} \left(c\nabla \times \mathbf{H} - \sum_i \frac{d\mathbf{P}_i}{dt}\right) \tag{1.11}$$

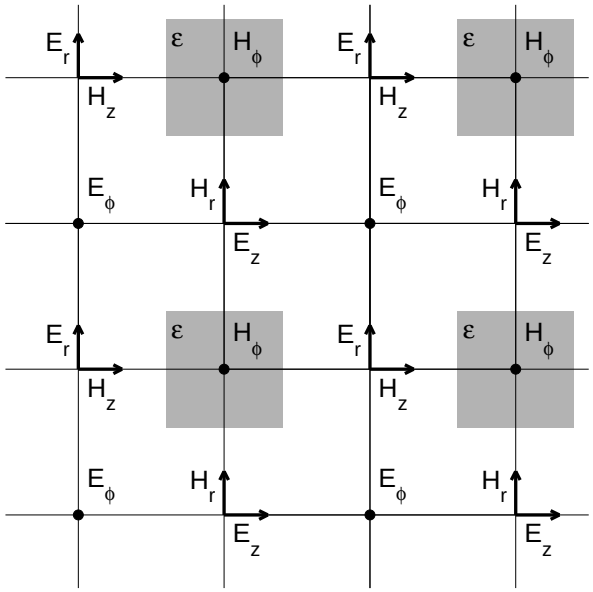
$$\frac{d^2\mathbf{P}_i}{dt^2} + \gamma_i \frac{d\mathbf{P}_i}{dt} + {\omega_0}_i^2 \mathbf{P}_i = \Delta\epsilon_i \, {\omega_0}_i^2 \mathbf{E} \tag{1.12}$$

## 1.1 The Yee lattice

In discretizing Maxwell's equations, we need to put $\mathbf{E}$ and $\mathbf{H}$ on a grid. Because we only need to calculate the curl of these quantities, we only need to know them at limited locations–this gives us the accuracy of a fine grid while only requiring as much data as a grid twice as coarse. This trick is called the Yee lattice. Figure 1.1 shows the Yee lattice in cylindrical coordinates (with $\hat{z}$ being to the right). The gray squares indicate the locations at which $\epsilon$ is stored.

The Yee lattice has the property that all the derivatives needed for $\nabla \times \mathbf{H}$ are known at the Yee lattice points of $\mathbf{E}$. For example, if you look at the $H_\phi$ location. $\frac{dH_\phi}{dt}$ depends on $\frac{dE_r}{dz}$ and $\frac{dE_z}{dr}$. This is great, because $E_r$ is known to the right and left of $H_\phi$, and $E_z$ is known above and below $H_\phi$.

Figure 1.1: Yee lattice in cylindrical coordinates.




The same principle that the Yee lattice does with space, we also do with time. **E** and **H** are known at different times, so that the time derivative of **E** is known at a **H** time and vice versa.

## 1.2 Maxwell's equations in cylindrical coordinates

Here are Maxwell's equations in cylindrical coordinates. We take the fields to be of the form:

$$\mathbf{E}(r, \phi, z) = \mathbf{E}_m(r, z)e^{im\phi}$$

Without further ado:

$$\frac{1}{c}\frac{dH_r}{dt} = \frac{dE_\phi}{dz} - \frac{im}{r}E_z \tag{1.13}$$

$$\frac{1}{c}\frac{dH_\phi}{dt} = \frac{dE_z}{dr} - \frac{dE_r}{dz} \tag{1.14}$$

$$\frac{1}{c}\frac{dH_z}{dt} = \frac{im}{r}E_r - \frac{1}{r}\frac{d(rE_\phi)}{dr} \tag{1.15}$$

$$\frac{\epsilon}{c}\frac{dE_r}{dt} = \frac{im}{r}H_z - \frac{dH_\phi}{dz} \tag{1.16}$$

$$\frac{\epsilon}{c}\frac{dE_\phi}{dt} = \frac{dH_r}{dz} - \frac{dH_z}{dr} \tag{1.17}$$

$$\frac{\epsilon}{c}\frac{dE_z}{dt} = \frac{1}{r}\frac{d(rH_\phi)}{dr} - \frac{im}{r}H_r \tag{1.18}$$

# Chapter 2

# PML

PML (Perfectly Matched Layers) is used to provide absorbing boundary conditions in either the $z$ or $r$ direction. PML consists of a material in which some of the field components are split into two fields, each of which has a conductivity associated with it, which is responsible for the absorption of the PML.

PML is a sort of material that contains a set of conductivities $\sigma_r$, $\sigma_\phi$ and $\sigma_z$. These conductivities are both **E** and **H** conductivities—yes, we have magnetic monopoles moving around in our PML. $\ddot\smile$ Each $\sigma$ causes absorption of radiation in the direction it is named after. Thus $\sigma_\phi$ is small, and almost unnecesary, and is only needed because of the curvature of the radial surface. The value of $\sigma_\phi$ at a given radius is equal to

$$\sigma_\phi(r) = \frac{1}{r}\int_0^r \sigma_r(r)dr \tag{2.1}$$

If we had a IDTD (Infinitesimal Difference Time Domain) code, PML would be perfectly absorbing, regardless of the variation of $\sigma$ with position. However, since meep is a lowly FDTD code, we have to make sure that $\sigma$ varies only slowly from one grid point to the next. We do this by making $\sigma_z$ (for example) vary as $z^2$, with a maximum value of $\sigma_{max}$ right in front of the boundary. At the edge of the PML region is a metalic boundary condition. The optimal value of $\sigma_{max}$ is determined by a tradeoff between reflection off the metallic boundary, caused by too little a $\sigma_{max}$, and reflection off the sigma itself, caused by too large a $\sigma_{max}$, which makes for a large variation of $\sigma$ from one grid point to the next.

Here are the field equations for a PML material:

$$\frac{dH_{r\phi}}{dt} = -c\frac{im}{r}E_z - \sigma_\phi H_{r\phi} \qquad \frac{dH_{rz}}{dt} = c\frac{dE_\phi}{dz} - \sigma_z H_{rz} \tag{2.2}$$

$$\frac{dH_{\phi z}}{dt} = -c\frac{dE_r}{dz} - \sigma_z H_{\phi z} \qquad \frac{dH_{\phi r}}{dt} = c\frac{dE_z}{dr} - \sigma_r H_{\phi r} \tag{2.3}$$

$$\frac{dH_{zr}}{dt} = -c\frac{1}{r}\frac{d(rE_\phi)}{dr} - \sigma_r H_{zr} \qquad \frac{dH_{z\phi}}{dt} = c\frac{im}{r}E_r - \sigma_\phi H_{z\phi} \tag{2.4}$$

$$\epsilon\frac{dE_{r\phi}}{dt} = c\frac{im}{r}H_z - \sigma_\phi E_{r\phi} \qquad \epsilon\frac{dE_{rz}}{dt} = -c\frac{dH_\phi}{dz} - \sigma_z E_{rz} \tag{2.5}$$

$$\epsilon\frac{dE_{\phi z}}{dt} = c\frac{dH_r}{dz} - \sigma_z E_{\phi z} \qquad \epsilon\frac{dE_{\phi r}}{dt} = -c\frac{dH_z}{dr} - \sigma_r E_{\phi r} \tag{2.6}$$

$$\epsilon\frac{dE_{zr}}{dt} = c\frac{1}{r}\frac{d(rH_\phi)}{dr} - \sigma_r E_{zr} \qquad \epsilon\frac{dE_{z\phi}}{dt} = -c\frac{im}{r}H_r - \sigma_\phi E_{z\phi} \tag{2.7}$$

# Chapter 3

# Of polaritons and plasmons

Most real materials, at least in some frequency range, have polarizations that are not actually instantaneously proportional to the local electric field. We model these polaritonic and plasmonic effects by introducing one or more additional polarization fields, to be propogated along with the electric and magnetic field. The polarization field, $\mathbf{P}$, is a vector field which exists on the electric field Yee lattice points.

The polarization field obeys a second order differential equation, which means that we need to keep track of the polarization at two time steps, in order to integrate it.

$$\frac{d^2\mathbf{P}}{dt^2} + \gamma\frac{d\mathbf{P}}{dt} + \omega^2\mathbf{P} = \Delta\epsilon\ \omega^2\mathbf{E} \tag{3.1}$$

To this, we need add one more term to maxwell's equation for $\mathbf{E}$:

$$c\nabla \times \mathbf{H} = \epsilon_\infty\frac{d\mathbf{E}}{dt} + \frac{d\mathbf{P}}{dt} \tag{3.2}$$

So far, the polarization is beautifully simple. However, we would love to be able to put polaritonic materials into our PML regions, and unfortunately in the PML region the electric field has been split into two components, so we need to figure out which of the two components gets the contribution from $\frac{d\mathbf{P}}{dt}$. The obvious solution to this (well, maybe not exactly obvious, but it is the solution) is to split the polarization field also into two components in the PML region, just as we split the electric and magnetic fields.

The electric field propogation equations in PML then become:

$$\epsilon\frac{dE_{r\phi}}{dt} = c\frac{im}{r}H_z - \sigma_\phi E_{r\phi} - \frac{dP_{r\phi}}{dt} \tag{3.3}$$

$$\epsilon\frac{dE_{\phi z}}{dt} = c\frac{dH_r}{dz} - \sigma_z E_{\phi z} - \frac{dP_{\phi z}}{dt} \tag{3.4}$$

$$\epsilon\frac{dE_{zr}}{dt} = c\frac{1}{r}\frac{d(rH_\phi)}{dr} - \sigma_r E_{zr} - \frac{dP_{zr}}{dt} \tag{3.5}$$

$$\epsilon \frac{dE_{rz}}{dt} = -c\frac{dH_\phi}{dz} - \sigma_z E_{rz} - \frac{dP_{rz}}{dt} \tag{3.6}$$

$$\epsilon \frac{dE_{\phi r}}{dt} = -c\frac{dH_z}{dr} - \sigma_r E_{\phi r} - \frac{dP_{\phi r}}{dt} \tag{3.7}$$

$$\epsilon \frac{dE_{z\phi}}{dt} = -c\frac{im}{r}H_r - \sigma_\phi E_{z\phi} - \frac{dP_{z\phi}}{dt} \tag{3.8}$$

# Chapter 4

# Hints for writing finite difference time domain code

(Or *Things I forgot many times, so I wrote down so maybe I won't make the same mistake again.*)

There is just one rule to remember when writing time domain code, and that is (as Lefteris has repeatedly told me) "Always know *when* each equation is evaluated." The trick, of course, lies in knowing how to apply this rule, and remembering to actually apply it (and I think the latter is perhaps harder than the former).

As an example, I'll convert a PML polariton equation into a finite difference equation taken from equation 3.8 of chapter 3.

$$\epsilon \frac{dE_{z\phi}}{dt} = -c\frac{im}{r}H_r - \sigma_\phi E_{z\phi} - \frac{dP_{z\phi}}{dt}$$

If we consider the **E** timesteps to be at times $n$, $n+1$ etc., then this equation needs to be evaluated at time $n+\frac{1}{2}$. This is no problem for most of the terms, but it means that the $\sigma_\phi E_{z\phi}$ term needs to be an average of its values at time $n$ and $n+1$. In short (taking $\Delta t$ to be unity)...

$$\epsilon(E_{z\phi}^{n+1} - E_{z\phi}^{n}) = -c\frac{im}{r}H_r^{n+\frac{1}{2}} - \sigma_\phi(E_{z\phi}^{n+1} + E_{z\phi}^{n}) - (dP_{z\phi}^{n+1} - dP_{z\phi}^{n})$$

Simplifying a tad gives

$$E_{z\phi}^{n+1} - E_{z\phi}^{n} = \frac{1}{\epsilon + \frac{1}{2}\sigma_\phi}\left(-c\frac{im}{r}H_r^{n+\frac{1}{2}} - \sigma_\phi E_{z\phi}^{n} - (dP_{z\phi}^{n+1} - dP_{z\phi}^{n})\right)$$

Basically, that is all there is to it. You now have the equation to determine $E_{z\phi}^{n+1}$ from $E_{z\phi}^{n}$, $\frac{im}{r}H_r^{n+\frac{1}{2}}$, $dP_{z\phi}^{n+1}$ and $dP_{z\phi}^{n}$.

# Chapter 5

# Tutorial

## 5.1 A simple 2D system.

This example is intended to let you quickly get started using meep to run a simple calculation. As such, it will include within it the complete code of the example itself. Meep is a C++ library, so your control file is a small C++ program.

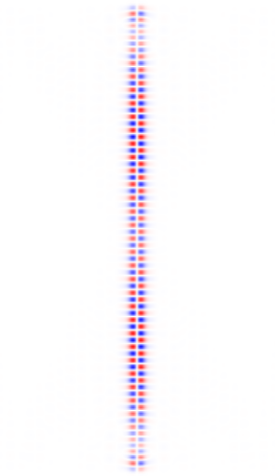
At the beginning of your control file, you have to include the "meep.h" header and use the "meep" namespace...

```
#include <meep.hpp>
using namespace meep;
```

Next we create a function to define epsilon. This function accepts a "vec" argument, and returns a double, which is the value of epsilon. For this example, we use an index-guided waveguide with some air slits cut in it. You can choose whatever units you like in which to define your structure. In this case we choose the width of the waveguide as our unit, which is also equal to 1 micron.

```
const double half_cavity_width = 0.5*0.68, air_slit_width = 0.38,
             grating_periodicity = 0.48,
             half_waveguide_width = 1.0,
             num_air_slits = 15.0,
             high_dielectric = 12.0, low_dielectric = 11.5;
const double pml_thickness = 1.0;
const double x_center = 7.7 + pml_thickness;
const double y_center = 10.5 + pml_thickness;
double eps(const vec &rr) {
  // Displacement from center of cavity is r:
  const vec r = rr - vec(x_center, y_center);
  // First the air slits:
  double dx = fabs(r.x()) - half_cavity_width;
  if (dx < num_air_slits*grating_periodicity && dx > 0.0) {
```

Figure 5.1: $E_z$



```
    while (dx > grating_periodicity) dx -= grating_periodicity;
    if (dx < air_slit_width) return 1.0;
  }
  // Now check if the y value is within the waveguide:
  if (fabs(r.y()) < half_waveguide_width) return high_dielectric;
  // Otherwise we must be in the surrounding low dielectric:
  return low_dielectric;
}
```

The main function should always start by creating an initialize object. This object is responsible for setting up MPI if we are running on multiple processors, and cleaning up properly when it is deleted (which means we are done).

```
int main(int argc, char *argv[]) {
  initialize mpi(argc, argv);
```

The "s" structure defines the contents of the unit cell. It needs a volume, which includes the size of the grid and the resolution, as well as the epsilon function we defined earlier. Here we also choose to use PML absorbing boundary conditions in all directions, since we are interested in the high Q mode in the cavity.

```
  const double amicron = 10; // a micron is this many grid points.
  const volume vol = voltwo(2*x_center, 2*y_center, amicron);
  const symmetry S = mirror(Y, vol) + rotate2(Y, vol);
  structure s(vol, eps, pml(pml_thickness), S);
```

To avoid clutter, we'll create a directory to hold the output. The function `make_output_directory` creates a directory based on the name of the example

program along with an extension. It also backs up the C++ source file if it can find it. If the directory already exists, then it reuses it, unless the C++ control file has changed, in which case it creates a new one.

```
  const char *dirname = make_output_directory(__FILE__);
  s.set_output_directory(dirname);
```

The structure only holds the epsilon. We will also need a "fields" object to hold our electric and magnetic fields. We add a point source oriented in the $E_z$ direction, located in the center of our cavity.

```
  fields f(&s);
  const double wavelength = 1.72;
  const double freq = 1.0/wavelength;
  f.add_point_source(Hy, freq, 10.0, 0.0, 5.0, vec(x_center,y_center));
```

I'm not interested in seeing the source itself, so I'll keep time stepping until the current time is greater than the last time at which the source is running.

```
  while (f.time() < f.last_source_time()) f.step();
```

Now we'll wait a bit (to let the low-Q modes die off) and then take a snapshot of the fields in HDF5 format.

```
  while (f.time() < 200.0) f.step();
  f.output_hdf5(Hx, f.total_volume());
  f.output_hdf5(Hy, f.total_volume());
  f.output_hdf5(Ez, f.total_volume());
}
```

And now we're done, although you might wonder if we've done anything worthwhile, since all we got out of this was a picture...

## 5.2 A considerably more complicated 2D example.

This example demonstrates a lot more of what you can do using meep. The system is the same as in the previous example, but this time we will calculate the quality factor of the cavity. Again, the entire control file will be included here, but I'll skip over sections that have already been explained.

```
#include <meep.hpp>
using namespace meep;

const double half_cavity_width = 0.5*0.68, air_slit_width = 0.38,
             grating_periodicity = 0.48,
             half_waveguide_width = 1.0,
             num_air_slits = 15.0,
```

```
            high_dielectric = 12.0, low_dielectric = 11.5;
const double pml_thickness = 1.0;
const double x_center = 7.7 + pml_thickness;
const double y_center = 10.5 + pml_thickness;
double eps(const vec &rr) {
  // Displacement from center of cavity is r:
  const vec r = rr - vec(x_center, y_center);
  // First the air slits:
  double dx = fabs(r.x()) - half_cavity_width;
  if (dx < num_air_slits*grating_periodicity && dx > 0.0) {
    while (dx > grating_periodicity) dx -= grating_periodicity;
    if (dx < air_slit_width) return 1.0;
  }
  // Now check if the y value is within the waveguide:
  if (fabs(r.y()) < half_waveguide_width) return high_dielectric;
  // Otherwise we must be in the surrounding low dielectric:
  return low_dielectric;
}
```

This time we use the `deal_with_ctrl_c();` function. This is a handy utility function that is useful when running your meep code interactively. It traps the SIGINT signal, so when you hit cntl-C, rather than simply exiting, the global variable `interrupt` is incremented. If you really want to exit, just hit cntl-C again, and when `interrupt` reaches 2, the program will exit.

```
int main(int argc, char *argv[]) {
  initialize mpi(argc, argv);
  deal_with_ctrl_c();
  const double amicron = 10; // a micron is this many grid points.
  const volume vol = voltwo(2*x_center, 2*y_center, amicron);
  const symmetry S = mirror(Y, vol) + rotate2(Y, vol);
  structure s(vol, eps, pml(pml_thickness), S);
  const char *dirname = make_output_directory(__FILE__);
  s.set_output_directory(dirname);
  fields f(&s);
  const double wavelength = 1.72;
  const double freq = 1.0/wavelength;
  f.add_point_source(Hy, freq, 5.0, 0.0, 5.0, vec(x_center,y_center));
```

We add an additional check below "`&&` interrupt!" so that when the user hits cntl-C we exit the loop.

```
  while (f.time() < f.last_source_time() && !interrupt) f.step();
  f.output_hdf5(Ez, f.total_volume());
  while (f.time() < 400.0 && !interrupt) f.step();
```

This time we're going to run the simulation longer, so we would like to get occasional informative messages. To do this we define a variable to hold the next time we want to print a message.

```
double next_print_time = 500.0;
```

To calculate the $Q$ of our cavity, we use a monitor point `p`. We also store the value of $H_y$ at our monitor point in a file named "hy" periodically. We create this file using `create_output_file`, which creates and opens a file for writing in the given output directory. This utility function works properly whether we are running in parallel or not.

```
monitor_point *p = NULL;
FILE *myout = create_output_file(dirname, "hy");
while (f.time() <= 2000.0 && !interrupt) {
  // Now we'll start taking data!
  f.step();
```

To get the monitor point data we use the `get_new_point` method of `fields`. This ends up creating a linked list containing the values of the field at the monitor point as a function of time, which we will later use to run harminv and get the $Q$.

```
  p = f.get_new_point(vec(x_center,y_center), p);
```

We use the `get_component` method to extract the (complex) fields from the monitor point so we can print them. We use the `master_fprintf` function because we only want to get one copy of the information even if we're running in parallel.

```
  master_fprintf(myout, "%g\t%g\t%g\n", f.time(),
                 real(p->get_component(Hy)),
                 imag(p->get_component(Hy)));
```

Every time we reach the `next_print_time` we print out a copy of the $E_z$ fields, along with a little message indicating the time and the total energy (which should be decaying at this point). The function `master_printf` is a utility function that works basically like `printf`, except that when running in parallel only one of the processors (the "master") does the printing. You should use this function rather than something like "`if (my_rank()==0) printf(...)`", since the latter can cause problems if the arguments to printf require synchronization between the processes.

```
  if (f.time() >= next_print_time) {
    f.output_hdf5(Ez, f.total_volume());
    master_printf("Energy is %g at time %g\n",
                  f.total_energy(), f.time());
    next_print_time += 500.0;
  }
}
```

Files which are opened with `create_output_file` need to be closed with `master_fclose`, which does the Right Thing when running in parallel.

Figure 5.2: Contents of "freqs" file

```
0.651297 7.26245e-07 -896801 0.00821062 -0.000426369
```

```
  master_fclose(myout);
```

Having collected all the monitor point data, we now want to run harminv[1] on it to find the $Q$ of our resonant cavity. The `harminv` method gives us the complex amplitudes, the frequencies and the decay rates. The decay rate is given in the same units as the frequency, so you could choose to view it as the imaginary part of the frequency if you like.

This harminv step is the real reason for using the ctrl-C trick, since if while running this example we get impatient and decide we have enough data we can just hit ctrl-C and get the results using what data we have. This means you can just set the code to run for an excessively long time without risking losing everything if you lose patience.

In case you're wondering about the "`\begin{verbatim}`", it's there so I can easily include the output in this manual (see Figure 5.2).

```
  complex<double> *amp, *freqs;
  int num;

  FILE *myfreqs = create_output_file(dirname, "freqs");
  master_fprintf(myfreqs, "\\begin{verbatim}\n");
  master_printf("Harminving Hy...\n");
  interrupt = 0; // Harminv even if we were interrupted.
  p->harminv(Hy, &amp, &freqs, &num, 0.8*freq, 1.2*freq, 5);
  for (int i=0;i<num;i++) {
    master_fprintf(myfreqs, "%g\t%g\t%g\t%g\t%g\n",
                   real(freqs[i]), imag(freqs[i]),
                   -real(freqs[i])/imag(freqs[i]),
                   real(amp[i]), imag(amp[i]));
  }
  master_fprintf(myfreqs, "%cend{verbatim}\n", '\\');
  master_fclose(myfreqs);
  delete[] dirname;
}
```

[1] If you don't know what harminv is, I'm not going to explain it here, so you may as well ask me in person (or even better, ask Steven...

## 5.3 Baby's First Bandstructure

In this example we calculate the lowest four TE modes of a simple hollow metallic waveguide of radius one.

```
int main(int argc, char *argv[]) {
  initialize mpi(argc, argv);
  FILE *ban = master_fopen("bands", "w");
  structure s(volcyl(1.0, 0.0, rad), eps);
  for (int m=0;m<3;m++) {
    for (double k=0.0; k<= 1.01; k += 0.25) {
      master_printf("Working on k of %g and m = %d with a=%d...\n",
                    k, m, rad);
      fields f(&s, m);
      f.use_bloch(k);
```

There are a few tricks you should know before you decide to go about calculating a band structure. One of the biggest problems in calculating a band structure in a time domain code is that of exciting all the modes you are interested in. Meep makes this easy with a couple of "fields" methods, `initialize_with_n_te`, and `initialize_with_n_tm`. These initialize the field with the n lowest TE and TM modes respectively.

```
      f.initialize_with_n_te(4);
```

The band structure code itself begins with a call to `prepare_for_bands`, which allocates space to store the field data, which is later used for the band structure calculation. Its third argument is the maximum frequency you are interested in.

```
      double fmax = 1.0, qmin = 200;
      f.prepare_for_bands(0, ttot, fmax, qmin);
      for (int t=0;t<ttot;t++) {
```

The second band structure function is `record_bands`, which just copies the fields into the already allocated arrays for future use.

```
        f.record_bands();
        f.step();
      }
```

Finally, the band structure is actually computed and output by the method `output_bands`. The key thing to know about `output_bands` is that its last argument should be something like twice the number of modes which have a frequency below your maximum. Rounding this number up slows the code down considerably, but can sometimes fix problems where harminv (which is used internally) doesn't find the modes correctly. Usually, however, when harminv fails it means you are misunderstanding something (for example, fmax may be less than the lowest frequency mode).

```
      f.output_bands(ban, "band", 35);
    }
  }
  master_fclose(ban);
}
```

## 5.4 Computing the band structure of an omniguide

In this section we give as an example of a more complicated band structure, a computation of the band structure of an omniguide. The output of this program is shown in Figure 5.4.

```
const int num_layers = 3;
const double rcore = 3.0;

double guided_eps(const vec &v) {
  double rr = v.r() - rcore;
  if (rr > num_layers + 0.3) return 1.0; // outside the entire waveguide
  if (rr < 0.0) return 1.0;   // vacuum in the core
  while (rr > 1.0) rr -= 1.0; // calculate (r - rcore) % 1
  if (rr < 0.3) return 21.16; // in the high dielectric
  return 1.6*1.6;             // in the low dielectric
}
double vacuum_eps(const vec &v) { return 1.0; }
```

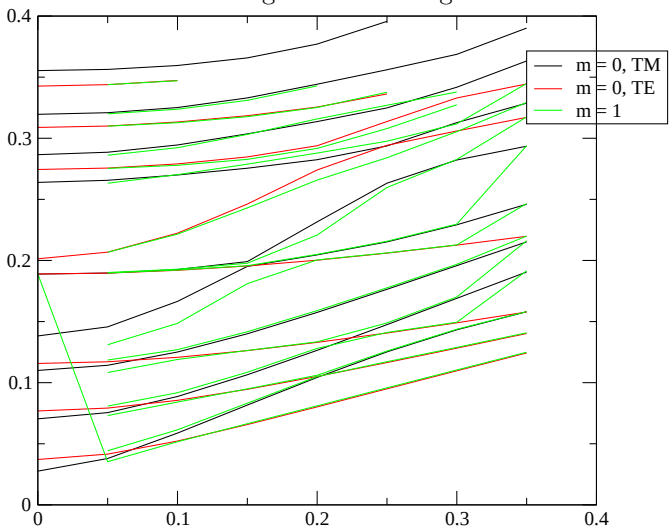
For this band structure example, we use the grace object to create our plot.

```
  grace g("bands", dirname);
  g.set_range(0.0, 0.35, 0.0, 0.35);
```

Since the m = 0 modes are pure TE or TM, it makes sense to calculate the two polarizations separately. Not only does this give us more interesting output, but it doesn't cost us any time, to speak of, and actually makes the band structure much easier to converge. However, for brevity, I won't include here in the manual computation of the TM modes, but will skip straight to the TE modes.

```
  for (int m=0;m<2 && !interrupt;m++) {
    g.new_set();
    char m_string[30];
    if (m) snprintf(m_string, 30, "m = %d", m);
    else snprintf(m_string, 30, "m = 0, TE");
    g.set_legend(m_string);
    for (double k=0.0;k<0.351 && !interrupt;k+=0.05) {
```

Figure 5.3: Omniguide band structure.




In order to populate the modes that we are interested in, we first populate the modes of an empty waveguide (whose modes are known), and then adiabatically transform from that waveguide into our omniguide structure.

```
printf("Working on k of %g and %s with a=%d...\n", k, m_string, a);
fields f(&vac, m);
f.use_bloch(k);
f.verbose(1);
f.phase_in_material(&s, 1000);
```

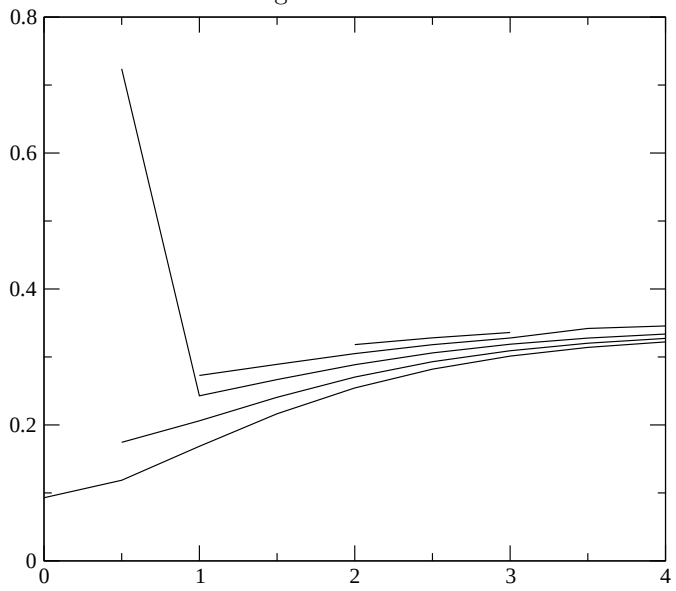
We initialize the fields with both TE and TM modes, and then phase in the epsilon as usual, and then do the actual phasing in of the structure.

```
f.initialize_with_n_te(9);
if (m) f.initialize_with_n_tm(9);
while (f.is_phasing() && !interrupt) f.step();
```

Again, the band structure code is pretty normal, with the only real difference being that in this case we *really* want to have specify a large $Q_{min}$, to help meep to distinguish between real modes and spurious noise. Note that we are using metallic boundary conditions, so all physical modes should have infinite lifetime.

```
f.prepare_for_bands(veccyl(4.801,0.0), ttot, .35, 300, 0.0);
f.prepare_for_bands(veccyl(1.801,0.0), ttot, .35, 300, 0.0);
f.prepare_for_bands(veccyl(2.801,0.0), ttot, .35, 300, 0.0);
const double stoptime = f.time() + ttot;
while (f.time() < stoptime && !interrupt) {
```

Figure 5.4: Polariton band structure.




```
    f.record_bands();
    f.step();
  }
```

Finally, we just need to compute and output the bands. We are careful here to keep in mind that when $m > 0$, there are twice as many bands, since there are both TM and TE modes.

```
  f.grace_bands(&g, m?80:40);
}
```

The band is actually printed to disk only when the grace object is destroyed, which in this case happens just before the program exits.

## 5.5 Band structure of a polariton

Here we compute and plot the band structure of a polariton material. We look at a simple metallic waveguide filled with a polaritonic material. The material we look at has an epsilon of 13.4 and a longitudinal phonon frequency of 0.7 and a transverse phonon frequency of 0.4.

```
double eps(const vec &) { return 13.4; }
double one(const vec &) { return 1; }
```

To create the polaritonic material, we add the polarizability to the material after we have created it.

```
double freq = 0.4, gamma = 0.01, delta_eps = 27.63;
s.add_polarizability(one, freq, gamma, delta_eps);

for (k=0.0;k<4.01 && !interrupt;k+=.5) {
  master_printf("Working on k of %g and m = %d...\n", k, m);
  fields f(&s, m);
  f.use_bloch(k);
```

Now we excite the first TE mode (we are only looking at m = 0 here), and remember to excite along with it the phonon with which it couples.

```
  f.initialize_with_nth_te(1);
  f.initialize_polarizations();
```

Finally, we compute the band structure as usual.

```
  f.prepare_for_bands(veccyl(0.501,0.0), ttot, .7+.15*k/3.0, 50, 1e-4);
  f.prepare_for_bands(veccyl(0.301,0.0), ttot, .7+.15*k/3.0, 50, 1e-4);

  while (f.time() < ttot && !interrupt) {
    f.record_bands();
    f.step();
  }
  f.grace_bands(&g, 16);
```

The final output of this routine (as calculated using the "plot" program) is shown in Figure 5.5.

## 5.6 Energy conservation in cylindrical coordinates

In this example, we compute the total energy over time for a polaritonic material in cylindrical coordinates. Eventually I figure I may extend this example to demonstrate energy/flux conservation using PML. That would definitely be more impressive.

For our example polaritonic material, we'll use an $\epsilon(0)$ of 13.4. We will put the polaritons in just one quarter of our system to add a little extra excitement.

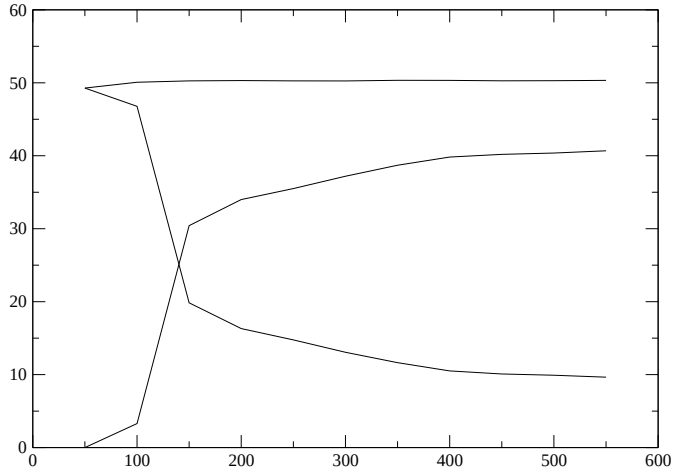
```
double eps(const vec &) { return 13.4; }
double one(const vec &p) { return (p.z() > 15.0)?1:0; }
```

We use a long and skinny system so as to exaggerate any errors that may crop up at small $r$.

```
structure s(volcyl(1.0,20.0, a), eps);
```

We use several point sources, to cover a broad frequency range, just for the heck of it.

Figure 5.5: Energy vs. Time.




```
f.add_point_source(Ep, 0.6 , 1.8, 0.0, 8.0, veccyl(0.5,2.0));
f.add_point_source(Ep, 0.4 , 1.8, 0.0, 8.0, veccyl(0.5,2.0));
f.add_point_source(Ep, 0.33, 1.8, 0.0, 8.0, veccyl(0.5,2.0));
```

We plot the total energy, the electromagnetic energy and the "thermodynamic energy" which is the energy that is either stored in the polarization, or has been converted into heat, or (if we had a saturating gain system) perhaps is stored in a population inversion.

```
g.output_out_of_order(0, f.time(), f.total_energy());
g.output_out_of_order(1, f.time(), f.field_energy_in_box(f.v.surroundings()));
g.output_out_of_order(2, f.time(), f.thermo_energy_in_box(f.v.surroundings()));
```

## 5.7 Energy conservation in one dimension

In this example, we compute the total energy over time for a polaritonic material in one dimension to verify that it is indeed conserved. This also demonstrates how to use a 1D system.

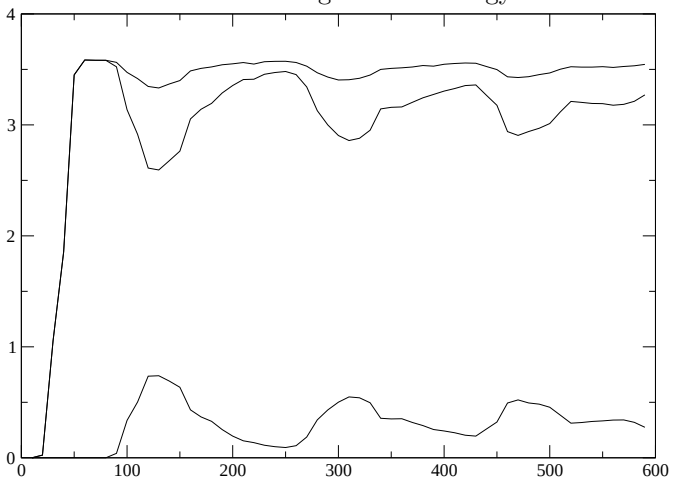
For our example polaritonic material, we'll use an $\epsilon(0)$ of 13.4. We will put the polaritons in just one quarter of our system to add a little extra excitement.

```
double eps(const vec &) { return 13.4; }
double one(const vec &p) { return (p.z() > 15.0)?1:0; }
```

We create a 1D system by making the volume with the "`volone`" function, and making sure any vecs we use are one dimensional.

```
structure s(volone(20.0, a), eps);
```

Figure 5.6: Energy vs. Time.




The polarizability is added as usual... in this case we use a very sharp resonance, which means that our energy will only be very slowly absorbed.

```
s.add_polarizability(one, 0.25, 0.0001, 3.0);
fields f(&s);
grace g("energy", dirname);
```

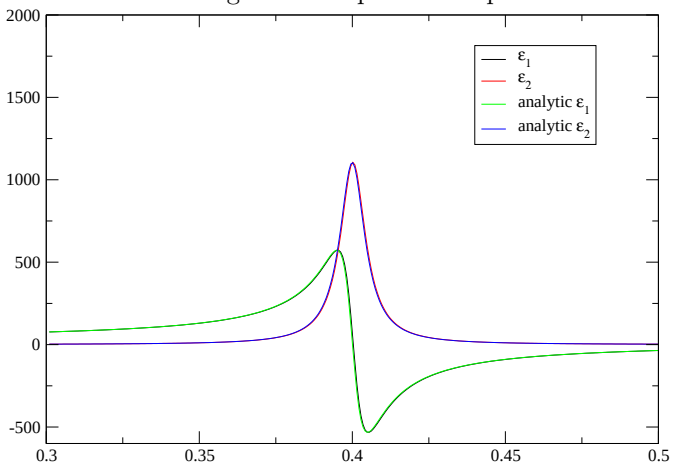
We use several point sources, to cover a broad frequency range, just for the heck of it.

```
f.add_point_source(Ex, 0.6 , 1.8, 0.0, 8.0, vec(2.0));
f.add_point_source(Ex, 0.4 , 1.8, 0.0, 8.0, vec(2.0));
f.add_point_source(Ex, 0.33, 1.8, 0.0, 8.0, vec(2.0));
```

We plot the total energy, the electromagnetic energy and the "thermodynamic energy" which is the energy that is either stored in the polarization, or has been converted into heat, or (if we had a saturating gain system) perhaps is stored in a population inversion.

```
    g.output_out_of_order(0, f.time(), f.total_energy() - ezero);
    g.output_out_of_order(1, f.time(),
                          f.electric_energy_in_box(f.v.surroundings())
                        + f.magnetic_energy_in_box(f.v.surroundings()));
    g.output_out_of_order(2, f.time(),
                          f.thermo_energy_in_box(f.v.surroundings()) - ezero);
```

Figure 5.7: Epsilon of a polaritonic material.




## 5.8 Epsilon of a polaritonic material in one dimension

In this example, we compute epsilon as a function of frequency for a simple polaritonic material. This example is done in one dimension for speed purposes.

One thing to be aware of when using polaritonic materials, is that generally you will be needing a rather higher grid resolution than you may be used to in order to properly model the material. Here I am using an $a$ of 40.

Altough in this calculation the polaritonic material will not be within the PML, it is all right to have polaritonic material within PML regions.

```
s.add_polarizability(one, 0.4, 0.01, 27.63);
```

We use a single rather high frequency (and very broad) point source, to cover a broad frequency range.

```
f.add_point_source(Ex, 0.9, 0.8, 0.0, 8.0, vec(sourceloc));
```

We use a couple of monitor points to determine epsilon.

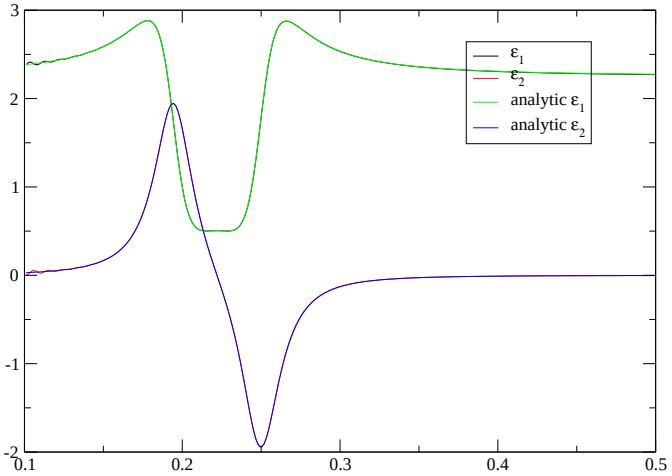
```
monitor_point *left = NULL, *right = NULL, *middle = NULL;
```

The monitor points are located one grid spacing from one another. The `get_new_point` method appends the fields at a given time to a monitor point linked list.

```
left  = f.get_new_point(vec(sourceloc+1.0/a), left );
middle = f.get_new_point(vec(sourceloc+2.0/a), middle);
right = f.get_new_point(vec(sourceloc+3.0/a), right);
```

When the time stepping is over, we take a fourier transform of the fields at the two monitor points.

Figure 5.8: Dielectric function of a material with loss and gain.




```
left->fourier_transform(Ex, &al, &freqs, &numl, 0.301, 0.5, 300);
```

Finally we calculate epsilon from the second derivative of the field using

$$-k^2 H_z(\omega) = \nabla^2 H_z(\omega) = \nabla^2$$

```
complex<double> *epsilon = new complex<double>[numl];
for (int i=0;i<numl;i++) {
  complex<double> ksqr = -(ar[i]+al[i]-2.0*am[i])*a*a/am[i];
  epsilon[i] = ksqr/freqs[i]/freqs[i]/(2*pi*2*pi);
}
for (int i=0;i<numl;i++)
  g.output_point(real(freqs[i]), real(epsilon[i]));
g.new_set();
g.set_legend("\\x\\e\\s2\\N");
for (int i=0;i<numl;i++)
  g.output_point(real(freqs[i]), imag(epsilon[i]));
```

## 5.9 Dielectric function of a material with loss and gain

In this section, we demonstrate a better way to calculate the dielectric function, and illustrate it by computing the dielectric function of a material with a normal lossy resonance as well as a gain line. Gain in the PML is a bad idea, so we restrict the polarizabilities to exist only in the middle of the system (which is surrounded by PML).

```
s.add_polarizability(one_in_middle, 0.195, 0.03, 0.25*.25/.195);
s.add_polarizability(one_in_middle, 0.25, 0.03,-0.25);
```

For sources, we use a series of broad point sources covering the frequency range of interest.

```
f.add_point_source(Ep, 0.3,  0.8, 0.0, 8.0, veccyl(rmax*0.5,sourceloc));
```

The calculation proceeds as usual, except that we now keep track of a set of monitor points that will allow us to take a laplacian of the $H_z$ field component. We choose $m = 0$, so we won't have to deal with the $\phi$ derivative.

```
  left   = f.get_new_point(veccyl(r_look    , z_look - d), left);
  middle = f.get_new_point(veccyl(r_look    , z_look    ), middle);
  top    = f.get_new_point(veccyl(r_look + d, z_look    ), top);
  bottom = f.get_new_point(veccyl(r_look - d, z_look    ), bottom);
  right  = f.get_new_point(veccyl(r_look    , z_look + d), right);
```

We fourier transform $H_z$ at each point:

```
left->fourier_transform(Hz, &al, &freqs, &numl, minfreq, maxfreq, numfreqs);
```

Finally, we calculate the laplacian of $H_z(\omega)$, which is equal to $-k^2 H_z(\omega)$, from which we extract epsilon.

```
for (int i=0;i<numl;i++) {
  complex<double> ksqr = -(ar[i]+al[i]-2.0*am[i]
                           + at[i]*0.5*(1+(r_look+d)/r_look) +
                             ab[i]*0.5*(1+(r_look-d)/r_look) - 2.0*am[i]
                           - m*m*1.0*am[i]/r_look/r_look*a*a
                           )*a*a/am[i];
  epsilon[i] = ksqr/freqs[i]/freqs[i]/(2*pi*2*pi);
}
```

I've left out the bulk of this example from the manual itself, since it is pretty much the same as the previous examples. Among other features, we use the grace functions to plot the result, which can be seen in Figure 5.9.

## 5.10 Nonlinear materials

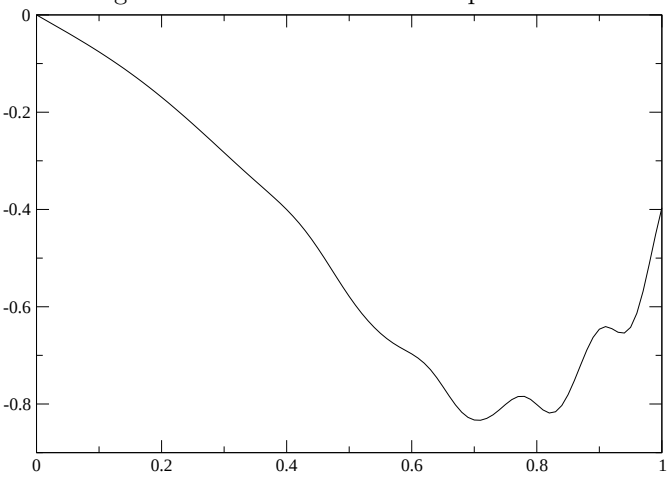
FIXME: Add a nice discussion of nonlinear materials here...

In this example, we will use a CW source and compute the amplitude of the field at a given position and time as a function of the source amplitude. The result will be linear as long as the material remains in the linear regime. Once we have departed from the linear regime, we get more complicated behavior. Yes, this is a stupid example...

The system is shown in Figure 5.10. It is a 2D metallic waveguide with vacuum $\epsilon$ of 2.25. In the center of the cell is a small region that is linear which contains the source, and the rest of the waveguide contains a nonlinear material. Both ends of the waveguide have PML absorbing boundary conditions.

Figure 5.9: Field vs. source amplitude with nonlinear material

Figure 5.10: Field vs. source amplitude with nonlinear material




```
const double alpha_value = 0.07;
double alpha(const vec &v) {
  if (fabs(v.x() - xmax/2.0) < .51) return 0.0;
  return alpha_value;
}
```

The set_chi3 method is used to set the chi3 coefficient.

```
  s.set_chi3(alpha);
```

We use a CW source at a frequency of 0.4, which gives single-mode behavior when the amplitude is small. Also note that we use real fields, since complex fields are incorrect for nonlinear materials.

```
  continuous_src_time my_source(0.4,  0.8);
  for (double amp = 0.05; amp <= 1.01 && !interrupt; amp += 0.01) {
    fields f(&s, m);
    f.use_real_fields();
    f.add_point_source(Ez, my_source, vec(xmax*0.5,ymax*0.5), amp);
```

Time stepping, etc, is done as usual. We monitor the field at one end of the cell, which gives Figure 5.10, which show the field versus source amplitude.

# Appendix A

# GNU General Public License

## Version 2, June 1991



## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

## GNU GENERAL PUBLIC LICENSE

## TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this

License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law

if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

## NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER

PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## END OF TERMS AND CONDITIONS