newsim-tutorial

August 26, 2019

1 FIRST SIMULATION

This tutorial covers:

- The basic usage of NANOCPP package for 2D simulations
- How to appropriately add a geometric shape
- How to calculate a Far-Field

We begin by importing all dependencies to analyze the output from NANOCPP.

```
In [128]: import numpy as np
          import matplotlib.pyplot as plt
          import os, sys
```

We start with the simplest situation of a 2D dipole source propagating in the free space.
In base directory there are several files:

```
In [67]: !ls
```

```
array_0.bin                    figs                 line_parser.o           nano
array_1.bin                    geometry.txt  line_parser_s.o   p_rand.o
blas_LINUX.a                    input.txt         makefile           README.md
build                            lattices.o         makefile_shaheen  res
build_input.o                    liblapack.a         myfile.cc          Untitled.ipynb
definitions.h                    liblapacke.a  myfile.cc~
dispersion_tutorial.ipynb  libtmglib.a         myfile.o
```

For this simulation, we need to edit the file input.txt, which contains all the high level options for the Nanocpp code. Open your favorite editor (we are linux friendly and assume the use of emacs for this tutorial) and edit input.txt.

As you see, the file is divided into two main sections:

COMPILE-TIME PARAMETERS

and

RUN-TIME PARAMETERS

the former contains all the parameters that need to be known at compile time (e.g., optimizations, dimensions, source types, ...), while the latter has all the data for running a Nanocpp simulation (geometry, number of cells, ...). One of the strength of Nanocpp, which allows it to scale over hundreds of thousands of processors, lies in its templated+macro structure. Nanocpp, in fact, self-generates the C++ code from the parameters of the compile-time section and produces an executable that is always optimized and free from any if statement.

We begin by selecting the dimensionality of the problem:

```
##### FDTD COMPILE DATA
N=2 # dimensionality of the problem (2,3)
```

and the source type

```
##### SOURCES
spatial=point+Ex
```

Nanocpp embodies a high level C++ parser class that allows to combine all the various primitives with a high-level syntax. The data will then be interpreted by the parser and traduced into low-level optimized C++ instructions. In this case, to specify a point source we use "point", which stands for point source, followed by the fields we want to activate, in this case only Ex. The order of this instructions is irrelevant, and the instructions are case insensitive. This give the maximum degree of flexibility.

We complete the compile-time section by selecting the measurements that we want to enable in the code. In this case, we choose just a point probe:

```
##### MEASUREMENT
measurements=probes+Ex  # probes (Ei and Hi) + farfield (only in 3D) + energy + poynting
```

We now type

```
$ make
```

and move to the run-time section of input.txt. As a first step, we define the computational domain of our system:

```
##### FDTD GEOMETRY
number_of_cells=100 100 60  # x y z number of unit cells
box_size=2 2 2  # physical size of simulation box along x y z
```

The number of cells gives the number of YEE cells of our simulation, while the Box size if the physical size of the box. In Nanocpp, SI units can be used, as well as adimensional units (like in this example). These are defined as follows:

We then move to the Time & I/O parameters:

```
##### TIME &amp;amp;amp; I/O
base_dir=./res  # directory where to save results
CFL=2.  # CFL Condition
total_steps=10000ů  # steps used for performing final averages (total steps = equilibration + 
io_screen=1000  # I/O on screen
chkpoint_every=10000  # checkpoint step
chk_load=0  # 1 true, 0 false
chk_numb=0  # chkpoint number: 0 or 1
```

The parameter base_dir is the root directory where to save all results (the dir is automatically created if does not exist). The checkpoint is enabled by the Boolean variable chk_load, and the checkpoint number to be loaded is selected by chk_numb, which has been introduced as a mechanism of redundancy to avoidů crashing problem during the writing of the checkpoint. In particular, Nanocpp uses a toroidal system of two checkpoints, which are labeled as “0” and “1”. In the unlikely event of a system crash during a checkpoint (e.g., checkpoint 0), the simulation can be safely restarted by using the previous file (checkpoint 1), which has been written before.

2 Energy and Poynting

To enable the calculation of energy and poynting vectors, we first add them in the measurement section of the compile-time parameters part of input.txt:

```
measurement=energy+probes+poynting+Ex
```

We then re-compile the code:

```
$ make clean
$ make
```

and finally select the frequency of the energy & poynting I/O operations:

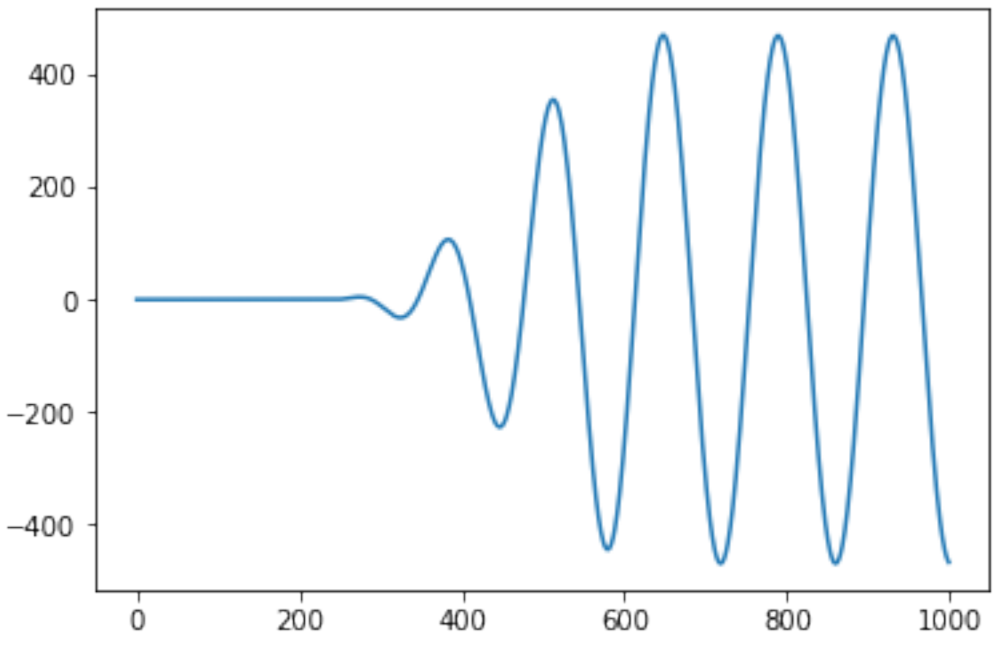
```
# ENERGY &amp;amp;amp; POYNTING
erg_io=10000  # frequency of io save
```

in this case every 10000 simulation steps. For nondispersive materials, the energy calculated by Nanocpp is the electromagnetic energy density.

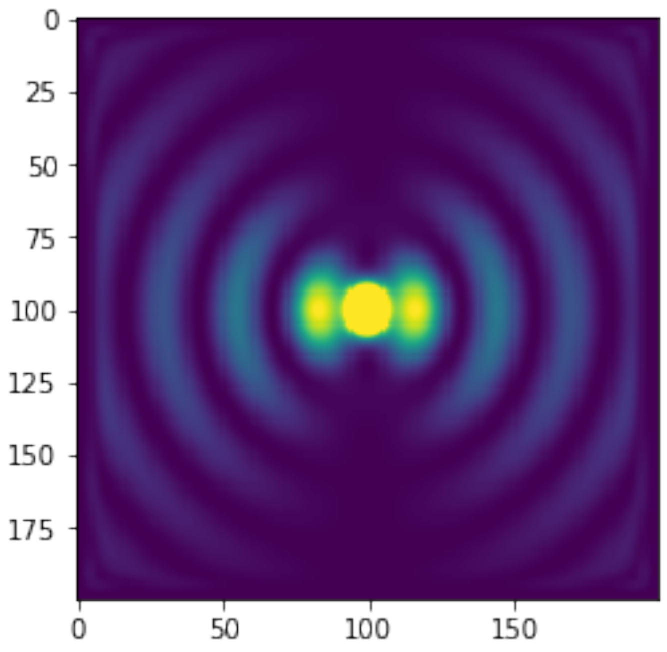
We can then launch Nanocpp (now employing 16 processors):

```
mpiexec -n 16 ./nanocpp input.txt
```

```
In [127]: Ex = np.fromfile(resdir+'probes/' + 'probe_0_ex.bin')
          plt.plot(Ex)
          plt.show()
```



```
In [116]: resdir = 'res/'
          erg = os.path.join(resdir,'erg_1.bin')
          px = os.path.join(resdir,'px_1.bin')
          pz = os.path.join(resdir,'pz_1.bin')
          e = np.fromfile(erg)
          px = np.fromfile(px)
          pz = np.fromfile(pz)
          vmax = e.max()
          e = e.reshape(200,200)
          pz = pz.reshape(200,200)
          px = px.reshape(200,200)
          #plt.subplot(121)
          plt.imshow(e,vmax=vmax/1e4)
          #plt.subplot(122)
          plt.show()
```



3 Sources

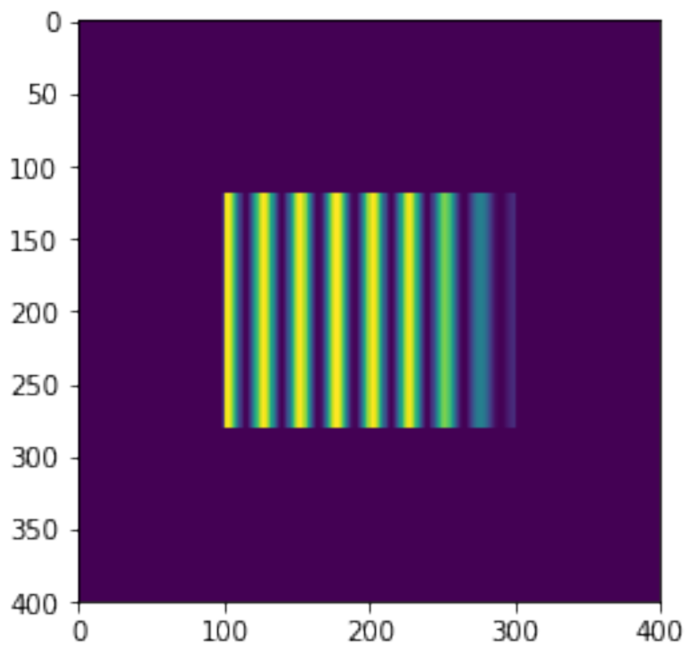
Other than just point sources, NANOCPP supports TFSF sources for plane waves and Gaussian beams. To enable a plane TFSF source with Ex polarization, go to the sources section in the compile-time parameters and type:

```
##### SOURCES
spatial=tfsf+ex

# TFSF SOURCE SPECIFIERS
tfsf_box=0.6 1.4 0.5 1.5 0.5 0.5  # xsta xend ysta yend zsta zend separated by just one space
tfsf_power=1e-3                              # average power of the source [W] at central frequen
```

The box is defined by the edges along x,y,z directions. The average power, conversely, represents the input power of the TFSF source in SI. After re-compiling, we run the simulation. Here is a plot of the energy obtained:

```
In [132]: resdir = 'res/'
          erg = os.path.join(resdir,'erg_1.bin')
          e = np.fromfile(erg)
          e = e.reshape(400,400)
          plt.imshow(e)
          plt.show()
```



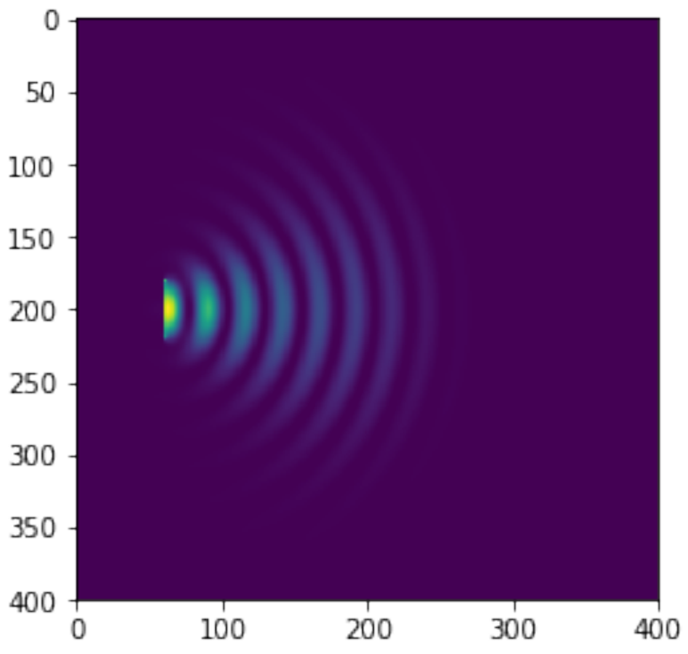
Localized TFSF sources are available, as well. To enable them, add the single specifier in the sources data list:

```
##### SOURCES
spatial=tfsf+Ex+single

tfsf_box=0.9 1.1 0.3 0.1 0.5 0.5
tfsf_power=1e-3
```

in this case, we selected a source at z=0.3, whose extent along x goes from 0.9 to 1.1. Then re-compile and run. This is the energy after 600 timesteps:

```
In [130]: e = np.fromfile(erg)
          e = e.reshape(400,400)
          plt.imshow(e)
          plt.show()
```



NANOCPP uses a slight modification to the standard TFSF formulation, which allows the plane wave to be numerically exact, up to double precision. In order to use a different time-waveform for the source, and pick up a Gaussian pulse:

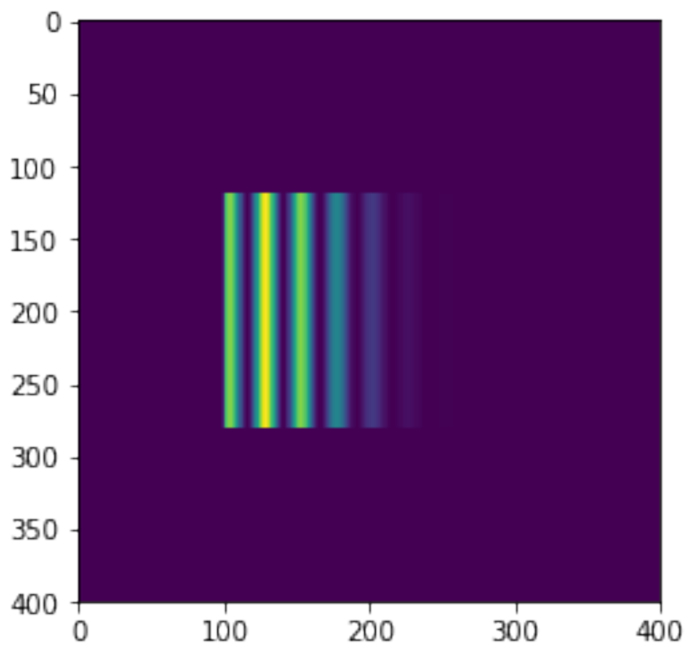
```
##### SOURCES
waveform=gaussian2+0.25+0.05
```

which selects a Gaussian envelope with $\lambda = 0.25$ and bandwidth $\Delta\lambda = 0.05$. We then increase the TFSF box size along z:

```
tfsf_box=0.6 1.4 0.2 1.8
tfsf_power=1e-3
```

to better see the tail of the energy field of the source. This is the energy distribution after 600 timesteps:

```
In [133]: e = np.fromfile(erg)
          e = e.reshape(400,400)
          plt.imshow(e)
          plt.show()
```



When using a TFSF source, NANOCPP assumes that the source propagates in the background material of the simulation, which is general is air. To use the TFSF in the case where there is more than one geometry, and the source is embedded into a material of dielectric constant ϵ_r, you can use the function:

```
void set_tfsf_background(double epsilon_r)
```

which sets the value of dielectric constant for the TFSF source.

4 Periodic boundaries

By deafult, NANOCPP assumes non-periodic boundaries terminated with UPML. Periodic boundaries are enabled by adding the directives periodic_ (direction=x,y, or z) in the problem_type compile-time parameter in input.txt:

```
problem_type=fdtd+periodic_x
```

different periodic directives periodic_x, periodic_y and periodic_z can be added together to specify more than one periodic boundary at the same time. In order to enable the periodicity, set also the corresponding UPML thickness to 0, e.g., along x:

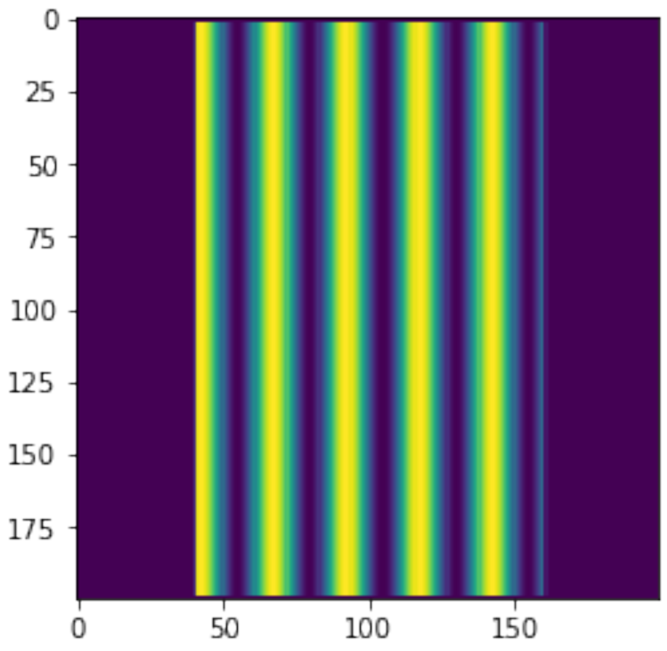
```
upml = 0 20
```

To enable a TFSF plane wave source, use the:

```
spatial=tfsf+single
```

directive, and specify a TFSF window from 0 to L-dx, being L the size of the computational domain along the chosen periodic direction and dx the grid spatial resolution.

```
In [53]: e = np.fromfile(erg)
         e = e.reshape(200,200)
         plt.imshow(e)
         plt.show()
```



5 Geometric shapes

Material parameters and shapes are contained in the file specified by the media_filename keyword:

```
##### FDTD GEOMETRY
number_of_cells=200 200 100  # x y z number of unit cells
box_size=2 2 2  # physical size of simulation box along x y z media_filename=geometry.txt
```

The file geometry.in is divided into two main sections:ů material parametersů and geometry. The standard file comes with two materials only, air and Silicon, which is enough for this example. Concerning the geometry section, the following shape primitives are supported:

```
### sphere  material id  (center  x y z) (radius)
### cylinder  material id  (center  x y z) (length  (radius) (orientation)
### cuboid  material id  (lower corner x y z) (upper corner x y z)
### elipsoid  material id  (center  x y z) (axis  x y z)
```

The first parameter of every shape is the material id. This is an integer number that connects the shape to a material defined in the material parameters section. The other geometrical parameters are self-explanatory. In the geometry.in file there is only one shape, a sphere centered in (1 1) which fills the whole space with air (material id 0).

```
#background sphere 0 1 1 100
```

In this example we will add an ellipse:

```
ellipsoid 1 1 1 0.3 0.2
```

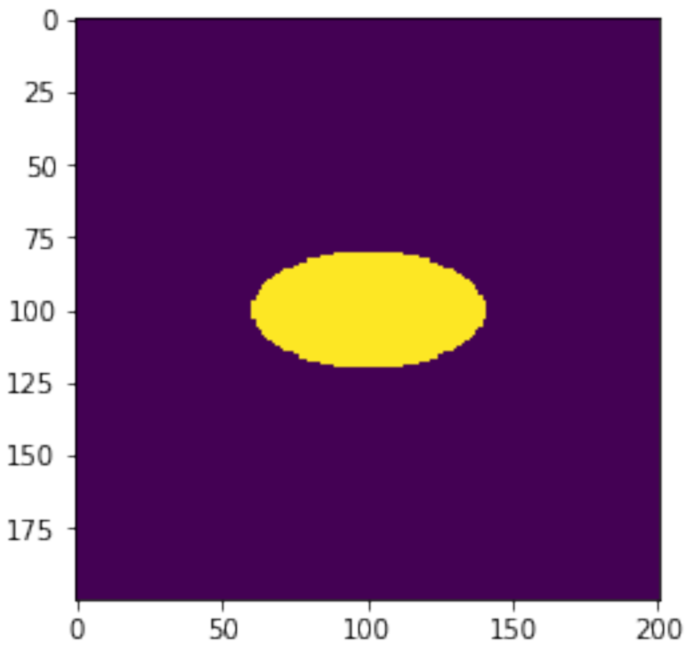
made of Silicon (material id 1), centered at (0.5 0.5) and with axes 0.1 along x and 0.2 along z. We then run NANOCPP again, and look at the output of NANOCPP:

```
SHAPES & MEDIA
filename: geometry.in
number of shapes: 1
sphere  type (0); center (0.5, 0.5); radius (100)
materials properties:
type: 0, eps_inf: 1
type: 1, eps_inf: 12

saved (z,x) 201 200 int on type_Ex.bin
saved (z,x) 200 201 int on type_Ez.bin
```

after every initialization, NANOCPP writes the spatial distribution of the materials id in the files (rootdir)/type_E(axis).bin. Data is saved as binary integers. Since the YEE grid uses different spatial positions for each electromagnetic field, a different file for each electromagnetic component is created and saved. In what follows we display the Ex media distribution, obtained from the 201 x 200 integer data of res/type_Ex.bin:

```
In [64]: e = np.fromfile(resdir+'type_Ex.bin',dtype=np.int32)
         e = e.reshape(200,201)
         plt.imshow(e)
         plt.show()
```



Geometrical operations on shapes are also supported. In NANOCPP, in particular, the rotation primitive is available:

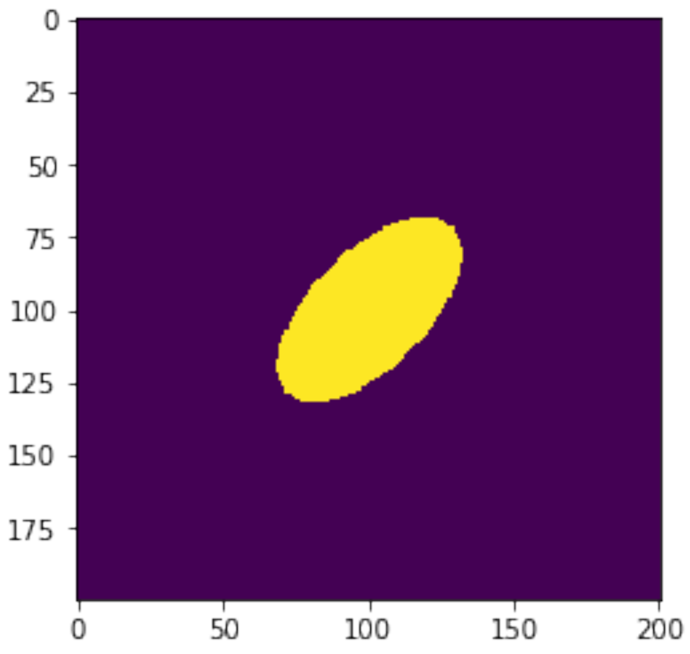
```
### rotation (centerů x y z) (rotation angles xz yz xy)
```

and it applies to all shapes defined below it. A rotation overwrites all the previous rotation directives. This allows to apply this primitive to single and multiple shapes, thus achieving the maximum degree of flexibility when defining a geometry. In the following example, we rotate the ellipse by 45 degrees along xz:

rotation 1. 1. 0.7854 ellipsoid 1 1 1 0.3 0.2

Re-run the simulation again. The file type_Ex.bin now shows a rotated geometry:

```
In [66]: e = np.fromfile(resdir+'type_Ex.bin',dtype=np.int32)
         e = e.reshape(200,201)
         plt.imshow(e)
         plt.show()
```



6 FAR FIELD

Parallel Far-Field calculation is available in Nanocpp by specifying the compile-time parameter farfield in the MEASUREMENT section of input.in:

```
##### MEASUREMENT
measurements=probes+Ex+farfield+energy
probes (Ei and Hi) + farfield + energy + poynting
```

Additional optional parameters bscatt and fscatt can be added to calculate backward and forward total electromagnetic far-fields. Please remember to compile the code with:

```
make clean
make
```

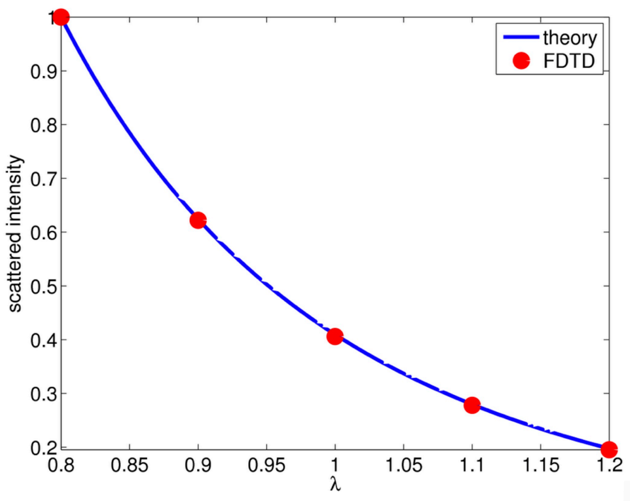
if not done before. Nanocpp implements the Time-Domain Near-To-Far-Field (NTFF) algorithm, explained in Chap. 8, Section 8.6 of the Taflove's Book. This allows to calculate the time evolution of the scattered electromagnetic fields at specific angles selected by the user. Several original optimization strategies have been developed in order to optimize both memory requirements and execution time. In order to set up the Far-Field computation routine, the user needs to specify a cuboid surface, to be used as a source for the surface equivalence theorem (see Chapter 8 of Taflove's for more details):

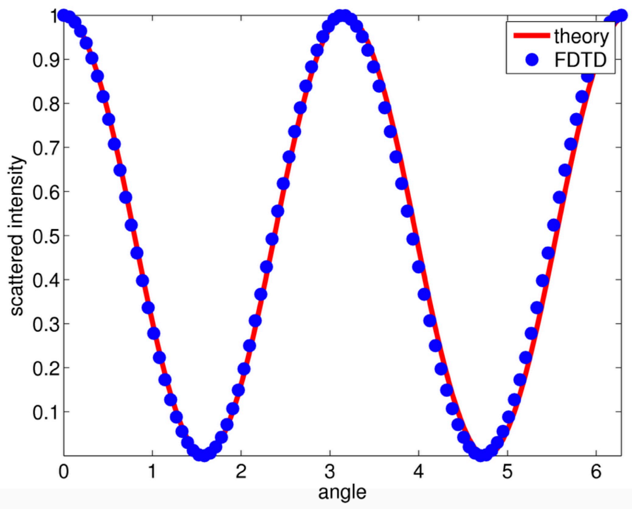
```
# FAR FIELD
farfield_box=0.2 1.8 0.2 1.8 0.5 0.5
# far field box xsta xend ysta yend zsta zend separated by just one space
```

Angles $[\theta, \phi]$, where the scattered fields have to be calculated, should be saved in a binary file that needs to be specified in the file input.in:

```
n_angles=100        # number of angles (theta,phi) to process
ff_file=angles.in   # binary filename containing angles [theta,phi, theta,phi, ...]
```

In this case the programs looks for the angles in the file angles.in. At the end of the simulation, Nanocpp saves the scattered electric field E_θ and E_ϕ in the files /eth.bin and /ephi.bin, respectively. These files input.in, geometry.in and angles.in calculate the scattered far field from an electric dipole radiator oscillating parallel to **x**, with a current $J \propto \frac{\sin(\omega t)}{\lambda}$. For this type of source, (see Chaper 9 of Jackson's Book) the far field intensity is $I \propto \frac{\sin \theta^2}{\lambda^4}$. Figures below show the results of Nanocpp:


theory
FDTD
scattered intensity
λ


theory
FDTD
scattered intensity
angle