# fdtd

*Release 0.2.2*

**Floris laporte**

**Oct 09, 2021**

# CONTENTS

A 3D electromagnetic FDTD simulator written in Python. The FDTD simulator has an optional PyTorch backend, enabling FDTD simulations on a GPU.

CHAPTER

# ONE

# DOCS

## 1.1 Examples

### 1.1.1 00. Quick Start

#### Imports

the fdtd library is simply imported as follows:

[1]:
```
import fdtd
```

#### Setting the backend

the fdtd library allows for setting a backend. There exist a Numpy backend and several PyTorch backends. The available backends are: - `"numpy"` (defaults to float64 arrays) - `"torch"` (defaults to float64 tensors) - `"torch.float32"` - `"torch.float64"` - `"torch.cuda"` (defaults to float64 tensors) - `"torch.cuda.float32"` - `"torch.cuda.float64"`

In general, the `"numpy"` backend is preferred for standard CPU calculations with `"float64"` precision. In general, `"float64"` precision is always preferred for FDTD simulations, however, `"float32"` might give a significant performance boost.

The `"cuda"` backends are only available for computers with a GPU.

[2]:
```
fdtd.set_backend("numpy")
```

#### The FDTD-grid

The FDTD grid defines the simulation region.

```
# signature
fdtd.Grid(
    shape: Tuple[Number, Number, Number],
    grid_spacing: float = 155e-9,
    permittivity: float = 1.0,
    permeability: float = 1.0,
    courant_number: float = None,
)
```

A grid is defined by its `shape`, which is just a 3D tuple of `Number`-types (integers or floats). If the shape is given in floats, it denotes the width, height and length of the grid in meters. If the shape is given in integers, it denotes the width, height and length of the grid in terms of the `grid_spacing`. Internally, these numbers will be translated to three integers: `grid.Nx`, `grid.Ny` and `grid.Nz`.

A `grid_spacing` can be given. For stability reasons, it is recommended to choose a grid spacing that is at least 10 times smaller than the *smallest* wavelength in the grid. This means that for a grid containing a source with wavelength `1550nm` and a material with refractive index of `3.1`, the recommended minimum `grid_spacing` turns out to be `50pm`

For the `permittivity` and `permeability` floats or arrays with the following shapes

- `(grid.Nx, grid.Ny, grid.Nz)`
- or `(grid.Nx, grid.Ny, grid.Nz, 1)`
- or `(grid.Nx, grid.Ny, grid.Nz, 3)`

are expected. In the last case, the shape implies the possibility for different permittivity for each of the major axes (so-called *uniaxial* or *biaxial* materials). Internally, these variables will be converted (for performance reasons) to their inverses `grid.inverse_permittivity` array and a `grid.inverse_permeability` array of shape `(grid.Nx, grid.Ny, grid.Nz, 3)`. It is possible to change those arrays after making the grid.

Finally, the `courant_number` of the grid determines the relation between the `time_step` of the simulation and the `grid_spacing` of the grid. If not given, it is chosen to be the maximum number allowed by the Courant-Friedrichs-Lewy Condition: `1` for `1D` simulations, `1/2` for `2D` simulations and `1/3` for `3D` simulations (the dimensionality will be derived by the shape of the grid). For stability reasons, it is recommended not to change this value.

[3]:
```
grid = fdtd.Grid(
    shape = (25e-6, 15e-6, 1), # 25um x 15um x 1 (grid_spacing) --> 2D FDTD
)

print(grid)
```

```
Grid(shape=(161,97,1), grid_spacing=1.55e-07, courant_number=0.70)
```

### Adding an object to the grid

An other option to locally change the `permittivity` or `permeability` in the grid is to add an `Object` to the grid.

```
# signature
fdtd.Object(
    permittivity: Tensorlike,
    name: str = None
)
```

An object defines a part of the grid with modified update equations, allowing to introduce for example absorbing materials or biaxial materials for which mixing between the axes are present through `Pockels coefficients` or many more. In this case we'll make an object with a different `permittivity` than the grid it is in.

Just like for the grid, the `Object` expects a `permittivity` to be a floats or an array of the following possible shapes

- `(obj.Nx, obj.Ny, obj.Nz)`
- or `(obj.Nx, obj.Ny, obj.Nz, 1)`
- or `(obj.Nx, obj.Ny, obj.Nz, 3)`

Note that the values `obj.Nx`, `obj.Ny` and `obj.Nz` are not given to the object constructor. They are in stead derived from its placing in the grid:

[4]:
```
grid[11:32, 30:84, 0] = fdtd.Object(permittivity=1.7**2, name="object")
```

Several things happen here. First of all, the object is given the space `[11:32, 30:84, 0]` in the grid. Because it is given this space, the object's `Nx`, `Ny` and `Nz` are automatically set. Furthermore, by supplying a name to the object, this name will become available in the grid:

[5]:
```
print(grid.object)
```

```
    Object(name='object')
        @ x=11:32, y=30:84, z=0:1
```

We can add a second object to the grid:

[6]:
```
grid[13e-6:18e-6, 5e-6:8e-6, 0] = fdtd.Object(permittivity=1.5**2)
```

Here we chose to slice the grid with floating point numbers, which will be replaced by integer `Nx`, `Ny` and `Nz` during the registration of the object. Since we didnt give the object a name, the object won't be available to us as an attribute of the grid. However, it is still available to us via the `grid.objects` list:

[7]:
```
print(grid.objects)
```

```
[Object(name='object'), Object(name=None)]
```

This list stores all objects (i.e. of type `fdtd.Object`) in the order that they were added to the grid.

### Adding a source to the grid

Similarly as to adding an object to the grid, an `fdtd.LineSource` can also be added:

```
# signature
fdtd.LineSource(
    period: Number = 15, # timesteps or seconds
    amplitude: float = 1.0,
    phase_shift: float = 0.0,
    name: str = None,
)
```

Similarly to an `fdtd.Object`, an `fdtd.Source` size is defined by its placement on the grid:

[8]:
```
grid[7.5e-6:8.0e-6, 11.8e-6:13.0e-6, 0] = fdtd.LineSource(
    period = 1550e-9 / (3e8), name="source"
)
```

However, it is important to note that in this case we are adding a `LineSource`, i.e. the source spans the diagonal of the cube defined by the slices. Internally, these slices will be converted into lists to ensure the expected behavior:

[9]:
```
print(grid.source)
```

```
    LineSource(period=14, amplitude=1.0, phase_shift=0.0, name='source')
        @ x=[48, ... , 51], y=[76, ... , 83], z=[0, ... , 0]
```

Note that one could have also supplied lists to index the grid in the first place. This feature could be useful to create a `LineSource` of arbitrary shape.

### Adding a detector to the grid

Adding a detector to the grid works the same as adding a source

```
# signature
fdtd.LineDetector(
    name=None
)
```

[10]:
```
grid[12e-6, :, 0] = fdtd.LineDetector(name="detector")
```

[11]:
```
print(grid.detector)
```

```
    LineDetector(name='detector')
        @ x=[77, ... , 77], y=[0, ... , 96], z=[0, ... , 0]
```

### Adding grid boundaries

Although, having an object, source and detector to simulate is in principle enough to perform an FDTD simulation, One also needs to define a grid boundary to prevent the fields to be reflected. One of those boundaries that can be added to the grid is a Perfectly Matched Layer or PML. These are basically absorbing boundaries.

```
fdtd.PML(
    a: float = 1e-8, # stability factor
    name: str = None
)
```

[12]:
```
# x boundaries
# grid[0, :, :] = fdtd.PeriodicBoundary(name="xbounds")
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")

# y boundaries
# grid[:, 0, :] = fdtd.PeriodicBoundary(name="ybounds")
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")
```

### Grid summary

A simple summary of the grid can be shown by printing out the grid:

[13]:
```
print(grid)
```

```
Grid(shape=(161,97,1), grid_spacing=1.55e-07, courant_number=0.70)

sources:
    LineSource(period=14, amplitude=1.0, phase_shift=0.0, name='source')
        @ x=[48, ... , 51], y=[76, ... , 83], z=[0, ... , 0]

detectors:
```

(continues on next page)

(continued from previous page)

```
    LineDetector(name='detector')
        @ x=[77, ... , 77], y=[0, ... , 96], z=[0, ... , 0]

boundaries:
    PML(name='pml_xlow')
        @ x=0:10, y=:, z=:
    PML(name='pml_xhigh')
        @ x=-10:, y=:, z=:
    PML(name='pml_ylow')
        @ x=:, y=0:10, z=:
    PML(name='pml_yhigh')
        @ x=:, y=-10:, z=:

objects:
    Object(name='object')
        @ x=11:32, y=30:84, z=0:1
    Object(name=None)
        @ x=84:116, y=32:52, z=0:1
```

### Running a simulation

Running a simulation is as simple as using the `grid.run` method.

```
grid.run(
    total_time: Number,
    progress_bar: bool = True
)
```

Just like for the the lengths in the grid, the `total_time` of the simulation can be specified as an integer (number of `time_steps`) or as a float (in seconds).

[14]:
```
grid.run(total_time=100)
```

```
100%|| 100/100 [00:00<00:00, 644.97it/s]
```

### Grid visualization

Let's visualize the grid. This can be done with the `grid.visualize` method:

```
# signature
grid.visualize(
    grid,
    x=None,
    y=None,
    z=None,
    cmap="Blues",
    pbcolor="C3",
    pmlcolor=(0, 0, 0, 0.1),
    objcolor=(1, 0, 0, 0.1),
    srccolor="C0",
```

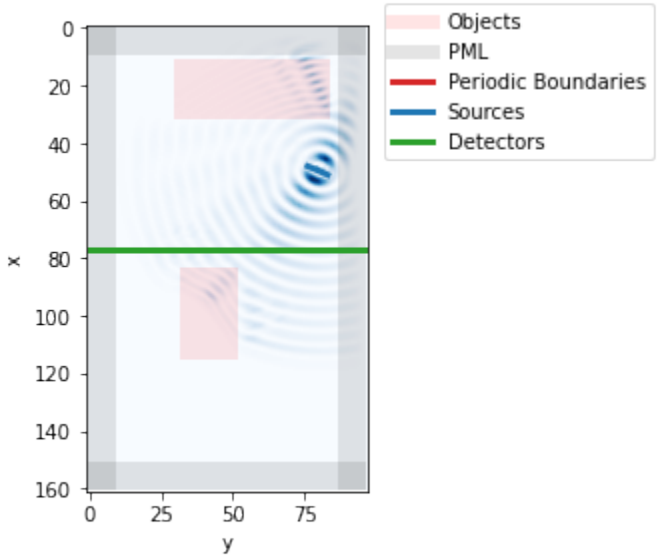
(continues on next page)

(continued from previous page)

```
    detcolor="C2",
    show=True,
)
```

This method will by default visualize all objects in the grid, as well as the current field intensity at a certain `x`, `y` **OR** `z`-plane. By setting `show=False`, one can disable the immediate visualization of matplotlib.

[15]:
```
grid.visualize(z=0, show=False)
import matplotlib.pyplot as plt
```




### 1.1.2 01. Basic Example

A simple example on how to use the FDTD Library

#### Imports

[1]:
```
import matplotlib.pyplot as plt

import fdtd
import fdtd.backend as bd
```

#### Set Backend

[2]:
```
fdtd.set_backend("numpy")
```

### Constants

[3]:
```
WAVELENGTH = 1550e-9
SPEED_LIGHT: float = 299_792_458.0  # [m/s] speed of light
```

### Simulation

create FDTD Grid

[4]:
```
grid = fdtd.Grid(
    (2.5e-5, 1.5e-5, 1),
    grid_spacing=0.1 * WAVELENGTH,
    permittivity=1.0,
    permeability=1.0,
)
```

boundaries

[5]:
```
# grid[0, :, :] = fdtd.PeriodicBoundary(name="xbounds")
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")

# grid[:, 0, :] = fdtd.PeriodicBoundary(name="ybounds")
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")

grid[:, :, 0] = fdtd.PeriodicBoundary(name="zbounds")
```

sources

[6]:
```
grid[50:55, 70:75, 0] = fdtd.LineSource(
    period=WAVELENGTH / SPEED_LIGHT, name="linesource"
)
grid[100, 60, 0] = fdtd.PointSource(
    period=WAVELENGTH / SPEED_LIGHT, name="pointsource",
)
```

detectors

[7]:
```
grid[12e-6, :, 0] = fdtd.LineDetector(name="detector")
```

objects

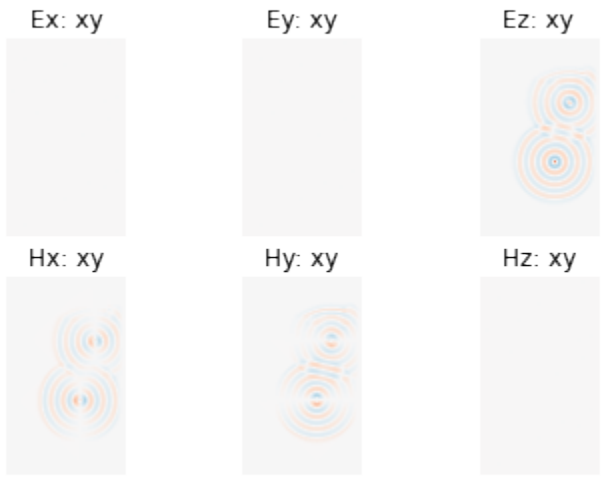
[8]:
```
grid[11:32, 30:84, 0:1] = fdtd.AnisotropicObject(permittivity=2.5, name="object")
```

### Run simulation

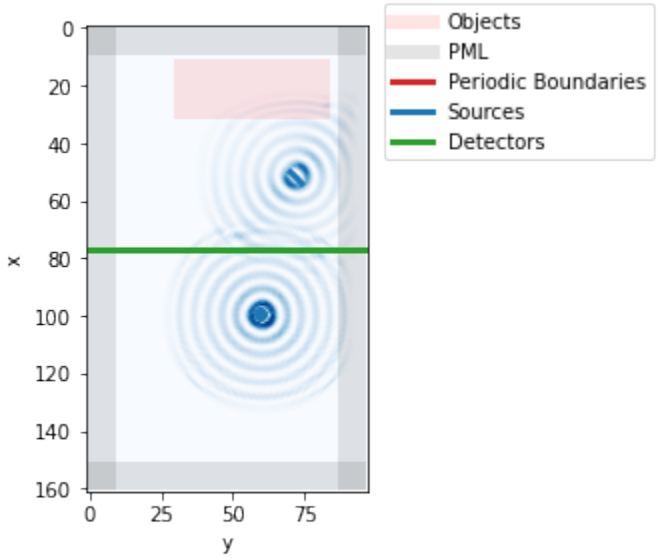
[9]:
```
grid.run(50, progress_bar=False)
```

### Visualization

[10]:
```
fig, axes = plt.subplots(2, 3, squeeze=False)
titles = ["Ex: xy", "Ey: xy", "Ez: xy", "Hx: xy", "Hy: xy", "Hz: xy"]

fields = bd.stack(
    [
        grid.E[:, :, 0, 0],
        grid.E[:, :, 0, 1],
        grid.E[:, :, 0, 2],
        grid.H[:, :, 0, 0],
        grid.H[:, :, 0, 1],
        grid.H[:, :, 0, 2],
    ]
)

m = max(abs(fields.min().item()), abs(fields.max().item()))

for ax, field, title in zip(axes.ravel(), fields, titles):
    ax.set_axis_off()
    ax.set_title(title)
    ax.imshow(bd.numpy(field), vmin=-m, vmax=m, cmap="RdBu")

plt.show()
```




[11]:
```
plt.figure()
grid.visualize(z=0)
```




### 1.1.3 02. Absorbing Object

A simple example using the AbsorbingObject

#### Imports

[1]:
```
import matplotlib.pyplot as plt

import fdtd
fdtd.set_backend("numpy")
```

#### Constants

[2]:
```
WAVELENGTH = 1550e-9
SPEED_LIGHT: float = 299_792_458.0  # [m/s] speed of light
```

#### Grid setup

[3]:
```
# create FDTD Grid
grid = fdtd.Grid(
    (1.5e-5, 1.5e-5, 1),  # 2D grid
    grid_spacing=0.1 * WAVELENGTH,
    permittivity=2.5,  # same as object
)

# sources
grid[15, :] = fdtd.LineSource(period=WAVELENGTH / SPEED_LIGHT, name="source")
```

(continues on next page)

(continued from previous page)

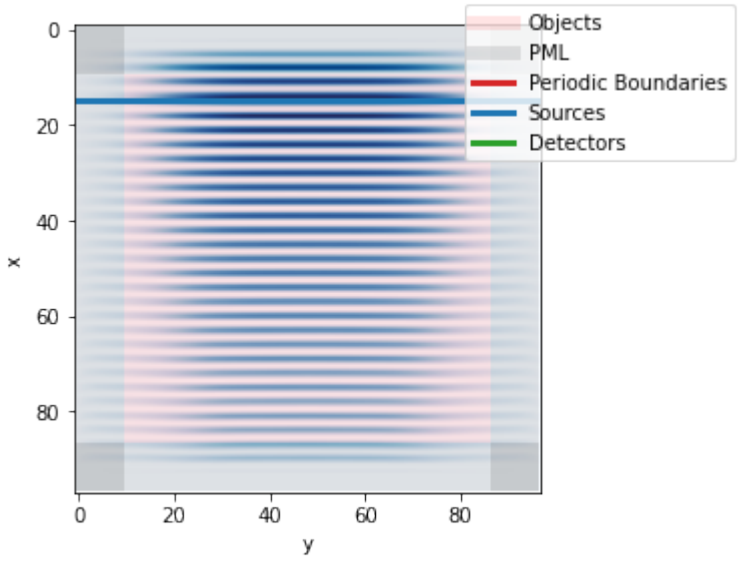
```
# detectors
grid[-15, :, 0] = fdtd.LineDetector(name="detector")

# x boundaries
# grid[0, :, :] = fdtd.PeriodicBoundary(name="xbounds")
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")

# y boundaries
# grid[:, 0, :] = fdtd.PeriodicBoundary(name="ybounds")
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")

# The absorbing object fills the whole grid
grid[10:-10, 10:-10, :] = fdtd.AbsorbingObject(
    permittivity=2.5, conductivity=1e-6, name="absorbin_object"
)
```

### Simulation without absorption:

[4]:
```
grid.run(250, progress_bar=False)
```

### Visualization

[5]:
```
grid.visualize(z=0)
```



### 1.1.4 03. Objects of arbitrary shape

#### Imports

[1]:
```
import fdtd
import numpy as np
import matplotlib.pyplot as plt
fdtd.set_backend("numpy")
```

#### Grid Setup

[2]:
```
grid = fdtd.Grid(
    shape = (300, 300, 1), # 25um x 15um x 1 (grid_spacing) --> 2D FDTD
    grid_spacing = 1e-7,
    permittivity = 1,
)

grid[50:250, 50, 0] = fdtd.LineSource(
    period = 1550e-9 / (3e8), name="source"
)

grid[50:250, 250, 0] = fdtd.LineDetector(name="detector")

grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")
```

#### Circular Object

[3]:
```
refractive_index = 1.7
x = y = np.linspace(-1,1,100)
X, Y = np.meshgrid(x, y)
circle_mask = X**2 + Y**2 < 1
permittivity = np.ones((100,100,1))
permittivity += circle_mask[:,:,None]*(refractive_index**2 - 1)
grid[170:270, 100:200, 0] = fdtd.Object(permittivity=permittivity, name="object")
```

#### Run Simulation

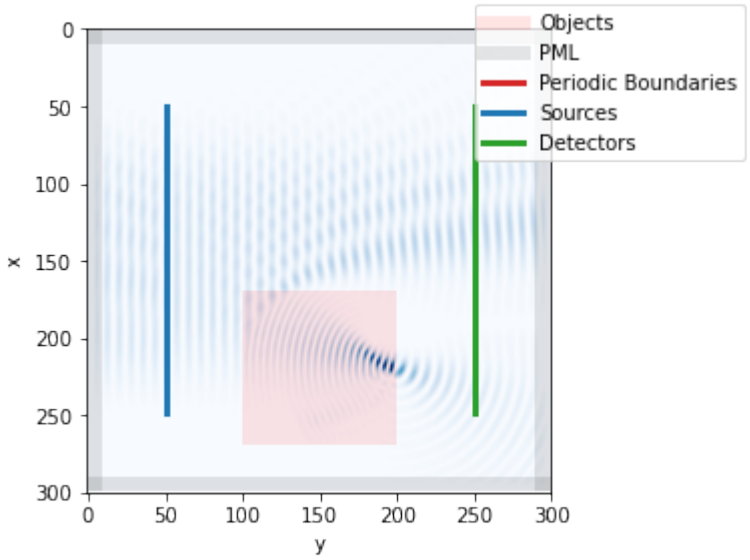
[4]:
```
grid.run(total_time=500)
```

```
100%|| 500/500 [00:03<00:00, 164.02it/s]
```

#### Visualize

Even though visualization of a circular object is not implemented (for now), one can clearly see the focussing.

[5]:
```
grid.visualize(z=0)
```




[6]:
```
grid.object.inverse_permittivity.min()
```

[6]:
```
0.34602076124567477
```

### 1.1.5 04. Performance Profiling

We can profile the performance with a 3D FDTD simulation:

#### Imports

[1]:
```
import matplotlib.pyplot as plt
from line_profiler import LineProfiler

import fdtd
import fdtd.backend as bd
```

#### Set Backend

Let's profile the impact of the backend. These are the possible backends:

- `numpy` (defaults to float64 arrays)
- `torch` (defaults to float64 tensors)
- `torch.float32`
- `torch.float64`
- `torch.cuda` (defaults to float64 tensors)

- `torch.cuda.float32`
- `torch.cuda.float64`

[2]:
```
fdtd.set_backend("numpy")
```

In general, the `numpy` backend is preferred for standard CPU calculations with `"float64"` precision as it is slightly faster than `torch` backend on CPU. However, a significant performance improvement can be obtained by choosing `torch.cuda` on large enough grids.

Note that, in FDTD, `float64` precision is generally preferred over `float32` to ensure numerical stability and prevent numerical dispersion. If this is of no concern to you, you can opt for `float32` precision, which especially on a GPU might yield a significant performance boost.

### Constants

[3]:
```
WAVELENGTH = 1550e-9
SPEED_LIGHT: float = 299_792_458.0  # [m/s] speed of light
```

### Setup Simulation

create FDTD Grid

[4]:
```
N = 100

grid = fdtd.Grid(
    (N, N, N),
    grid_spacing=0.05 * WAVELENGTH,
    permittivity=1.0,
    permeability=1.0,
)
```

add boundaries

[5]:
```
# x boundaries
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")

# y boundaries
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")

# z boundaries
grid[:, :, 0:10] = fdtd.PML(name="pml_zlow")
grid[:, :, -10:] = fdtd.PML(name="pml_zhigh")
```

add sources

[6]:
```
grid[10+N//10:10+N//10, :, :] = fdtd.PlaneSource(
    period=WAVELENGTH / SPEED_LIGHT, name="source"
)
```

```
[1, 100, 100]
```

add objects

[7]:
```
grid[10+N//5:4*N//5-10, 10+N//5:4*N//5-10, 10+N//5:4*N//5-10] = fdtd.
↪Object(permittivity=2.5, name="center_object")
```

grid summary

[8]:
```
print(grid)
```

```
Grid(shape=(100,100,100), grid_spacing=7.75e-08, courant_number=0.57)

sources:
    PlaneSource(period=35, amplitude=1.0, phase_shift=0.0, name='source')
        @ x=[20, ... , 21], y=[0, ... , 100], z=[0, ... , 100]

boundaries:
    PML(name='pml_xlow')
        @ x=0:10, y=:, z=:
    PML(name='pml_xhigh')
        @ x=-10:, y=:, z=:
    PML(name='pml_ylow')
        @ x=:, y=0:10, z=:
    PML(name='pml_yhigh')
        @ x=:, y=-10:, z=:
    PML(name='pml_zlow')
        @ x=:, y=:, z=0:10
    PML(name='pml_zhigh')
        @ x=:, y=:, z=-10:

objects:
    Object(name='center_object')
        @ x=30:70, y=30:70, z=30:70
```

### Setup LineProfiler

create and enable profiler

[9]:
```
profiler = LineProfiler()
profiler.add_function(grid.update_E)
profiler.enable()
```

### Run Simulation

run simulation

[10]:
```
grid.run(50, progress_bar=True)
```

```
100%|| 50/50 [00:12<00:00,  4.03it/s]
```

### Profiler Results

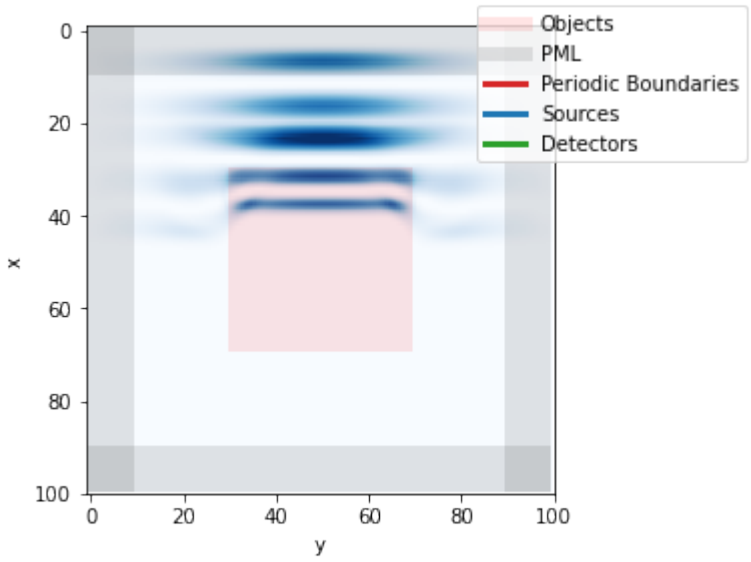
print profiler summary

[11]:
```
profiler.print_stats()
```

```
Timer unit: 1e-06 s

Total time: 6.15308 s
File: /home/docs/checkouts/readthedocs.org/user_builds/fdtd/checkouts/stable/docs/
↪examples/fdtd/grid.py
Function: update_E at line 291

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   291                                           def update_E(self):
   292                                               """ update the electric field␣
↪by using the curl of the magnetic field """
   293
   294                                               # update boundaries: step 1
   295       350       1045.0      3.0      0.0          for boundary in self.boundaries:
   296       300    2281803.0   7606.0     37.1              boundary.update_phi_E()
   297
   298        50    2663606.0  53272.1     43.3          curl = curl_H(self.H)
   299        50     723994.0  14479.9     11.8          self.E += self.courant_number *␣
↪self.inverse_permittivity * curl
   300
   301                                               # update objects
   302       100        435.0      4.3      0.0          for obj in self.objects:
   303        50      59449.0   1189.0      1.0              obj.update_E(curl)
   304
   305                                               # update boundaries: step 2
   306       350        632.0      1.8      0.0          for boundary in self.boundaries:
   307       300     419090.0   1397.0      6.8              boundary.update_E()
   308
   309                                               # add sources to grid:
   310       100        141.0      1.4      0.0          for src in self.sources:
   311        50       2852.0     57.0      0.0              src.update_E()
   312
   313                                               # detect electric field
   314        50         36.0      0.7      0.0          for det in self.detectors:
   315                                                   det.detect_E()
```

#### Visualization

[12]:
```
plt.figure()
grid.visualize(z=N//2)
```




### 1.1.6 05. Lenses and analysing lensing actions

submitted by substancia, adapted by flaport

#### Imports

[1]:
```
import os
import fdtd
import numpy as np
import matplotlib.pyplot as plt
```

#### Grid

[2]:
```
grid = fdtd.Grid(shape=(260, 15.5e-6, 1), grid_spacing=77.5e-9)
# x boundaries
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")
# y boundaries
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")
simfolder = grid.save_simulation("Lenses")  # initializing environment to save␣
↪simulation data
print(simfolder)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fdtd/checkouts/stable/docs/examples/
↪fdtd_output/fdtd_output_2021-10-9-18-9-10 (Lenses)
```

### Objects

defining a biconvex lens

[3]:
```
x, y = np.arange(-200, 200, 1), np.arange(190, 200, 1)
X, Y = np.meshgrid(x, y)
lens_mask = X ** 2 + Y ** 2 <= 40000
for j, col in enumerate(lens_mask.T):
    for i, val in enumerate(np.flip(col)):
        if val:
            grid[30 + i : 50 - i, j - 100 : j - 99, 0] = fdtd.Object(permittivity=1.5 **␣
↪2, name=str(i) + "," + str(j))
            break
```

### Source

using a continuous source (not a pulse)

[4]:
```
grid[15, 50:150, 0] = fdtd.LineSource(period=1550e-9 / (3e8), name="source")
```

### Detectors

using a BlockDetector

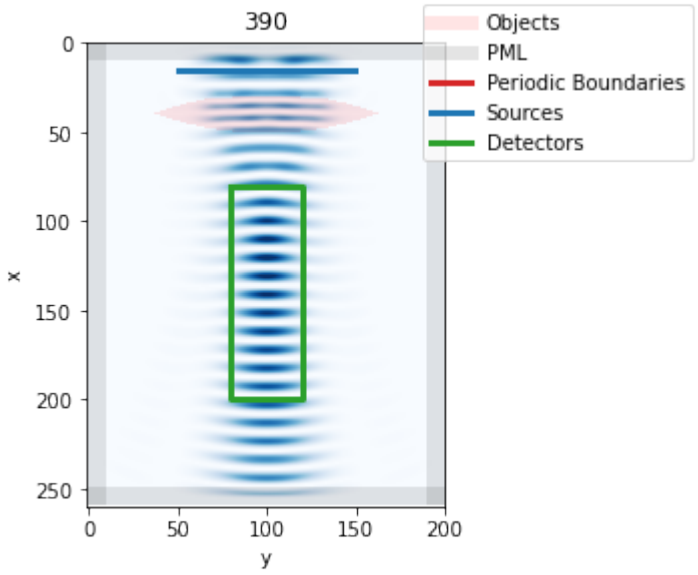
[5]:
```
grid[80:200, 80:120, 0] = fdtd.BlockDetector(name="detector")
```

Saving grid geometry for future reference

[6]:
```
with open(os.path.join(simfolder, "grid.txt"), "w") as f:
    f.write(str(grid))
    wavelength = 3e8/grid.source.frequency
    wavelengthUnits = wavelength/grid.grid_spacing
    GD = np.array([grid.x, grid.y, grid.z])
    gridRange = [np.arange(x/grid.grid_spacing) for x in GD]
    objectRange = np.array([[gridRange[0][x.x], gridRange[1][x.y], gridRange[2][x.z]]␣
↪for x in grid.objects], dtype=object).T
    f.write("\n\nGrid details (in wavelength scale):")
    f.write("\n\tGrid dimensions: ")
    f.write(str(GD/wavelength))
    f.write("\n\tSource dimensions: ")
    f.write(str(np.array([grid.source.x[-1] - grid.source.x[0] + 1, grid.source.y[-1] -␣
↪grid.source.y[0] + 1, grid.source.z[-1] - grid.source.z[0] + 1])/wavelengthUnits))
    f.write("\n\tObject dimensions: ")
    f.write(str([(max(map(max, x)) - min(map(min, x)) + 1)/wavelengthUnits for x in␣
↪objectRange]))
```

### Simulation

[7]:
```
from IPython.display import clear_output # only necessary in jupyter notebooks
for i in range(400):
    grid.step()  # running simulation 1 timestep a time and animating
    if i % 10 == 0:
        # saving frames during visualization
        grid.visualize(z=0, animate=True, index=i, save=True, folder=simfolder)
        plt.title(f"{i:3.0f}")
        clear_output(wait=True) # only necessary in jupyter notebooks

grid.save_data()  # saving detector readings
```




We can generate a video with ffmpeg:

[8]:
```
try:
    video_path = grid.generate_video(delete_frames=False)  # rendering video from saved␣
↪frames
except:
    video_path = ""
    print("ffmpeg not installed?")
```

```
ffmpeg not installed?
```

[9]:
```
if video_path:
    from IPython.display import Video
    display(Video(video_path, embed=True))
```

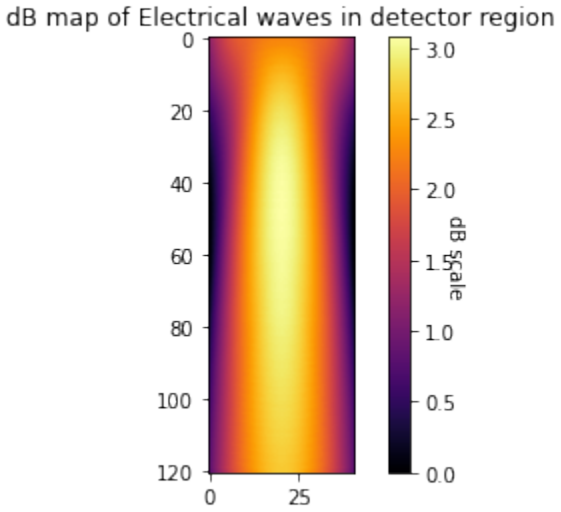
#### Analyse

analysing data stored by above simulation by plotting a 2D decibel map

[10]:
```
df = np.load(os.path.join(simfolder, "detector_readings.npz"))
fdtd.dB_map_2D(df["detector (E)"])
```

```
100%|| 121/121 [00:01<00:00, 105.38it/s]
```

```
Peak at: [[[45, 20]]]
```




### 1.1.7 06. GRIN medium and analysing refraction

submitted by substancia, adapted by flaport

#### Imports

[1]:
```
import os
import fdtd
import numpy as np
import matplotlib.pyplot as plt
```

#### Grid

[2]:
```
grid = fdtd.Grid(shape=(9.3e-6, 15.5e-6, 1), grid_spacing=77.5e-9)
# x boundaries
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")
# y boundaries
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")
simfolder = grid.save_simulation("GRIN")  # initializing environment to save simulation␣
↪data
print(simfolder)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fdtd/checkouts/stable/docs/examples/
↪fdtd_output/fdtd_output_2021-10-9-18-10-2 (GRIN)
```

### Objects

defining a graded refractive index slab, with homogenous slab extensions outwards from both ends

[3]:
```
n0, theta, t = 1, 30, 0.5
for i in range(50):
    x = i * 0.08
    epsilon = n0 + x * np.sin(np.radians(theta)) / t
    epsilon = epsilon ** 0.5
    grid[
        5.1e-6:5.6e-6, (5 + i * 0.08) * 1e-6 : (5.08 + i * 0.08) * 1e-6, 0
    ] = fdtd.Object(permittivity=epsilon, name="object" + str(i))

# homogenous slab extensions
grid[5.1e-6:5.6e-6, 0.775e-6:5e-6, 0] = fdtd.Object(
    permittivity=n0 ** 2, name="objectLeft"
)
grid[5.1e-6:5.6e-6, 9e-6 : (15.5 - 0.775) * 1e-6, 0] = fdtd.Object(
    permittivity=epsilon, name="objectRight"
)
```

### Source

using a pulse (hanning window pulse)

[4]:
```
grid[3.1e-6, 1.5e-6:14e-6, 0] = fdtd.LineSource(period=1550e-9 / (3e8), name="source",␣
↪pulse=True, cycle=3, hanning_dt=4e-15)
```

### Detectors

using a linear array of LineDetector

[5]:
```
for i in range(-4, 8):
    grid[5.8e-6, 84 + 4 * i : 86 + 4 * i, 0] = fdtd.LineDetector(name="detector" +␣
↪str(i))
```

### Saving grid geometry

[6]:
```
with open(os.path.join("./fdtd_output", grid.folder, "grid.txt"), "w") as f:
    f.write(str(grid))
    wavelength = 3e8/grid.source.frequency
    wavelengthUnits = wavelength/grid.grid_spacing
    GD = np.array([grid.x, grid.y, grid.z])
    gridRange = [np.arange(x/grid.grid_spacing) for x in GD]
    objectRange = np.array([[gridRange[0][x.x], gridRange[1][x.y], gridRange[2][x.z]]␣
↪for x in grid.objects], dtype=object).T
```

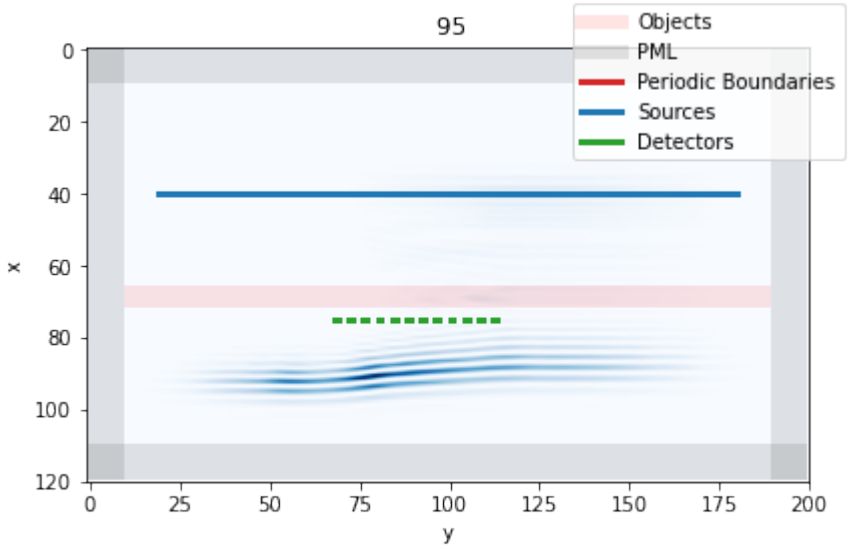
(continues on next page)

(continued from previous page)

```
    f.write("\n\nGrid details (in wavelength scale):")
    f.write("\n\tGrid dimensions: ")
    f.write(str(GD/wavelength))
    f.write("\n\tSource dimensions: ")
    f.write(str(np.array([grid.source.x[-1] - grid.source.x[0] + 1, grid.source.y[-1] -␣
↪grid.source.y[0] + 1, grid.source.z[-1] - grid.source.z[0] + 1])/wavelengthUnits))
    f.write("\n\tObject dimensions: ")
    f.write(str([(max(map(max, x)) - min(map(min, x)) + 1)/wavelengthUnits for x in␣
↪objectRange]))
```

### Simulation

[7]:
```
from IPython.display import clear_output  # only necessary in jupyter notebooks

for i in range(100):
    grid.step()  # running simulation 1 timestep a time and animating
    if i % 5 == 0:
        # saving frames during visualization
        grid.visualize(z=0, animate=True, index=i, save=True, folder=simfolder)
        plt.title(f"{i:3.0f}")
        clear_output(wait=True)  # only necessary in jupyter notebooks
grid.save_data()  # saving detector readings
```




We can generate a video with ffmpeg:

[8]:
```
try:
    video_path = grid.generate_video(delete_frames=False)  # rendering video from saved␣
↪frames
except:
    video_path = ""
    print("ffmpeg not installed?")
```

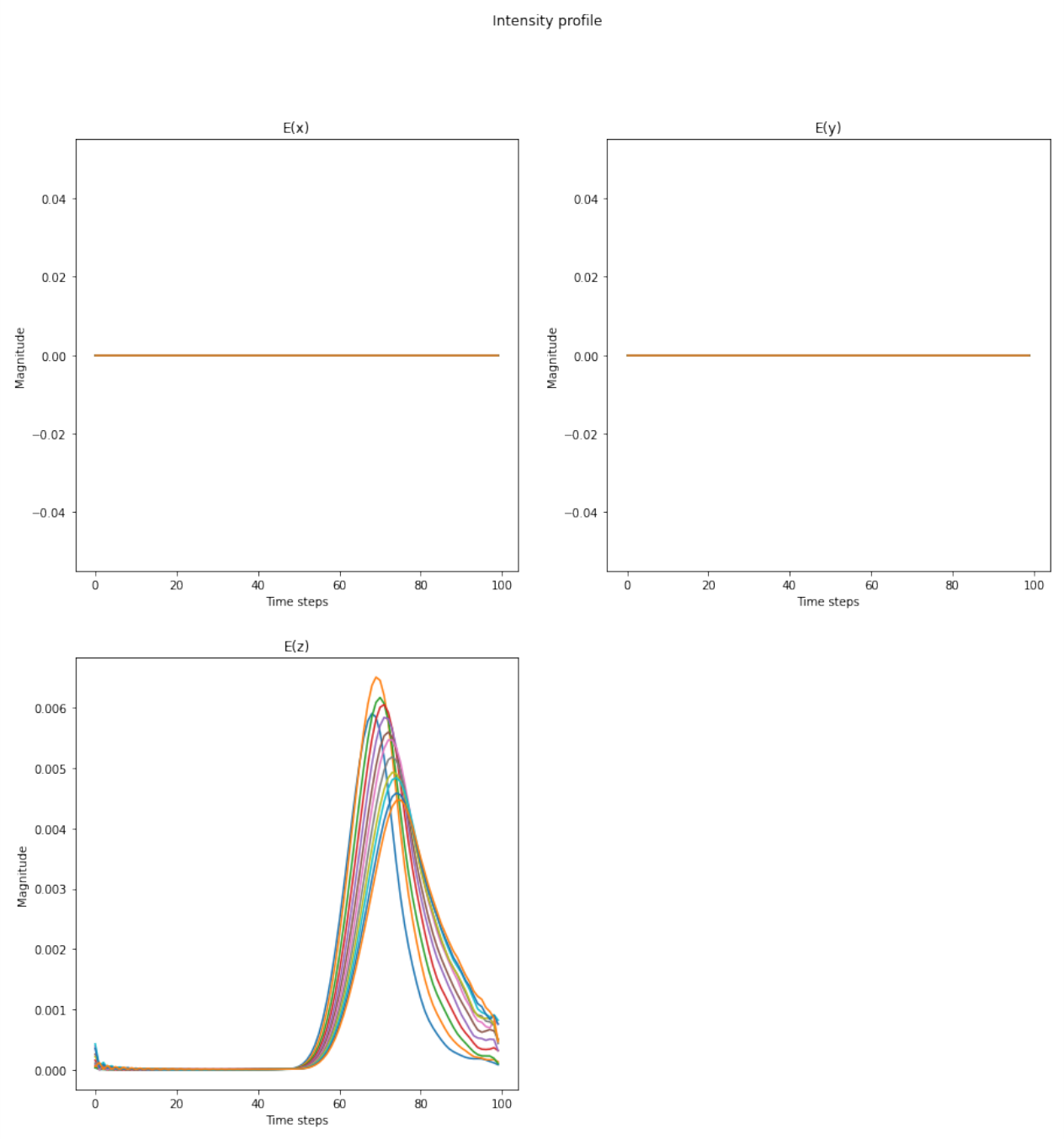
```
ffmpeg not installed?
```

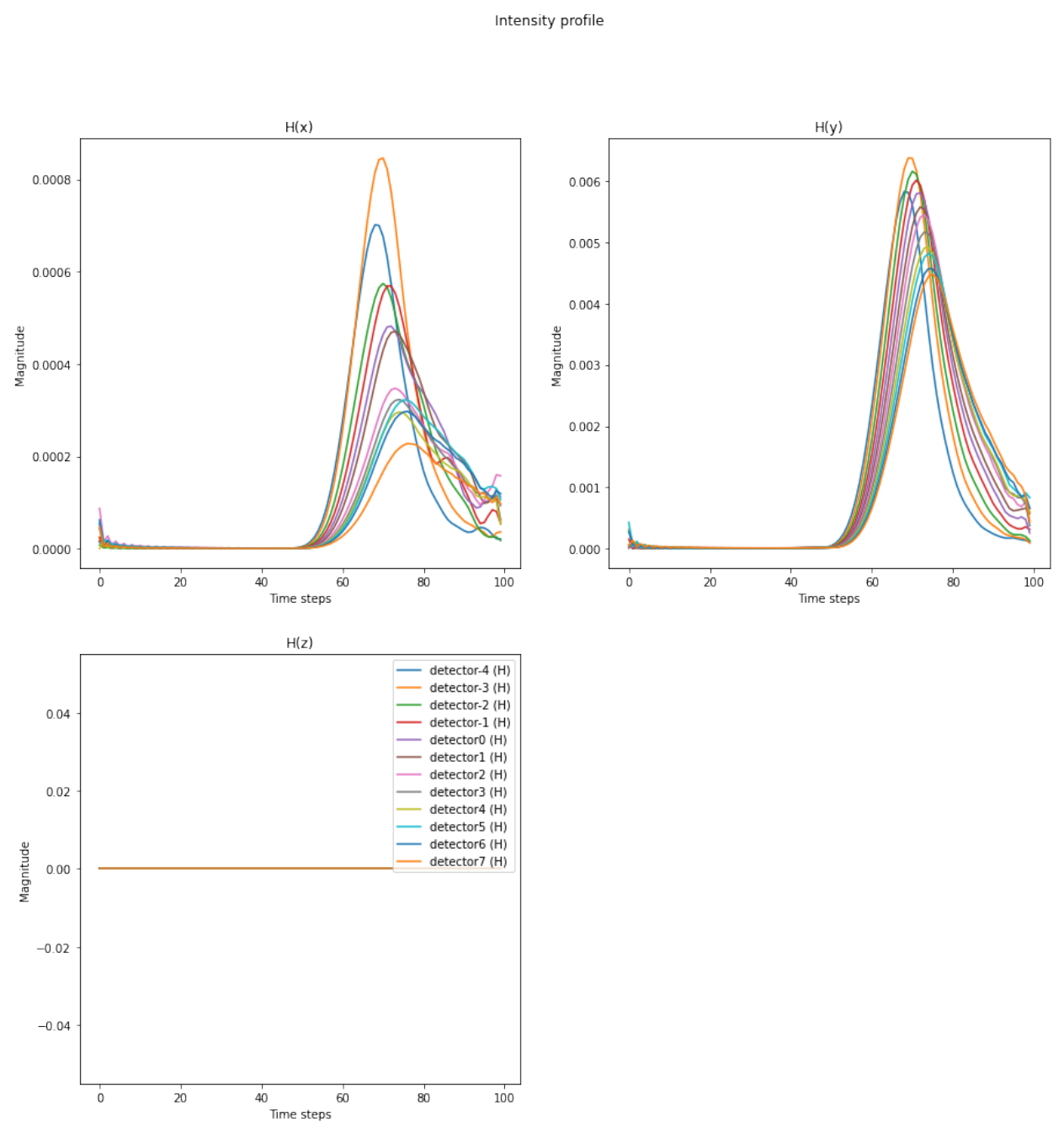
[9]:
```
if video_path:
    from IPython.display import Video
    display(Video(video_path, embed=True))
```

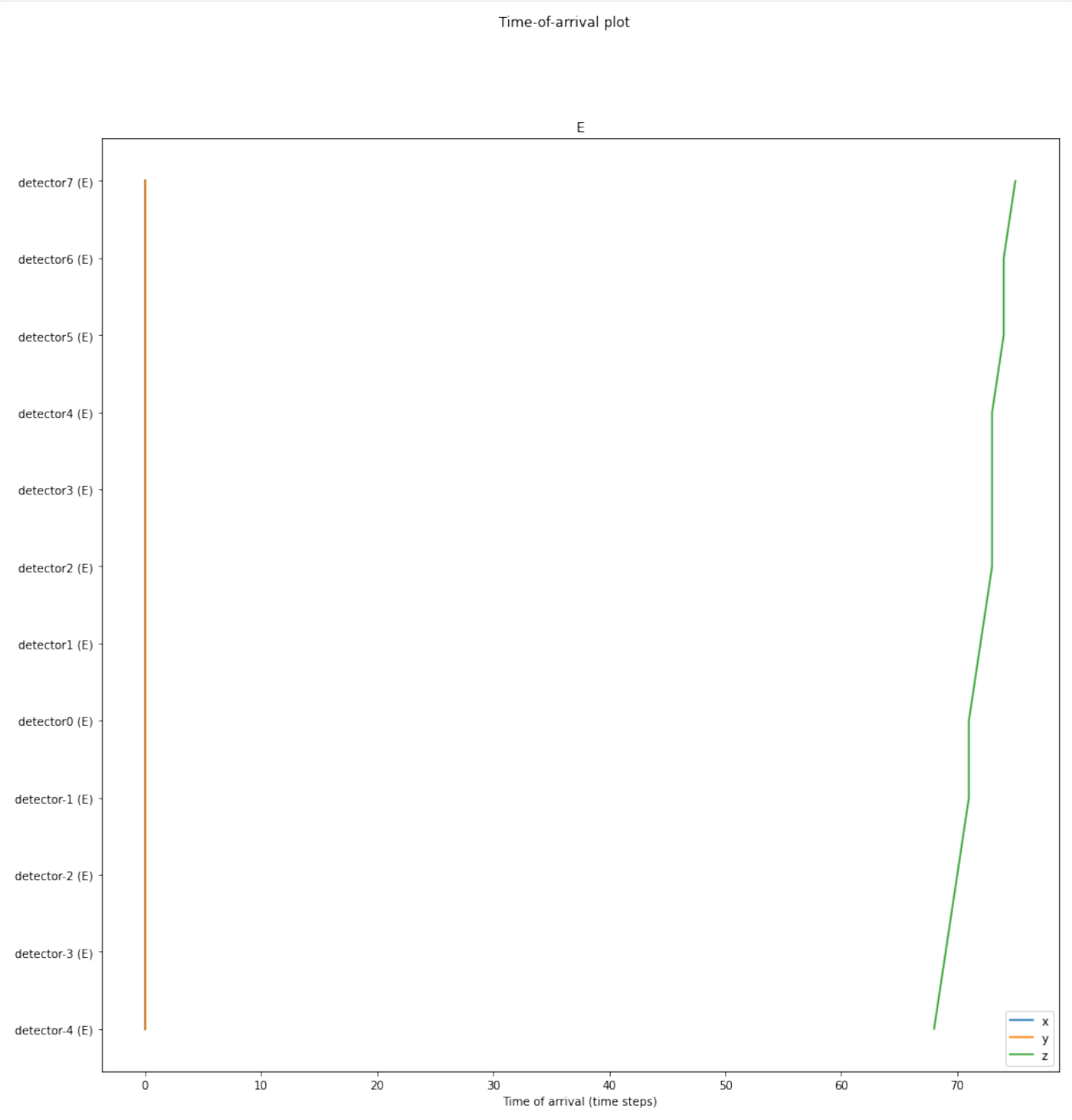
### Analyse

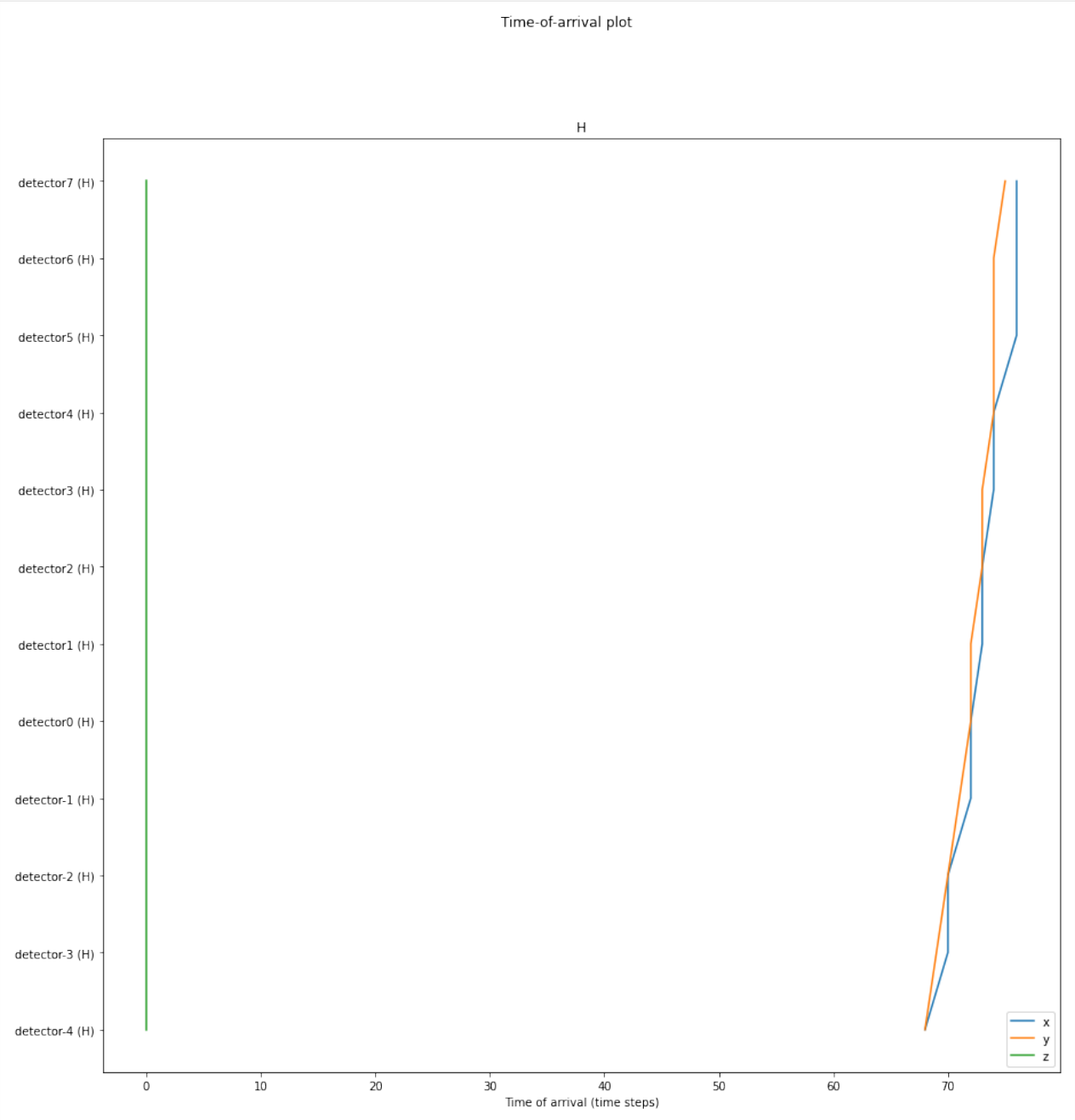
analysing data stored by above simulation to find intensity profile and time-of-arrival plot

[10]:
```
dic = np.load(os.path.join(simfolder, "detector_readings.npz"))
import warnings; warnings.filterwarnings("ignore") # TODO: fix plot_detection to prevent␣
↪warnings
fdtd.plot_detection(dic)
```


Intensity profile
E(x)
E(y)
E(z)
Magnitude
Time steps
0.04
0.02
0.00
−0.02
−0.04
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
20
40
60
80
100


Intensity profile
H(x)
H(y)
H(z)
Magnitude
Time steps
detector-4 (H)
detector-3 (H)
detector-2 (H)
detector-1 (H)
detector0 (H)
detector1 (H)
detector2 (H)
detector3 (H)
detector4 (H)
detector5 (H)
detector6 (H)
detector7 (H)


Time-of-arrival plot
E
detector7 (E)
detector6 (E)
detector5 (E)
detector4 (E)
detector3 (E)
detector2 (E)
detector1 (E)
detector0 (E)
detector-1 (E)
detector-2 (E)
detector-3 (E)
detector-4 (E)
0
10
20
30
40
50
60
70
Time of arrival (time steps)
x
y
z




## 1.2 fdtd package

### 1.2.1 backend module

Selects the backend for the fdtd-package.

The *fdtd* library allows to choose a backend. The `numpy` backend is the default one, but there are also several additional PyTorch backends:

- `numpy` (defaults to float64 arrays)

- `torch` (defaults to float64 tensors)
- `torch.float32`
- `torch.float64`
- `torch.cuda` (defaults to float64 tensors)
- `torch.cuda.float32`
- `torch.cuda.float64`

For example, this is how to choose the *"torch"* backend:

```
fdtd.set_backend("torch")
```

In general, the `numpy` backend is preferred for standard CPU calculations with *"float64"* precision. In general, `float64` precision is always preferred over `float32` for FDTD simulations, however, `float32` might give a significant performance boost.

The `cuda` backends are only available for computers with a GPU.

**class** fdtd.backend.**Backend**

Bases: `object`

Backend Base Class

**pi = 3.141592653589793**

**class** fdtd.backend.**NumpyBackend**

Bases: *fdtd.backend.Backend*

Numpy Backend

**static arange**([*start*], *stop*[, *step*], *dtype=None*, ***, *like=None*)

create a range of values

**static array**()

create an array from an array-like sequence

**static asarray**(*a*, *dtype=None*, *order=None*, ***, *like=None*)

create an array

**static bmm**(*arr1*, *arr2*)

batch matrix multiply two arrays

**static broadcast_arrays**(**args*, *subok=False*)

broadcast arrays

**static broadcast_to**(*array*, *shape*, *subok=False*)

broadcast array into shape

**cos = <ufunc 'cos'>**

cosine of all elements in array

**divide = <ufunc 'true_divide'>**

**exp = <ufunc 'exp'>**

exponential of all elements in array

**static fft**(*a*, *n=None*, *axis=- 1*, *norm=None*)

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

**Parameters**

- **a** (*array_like*) – Input array, can be complex.
- **n** (*int, optional*) – Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.
- **axis** (*int, optional*) – Axis over which to compute the FFT. If not given, the last axis is used.
- **norm** (*{"backward", "ortho", "forward"}, optional*) – New in version 1.10.0.

  Normalization mode (see *numpy.fft*). Default is "backward". Indicates which direction of the forward/backward pair of transforms is scaled and with what normalization factor.

  New in version 1.20.0: The "backward", "forward" values were added.

**Returns** **out** – The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

**Return type** complex ndarray

**Raises** **IndexError** – If *axis* is not a valid axis of *a*.

**See also:**

**numpy.fft** for definition of the DFT and conventions used.

**ifft** The inverse of *fft*.

**fft2** The two-dimensional FFT.

**fftn** The *n*-dimensional FFT.

**rfftn** The *n*-dimensional FFT of real input.

*fftfreq* Frequency bins for given FFT parameters.

### Notes

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when *n* is a power of 2, and the transform is therefore most efficient for these sizes.

The DFT is defined, with the conventions used in this implementation, in the documentation for the *numpy.fft* module.

### References

### Examples

```
>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
array([-2.33486982e-16+1.14423775e-17j,  8.00000000e+00-1.25557246e-15j,
        2.33486982e-16+2.33486982e-16j,  0.00000000e+00+1.22464680e-16j,
       -1.14423775e-17+2.33486982e-16j,  0.00000000e+00+5.20784380e-16j,
        1.14423775e-17+1.14423775e-17j,  0.00000000e+00+1.22464680e-16j])
```

In this example, real input has an FFT which is Hermitian, i.e., symmetric in the real part and anti-symmetric in the imaginary part, as described in the *numpy.fft* documentation:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(256)
>>> sp = np.fft.fft(np.sin(t))
>>> freq = np.fft.fftfreq(t.shape[-1])
>>> plt.plot(freq, sp.real, freq, sp.imag)
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at␣
↪0x...>]
>>> plt.show()
```

**static fftfreq**(*n*, *d=1.0*)

Return the Discrete Fourier Transform sample frequencies.

The returned float array $f$ contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length $n$ and a sample spacing $d$:

```
f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)   if n is odd
```

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar, optional*) – Sample spacing (inverse of the sampling rate). Defaults to 1.

**Returns** **f** – Array of length $n$ containing the sample frequencies.

**Return type** ndarray

### Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0.  ,  1.25,  2.5 , ..., -3.75, -2.5 , -1.25])
```

**float**

alias of `numpy.float64`

**int**

alias of `numpy.int64`

**static is_array**(*arr*)

check if an object is an array

**static linspace**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*, *axis=0*)

create a linearly spaced array between two points

**static max**(*a*, *axis=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

max element in array

**static numpy**()

convert the array to numpy array

static ones(*shape*, *dtype=None*, *order='C'*, ***, *like=None*)
create an array filled with ones

static pad(*array*, *pad_width*, *mode='constant'*, ***kwargs*)
Pad an array.

**Parameters**

- **array** (*array_like of rank N*) – The array to pad.
- **pad_width** (*{sequence, array_like, int}*) – Number of values padded to the edges of each axis. ((before_1, after_1), … (before_N, after_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes.
- **mode** (*str or function, optional*) – One of the following string values or a user supplied function.

  **'constant' (default)** Pads with a constant value.

  **'edge'** Pads with the edge values of array.

  **'linear_ramp'** Pads with the linear ramp between end_value and the array edge value.

  **'maximum'** Pads with the maximum value of all or part of the vector along each axis.

  **'mean'** Pads with the mean value of all or part of the vector along each axis.

  **'median'** Pads with the median value of all or part of the vector along each axis.

  **'minimum'** Pads with the minimum value of all or part of the vector along each axis.

  **'reflect'** Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

  **'symmetric'** Pads with the reflection of the vector mirrored along the edge of the array.

  **'wrap'** Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

  **'empty'** Pads with undefined values.

  New in version 1.17.

  **<function>** Padding function, see Notes.
- **stat_length** (*sequence or int, optional*) – Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value.

  ((before_1, after_1), … (before_N, after_N)) unique statistic lengths for each axis.

  ((before, after),) yields same before and after statistic lengths for each axis.

  (stat_length,) or int is a shortcut for before = after = statistic length for all axes.

  Default is `None`, to use the entire axis.
- **constant_values** (*sequence or scalar, optional*) – Used in 'constant'. The values to set the padded values for each axis.

  `((before_1, after_1), ... (before_N, after_N))` unique pad constants for each axis.

  `((before, after),)` yields same before and after constants for each axis.

  `(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes.

Default is 0.

- **end_values** (*sequence or scalar, optional*) – Used in 'linear_ramp'. The values used for the ending value of the linear_ramp and that will form the edge of the padded array.

  `((before_1, after_1), ... (before_N, after_N))` unique end values for each axis.

  `((before, after),)` yields same before and after end values for each axis.

  `(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes.

  Default is 0.

- **reflect_type** (*{'even', 'odd'}, optional*) – Used in 'reflect', and 'symmetric'. The 'even' style is the default with an unaltered reflection around the edge value. For the 'odd' style, the extended part of the array is created by subtracting the reflected values from two times the edge value.

**Returns** **pad** – Padded array of rank equal to *array* with shape increased according to *pad_width*.

**Return type** ndarray

### Notes

New in version 1.7.0.

For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should modify a rank 1 array in-place. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, kwargs)
```

where

**vector** [ndarray] A rank 1 array already padded with zeros. Padded values are vector[:iaxis_pad_width[0]] and vector[-iaxis_pad_width[1]:].

**iaxis_pad_width** [tuple] A 2-tuple of ints, iaxis_pad_width[0] represents the number of values padded at the beginning of vector where iaxis_pad_width[1] represents the number of values padded at the end of vector.

**iaxis** [int] The axis currently being calculated.

**kwargs** [dict] Any keyword arguments the function requires.

### Examples

```
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2, 3), 'constant', constant_values=(4, 6))
array([4, 4, 1, ..., 6, 6, 6])
```

```
>>> np.pad(a, (2, 3), 'edge')
array([1, 1, 1, ..., 5, 5, 5])
```

```
>>> np.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
```

```
>>> np.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```
>>> np.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> np.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> a = [[1, 2], [3, 4]]
>>> np.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [3, 3, 3, 4, 3, 3, 3],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1]])
```

```
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
>>> np.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

```
>>> np.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
>>> np.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
>>> np.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```
>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
>>> a = np.arange(6)
>>> a = a.reshape((2, 3))
>>> np.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10,  0,  1,  2, 10, 10],
       [10, 10,  3,  4,  5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
>>> np.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
```

(continues on next page)

(continued from previous page)

```
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100,   0,   1,   2, 100, 100],
       [100, 100,   3,   4,   5, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])
```

**static reshape**(*a*, *newshape*, *order='C'*)
reshape array into given shape

**sin = <ufunc 'sin'>**
sine of all elements in array

**static squeeze**(*a*, *axis=None*)
remove dim-1 dimensions

**static stack**(*arrays*, *axis=0*, *out=None*)
stack multiple arrays

**static sum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)
sum elements in array

**static transpose**(*a*, *axes=None*)
transpose array by flipping two dimensions

**static zeros**(*shape*, *dtype=float*, *order='C'*, ***, *like=None*)
create an array filled with zeros

**static zeros_like**(*a*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)
create an array filled with zeros

fdtd.backend.**set_backend**(*name: str*)
Set the backend for the FDTD simulations

This function monkeypatches the backend object by changing its class. This way, all methods of the backend object will be replaced.

**Parameters** **name** – name of the backend. Allowed backend names: - `numpy` (defaults to float64 arrays) - `torch` (defaults to float64 tensors) - `torch.float32` - `torch.float64` - `torch.cuda` (defaults to float64 tensors) - `torch.cuda.float32` - `torch.cuda.float64`

### 1.2.2 boundaries module

Boundaries for the FDTD Grid.

Available Boundaries:

- PeriodicBoundary
- PML

**class** fdtd.boundaries.**Boundary**(*name: Optional[str] = None*)
Bases: `object`

an FDTD Boundary [base class]

**__init__**(*name: Optional[str] = None*)
Create a boundary

**Parameters** **name** – name of the boundary

**update_E**()
Update electric field of the grid

**Note:** this method is called *after* the grid fields are updated

**update_H**()
Update magnetic field of the grid

**Note:** this method is called *after* the grid fields are updated

**update_phi_E**()
Update convolution [phi_E]

**Note:** this method is called *before* the electric field is updated

**update_phi_H**()
Update convolution [phi_H]

**Note:** this method is called *before* the magnetic field is updated

fdtd.boundaries.**DomainBorderPML**(*grid*, *border_cells=5*)
Some problem setups require a layer of PML all the way around the problem. This is a convenience function to add such a layer to an existing grid. Caution: Alters grid in-place.

**class** fdtd.boundaries.**PML**(*a: float = 1e-08*, *name: Optional[str] = None*)
Bases: *fdtd.boundaries.Boundary*

A perfectly matched layer (PML)

a PML is an impedence-matched area at the boundary of the grid for which all fields incident perpendicular to the area are absorbed without reflection.

**Note:** Registering a PML to the grid will monkeypatch the PML to become one of its subclasses: `_PMLXlow`, `_PMLYlow` or `_PMLZlow`, `_PMLXhigh`, `_PMLYhigh`, `_PMLZhigh` depending on the position in the grid.

**__init__**(*a: float = 1e-08*, *name: Optional[str] = None*)
Perfectly Matched Layer

**Parameters**

- **a** – stability parameter
- **name** – name of the PML

**update_E**()
Update electric field of the grid

**Note:** this method is called *after* the electric field is updated

**update_H**()
Update magnetic field of the grid

**Note:** this method is called *after* the magnetic field is updated

**update_phi_E**()
Update convolution [phi_E]

**Note:** this method is called *before* the electric field is updated

**update_phi_H**()
Update convolution [phi_H]

**Note:** this method is called *before* the magnetic field is updated

**class** fdtd.boundaries.**PeriodicBoundary**(*name: Optional[str] = None*)
Bases: *fdtd.boundaries.Boundary*

An FDTD Periodic Boundary

**Note:** Registering a periodic boundary to the grid will change the periodic boundary in one of its subclasses: `_PeriodicBoundaryX`, `_PeriodicBoundaryY` or `_PeriodicBoundaryY`, depending on the position in the grid.

### 1.2.3 detectors module

Detectors for the FDTD Grid.

Available Detectors:

- LineDetector

**class** fdtd.detectors.**BlockDetector**(*name=None*)
Bases: `object`

A detector along a block in the FDTD grid

**__init__**(*name=None*)
Create a block detector

**Parameters** **name** – name of the Detector

**detect_E**()
detect the electric field at a certain location in the grid

**detect_H**()
detect the magnetic field at a certain location in the grid

**detector_values**()
outputs what detector detects

**class** fdtd.detectors.**CurrentDetector**(*name=None*)
Bases: `object`

A current detector.

**__init__**(*name=None*)
Create a block detector

**Parameters** **name** – name of the Detector

**detect_E**()
detect the electric field at a certain location in the grid

**detect_H**()

**detector_values**()
outputs what detector detects

**single_point_current**(*px*, *py*, *pz*)
Only Z-polarized for now. Can probably do a cross product to get arbitrary polarizations

X—->

TODO: FIXME: IMPORTANT: material magnetic permeability? find test cases!

Implements the first correction from [Fang 1994] (two cells are spatially averaged to account for Yee cell half-step inaccuracies), but not the second one (minor loss of accuracy).

Jiayuan Fang, Danwei Xue. Precautions in the calculation of impedance in FDTD computations. Proceedings of IEEE Antennas and Propagation Society International Symposium and URSI National Radio Science Meeting, vol. 3, 1994, p. 1814–7 vol.3. https://doi.org/10.1109/APS.1994.408185.

Luebbers RJ, Langdon HS. A simple feed model that reduces time steps needed for FDTD antenna and microstrip calculations. IEEE Trans Antennas Propagat 1996;44:1000–5. https://doi.org/10.1109/8.504308.

**class** fdtd.detectors.**LineDetector**(*name=None*)
Bases: object

A detector along a line in the FDTD grid

**__init__**(*name=None*)
Create a line detector

**Parameters** **name** – name of the Detector

**detect_E**()
detect the electric field at a certain location in the grid

**detect_H**()
detect the magnetic field at a certain location in the grid

**detector_values**()
outputs what detector detects

### 1.2.4 grid module

The FDTD Grid

The grid is the core of the FDTD Library. It is where everything comes together and where the biggest part of the calculations are done.

**class** fdtd.grid.**Grid**(*shape: Tuple[numbers.Number, numbers.Number, numbers.Number]*, *grid_spacing: float = 1.55e-07*, *permittivity: float = 1.0*, *permeability: float = 1.0*, *courant_number: Optional[float] = None*)
Bases: object

The FDTD Grid

The grid is the core of the FDTD Library. It is where everything comes together and where the biggest part of the calculations are done.

**__init__**(*shape: Tuple[numbers.Number, numbers.Number, numbers.Number]*, *grid_spacing: float = 1.55e-07*, *permittivity: float = 1.0*, *permeability: float = 1.0*, *courant_number: Optional[float] = None*)

**Parameters**

- **shape** – shape of the FDTD grid.
- **grid_spacing** – distance between the grid cells.
- **permittivity** – the relative permittivity of the background.
- **permeability** – the relative permeability of the background.
- **courant_number** – the courant number of the FDTD simulation. Defaults to the inverse of the square root of the number of dimensions > 1 (optimal value). The timestep of the simulation will be derived from this number using the CFL-condition.

**add_boundary**(*name*, *boundary*)
add a boundary to the grid

**add_detector**(*name*, *detector*)
add a detector to the grid

**add_object**(*name*, *obj*)
add an object to the grid

**add_source**(*name*, *source*)
add a source to the grid

**generate_video**(*delete_frames=False*)
Compiles frames into a video

These framed should be saved through `fdtd.Grid.visualize(save=True)` while having `fdtd.Grid.save_simulation()` enabled.

**Parameters** **delete_frames** (*optional, bool*) – delete stored frames after conversion to video.

**Returns** the filename of the generated video.

**Note:** this function requires `ffmpeg` to be available in your path.

**reset**()
reset the grid by setting all fields to zero

**run**(*total_time: numbers.Number*, *progress_bar: bool = True*)
run an FDTD simulation.

**Parameters**

- **total_time** – the total time for the simulation to run.
- **progress_bar** – choose to show a progress bar during simulation

save_data()
Saves readings from all detectors in the grid into a numpy zip file. Each detector is stored in separate arrays. Electric and magnetic field field readings of each detector are also stored separately with suffix " (E)" and " (H)" (Example: ['detector0 (E)', 'detector0 (H)']). Therefore, the numpy zip file contains arrays twice the number of detectors. REQUIRES 'fdtd.Grid.save_simulation()' to be run before this function.

Parameters: None

save_simulation(*sim_name=None*)
Creates a folder and initializes environment to store simulation or related details. saveSimulation() needs to be run before running any function that stores data (generate_video(), save_data()).

**Parameters:-** (optional) sim_name (string): Preferred name for simulation

property shape: Tuple[int, int, int]
get the shape of the FDTD grid

step()
do a single FDTD step by first updating the electric field and then updating the magnetic field

property time_passed: float
get the total time passed

update_E()
update the electric field by using the curl of the magnetic field

update_H()
update the magnetic field by using the curl of the electric field

visualize(*x=None*, *y=None*, *z=None*, *cmap='Blues'*, *pbcolor='C3'*, *pmlcolor=(0, 0, 0, 0.1)*, *objcolor=(1, 0, 0, 0.1)*, *srccolor='C0'*, *detcolor='C2'*, *norm='linear'*, *show=False*, *animate=False*, *index=None*, *save=False*, *folder=None*)
visualize a projection of the grid and the optical energy inside the grid

**Parameters**

- **x** – the x-value to make the yz-projection (leave None if using different projection)
- **y** – the y-value to make the zx-projection (leave None if using different projection)
- **z** – the z-value to make the xy-projection (leave None if using different projection)
- **cmap** – the colormap to visualize the energy in the grid
- **pbcolor** – the color to visualize the periodic boundaries
- **pmlcolor** – the color to visualize the PML
- **objcolor** – the color to visualize the objects in the grid
- **srccolor** – the color to visualize the sources in the grid
- **detcolor** – the color to visualize the detectors in the grid
- **norm** – how to normalize the grid_energy color map ('linear' or 'log').
- **show** – call pyplot.show() at the end of the function
- **animate** – see frame by frame state of grid during simulation
- **index** – index for each frame of animation (typically a loop variable is passed)
- **save** – save frames in a folder
- **folder** – path to folder to save frames

**property x: int**
get the number of grid cells in the x-direction

**property y: int**
get the number of grid cells in the y-direction

**property z: int**
get the number of grid cells in the y-direction

fdtd.grid.**curl_E**(*E: numpy.ndarray*) → numpy.ndarray
Transforms an E-type field into an H-type field by performing a curl operation

**Parameters** **E** – Electric field to take the curl of (E-type field located on the edges of the grid cell [integer gridpoints])

**Returns** The curl of E (H-type field located on the faces of the grid [half-integer grid points])

fdtd.grid.**curl_H**(*H: numpy.ndarray*) → numpy.ndarray
Transforms an H-type field into an E-type field by performing a curl operation

**Parameters** **H** – Magnetic field to take the curl of (H-type field located on half-integer grid points)

**Returns** The curl of H (E-type field located on the edges of the grid [integer grid points])

**class** fdtd.grid.**d_**
Bases: object

Just a convenience function for indexing polarizations of arrays.

**X = 0**

**Y = 1**

**Z = 2**

### 1.2.5 objects module

The objects to place in the grid.

Objects define all the regions in the grid with a modified update equation, such as for example regions with anisotropic permittivity etc.

**Available Objects:**

- Object
- AnisotropicObject

**class** fdtd.objects.**AbsorbingObject**(*permittivity: numpy.ndarray*, *conductivity: numpy.ndarray*, *name: Optional[str] = None*)
Bases: *fdtd.objects.Object*

An absorbing object takes conductivity into account

**__init__**(*permittivity: numpy.ndarray*, *conductivity: numpy.ndarray*, *name: Optional[str] = None*)

**Parameters**

- **permittivity** – permittivity tensor
- **conductivity** – conductivity tensor (will introduce the loss)
- **name** – name of the object (will become available as attribute to the grid)

**update_E**(*curl_H*)
custom update equations for inside the absorbing object

**Parameters** **curl_H** – the curl of magnetic field in the grid.

**update_H**(*curl_E*)
custom update equations for inside the absorbing object

**Parameters** **curl_E** – the curl of electric field in the grid.

**class** fdtd.objects.**AnisotropicObject**(*permittivity: numpy.ndarray*, *name: Optional[str] = None*)
Bases: *fdtd.objects.Object*

An object with anisotropic permittivity tensor

**update_E**(*curl_H*)
custom update equations for inside the anisotropic object

**Parameters** **curl_H** – the curl of magnetic field in the grid.

**update_H**(*curl_E*)
custom update equations for inside the anisotropic object

**Parameters** **curl_E** – the curl of electric field in the grid.

**class** fdtd.objects.**Object**(*permittivity: numpy.ndarray*, *name: Optional[str] = None*)
Bases: `object`

An object to place in the grid

**__init__**(*permittivity: numpy.ndarray*, *name: Optional[str] = None*)

**Parameters**

- **permittivity** – permittivity tensor
- **name** – name of the object (will become available as attribute to the grid)

**update_E**(*curl_H*)
custom update equations for inside the object

**Parameters** **curl_H** – the curl of magnetic field in the grid.

**update_H**(*curl_E*)
custom update equations for inside the object

**Parameters** **curl_E** – the curl of electric field in the grid.

### 1.2.6 sources module

Sources are objects that inject the fields into the grid.

Available sources:

- PointSource
- LineSource

**class** fdtd.sources.**LineSource**(*period: numbers.Number = 15*, *amplitude: float = 1.0*, *phase_shift: float = 0.0*, *name: Optional[str] = None*, *pulse: bool = False*, *cycle: int = 5*, *hanning_dt: float = 10.0*)
Bases: `object`

A source along a line in the FDTD grid

__init__(*period: numbers.Number = 15*, *amplitude: float = 1.0*, *phase_shift: float = 0.0*, *name: Optional[str] = None*, *pulse: bool = False*, *cycle: int = 5*, *hanning_dt: float = 10.0*)
Create a LineSource with a gaussian profile

**Parameters**

- **period** – The period of the source. The period can be specified as integer [timesteps] or as float [seconds]
- **amplitude** – The amplitude of the source in simulation units
- **phase_shift** – The phase offset of the source.
- **pulse** – Set True to use a Hanning window pulse instead of continuous wavefunction.
- **cycle** – cycles for Hanning window pulse.
- **hanning_dt** – timestep used for Hanning window pulse width (optional).

update_E()
Add the source to the electric field

update_H()
Add the source to the magnetic field

class fdtd.sources.PlaneSource(*period: numbers.Number = 15*, *amplitude: float = 1.0*, *phase_shift: float = 0.0*, *name: Optional[str] = None*)
Bases: object

A source along a plane in the FDTD grid

__init__(*period: numbers.Number = 15*, *amplitude: float = 1.0*, *phase_shift: float = 0.0*, *name: Optional[str] = None*)
Create a LineSource with a gaussian profile

**Parameters**

- **period** – The period of the source. The period can be specified as integer [timesteps] or as float [seconds]
- **amplitude** – The amplitude of the source in simulation units
- **phase_shift** – The phase offset of the source.

update_E()
Add the source to the electric field

update_H()
Add the source to the magnetic field

class fdtd.sources.PointSource(*period: numbers.Number = 15*, *amplitude: float = 1.0*, *phase_shift: float = 0.0*, *name: Optional[str] = None*, *pulse: bool = False*, *cycle: int = 5*, *hanning_dt: float = 10.0*)
Bases: object

A source placed at a single point (grid cell) in the grid

__init__(*period: numbers.Number = 15*, *amplitude: float = 1.0*, *phase_shift: float = 0.0*, *name: Optional[str] = None*, *pulse: bool = False*, *cycle: int = 5*, *hanning_dt: float = 10.0*)
Create a LineSource with a gaussian profile

**Parameters**

- **period** – The period of the source. The period can be specified as integer [timesteps] or as float [seconds]

- **amplitude** – The electric field amplitude in simulation units
- **phase_shift** – The phase offset of the source.
- **name** – name of the source.
- **pulse** – Set True to use a Hanning window pulse instead of continuous wavefunction.
- **cycle** – cycles for Hanning window pulse.
- **hanning_dt** – timestep used for Hanning window pulse width (optional).

**update_E**()
Add the source to the electric field

**update_H**()
Add the source to the magnetic field

class fdtd.sources.**SoftArbitraryPointSource**(*waveform_array: numpy.ndarray*, *name: Optional[str] = None*, *impedance: float = 0.0*)

Bases: object

A source placed at a single point (grid cell) in the grid. This source is special: it's both a source and a detector.

Unlike the other sources, the input is a voltage, not an electric field. (really? why? should we convert back and forth?)

For electrical measurements I've only needed a single-index source, so I don't know how the volume/line sources above work. We want the FFT function to operate over any detector. Maybe all sources should take an arbitary waveform argument?

Each index in the *waveform* array represents 1 value at a timestep.

There are many different *geometries* of "equivalent sources". The detector/source paradigm used in /fdtd might perhaps not correspond to this in an ideal fashion.

It's not intuitively clear to me what a "soft" source would imply in the optical case, or what impedance even means for a laser.

/fdtd/ seems to have found primary use in optical circles, so the default Z should probably be 0.

"Whilst established for microwaves and electrical circuits, this concept has only very recently been observed in the optical domain, yet is not well defined or understood."[1]

[1]: Optical impedance of metallic nano-structures, M. Mazilu and K. Dholakia https://doi.org/10.1364/OE.14.007709

[2]: http://www.gwoptics.org/learn/02_Plane_waves/01_Fabry_Perot_cavity/02_Impedance_matched.php

//-

**__init__**(*waveform_array: numpy.ndarray*, *name: Optional[str] = None*, *impedance: float = 0.0*)
Create

**Parameters** **waveform_array** –

**update_E**()

**update_H**()

CHAPTER

TWO

# INSTALLATION

The `fdtd`-library can be installed with `pip`:

```
pip install fdtd
```

CHAPTER

# THREE

# DEPENDENCIES

- python 3.6+
- numpy
- matplotlib
- tqdm
- pytorch (optional)

CHAPTER

# FOUR

# QUICK INTRO

The `fdtd` library is simply imported as follows:

```
import fdtd
```

## 4.1 Setting the backend

*fdtd.backend*

The `fdtd` library allows to choose a backend. The `"numpy"` backend is the default one, but there are also several additional PyTorch backends:

- `numpy` (defaults to float64 arrays)
- `torch` (defaults to float64 tensors)
- `torch.float32`
- `torch.float64`
- `torch.cuda` (defaults to float64 tensors)
- `torch.cuda.float32`
- `torch.cuda.float64`

For example, this is how to choose the `torch` backend:

```
fdtd.set_backend("torch")
```

In general, the `numpy` backend is preferred for standard CPU calculations with `float64` precision. In general, `float64` precision is always preferred over `float32` for FDTD simulations, however, `float32` might give a significant performance boost.

The `cuda` backends are only available for computers with a GPU.

## 4.2 The FDTD-grid

*fdtd.grid*

The FDTD grid defines the simulation region.

```
# signature
fdtd.Grid(
    shape: Tuple[Number, Number, Number],
    grid_spacing: float = 155e-9,
    permittivity: float = 1.0,
    permeability: float = 1.0,
    courant_number: float = None,
)
```

A grid is defined by its `shape`, which is just a 3D tuple of `Number`-types (integers or floats). If the shape is given in floats, it denotes the width, height and length of the grid in meters. If the shape is given in integers, it denotes the width, height and length of the grid in terms of the `grid_spacing`. Internally, these numbers will be translated to three integers: `grid.Nx`, `grid.Ny` and `grid.Nz`.

A `grid_spacing` can be given. For stability reasons, it is recommended to choose a grid spacing that is at least 10 times smaller than the _smallest_ wavelength in the grid. This means that for a grid containing a source with wavelength `1550nm` and a material with refractive index of `3.1`, the recommended minimum `grid_spacing` turns out to be `50pm`

For the `permittivity` and `permeability` floats or arrays with the following shapes

- `(grid.Nx, grid.Ny, grid.Nz)`
- or `(grid.Nx, grid.Ny, grid.Nz, 1)`
- or `(grid.Nx, grid.Ny, grid.Nz, 3)`

are expected. In the last case, the shape implies the possibility for different permittivity for each of the major axes (so-called _uniaxial_ or _biaxial_ materials). Internally, these variables will be converted (for performance reasons) to their inverses `grid.inverse_permittivity` array and a `grid.inverse_permeability` array of shape `(grid.Nx, grid.Ny, grid.Nz, 3)`. It is possible to change those arrays after making the grid.

Finally, the `courant_number` of the grid determines the relation between the `time_step` of the simulation and the `grid_spacing` of the grid. If not given, it is chosen to be the maximum number allowed by the Courant-Friedrichs-Lewy Condition: `1` for `1D` simulations, `1/2` for `2D` simulations and `1/3` for `3D` simulations (the dimensionality will be derived by the shape of the grid). For stability reasons, it is recommended not to change this value.

```
grid = fdtd.Grid(
    shape = (25e-6, 15e-6, 1), # 25um x 15um x 1 (grid_spacing) --> 2D FDTD
)
print(grid)
```

```
Grid(shape=(161,97,1), grid_spacing=1.55e-07, courant_number=0.70)
```

## 4.3 Objects

*fdtd.objects*

An other option to locally change the `permittivity` or `permeability` in the grid is to add an `Object` to the grid.

```
# signature
fdtd.Object(
    permittivity: Tensorlike,
    name: str = None
)
```

An object defines a part of the grid with modified update equations, allowing to introduce for example absorbing materials or biaxial materials for which mixing between the axes are present through `Pockels coefficients` or many more. In this case we'll make an object with a different `permittivity` than the grid it is in.

Just like for the grid, the `Object` expects a `permittivity` to be a floats or an array of the following possible shapes

- `(obj.Nx, obj.Ny, obj.Nz)`
- or `(obj.Nx, obj.Ny, obj.Nz, 1)`
- or `(obj.Nx, obj.Ny, obj.Nz, 3)`

Note that the values `obj.Nx`, `obj.Ny` and `obj.Nz` are not given to the object constructor. They are in stead derived from its placing in the grid:

```
grid[11:32, 30:84, 0] = fdtd.Object(permittivity=1.7**2, name="object")
```

Several things happen here. First of all, the object is given the space `[11:32, 30:84, 0]` in the grid. Because it is given this space, the object's `Nx`, `Ny` and `Nz` are automatically set. Furthermore, by supplying a name to the object, this name will become available in the grid:

```
print(grid.object)
```

```
Object(name='object')
    @ x=11:32, y=30:84, z=0:1
```

A second object can be added to the grid:

```
grid[13e-6:18e-6, 5e-6:8e-6, 0] = fdtd.Object(permittivity=1.5**2)
```

Here, a slice with floating point numbers was chosen. These floats will be replaced by integer `Nx`, `Ny` and `Nz` during the registration of the object. Since the object did not receive a name, the object won't be available as an attribute of the grid. However, it is still available via the `grid.objects` list:

```
print(grid.objects)
```

```
[Object(name='object'), Object(name=None)]
```

This list stores all objects (i.e. of type `fdtd.Object`) in the order that they were added to the grid.

## 4.4 Sources

*fdtd.sources*

Similarly as to adding an object to the grid, an `fdtd.LineSource` can also be added:

```
# signature
fdtd.LineSource(
    period: Number = 15, # timesteps or seconds
    amplitude: float = 1.0,
    phase_shift: float = 0.0,
    name: str = None,
)
```

And also just like an `fdtd.Object`, an `fdtd.LineSource` size is defined by its placement on the grid:

```
grid[7.5e-6:8.0e-6, 11.8e-6:13.0e-6, 0] = fdtd.LineSource(
    period = 1550e-9 / (3e8), name="source"
)
```

However, it is important to note that in this case a `LineSource` is added to the grid, i.e. the source spans the diagonal of the cube defined by the slices. Internally, these slices will be converted into lists to ensure this behavior:

```
print(grid.source)
```

```
LineSource(period=14, amplitude=1.0, phase_shift=0.0, name='source')
    @ x=[48, ... , 51], y=[76, ... , 83], z=[0, ... , 0]
```

Note that one could also have supplied lists to index the grid in the first place. This feature could be useful to create a `LineSource` of arbitrary shape.

## 4.5 Detectors

*fdtd.detectors*

```
# signature
fdtd.LineDetector(
    name=None
)
```

Adding a detector to the grid works the same as adding a source

```
grid[12e-6, :, 0] = fdtd.LineDetector(name="detector")
print(grid.detector)
```

```
LineDetector(name='detector')
    @ x=[77, ... , 77], y=[0, ... , 96], z=[0, ... , 0]
```

## 4.6 Boundaries

*fdtd.boundaries*

```
# signature
fdtd.PML(
    a: float = 1e-8, # stability factor
    name: str = None
)
```

Although, having an object, source and detector to simulate is in principle enough to perform an FDTD simulation, One also needs to define a grid boundary to prevent the fields to be reflected. One of those boundaries that can be added to the grid is a Perfectly Matched Layer: or `PML`. These are basically absorbing boundaries.

```
# x boundaries
grid[0:10, :, :] = fdtd.PML(name="pml_xlow")
grid[-10:, :, :] = fdtd.PML(name="pml_xhigh")

# y boundaries
grid[:, 0:10, :] = fdtd.PML(name="pml_ylow")
grid[:, -10:, :] = fdtd.PML(name="pml_yhigh")
```

## 4.7 Grid Summary

A simple summary of the grid can be shown by printing out the grid:

```
print(grid)
```

```
Grid(shape=(161,97,1), grid_spacing=1.55e-07, courant_number=0.70)

sources:
    LineSource(period=14, amplitude=1.0, phase_shift=0.0, name='source')
        @ x=[48, ... , 51], y=[76, ... , 83], z=[0, ... , 0]

detectors:
    LineDetector(name='detector')
        @ x=[77, ... , 77], y=[0, ... , 96], z=[0, ... , 0]

boundaries:
    PML(name='pml_xlow')
        @ x=0:10, y=:, z=:
    PML(name='pml_xhigh')
        @ x=-10:, y=:, z=:
    PML(name='pml_ylow')
        @ x=:, y=0:10, z=:
    PML(name='pml_yhigh')
        @ x=:, y=-10:, z=:

objects:
    Object(name='object')
        @ x=11:32, y=30:84, z=0:1
```

(continues on next page)

(continued from previous page)

```
    Object(name=None)
        @ x=84:116, y=32:52, z=0:1
```

## 4.8 Running a simulation

Running a simulation is as simple as using the `grid.run` method.

```
grid.run(
    total_time: Number,
    progress_bar: bool = True
)
```

Just like for the lengths in the grid, the `total_time` of the simulation can be specified as an integer (number of `time_steps`) or as a float (in seconds).

```
grid.run(total_time=100)
```

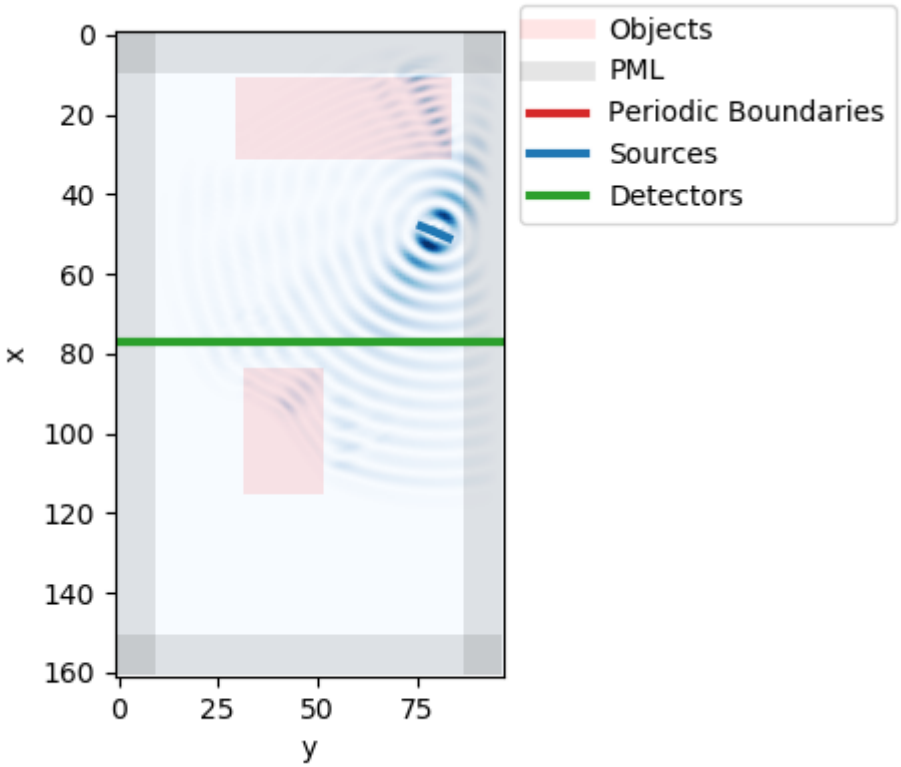
## 4.9 Grid visualization

Let's visualize the grid. This can be done with the `grid.visualize` method:

```
# signature
grid.visualize(
    grid,
    x=None,
    y=None,
    z=None,
    cmap="Blues",
    pbcolor="C3",
    pmlcolor=(0, 0, 0, 0.1),
    objcolor=(1, 0, 0, 0.1),
    srccolor="C0",
    detcolor="C2",
    show=True,
)
```

This method will by default visualize all objects in the grid, as well as the field intensity at the current `time_step` at a certain `x`, `y` **OR** `z`-plane. By setting `show=False`, one can disable the immediate visualization of the matplotlib image.

```
grid.visualize(z=0)
```


Objects
PML
Periodic Boundaries
Sources
Detectors
0
20
40
60
80
100
120
140
160
x
0
25
50
75
y

# BIBLIOGRAPHY

[CT] Cooley, James W., and John W. Tukey, 1965, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19: 297-301.

# PYTHON MODULE INDEX

## f

# INDEX

## Symbols



## A



## B



## C



## D



## E



## F



## G

## Z