Tight focusing

September 13, 2020

1 Basic example of using the *tight focusing* tool

In this example we show how one can simulate focusing of an electromagnetic pulse given by an arbitrary field state, i.e. arbitrary longitudinal and transverse variation of the amplitude, phase and polarization state, as well as arbitrary value of f-number. The method uses spectral solver and gives the possibility to evolve the field state to arbitrary time instant within one step and/or iterate it with a given time step. A significant optimization (memory and cpu time) is achieved by back-and-forth mapping to a spatial region that is reduced based on periodicity (see details in A. Panova et al. arXiv:<???> (2020)).

2 Importing modules

To import the hi-χ module we need to provide the path to the locally compiled (C++) binaries and also to some primitive elements written in Python for convinience:

```
[1]:  import sys
      # all paths are relative to the current directory 'doc'
      sys.path.append("../bin")
      sys.path.append("../python_modules")
```

```
[2]:  import pyHiChi as hichi
```

The *hichi_primitives* module contains some auxiliary Python functions and classes that will be used later.

```
[3]:  import hichi_primitives as hp
```

The *numpy* and *time* modules are necessary to work with arrays and to take time measurements, respectively.

```
[4]:  import numpy as np
```

```
[5]:  import time
```

3 Setting the pulse to be focused

Let's set the next parameters of the spherical pulse (by default we use CGS units).

1

```
[6]: wavelength = 1   # the wavelength of the pulse [micron] (=1e-4 cm)
     pulselength = 2.0*wavelength   # the pulse length
     R0 = 16*wavelength   # the distance to the geometrical center
     total_power = 1e7   # the total peak power of the incoming
        # electromagnetic radiation (over speherical surface) [erg/s] (= 1 W)
     f_number = 0.3   # f-number defines how tight the focusing is
     opening_angle = np.arctan(1.0/(2.0*f_number)) # we calculate the angle
        # to restrict the pulse in transverse direction
     edge_smoothing_angle = 0.1   # the angle over which
        # the transverse profile smoothly reaches zero at the edges [rad]
```

To set a limited profile of the pulse we use a rectangular function from the *hichi_primitives*:

$$\text{hp.block}(x, a, b) = \begin{cases} 1, x \in [a, b] \\ 0, x \notin [a, b] \end{cases}$$

We consider the longitudinal shape of the following form.

```
[7]: def longitudinal_field_variation(x):
        return np.sin(2*hichi.pi*x/wavelength) * \
               (np.cos(hichi.pi*x/pulselength))**2 * \
               hp.block(x, -0.5*pulselength, 0.5*pulselength)
```

To plot this function on a graph, we use *matplotlib*. The following code sets up the plotting area and plots the longitudinal field variation function.

```
[8]: import matplotlib
     import matplotlib.pyplot as plt
     from matplotlib.ticker import MultipleLocator, AutoLocator, \
                                   FixedLocator, FormatStrFormatter, \
                                   NullFormatter
```

```
[9]: def plot_graph(X, Y, xlabel, ylabel, style="-b", label=""):
        fig = plt.figure(dpi=500, figsize = (12, 4))
        ax = fig.add_subplot(1, 1, 1)
        ax.plot(X, Y, style, label=label)
        ax.grid(which='major', color='black')
        ax.grid(which='minor', color='lightgray')
        ax.xaxis.set_major_locator(FixedLocator([0.0]))
        ax.yaxis.set_major_locator(FixedLocator([0.0]))
        ax.xaxis.set_minor_locator(AutoLocator())
        ax.xaxis.set_minor_formatter(FormatStrFormatter("%0.1f"))
        ax.yaxis.set_minor_locator(AutoLocator())
        ax.yaxis.set_minor_formatter(FormatStrFormatter("%0.2f"))
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        if (label != ""): ax.legend()
        plt.show()
```

```
[10]: func = np.vectorize(longitudinal_field_variation)
      n_points = 1000
      X = np.linspace(-1.0, 1.0, n_points)
      Y = func(X)
      plot_graph(X, Y, "$x$", "$u_l(x)$", label="longitudinal field variation")
```
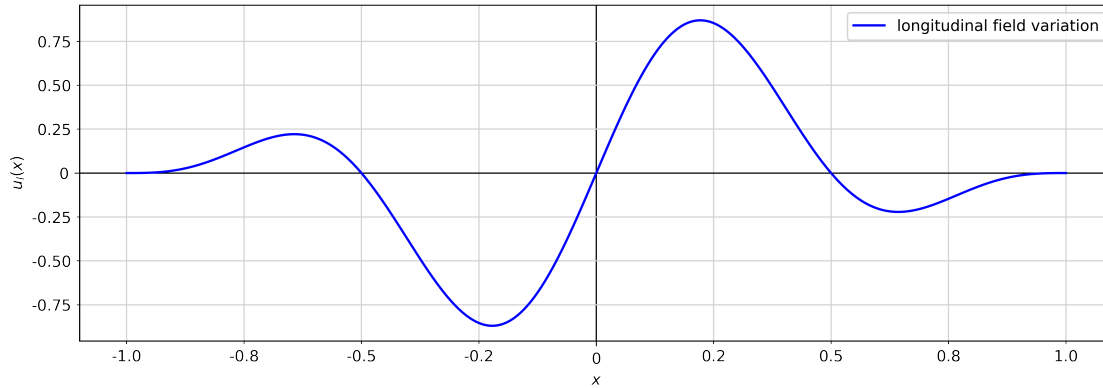
Transverse shape of the pulse in a simple case is just a rectangle bounding a spherical sector. However, we introduce an edge smoothing angle to provide a smooth transition to the zero field in the outer region.

```
[11]: def transverse_shape(angle):  # angle >= 0
          angle1 = opening_angle - edge_smoothing_angle*0.5
          angle2 = opening_angle + edge_smoothing_angle*0.5
          return hp.block(angle, -1.0, angle1) + \
                  (np.cos(0.5*hichi.pi*(angle - angle1)/edge_smoothing_angle)**2 * \
                   hp.block(angle, angle1, angle2) if (not edge_smoothing_angle == 0.
      →0) \
                  else 0.0)
```

You can see the graphical representation of the transverse shape function below.

```
[12]: def sym_transverse_shape(angle):
          return transverse_shape(abs(angle))

      func = np.vectorize(sym_transverse_shape)
      n_points = 1000
      X = np.linspace(-hichi.pi, hichi.pi, n_points)
      Y = func(X)
      plot_graph(X, Y, "$\\alpha$", "$u_{ts}(\\alpha)$", label="transverse shape")
```
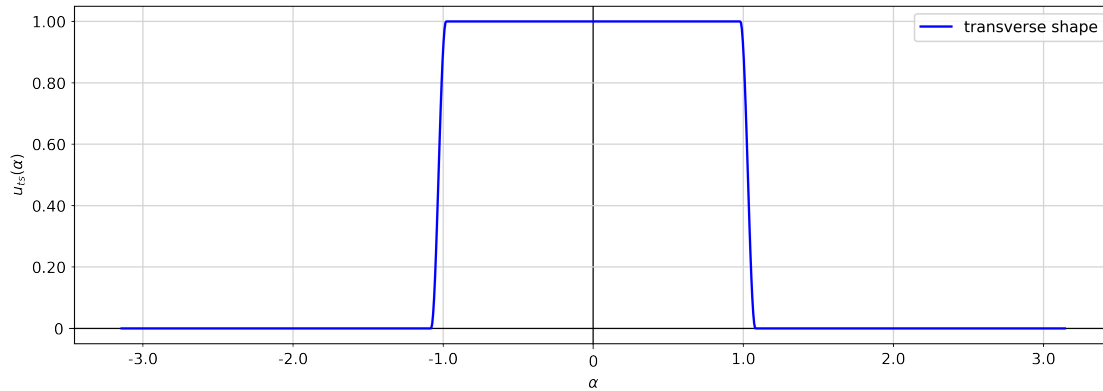
The *mask* function combines the longitudinal and transverse shapes of a spherical pulse. The amplitude is calculated based on the given input power.

```
[13]: amp = np.sqrt(total_power*4.0/(hichi.c*(1.0 - np.cos(opening_angle))))
```

```
[14]: def mask(x, y, z):
          R = np.sqrt(x*x + y*y + z*z)
          if(R > 1e-5):
              angle = np.arcsin(np.sqrt(y*y + z*z)/R)
              return (amp/R) * longitudinal_field_variation(R - R0) * \
                                      transverse_shape(angle)*(x < 0)
          else:
              return 0
```

To show the final pulse a colormap can be used.

```
[15]: N = 1024
      a = -20
      b = 20
      step = (b - a)/N
      u_values = np.array([[mask(a + i*step, a + j*step, 0) for i in range(N)] \
                                                    for j in range(N)])
```

```
[16]: def plot_im(values, vmin=0.0, vmax=1e-2, a=(-20, -20), b=(20, 20)):
          fig = plt.figure(dpi=500, figsize = (12, 4))
          ax = fig.add_subplot(1, 1, 1)
          ax.tick_params(axis='both', which='major')
          im = ax.imshow(values, cmap='RdBu', interpolation='none',
                        extent=(a[0], b[0], a[1], b[1]), vmin=vmin, vmax=vmax)
          fig.colorbar(im, ax=ax)
          plt.show()
```

```
[17]: plot_im(np.abs(u_values))
```
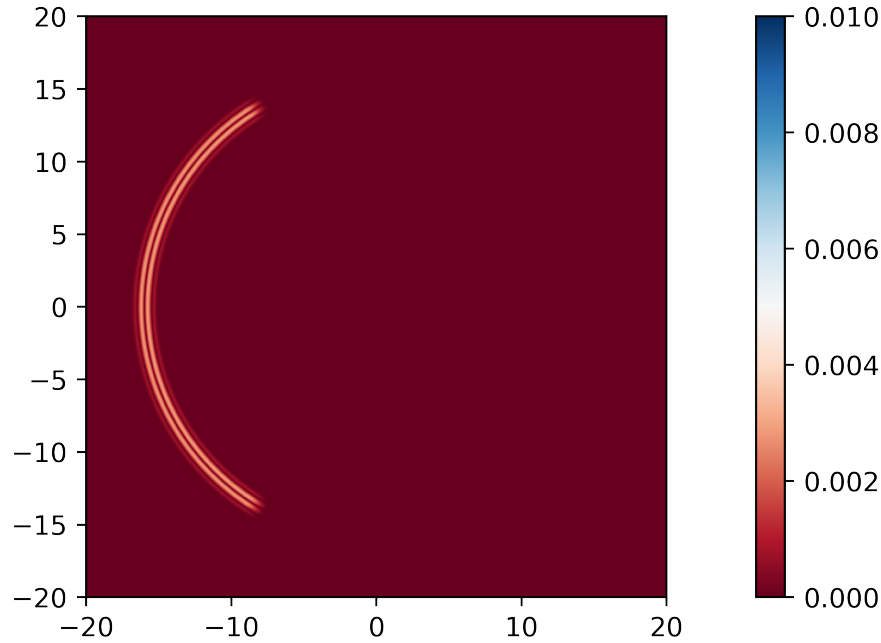
4 Creation of a computational grid and its initialization

A computational grid is required to perform a tight focusing simulation. The hi-chi module contains several types of computational grid for each field solver. To create a grid you need to specify the grid size, the spatial step of the grid, and the start coordinates of a computational area.

```
[18]: N = 256
      grid_size = hichi.Vector3d(N, N, N)
      min_coords = hichi.Vector3d(a, a, a)
      max_coords = hichi.Vector3d(b, b, b)
      grid_step = (max_coords - min_coords) / grid_size
```

Next we set the time step for evolving the field state.

```
[19]: time_step = R0*0.5/hichi.c
```

The following line initializes the spectral field solver (PSATD implementation from http://dx.doi.org/10.1016/j.jcp.2013.03.010) and creates a grid instance for further calculations.

```
[20]: field = hichi.PSATDField(grid_size, min_coords, grid_step, time_step)
```

4.1 Simple initialisation

The easiest way to initialize the grid is by using a Python function. The next function sets up the electomagnetic field for a point with coordinates (x, y, z).

```
[21]: polarisation = hichi.Vector3d(0.0, 1.0, 0.0)

      def get_EM_field_python_func(x, y, z):
          r = hichi.Vector3d(x, y, z)
          s1 = hichi.cross(polarisation, r)
          s0 = hichi.cross(r, s1)
          s0.normalize()
          s1.normalize()
          value = mask(x, y, z)
          E = value*s0
          B = value*s1
          return hichi.Field(E, B)
```

However, the grid initialisation takes a long time due to Python-C++ interaction costs. So the grid of size $256 \times 256 \times 256$ is initialized in about 20 minites.

```
[22]: start_time = time.process_time()   # the time before performing initialization

      field.set(get_EM_field_python_func)   # grid initialisation

      final_time = time.process_time()   # the time after performing initialization

      # display the time spent on initialization
      print("Time is %0.2f sec" % (final_time - start_time))
```

```
Time is 1525.08 sec
```

We can get the field value for a point with coordinates (x, y, z) from the grid. If the point is not a grid node, linear interpolation is performed. The next function reads the field values from the grid and writes them to the numpy array. The output array size and the grid size can be different.

```
[23]: def get_E_norm(field, shape, min_coords, max_coords):

          # 'shape' is the size of the output array 'res'
          res = np.zeros(shape=(shape[1], shape[0]))
          step = ((max_coords.x - min_coords.x)/shape[0], \
                  (max_coords.y - min_coords.y)/shape[1])

          for i in range(shape[0]):
              for j in range(shape[1]):

                  # computing the x, y coordinates by index (i, j)
                  x = min_coords.x + i*step[0]
                  y = min_coords.y + j*step[1]
                  coord = hichi.Vector3d(x, y, 0.0)   # z=0

                  # getting the norm of electric field at the point (x, y, z)
                  res[shape[1]-j-1, i] = field.get_E(coord).norm()
```
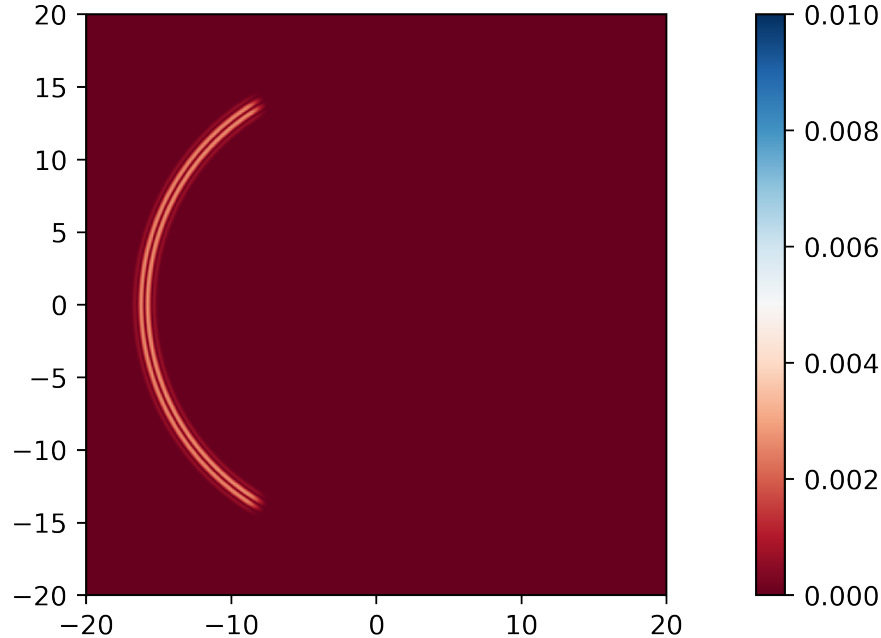
```
        return res
```

`[24]:`
```python
plot_im(get_E_norm(field, (4*N, 4*N), min_coords, max_coords))
```


4.2 Fast initialisation

As stated above, a long time is taken to Python-C++ interactions. One way to speed up the process is to avoid them by porting a function to the C++ level, that can be performed using *numba*. *Numba* is an open source JIT compiler that translates Python functions to optimized machine code at runtime. In addition, it can create a compiled function called from C code. *Numba* has a number of drawbacks, the main one being the inability to use external libraries, except for *numpy*. Nevertheless, compilation can significantly speed up the execution of the function.

`[25]:`
```python
from numba import cfunc, float64, jit, njit, jitclass, types, carray
```

The Numba limitation on the ability to use external modules prevents the use of the hi-chi *vector3d* class. Instead, a similar custom-designed class *vector3d* from the *hichi_primitives* module can be used. All helper functions that will be used for initialization must also be compiled. To mark a function for optimization by Numba's compiler it is enough to set *@jit* or *@njit* decorator above the function.

`[26]:`
```python
@njit
def get_polarisation():
    return hp.Vector3d(0.0, 1.0, 0.0)
```

The slightly modified auxulary functions below can also be compiled by Numba.

```python
[27]: @njit
      def longitudinal_field_variation(x):
           return np.sin(2*hichi.pi*x/wavelength) * \
                  np.cos(hichi.pi*x/pulselength)**2 * \
                  hp.block(x, -0.5*pulselength, 0.5*pulselength)


      @njit
      def transverse_shape(angle):
          angle1 = opening_angle - edge_smoothing_angle*0.5
          angle2 = opening_angle + edge_smoothing_angle*0.5
          return hp.block(angle, -1.0, angle1) + \
                  (np.cos(0.5*hichi.pi*(angle - angle1)/edge_smoothing_angle)**2 * \
                  hp.block(angle, angle1, angle2) if (not edge_smoothing_angle == 0.0)␣
       ↪\
                  else 0.0)


      @njit
      def mask(x, y, z):
          R = np.sqrt(x*x + y*y + z*z)
          if(R > 1e-5):
              angle = np.arcsin(np.sqrt(y*y + z*z)/R)
              return (amp/R) * longitudinal_field_variation(R - R0) * \
                                      transverse_shape(angle)*(x < 0)
          else:
              return 0.0
```

To compile a function that can be called from C++ level, the *@cfunc* decorator can be used. This decorator is similar to *@jit* decorator, however, the function signature has to be passed obligatorily. The *get_EM_field_numba_func()* function takes 4 arguments. The first three arguments are the coordinates of the point, and the fourth one is a pointer to an array of 6 elements where the result will be written as a sequence of field values E_x, E_y, E_z, B_x, B_y, B_z.

```python
[28]: @cfunc("void(float64,float64,float64,types.CPointer(float64))", nopython=True)
      def get_EM_field_numba_func(x, y, z, field_arr_):
          r = hp.Vector3d(x, y, z)
          s1 = hp.cross(get_polarisation(), r)
          s1.normalize()
          s0 = hp.cross(r, s1)
          s0.normalize()
          m = mask(x, y, z)
          field_arr = carray(field_arr_, (6))
          field_arr[0] = m*s0.x   # Ex
          field_arr[1] = m*s0.y   # Ey
          field_arr[2] = m*s0.z   # Ez
          field_arr[3] = m*s1.x   # Bx
          field_arr[4] = m*s1.y   # By
```

```
        field_arr[5] = m*s1.z   # Bz
```

The hi-chi module has a possibility to take a pointer to the compiled Python function.

```
[29]: start_time = time.process_time()

      # the 'address' property is a pointer to the compiled function
      field.set(get_EM_field_numba_func.address)

      final_time = time.process_time()
      numba_func_init_time = final_time - start_time
      print("Time is %0.2f sec" % (numba_func_init_time))
```
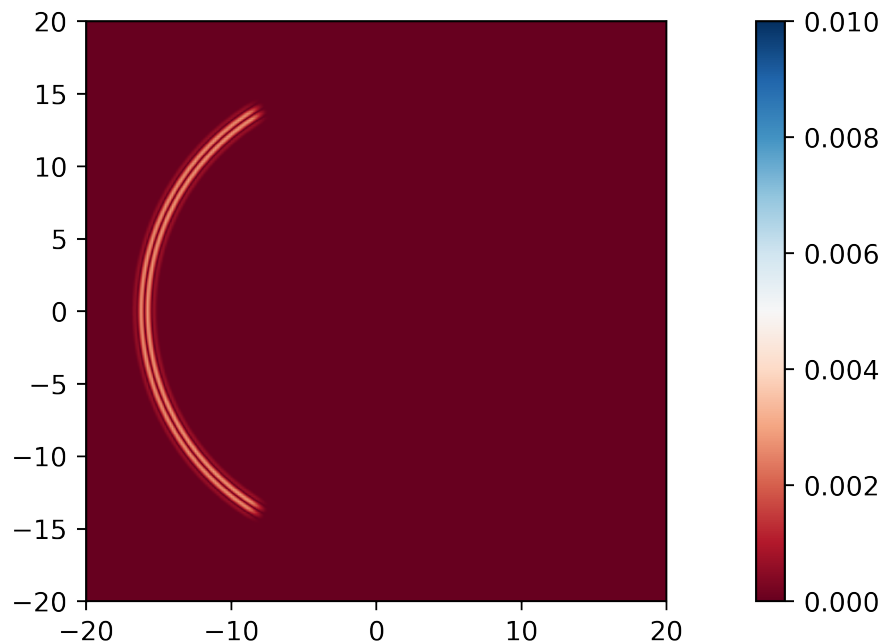
```
Time is 48.89 sec
```

We can see that the grid is initialized about 80 times faster. The image below confirms that everything is working correctly.

```
[30]: plot_im(get_E_norm(field, (4*N, 4*N), min_coords, max_coords))
```


4.3 Very fast initialisation

Obviously, the fastest way to initialize a computational grid is to use optimized multithreaded native C++ code. The hi-chi module provides such option for a given field configuration. However, it leads to loss of flexibility on the Python level.

```
[31]: tight_focusing_conf = hichi.TightFocusingField(f_number,
                                                      R0,
                                                      wavelength,
                                                      pulselength,
                                                      total_power,
                                                      edge_smoothing_angle
                                                      )
```
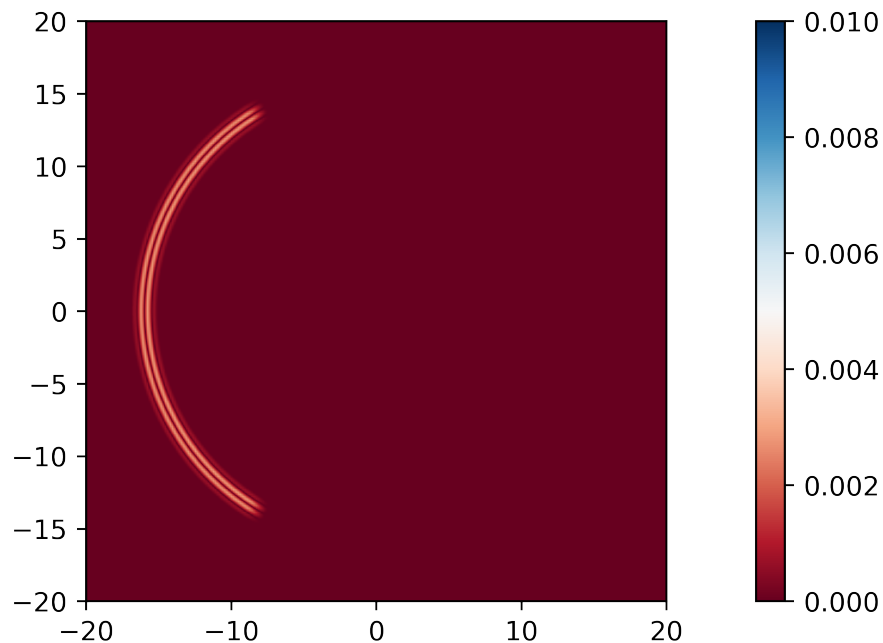
As we can see, C++ code runs an order of magnitude faster in one thread.

```
[32]: start_time = time.process_time()

      field.set(tight_focusing_conf)

      final_time = time.process_time()
      c_init_time = final_time - start_time
      print("Time is %0.2f sec" % (c_init_time))
```

    Time is 2.64 sec

```
[33]: plot_im(get_E_norm(field, (4*N, 4*N), min_coords, max_coords))
```

5 Start field solver

The next step is to run the simulation. To do this, just call the *update_fields()* method of the field. The spectral solver PSATD has no numerical dispersion and Courant condition, so we can perform simulation using a large time step.
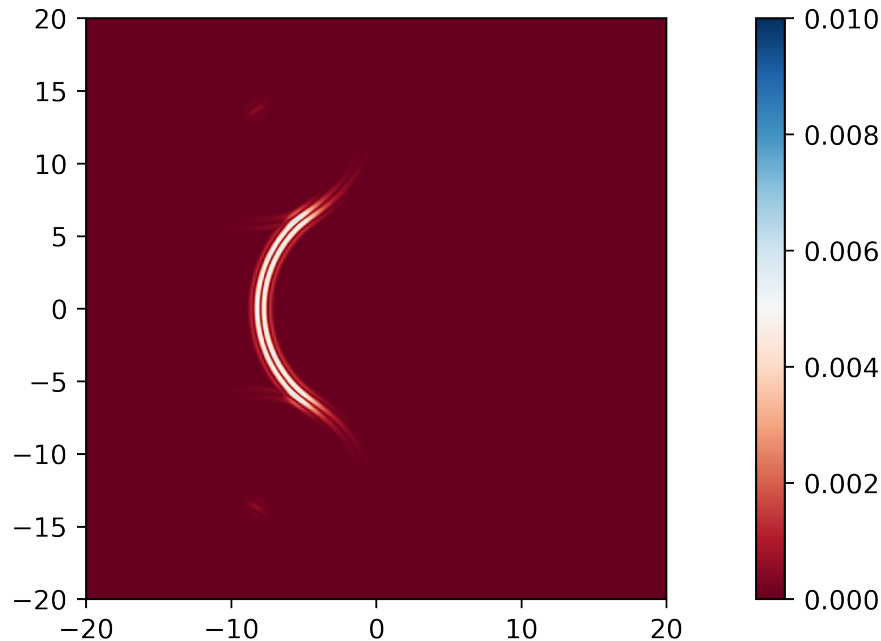
```python
start_time = time.process_time()

field.update_fields()

final_time = time.process_time()
field_solver_time = final_time - start_time
print("Time is %0.2f sec" % (field_solver_time))
```

[34]:

Time is 14.48 sec

[35]: `plot_im(get_E_norm(field, (4*N, 4*N), min_coords, max_coords))`

In the picture above, to the left of the pulse, we can see static fields, the appearance of which is due to the failure to satisfy the Poisson equation for the initial conditions. The code below performs the approciate adjustment.
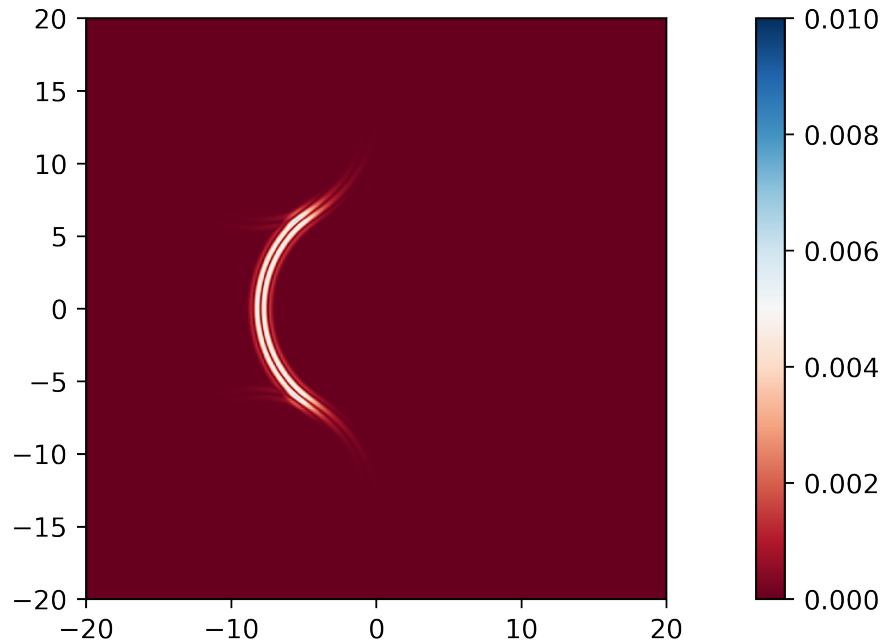
```python
start_time = time.process_time()

field.convert_fields_poisson_equation()
```

[36]:

```
final_time = time.process_time()
print("Time is %0.2f sec" % (final_time - start_time))
```

Time is 9.72 sec

[37]:
```
plot_im(get_E_norm(field, (4*N, 4*N), min_coords, max_coords))
```


The large opening angle allows us to assume that the peak intensity is reached at time R_0/c from the beginning of the calculation. The next code computes the peak value of the electric field.

[38]:
```
field.update_fields()
field_arr = get_E_norm(field, (4*N, 4*N), min_coords, max_coords)
plot_im(field_arr)
peak_field = field_arr.max()
print("Peak field value is %f" % (peak_field))
```
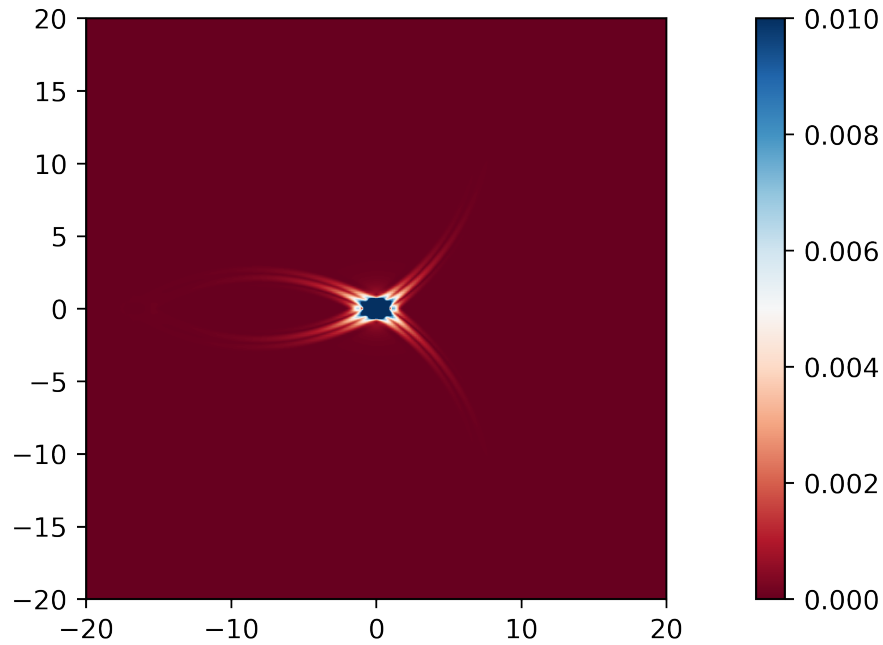
Peak field value is 0.141458

You can plot the result on a larger scale. To get more accurate results, you need to increase the computational grid resolution.

```
[39]: field_arr = get_E_norm(field, (4*N, 4*N),
                    hichi.Vector3d(-4, -4, -4),
                    hichi.Vector3d(4, 4, 4)
                    )
      plot_im(field_arr, vmax=peak_field, a=(-4, -4), b=(4, 4))
```
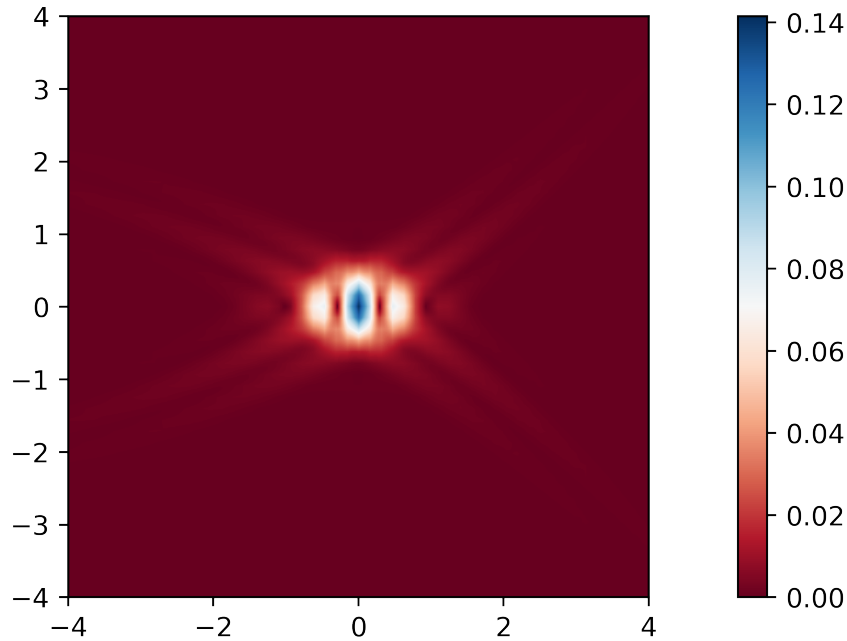
6   Mappings

The hi-chi module allows to use some mapping for the computational domain, such as a scale mapping, or a shift mapping, or a rotation mapping. Let's consider an electromagnetic pulse of the next configuration.

```
[40]: @cfunc("float64(float64,float64,float64)", nopython=True)
      def field_value(x, y, z):
          return np.exp(-x**2-y**2-z**2)*np.sin(5*x)
```

```
[41]: @cfunc("float64(float64,float64,float64)", nopython=True)
      def null_value(x, y, z):    # the auxulary function
          return 0.0
```

The computational area will be limited by the next points.

```
[42]: min_coords = hichi.Vector3d(-5, -5, -5)
      max_coords = hichi.Vector3d(5, 5, 5)
```

The code below creates the computational grid and initializes it.

```
[43]: grid_size = hichi.Vector3d(128, 128, 128)
      grid_step = (max_coords - min_coords) / grid_size
      time_step = 0.75/hichi.c
      field = hichi.PSATDField(grid_size, min_coords, grid_step, time_step)
```

14

```
[44]: field.set_E(null_value.address, field_value.address, null_value.address)
      field.set_B(null_value.address, null_value.address, field_value.address)
```

```
[45]: shape = (2*int(grid_size.x), 2*int(grid_size.y))
      plot_im(get_E_norm(field, shape, min_coords, max_coords),
              vmin=None, vmax=None,
              a=(min_coords.x, min_coords.y),
              b=(max_coords.x, max_coords.y)
             )
```
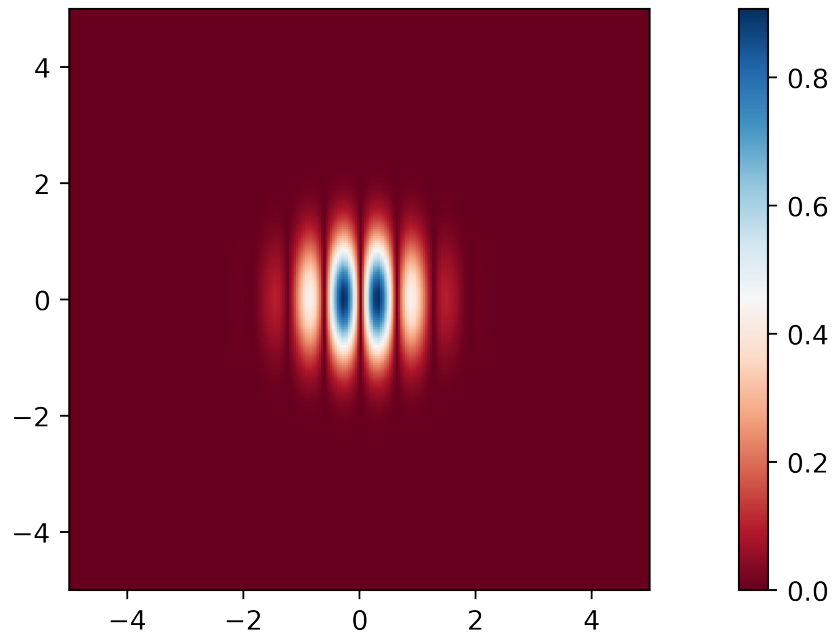

Let's demonstrate the rotation transform as an example. The following mapping rotates $\pi/3$ rad around the z-axis.

```
[46]: angle = hichi.pi/3
      rotation_z_mapping = hichi.RotationMapping(hichi.Axis.Z, angle)
```

The following code creates a grid wrapper that allows one or more mappings to be set and performed in sequence.

```
[47]: field_mapping = field.apply_mapping(rotation_z_mapping)
```
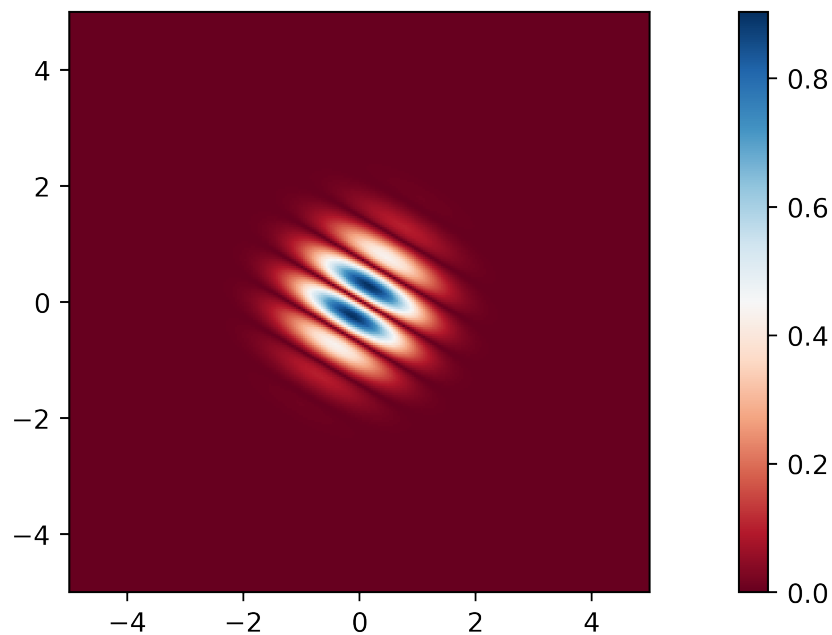
Now all the setted mappings will be performed automatically when the *get* and *set* grid methods are called..

```
[48]: plot_im(get_E_norm(field_mapping, shape, min_coords, max_coords),
              vmin=None, vmax=None,
              a=(min_coords.x, min_coords.y),
```

```
        b=(max_coords.x, max_coords.y)
        )
```


We also can simulate the process, and the beam will propagate along the specified direction.
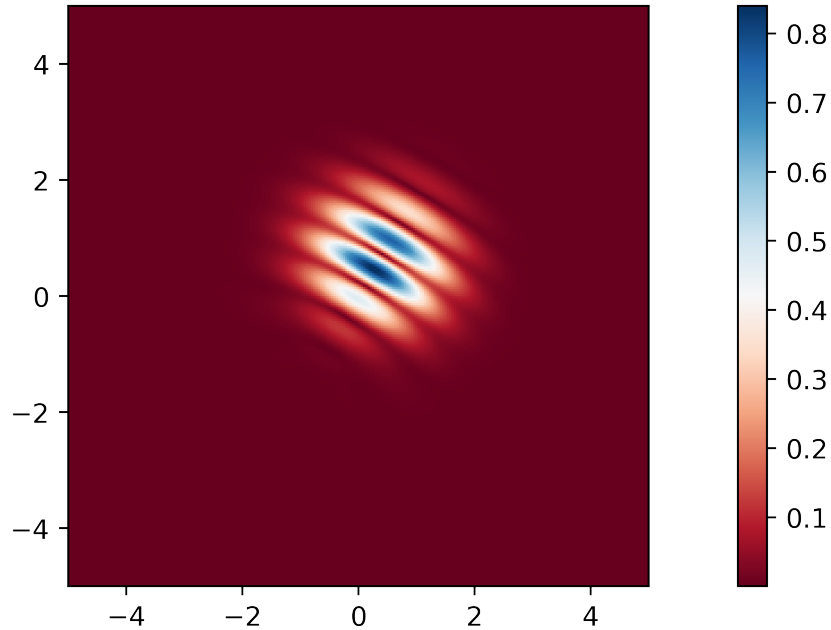
```
[49]: field_mapping.convert_fields_poisson_equation()
      # == field.convert_fields_poisson_equation()
      field_mapping.update_fields()
      # == field.update_fields()
```

```
[50]: plot_im(get_E_norm(field_mapping, shape, min_coords, max_coords),
              vmin=None, vmax=None,
              a=(min_coords.x, min_coords.y),
              b=(max_coords.x, max_coords.y)
              )
```

6.1   ombining fields

In addition to mappings, the hi-chi module has the ability to sum fields and multiply them by a constant. Taking into account the linearity of Maxwell's equations, this can make it possible to conveniently create various field configurations.

As an example, let's create the following additional field configuration, independent of the previous one.

```python
[51]:  @cfunc("float64(float64,float64,float64)", nopython=True)
       def field_value_2(x, y, z):
           return np.exp(2.0*(-x**2-y**2-z**2))*np.sin(10*x)
```

```python
[52]:  grid_size_2 = hichi.Vector3d(256, 256, 256)  # the grid size is different
       grid_step_2 = (max_coords - min_coords) / grid_size_2
       time_step_2 = 1.5/hichi.c                      # the time step is different too
       # hichi.PSATDPoissonField is a field that
       # takes into account the Poisson's equation automatically
       field_2 = hichi.PSATDPoissonField(grid_size_2, min_coords,
                                         grid_step_2, time_step_2)
```

```python
[53]:  field_2.set_E(null_value.address, field_value_2.address, null_value.address)
       field_2.set_B(null_value.address, null_value.address, field_value_2.address)
```

Let's add a new pulse to the previous field configuration.

```python
[54]:  shift_mapping = hichi.ShiftMapping(hichi.Vector3d(-2.5, -2.5, 0.0))
```

```
[55]: resulting_field = field_mapping + 0.8*field_2.apply_mapping(shift_mapping)
```

```
[56]: plot_im(get_E_norm(resulting_field, shape, min_coords, max_coords),
              vmin=None, vmax=None,
              a=(min_coords.x, min_coords.y),
              b=(max_coords.x, max_coords.y)
             )
```
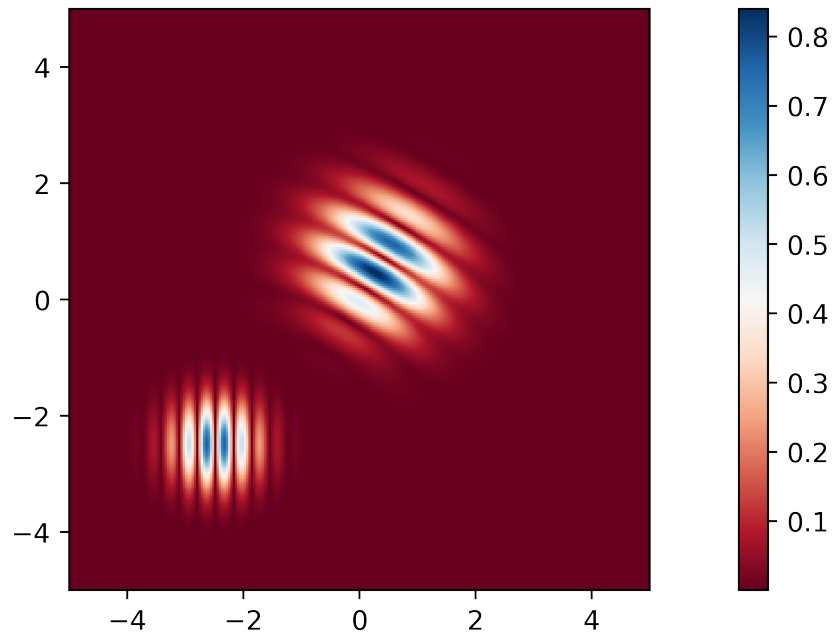

You can also perform summation, multiplication by a constant and apply mapping to the resulting field. Be aware of the time step difference when updating fields.

```
[57]: def update_fields():
          field.update_fields()    # dt = 0.75/hichi.c
          field.update_fields()    # dt = 0.75/hichi.c
          field_2.update_fields()  # dt = 1.5/hichi.c
```

```
[58]: update_fields()
```

```
[59]: plot_im(get_E_norm(resulting_field, shape, min_coords, max_coords),
              vmin=None, vmax=None,
              a=(min_coords.x, min_coords.y),
              b=(max_coords.x, max_coords.y)
             )
```
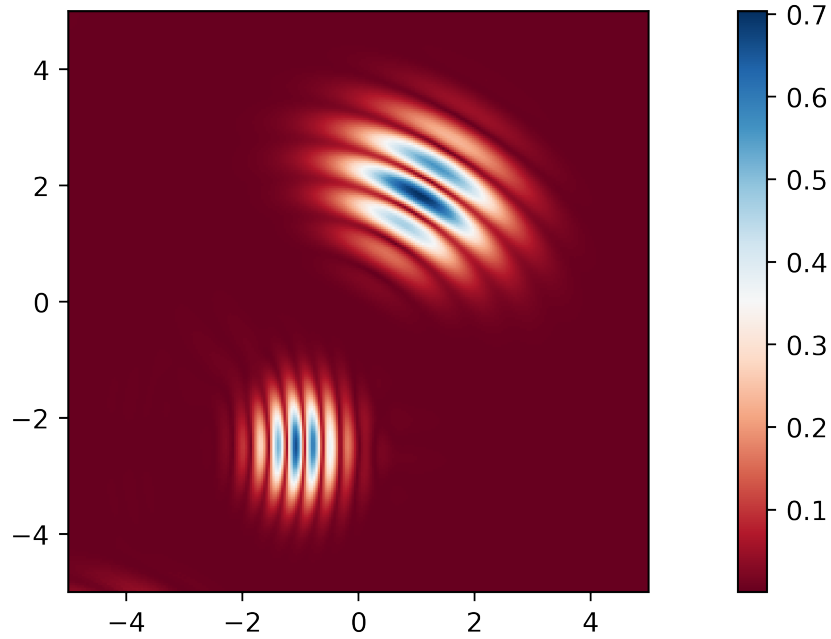
6.2 Tight Focusing Mapping

Based on the geometric properties of the tight focusing problem, only a small fraction of the computational grid accommodates the pulse. The main idea is to reduce computational costs due to areas with zero field values. However, to ensure the absence of numerical dispersion, the calculation is performed in the spectral space, and, therefore, it is not possible to exclude these regions from the calculation in an obvious way. We propose to get rid of additional computational costs by introducing a periodic structure of the field and, as a consequence, reducing the initial region to a thin layer along the x-axis.

Consider the original computational domain.

```
[60]: full_min_coords = hichi.Vector3d(-20, -20, -20)
      full_max_coords = hichi.Vector3d(20, 20, 20)
      full_grid_size = hichi.Vector3d(256, 256, 256)
```

Let's set the width of the band D along x-axis. In practice, it is usually sufficient to take the parameter D equal to $3-4$ pulse lengths. We choose D equal to 3.75 pulse lengths to get the grid size equal to 48 nodes along x-axis.

```
[61]: D = 3.75*pulselength   # the band width
      print(D)
```

7.5

```
[62]: new_grid_size_X = int(full_grid_size.x*D/(full_max_coords.x - full_min_coords.
      ↪x))
      print(new_grid_size_X)
```

       48

The next step is to create the appropriate mapping.

```
[63]: mapping = hichi.TightFocusingMapping(R0, pulselength, D, hichi.Axis.X)
```

The mapping sets up the band boundaries independently. We can get them by calling the *get_min_coord()* and *get_max_coords()* methods.

```
[64]: x_min = mapping.get_min_coord()
      x_max = mapping.get_max_coord()
      print(x_min, x_max)
```

       -22.5 -15.0

Knowing all the simulation parameters, we can create a computational grid.

```
[65]: min_coords = hichi.Vector3d(x_min, -20, -20)
      max_coords = hichi.Vector3d(x_max, 20, 20)
      grid_size = hichi.Vector3d(new_grid_size_X, full_grid_size.y, full_grid_size.z)
      grid_step = (max_coords - min_coords) / grid_size
      time_step = R0*0.5/hichi.c
```

The following code creates an instance of the PSATD field that takes Poisson's correction into account for each iteration and applies the mapping to it. This allows us not to call the *field.convert_fields_poisson_equation()* function and thereby save on one forward and one inverse fast Fourier transform.

```
[66]: field = hichi.PSATDPoissonField(grid_size, min_coords, grid_step, time_step)\
              .apply_mapping(mapping)
```

Since the grid size has decreased, we can expect that initialization costs will decrease by $256/48 \approx$ 5.33 times. It should be noted that there is additional overhead for performing the transform.

```
[67]: start_time = time.process_time()

      # initialisation using a compiled Python function
      field.set(get_EM_field_numba_func.address)

      final_time = time.process_time()
      numba_func_init_time_band = final_time - start_time
      print("Time is %0.2f sec" % (numba_func_init_time_band))
      print("Computational costs are decreased by %0.2f times" % \
              (numba_func_init_time/numba_func_init_time_band))
```
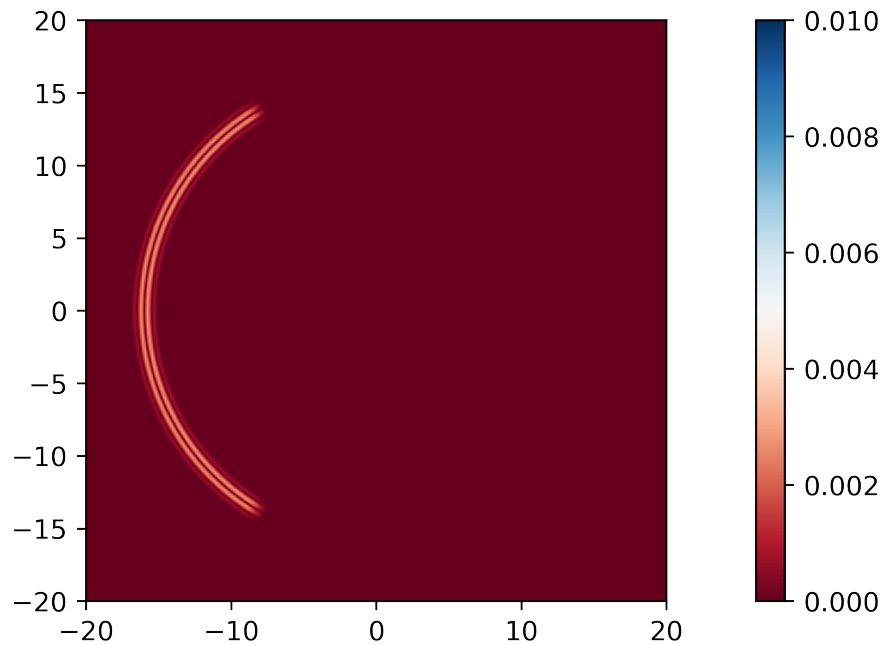
```
Time is 9.16 sec
Computational costs are decreased by 5.34 times
```

Since the mapping is done automatically, we can get the values of the electromagnetic field in the original area.

```
[68]: def get_field():
          return get_E_norm(field,
                            (4*int(full_grid_size.x), 4*int(full_grid_size.y)),
                            full_min_coords,
                            full_max_coords
                            )
```
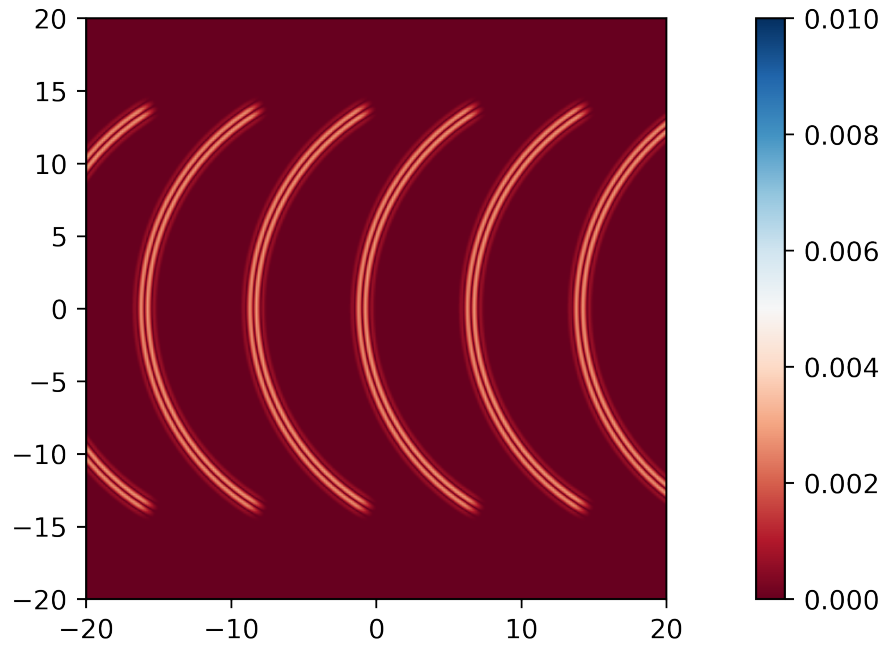
```
[69]: plot_im(get_field())
```


To illustrate how the mapping works, let's disable the inverse transform. To do this, we need to modify the mapping instance accordingly by calling the *if_perform_inverse_mapping()* method.

```
[70]: mapping.if_perform_inverse_mapping(False)
```

```
[71]: plot_im(get_field())
```

We can see that the initial pulse was duplicated along the x-axis, as a result of which the space became periodic. This allows the simulation to be performed out only in the band shown below.

```
[72]:  shape = (4*int(grid_size.x), 4*int(grid_size.y))
       plot_im(get_E_norm(field, shape, min_coords, max_coords),
               a=(min_coords.x, min_coords.y),
               b=(max_coords.x, max_coords.y)
              )
```
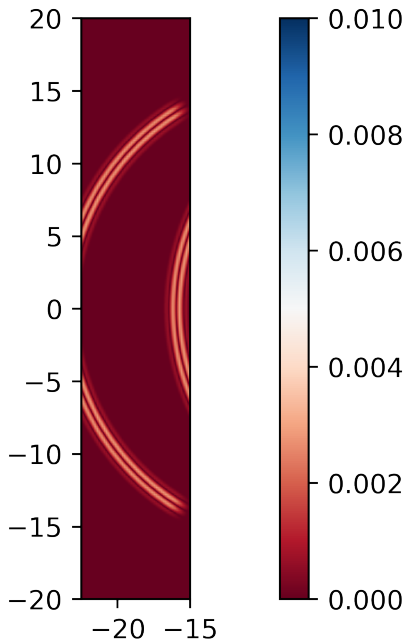
Since the mapping is performed only when the grid is initialized and the field values are obtained, then we can expect the maximum gain of the scheme in time when the field solver is executed.

```
[73]: start_time = time.process_time()

    field.update_fields()

    final_time = time.process_time()
    field_solver_time_band = final_time - start_time
    print("Time is %0.2f sec" % (final_time - start_time))
    print("Computational costs decreased by %0.2f times" % \
                    (field_solver_time/field_solver_time_band))
```
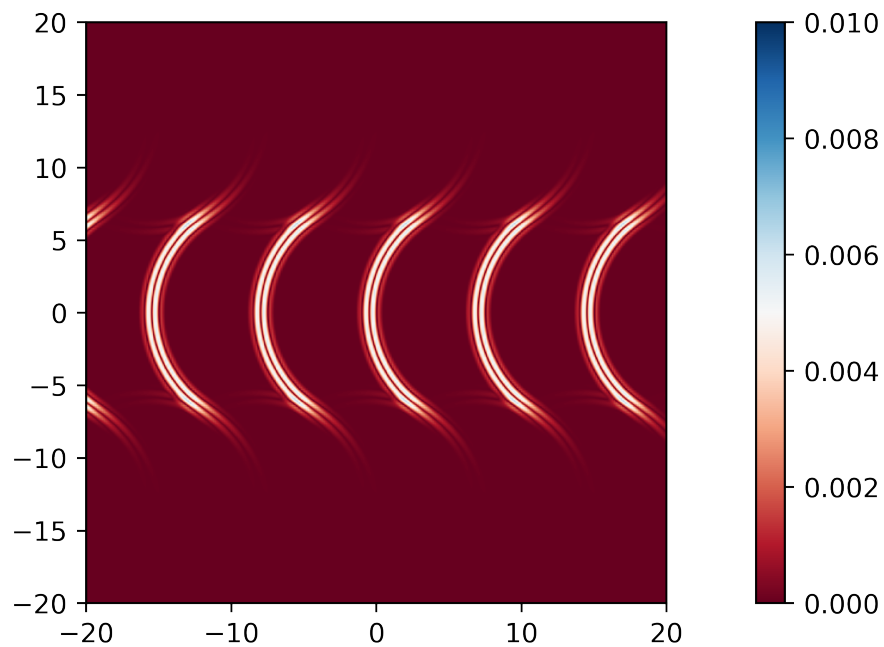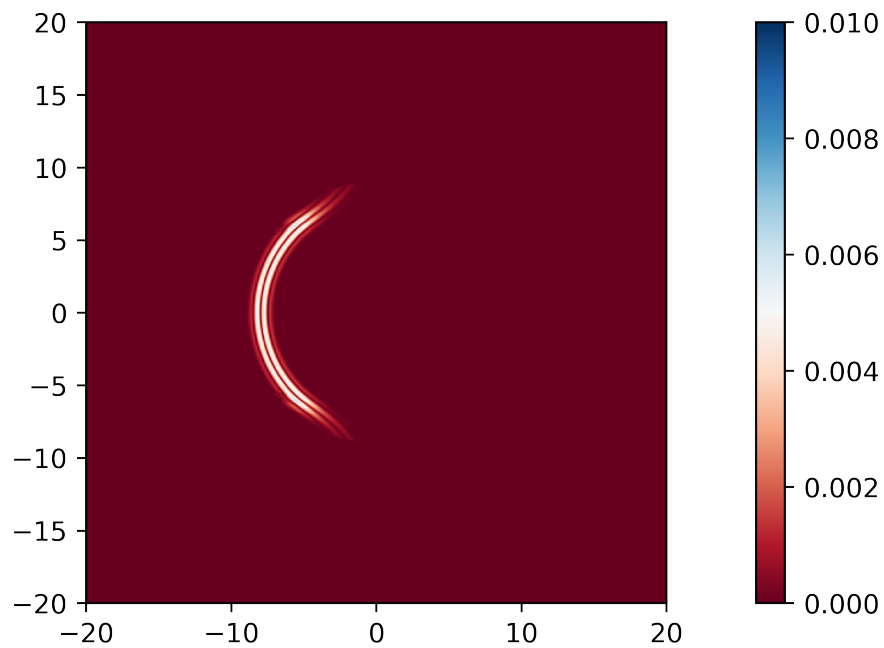
```
Time is 2.69 sec
Computational costs decreased by 5.39 times
```

```
[74]: plot_im(get_field())
```

Let's turn on the inverse mapping again, which 'cuts' the original pulse from the periodic space.

```
[75]: mapping.if_perform_inverse_mapping(True)
      plot_im(get_field())
```

The code below does a few more iterations of the solver and determines the peak intensity.

```
[76]: for iter in range(3):
          field.update_fields()
          plot_im(get_field())
          if iter == 0:   # time = R0*hichi.c
              peak_field_band = get_field().max()
              print("Peak field value is %f" % (peak_field_band))
```
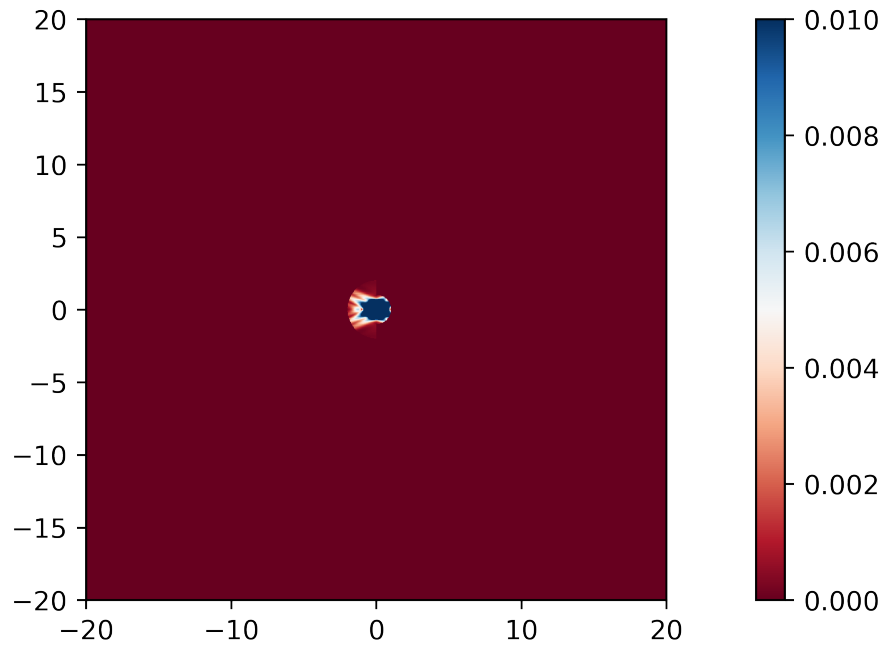


```
Peak field value is 0.141433
```
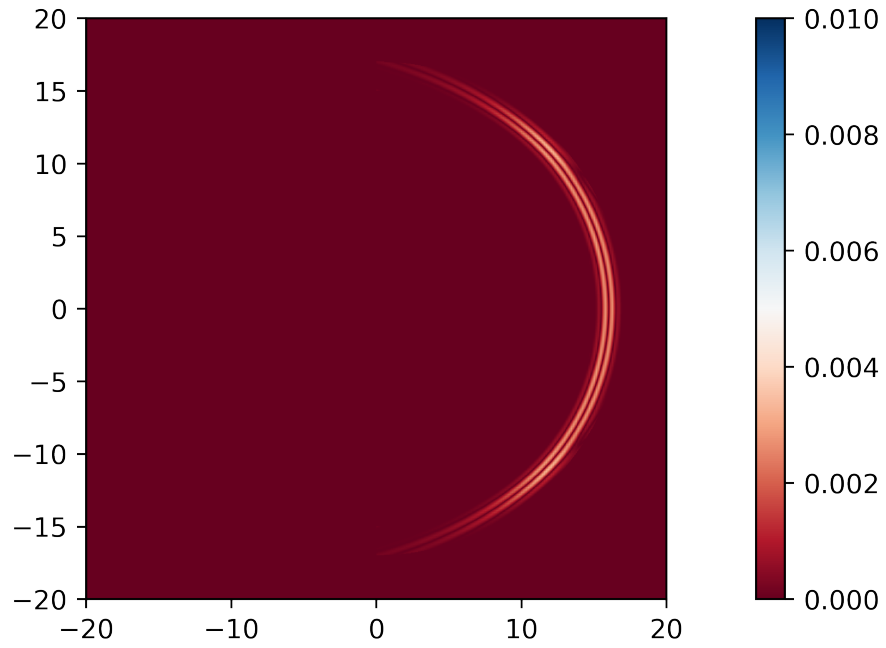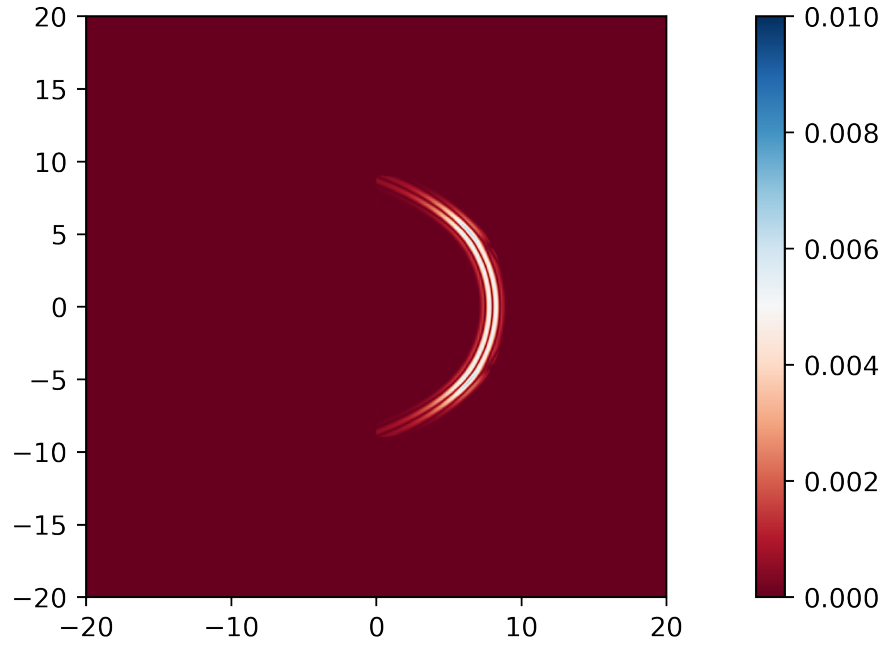
It can be seen that when determining the peak intensity, the error of the scheme was only 0.02% relative to the simulation in full entire region.

```
[77]: error = abs(peak_field_band - peak_field)/peak_field*100
      print(("Error of scheme is %0.3f" % (error)) + "%")
```

Error of scheme is 0.018%

7 Error depending on the band width D

Let us show that the error of the circuit with mapping decreases with increasing bandwidth D.
The grid resolution will be increase along x-axis, so that at small D the number of cells is sufficient.

```
[78]: full_grid_size = hichi.Vector3d(320, 256, 256)
      grid_step = (full_max_coords - full_min_coords) / full_grid_size
      time_step = R0/hichi.c
```

The next function is auxiliary.

```
[79]: def get_field(field):
          return get_E_norm(field,
                            (4*int(full_grid_size.x), 4*int(full_grid_size.y)),
                            full_min_coords,
                            full_max_coords
                            )
```

Let's set an array of points D for which simulation and error calculation will be carried out.

```
[80]: Rmin = R0 - pulselength*0.5
      Rmax = R0 + pulselength*0.5
      angle = opening_angle + edge_smoothing_angle
      Dmin = -Rmin*np.cos(angle) + (Rmax**2 - (Rmin*np.sin(angle))**2)**0.5
      Dmax = 10*pulselength
```

```
[81]: nx_arr = np.linspace(Dmin/grid_step.x, Dmax/grid_step.x, 12).astype(np.int)
      print("grid_size.x =", list(nx_arr))
      D_arr = nx_arr * grid_step.x
      print("D =", list(D_arr))
```

grid_size.x = [30, 42, 54, 66, 77, 89, 101, 113, 124, 136, 148, 160]
D = [3.75, 5.25, 6.75, 8.25, 9.625, 11.125, 12.625, 14.125, 15.5, 17.0, 18.5,
20.0]

The following function performs simulation for a fixed grid size along x-axis and returns values of
the electric field.

```
[82]: def run(nx):

          # the band width
          D = nx * grid_step.x

          # the mapping
```

```
    mapping = hichi.TightFocusingMapping(R0, pulselength, D, hichi.Axis.X)

    # the bounds of the band
    x_min = mapping.get_min_coord()
    x_max = mapping.get_max_coord()
    band_min_coords = hichi.Vector3d(x_min, full_min_coords.y, full_min_coords.
↪z)
    band_max_coords = hichi.Vector3d(x_max, full_max_coords.y, full_max_coords.
↪z)

    # the grid size for the periodic space (that is the real grid size)
    grid_size = hichi.Vector3d(nx, full_grid_size.y, full_grid_size.z)

    # the creation of the computational grid with the approciate mapping
    field = hichi.PSATDPoissonField(grid_size, band_min_coords,
                                    grid_step, time_step)
    field = field.apply_mapping(mapping)

    # the grid initialisation
    field.set(tight_focusing_conf)

    # the performing one iteration of the field solver
    field.update_fields()

    return get_field(field)
```

The error is calculated as the maximum field difference for full and thin band simulation.

```
[83]: def compute_error(res_band, res_full):
          return np.abs(np.subtract(res_band, res_full)).max()
```

Compute the result for full simulation.

```
[84]: res_full = run(full_grid_size.x)  # the full simulation result
```

In a loop, the results are calculated for each parameter value.

```
[85]: start_time = time.process_time()

      error_arr = []
      for nx in nx_arr:
          res_band = run(nx)  # the band simulation result
          error_arr.append(compute_error(res_band, res_full))

      final_time = time.process_time()
      print("Time is %0.2f sec" % (final_time - start_time))
```

```
Time is 208.11 sec
```

We can see that error does not exceed 1% and decreases with the D parameter growth.

```
[86]: def plot_error_graph():
          fig = plt.figure(dpi=500)
          ax = fig.add_subplot(1,1,1)
          ax.plot(D_arr/pulselength, error_arr/res_full.max()*100, "->b")
          ax.set_xlabel('$D/L$')
          ax.set_ylabel('relative error, %')
          ax.xaxis.set_minor_locator(FixedLocator([Dmin/pulselength]))
          ax.xaxis.set_major_locator(FixedLocator([2, 4, 6, 8, 10]))
          ax.yaxis.set_major_locator(MultipleLocator(0.1))
          ax.grid(which='major', linestyle='--')
          ax.grid(which='minor', linestyle='-')
          plt.show()
```

```
[87]: plot_error_graph()
```
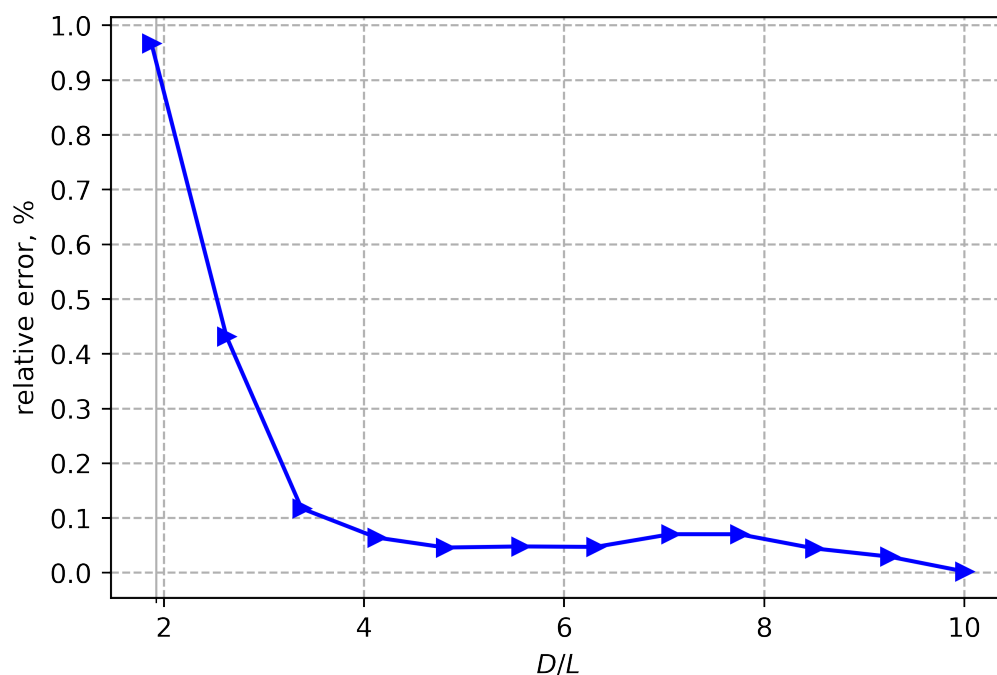

8 Peak field value depending on f-number

Let us calculate the peak field amplitude depending on the opening angle, or, equivalently, on the f-number, and show, that as the f-number decreases, the focus is located closer to the geometric center, and in this case the highest radiation intensity is achieved.

Since the configuration of the spherical wave will change, we create the following helper class.

```
[88]:  class SphericalPulse():

           def __init__(self, f_number=f_number,
                        R0=R0,
                        pulselength=pulselength,
                        edge_smoothing_angle=edge_smoothing_angle,
                        wavelength=wavelength,
                        total_power=total_power
                       ):

               self.wavelength = wavelength
               self.pulselength = pulselength*wavelength
               self.R0 = R0*wavelength
               self.total_power = total_power
               self.f_number = f_number
               self.edge_smoothing_angle = edge_smoothing_angle
               self.opening_angle = np.arctan(1.0/(2.0*f_number))

               self.tight_focusing_conf = \
                   hichi.TightFocusingField(self.f_number,
                                            self.R0,
                                            self.wavelength,
                                            self.pulselength,
                                            self.total_power,
                                            self.edge_smoothing_angle
                                           )

           def initialize(self, field):
               field.refresh()
               field.set(self.tight_focusing_conf)
```

```
[89]:  spherical_pulse = SphericalPulse()
```

The following code sets the simulation parameters.

```
[90]:  min_coords = hichi.Vector3d(-20, -20, -20)
       max_coords = hichi.Vector3d(20, 20, 20)
       full_grid_size = hichi.Vector3d(320, 256, 256)
```

```
[91]:  D = 2.5*pulselength
       grid_size = hichi.Vector3d(int(full_grid_size.x * D/(max_coords.x - min_coords.
       ↪x)),

                                  full_grid_size.y, full_grid_size.z
                                 )
```

```
[92]:  def create_field(spherical_pulse):
```

```
        mapping = hichi.TightFocusingMapping(spherical_pulse.R0,
                                             spherical_pulse.pulselength,
                                             D, hichi.Axis.X
                                             )

        x_min = mapping.get_min_coord()
        x_max = mapping.get_max_coord()

        min_band_coords = hichi.Vector3d(x_min, min_coords.y, min_coords.z)
        max_band_coords = hichi.Vector3d(x_max, max_coords.y, max_coords.z)

        grid_step = (max_band_coords - min_band_coords) / grid_size

        field = hichi.PSATDPoissonField(grid_size, min_band_coords, grid_step, 0.0)

        return field.apply_mapping(mapping)
```

[93]:
```
field = create_field(spherical_pulse)
```

Next, we determine the set of the f-number values for which the simulation will be performed.

[94]:
```
min_f_number = 0.2
max_f_number = 2.5
f_number_arr = np.linspace(min_f_number, max_f_number, 15)
print("f_number =", list(f_number_arr))
```

f_number = [0.2, 0.3642857142857143, 0.5285714285714286, 0.6928571428571428, 0.8571428571428572, 1.0214285714285714, 1.1857142857142857, 1.3499999999999999, 1.5142857142857142, 1.6785714285714286, 1.8428571428571427, 2.007142857142857, 2.1714285714285717, 2.335714285714286, 2.5]

The simulation will be performed in two stages. First, 16 iterations will be performed with a time step of $1/c$, then, among the obtained field configurations, the one with the highest intensity of the electromagnetic field will be selected. Finally, in its vicinity a simulation will be performed with a smaller time step $0.1/c$ to obtain a more accurate result.

[95]:
```
def get_field(field):
    res = np.zeros(shape=(int(full_grid_size.x)))
    coord = np.arange(min_coords.x, max_coords.x, grid_step.x)
    for ix in range(int(full_grid_size.x)):
        E = field.get_E(hichi.Vector3d(coord[ix], 0.0, 0.0))
        res[ix] = E.norm()
    return res, coord
```

[96]:
```
def update_fields(field, time_step):
    field.change_time_step(time_step / hichi.c)
    field.update_fields()
```

```
[97]: def run_in_time_interval(field, time_start, time_end, time_step):
          res = []
          cur_time = time_start
          while cur_time < time_end:
              field_arr, coord = get_field(field)
              peak = field_arr.max()
              x_peak = coord[np.where(field_arr == peak)[0]][0]
              res.append((cur_time, peak, x_peak))
              cur_time += time_step
              update_fields(field, time_step)
          return res
```

```
[98]: def run(field, spherical_pulse):
          res = []    # res is a list of touples
                      # (time, peak field value, peak coordinate)
          # dt = 1/c
          start_time = 0.0
          final_time = spherical_pulse.R0
          n_iter = spherical_pulse.R0
          time_step = (final_time - start_time) / n_iter
          res_interval = run_in_time_interval(field, start_time,
                                                  final_time, time_step)
          peak = max(res_interval, key = lambda a: a[1])
          time_max = peak[0]
          res.extend(res_interval)

          # dt = 0.1/c
          eps = 1.0*spherical_pulse.R0/16
          start_time = time_max - eps
          update_fields(field, start_time - final_time)
          final_time = time_max + eps
          time_step = 0.1
          res_interval = run_in_time_interval(field, start_time,
                                                  final_time, time_step)
          res.extend(res_interval)

          res.sort()

          return res
```

The following code starts the calculation.

```
[99]: start_time = time.process_time()

      peak_arr = []    # a list of tuples (time, peak field value, peak coordinate)
      for f_number in f_number_arr:
```

```
    spherical_pulse = SphericalPulse(f_number=f_number)
    spherical_pulse.initialize(field)

    res = run(field, spherical_pulse)

    peak_arr.append(max(res, key=lambda a: a[1]))

final_time = time.process_time()
print("Time is %0.2f sec" % (final_time - start_time))
```

```
Time is 1493.80 sec
```

Let's plot the graphs of the dependence of the peak amplitude, as well as the x coordinate and time at which it is reached, on the f-number. To get a more accurate result, you need to increase the grid resolution and simulate with a smaller time step.

[100]:
```python
def plot_peak_graphs():
    plot_graph(f_number_arr, [elem[1] for elem in peak_arr],
               'f-number', 'peak amplitude', style="-ob")
    plot_graph(f_number_arr, [elem[2] for elem in peak_arr],
               'f-number', 'peak coordinate', style="-ob")
```

[101]:
```python
plot_peak_graphs()
```