# Python module for optical setup simulation in the paraxial approximation - PyParax

August 24, 2021

**Abstract**

This module is intended as an optical system simulation tool for linear paraxial approximation propagation scenarios, and it also has routines for phase mask computation using a forward-backward propagation approach for one or two masks. The module includes three types of optical elements (free space, phase mask, and amplitude mask), allows for forward and backward propagation through the optical system, takes into consideration alignment tolerances, allows for Monte-Carlo type of optical system characterization if tolerances are given, and it allows for full beam profile access, based on the specifications of the computer in use.

# 1 General notions

## 1.1 The propagation model

### 1.1.1 Propagation and optical elements

The propagation is described in two cases. First there is the free space equation which is modeled using the wave equation in the paraxial approximation

$$\partial_z \psi = \frac{i}{2k}\left(\partial_x^2 \psi + \partial_y^2 \psi\right) \tag{1}$$

where $\psi$ is the wave envelope that propagates along the $z$ axis with two transverse axes $x$ and $y$, and $k = 2\pi/\lambda$ is the wavenumber. For only one transverse dimension the second term on the RHS can be ignored.

Eq. (1) can be easily solved by considering the Fourier transform (FT) along the transverse axes which gives solution

$$\bar{\psi} = \bar{\psi}_0 \exp\left(-\frac{2i\pi^2 z(\xi^2+\eta^2)}{k}\right) \tag{2}$$

where $\xi$ and $\eta$ are the spatial frequencies associated to the transverse axes, $\psi_0$ is the initial condition, and the notation $\bar{f}$ denotes the FT of the general function $f$.

The second case is the interaction with the amplitude and phase masks. The amplitude mask $M_a$ is a real-valued function of as many variables as the transverse space that modulates the amplitude of the wave by multiplication at a given position $z_0$ on the propagation axis as

$$\psi(z_0) \to M_a \cdot \psi(z_0). \tag{3}$$

For physical reasons the amplitude mask should contain values bounded by the interval $[0, 1]$ although the module does not impose it. The phase mask $M_p$ is similar to the amplitude mask with the main difference being that it is applied to the wave by

$$\psi(z_0) \to \exp\left(iM_p\right) \cdot \psi(z_0). \tag{4}$$

Although both masks can be implemented in an identical manner given by eq. 3, from a user perspective it is much easier to visualize and manipulate a real valued function, in this case the phase $M_p$, rather than the complex function that is used, i.e. $\exp(iM_p)$.

### 1.1.2 Numeric implementation

The numeric implementations relies on computation on grids of equal steps along each axis. This allows for a fast computation of the numeric solution in Fourier space (2) using the Fast Fourier Transform (FFT). Written in steps, the propagation is computed as follows:

- Compute the FFT of the initial condition $\psi_0$.
- Use $\bar{\psi}_0$ and eq. (2) to calculate the solution $\bar{\psi}$.
- Compute the inverse FFT (iFFT) of $\bar{\psi}$.

The amplitude and phase masks are applied by multiplication in real space (labeled in the following as normal space), which translates to a convolution in Fourier space. For computational reasons, whenever a mask is applied, a switch from Fourier space to normal space is made, with one exception [see section 1.2.3]. On the grid, the multiplication is implemented as a pointwise product.

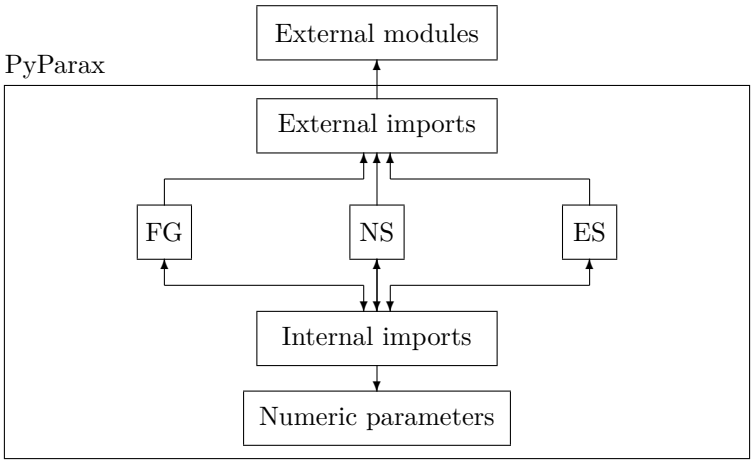


Figure 1: The structure of the PyParax module. Arrows indicate the import logic between the submodules and with the external modules.

## 1.2 Module structure

The module consists of four submodules that handle different types of variables or functions. Those modules are linked through an auxiliary module that makes intermodule imports cleaner. The import of external modules (e.g. NumPy, Matplotlib, SciPy) are handled by a second auxiliary module.

The structure of the PyParax module is given in figure 1. There are three submodules that contain computational parts within PyParax, the Function Generator (FG), the Numerical Solver (NS), and the Experimental Simulator (ES). The FG is a collection of frequently used functions e.g. common initial condition (Gaussian, Airy), lens phase masks, amplitude masks for circular apertures. The NS handles the computation of the solution for free space propagation. The ES is the main submodule since it contains all the functions that are related to the propagation through optical systems, optimization based on some parameters of the lenses (if they exist within the optical system), plots, Monte-Carlo calculations, and system RAM requirements estimations.

All the above mentioned submodules communicate via an internal imports submodule. They also import from a Numeric Parameters submodule the required values of various parameters such as the scale of the spatial domain by the choice of the unit of measurement (millimeters by default), the number of points and step values on each axis, the wavelength, the refractive index of the medium and values for shifting the transverse axes. In the following any distance related term will be referred to as *unit* implying that its order of magnitude is given by the Numeric parameters submodule.

In order to make use of existing optimized modules for linear algebra operations and data plotting, the three submodules FG, NS, and ES can import via an external imports submodule the required modules.

We consider that this structure offers good flexibility in customizing PyParax for specific user needs by offering the possibility of adding additional functions in FG, changing the numerical parameters as needed while preserving them as global variables and even introducing new solvers in NS.

### 1.2.1 Function Generator

The FG submodule consists of a collection of functions that are frequently used. What this submodule consists of should be user specific, thus we highly recommend that a user should personalize its collection of functions. The vanilla FG submodule consists of two classes, *standard_initial_conditions* for functions that are meant to be used as initial condition, and *optical_elements* for functions that define various optical elements as amplitude or phase masks, based on the type of element.

**Default functions in the *standard_initial_conditions* class:**

- 
```
spatial_domain_generator(dim = 1):
    """
    Generate the transversal spatial domain using the parameters in numeric_parameters.py
        ↪ .

    Inputs:
        dim = int - The number of dimensions of the transversal space.
    Outputs:
        if dim = 1
            spatial_domain = 1-dimensional array; float - The 1-dimensional spatial
                ↪ domain.
        if dim = 2
            spatial_domain_x, spatial_domain_y = tuple of two 2-dimensional arrays; float
                ↪ - The arrays with the coordinates for the 2-dimensional spatial domain
                ↪ .
    """
```

```
generate_gauss_1d(mean, sigma):
    """
    Generate a 1-dimensional Gaussian.

    Inputs:
        mean = float - The mean of the Gaussian.
        sigma = float - The standard deviation of the Gaussian.
    Outputs:
        f = 1-dimensional array; float - The Gaussian function.
    """
```

- `generate_gauss_2d(mean_x, mean_y, sigma_x, sigma_y):`

```
    """
    Generate a 2-dimensional Gaussian.

    Inputs:
        mean_x = float - The mean on the X axis of the Gaussian.
        mean_y = float - The mean on the Y axis of the Gaussian.
        sigma_x = float - The standard deviation the X axis of the Gaussian.
        sigma_y = float - The standard deviation the Y axis of the Gaussian.
    Outputs:
        f = 2-dimensional array; float - The Gaussian function.
    """
```

- `generate_airy_1d(scale, offset, decay):`

```
    """
    Generate a 1-dimensional Airy function.

    Inputs:
        scale = float - For spatial domain rescaling
        offset = float - For shifting the spatial domain
        decay = float positive - For truncating the function
    Outputs:
        f = 1-dimensional array; float - The Airy function.
    """
```

- `generate_airy_2d(scale, offset, decay):`

```
    """
    Generate a 2-dimensional Airy function by multiplication of two Airy functions i.e.
    ↪ Ai(x,y) = Ai(x)*Ai(y).

    Inputs:
        scale = float - For spatial domain rescaling
        offset = float - For shifting the spatial domain
        decay = float positive - For truncating the function
    Outputs:
        f = 2-dimensional array; float - The Airy function.
    """
```

**Default functions in the *optical_elements* class:**

- `lens_theory(focal, x_dom = None, y_dom = None, wavelength = None, n0 = None, offset_x = 0, offset_y = 0, dim = 1, fft_flag = False):`

```
    """
    Computes the phase mask (complex function) of a lens in the paraxial approximation.

    Inputs:
        focal = float - The focal length of the lens.
        x_dom = array; float - The transversal spatial domain on the X axis.
        y_dom = array; float - The transversal spatial domain on the Y axis.
        wavelength = float - The wavelength of the beam in vacuum.
        n0 = float - The refractive index of the medium.
        offset_x = float - The offset along the X axis of the lens.
        offset_y = float - The offset along the Y axis of the lens.
        dim = int - The number of dimensions for the transverse space.
```

```
        fft_flag = boolean - if True, the phase mask is computed in Fourier space.
    Outputs:
        if dim = 1
            phase = 1-dimensional array; complex - The phase mask (complex function) of
                ↪ the lens
        if dim = 2
            phase = 2-dimensional array; complex - The phase mask (complex function) of
                ↪ the lens
    """
```

- `linear_phase(tilt_x = 0, tilt_y = 0, wavelength = None, n0 = None, dim = 1, fft_flag = False):`
  ↪

```
    """
    Computes a phase mask that tilts the propagation direction with a given angle.

    Inputs:
        tilt_x = float - The tilt parameter for the X axis.
        tilt_y = float - The tilt parameter for the Y axis.
        wavelength = float - The wavelength of the beam in vacuum.
        n0 = float - The refractive index of the medium.
        dim = int - The number of dimensions for the transverse space.
        fft_flag = boolean - if True, the phase mask is computed in Fourier space.
    Outputs:
        if dim = 1
            phase = 1-dimensional array; complex - The phase mask (complex function) of
                ↪ the lens
        if dim = 2
            phase = 2-dimensional array; complex - The phase mask (complex function) of
                ↪ the lens
    """
```

- `amplitude_mask_circular(width, smooth = 1, x_size = None, y_size = None, offset_x = 0, offset_y = 0, dim = 1):`
  ↪

```
    """
    Computes a circular amplitude mask.

    Inputs:
        width = float - The diameter of the apperture.
        smooth = int - Number of pixels and interations on which the edges of the
            ↪ apperture are smoothened.
        x_size = int - Number of pixels on the X axis.
        y_size = int - Number of pixels on the y axis.
        offset_x = float - The offset along the X axis of the lens.
        offset_y = float - The offset along the Y axis of the lens.
        dim = int - The number of dimensions for the transverse space.
    Outputs:
        if dim = 1
            mask = 1-dimensional array; real - The amplitude mask (real function).
        if dim = 2
            mask = 2-dimensional array; real - The amplitude mask (real function).
    """
```

### 1.2.2 Numerical Solver

In the default version, the NS submodule contains two solvers, based on the number of transverse dimensions. It should be mentioned that each solver is built around the possibility of having inputs and outputs in both normal or Fourier spaces. The choice for this approach is given in section 1.2.3.

The solvers have been structured into two distinct classes namely *numeric_fourier_solver_1d* and *numeric_fourier_solver_2d* because the inclusion of other solvers in the future has been considered.

**Default functions in the *numeric_fourier_solver_1d* class:**

- `linear(f0,`
  `wavelength = None,`
  `n0 = 1,`

```
    dx = None,
    dz = None,
    steps = None,
    forward = True,
    print_progress = False,
    output_full = True,
    fft_input = False,
    fft_output = False):
"""
Computes the solution for the partial differential equation (PDE) for the 1-
    ↪ dimensional wave equation in the paraxial approximation using the Fourier
    ↪ Transform.
Inputs:
    f0 = 1-dimensional array; complex - The initial condition of the PDE.
    wavelength = float; real - The wavelength of the electromagnetic wave in vacuum.
    n0 = float; real - The refractive index of the medium.
    dx = float; real - The step for the spatial transverse axis.
    dz = float; real - The step for the spatial propagation axis.
    steps = int; natural - The number of iterations the solver has to compute.
    forward = boolean - If False, the propagation direction is reversed.
    print_progress = boolean - If True, the iteration at which the solver computing
        ↪ is printed.
    output_full = boolean - If True, the function returns the solution for each
        ↪ iteration. If False, only the last iteration is returned.
    fft_input = boolean - If True, f0 is considered to be the FFT of the initial
        ↪ condition, so that no extra FFT is applied on it.
    fft_output = boolean - If True, the output is returned in Fourier space.
Outputs:
    if output_full = True
        f = 2-dimensional array; complex - The solution of the PDE for every
            ↪ iteration. The shape is steps x size(f0)
    if output_full = False
        f = 1-dimensional array; complex - The solution of the PDE for the last
            ↪ iteration.
"""
```

**Default functions in the *numeric_fourier_solver_2d* class:**

- 
```
linear(f0,
    wavelength = None,
    n0 = 1,
    dx = None,
    dy = None,
    dz = None,
    steps = None,
    forward = True,
    print_progress = False,
    output_full = True,
    fft_input = False,
    fft_output = False):
"""
Computes the solution for the partial differential equation (PDE) for the 2-
    ↪ dimensional wave equation in the paraxial approximation using the Fourier
    ↪ Transform.
Inputs:
    f0 = 2-dimensional array; complex - The initial condition of the PDE.
    wavelength = float; real - The wavelength of the electromagnetic wave in vacuum.
    n0 = float; real - The refractive index of the medium.
    dx = float; real - The step for one of the spatial transverse axis.
    dy = float; real - The step for the other spatial transverse axis.
    dz = float; real - The step for the spatial propagation axis.
    steps = int; natural - The number of iterations the solver has to compute.
    forward = boolean - If False, the propagation direction is reversed.
    print_progress = boolean - If True, the iteration at which the solver computing
        ↪ is printed.
```

```
        output_full = boolean - If True, the function returns the solution for each
            ↪ iteration. If False, only the last iteration is returned.
        fft_input = boolean - If True, f0 is considered to be the FFT of the initial
            ↪ condition, so that no extra FFT is applied on it.
        fft_output = boolean - If True, the output is returned in Fourier space.
    Outputs:
        if output_full = True
            f = 3-dimensional array; complex - The solution of the PDE for every
                ↪ iteration. The shape is steps x size_on_x(f0) x size_on_y(f0)
            !!! Consumes lots of RAM !!!
        if output_full = False
            f = 2-dimensional array; complex - The solution of the PDE for the last
                ↪ iteration.
    """
```

### 1.2.3 Experimental Simulator

As it has been mentioned before, this is the main submodule for the simulation of a beam propagating through an optical system. This implies that the functions defined for simulating the propagation depend on all the other submodules presented so far while also specifying the rules of their use. In order to better emphasize this the first type of data that has to be presented is the optical elements and their properties, followed by how they are used to create an optical system.

The optical elements used can be categorized into the following types:

**Free space domains:** They are defined by a float positive number that indicates the number of units along the propagation axis the beam can propagate without encountering another optical element. This element is used in order to separate spatially the other optical elements. Although a 0 distance is allowed by the solver, it is recommended to avoid such scenarios.

**Lenses:** They are defined by a list having the structure

```
['l', f, error_x, error_y]
```

where "l" is a label used to identify the specific optical element as being a lens, f is the focal length in units, error_x is the shift of the lens along the $x$-axis, and error_y is the shift of the lens along the $y$-axis. The error terms are introduced in order to take into consideration the possibility of misalignment, which is used in the Monte-Carlo function in order to check how tolerances can influence the result of the output optical beam.

It should be noted that a lens is basically a phase mask, which is defined below. The reason for creating a specific structure for this optical element is given by its frequent use and the ability to compute the phase mask of a lens theoretically, thus reducing the RAM requirement for storing the optical system from the size of a typically $10^3 \times 10^3$ complex matrix, or greater, to that of one character.

**Phase masks:** This type of optical element is defined similarly to the lens

```
['mp', M, error_x, error_y]
```

with the main differences being the label "mp" and the matrix M that contains the phase matrix. It should be noted that M should be real valued.

**Amplitude masks:** As in the case of the phase masks, an amplitude mask is defined as

```
['ma', M, error_x, error_y]
```

with the label "ma" and the matrix M that contains the amplitude values. M should be real valued in this case also.

Although the implementation allows for other methods of utilization, the amplitude mask is meant to act as a controllable spatial filter in the sense that it should consist of values in the interval $[0, 1]$. This way the amplitude mask limits the amplitude of the beam based on the values of the matrix elements. A scenario where the values are defined as greater than 1 should be interpreted as an amplification of the incident beam.

Optical systems can now be initialized by creating lists of optical elements. The order in which the optical elements are placed is the same as the order in which the beam propagates through them. An example here should help to better showcase this. consider the case of the optical system

```
optical_system = [100, ['l', 50, 0.5, 0.5], 400, ['mp', M, -0.2, 0.2], 200]
```

From left to write, a beam propagating through optical_system encounters:

- 100: free space of 100 units
- ['l', 50, 0.5, 0.5]: lens with 50 units focal length that is shifted +0.5 on both $x$ and $y$ axes
- 400: free space of 400 units
- ['mp', M, −0.2, 0.2]: phase mask given by M that is shifted by -0.2 units on the $x$ axis and 0.2 units on the $y$ axis

- 200: free space of 200 units

Following this procedure any optical system composed of the above mentioned elements can be defined. The only restrictions that have to be taken into consideration are related to RAM requirements and the design of the setup in order to maintain the optical system inside the finite transverse domain.

For a Monte-Carlo analysis of the optical system some tolerances have to be given for each optical element. This can be done using a second list that has to have the same structure as the optical system. For example an optical system defined as

```
optical_system = [100, ['l', 50, 0.5, 0.5], 400, ['mp', M, -0.2, 0.2], 200]
```

can have an error defined as

```
errors_optical_system = [5, [2, 0.5, 0.4], 4, [0.3, 0.2], 2]
```

where the following interpretation is made:

- 100 units → 100±5 units;
- ['l', 50, 0.5, 0.5] → ['l', 50±2, 0.5±0.5, 0.5±0.4];
- 400 units → 400±3 units;
- ['mp', M, −0.2, 0.2] → ['l', M, −0.2±0.3, 0.2±0.2];
- 200 units → 200±2 units.

The final observation in regard to the ES submodule is related to the way lenses are handled. Since the spatial grid is discrete and the phase of the lens is given by a phase function of the form

$$\exp\left(iax^2\right)$$

a user can encounter situations where the change in phase cannot be covered because of the spacing of the grid. In these scenarios the phase function of the lens does not produce the desired result thus affecting the numeric propagation of the optical beam. This event is referred to in the following as **alias**.

A solution to this problem has been implemented in the FG, NS and ES submodules by allowing for computing the lens phase mask in both normal and Fourier space. This is handled by the **fft_flag** variable. Before the numerical solver is called, an analysis of the optical system is made. For each lens that is identified in the system a check is made in order to see if the identified lenses have alias based on the numeric values found in the Numeric parameters submodule. The result of this checking procedure is a list that specifies how the solution is computed through the optical system.

Consider the following optical system

```
system = [100, ['l', 50, 0,0], 150, ['l', 80, 0, 1], 50]
```

The result of the check function (described in the following) can be a list defined as

```
[['normal', 'fourier'], 'n', ['fourier', 'normal'], 'f', ['normal', 'normal']]
```

which should be interpreted as follows. For each lens the check is made and, under most scenarios, an alias is found in either the normal or the Fourier spaces. Based on this a flag is assigned to the lens "n" for alias in normal space and "f" for alias in Fourier space. This flag is then used to tell the numeric solver how to compute the solution upto and after the lens. If an alias in the normal space is identified, as it is the case for the first lens in the example above, then the solver computes the solution in Fourier space without performing an inverse transform. After the solution is computed as mentioned, the lens phase is computed also in Fourier space and then applied to the solution using convolution. Since the result is still in Fourier space, the solver computes the solution for the next free space region by taking its new initial condition accordingly.

But after this second propagation through free space comes a lens that has alias in Fourier space. This implies that the numerical solver has to return its solution in normal space, thus the inverse Fourier transform is applied. Next the lens phase mask is applied by multiplication after which the solver computes the solution for the last free space its now initial condition in normal space and the result also in normal space.

This entire process is handled through the **fft_input** and **fft_output** variables that are present in the solvers, and the **fft_flag** variable present in the lens_theory and linear_phase functions from FG.

With this last observation, the ES submodule is split in classes that handle the propagation, phase mask retrieval, estimators for RAM usage, and a plotter function. In the order they are listed in the ES submodule they contain the following functions.

### Default functions in the *experimental_simulator_1d* class:

- **def** `check_lens_for_alias(focal, x_dom = None, use_prints = True):`

```
"""
Based on the parameters in numeric_parameters.py and the desired focal length, the
    ↪ lens phase mask is checked for alias. The check is done in normal and fourier
    ↪ space.
Inputs:
    focal = float; real - The focal length of the lens
    x_dom = 1_dimensional array; float - The transversal spatial domain on the X axis
        ↪ .
    use_prints = boolean - If True, all the print calls in the function are activated
        ↪ . Used for debugging.
```

```
    Outputs:
        output = string - Returns one of the 4 possible scenarios:
            1) 'pass' - if no alias is found in both normal and Fourier space
            2) 'n' - if alias is found in normal space
            3) 'f' - if alias is found in Fourier space
    Note: Based on my experience, scenarios 2 and 3 are the most common. Scenario 1 is
        ↪ expected for small values of internal_imports.p.N_x variable.
    """
```

- **def** check_lenses_in_system_for_alias(system, use_prints = True):

```
    """
    Identifies all the aliases for each lens in an optical system by using
        ↪ check_lens_for_alias function. Based on the alias information, the lens phase
        ↪ maskes are computed in the space where no alias is found.
    Inputs:
        system = list - The optical system. Check class documentation for more
            ↪ information on how the optical system is defined.
        use_prints = boolean - If True, all the print calls in the function are activated
            ↪ . Used for debugging.
    Outputs:
        system_warnings = list - A list that speficies the space in which the computation
            ↪  is made.

    Note: The output format is related to the optical system. Consider the optical system

        optical_system = [100, ['l', 10, 0, 0], 110, ['l', 100, 0, 0], 100]

    and consider that the first lens has an alias in normal space, while the second lens
        ↪ has an alias in fourier space. The output in this case is

        system_warnings = [['normal', 'fourier'], 'n', ['fourier', 'normal'], 'f', ['
            ↪ normal', 'normal']].

    See solver.numeric_fourier_solver_1d() variables fft_input and fft_output, and
        ↪ experimental_simulator_1d.propagate() function for more information.
    """
```

- **def** propagate(f0, system, forward = True, norm = True, output_full = True, print_output =
  ↪ False, use_prints = True):

```
    """
    Propagates the beam through the optical system.
    Inputs:
        f0 = 1-dimensional array; complex - The input beam profile.
        system = list - The optical system.
        forward = boolean - If False, the propagation direction through the optical
            ↪ system is reversed.
        norm = boolean - If True, the beam profile is normalized at each step on the
            ↪ propagation axis when print_output = True. Good for visualising the profile
            ↪  on the entire 1+1-dimensional space.
        output_full = boolean - If True, the function returns the solution for each
            ↪ iteration. If False, only the last iteration is returned.
        print_output = boolean - If True, the solution is ploted. Depends on output_full
            ↪ variable.
        use_prints = boolean - If True, all the print calls in the function are activated
            ↪ . Used for debugging.
    Outputs:
        if output_full = True
            f_total = 2-dimensional array; complex - The beam profile for the entire
                ↪ optical system.
        if output_full = False
            f_final = 1-dimensional array; complex - The beam profile at the end of the
                ↪ optical system.
    """
```

- **def** monte_carlo_precision_test(f_in, system, system_errors, number_of_samples):

```
    """
```

```
    Simulate the propagation of a beam considering errors regarding the positioning of
        ↪ optical elements. Based of the errors, various scenarios are considered and
        ↪ simulated in order to check how errors influence the optical system.

    The sampling assumes a uniform distribution inside the intervals.

    Inputs:
        f_in = 1-dimensional array; complex - The input beam profile.
        system = list - The optical system.
        system_errors = list - The errors related to the optical elements.
        number_of_samples = float - Number of parameters sampled and of simulations done.
    Outputs:
        f_out = 1-dimensional array; complex - The beam profile at the end of the optical
            ↪ system.
    """
```

**Default functions in the *mask_generator_1d* class:**

- **def** `compute_mask_dual_system(system1, system2, f_in, f_out, check_mask = False,` `print_output = False, norm = True, use_prints = True):`

```
    """
    Computes the phase mask for a specific optical system that is split in 2 subsystems.

    System diagram:
                Subsystem1                      Subsystem2
    Input[-------------------------]MASK[-------------------------]Output

    The entire optical system is given by:

        optical_system = system1 + ['mp', MASK] + system2

    Inputs:
        system1 = list - Subsystem1.
        system2 = list - Subsystem2.
        f_in = 1-dimensional array; complex - The initial condition.
        f_out = 1-dimensional array; complex - The desired output beam.
        check_mask = boolean - If True, calls check_output_for_mask function.
        print_output = boolean - If True, the whole beam profile is ploted.
        use_prints = boolean - If True, all the print calls in the function are activated
            ↪ . Used for debugging.
    Outputs:
        mask = 1-dimensional array; real - The phase mask
    """
```

- **def** `check_output_for_mask(system, f_in, f_out, output = True, use_prints = True):`

```
    """
    Computes the beam profile through an optical system given by

        optical_system = system1 + [['mp', mask]] + system2

    and then compare the computed output with the desired one (f_out).

    Inputs:
        system = list - The entire optical system with phase mask.
        f_in = 1-dimensional array; complex - The initial condition.
        f_out = 1-dimensional array; complex - The desired output beam.
        output = boolean - If True, the function returns the convolution of the 2 outputs
            ↪ and the shift.
        use_prints = boolean - If True, all the print calls in the function are activated
            ↪ . Used for debugging.
    Outputs:
        if output = True
            corr = 1-dimensional array; complex - The cross-correlation between the
                ↪ computed output and the desired one
            shift = int - The difference in position between the 2 outputs, measured in
                ↪ pixels.
```

```
        if output = False
            None
    """
```

- **def** compute_mask_triple_system(system1, system2, system3, f_in, f_out, check_mask = False, print_output = False, norm = True, use_prints = True):

```
    """
    Computes the phase mask for a specific optical system that is split in 3 subsystems.

    System diagram:
              Subsystem1                    Subsystem2                    Subsystem3
    Input[————————————————————]MASK_1[————————————————————]MASK_2[————————————————————]Output

    The entire optical system is given by:

        optical_system = system1 + [['mp', MASK_1]] + system2 + [['mp', MASK_2]] +
            ↪ system3

    Inputs:
        system1 = list - Subsystem1.
        system2 = list - Subsystem2.
        system3 = list - Subsystem3.
        f_in = 1-dimensional array; complex - The initial condition.
        f_out = 1-dimensional array; complex - The desired output beam.
        check_mask = boolean - If True, calls check_output_for_mask function.
        print_output = boolean - If True, the whole beam profile is ploted.
        use_prints = boolean - If True, all the print calls in the function are activated
            ↪ . Used for debugging.
    Outputs:
        mask1 = 1-dimensional array; real - The phase mask
        mask2 = 1-dimensional array; real - The phase mask
    """
```

- **def** mask_shift(mask, shift_x):

```
    """
    Shift the phase mask on the X direction by an amount given by the user.

    Inputs:
        mask = 1-dimensional array; real - The phase mask.
        shift_x = float; real - Shift on the X axis. Units are the same as in
            ↪ numeric_parameters.py file.
    Outputs:
        mask = 1-dimensional array; real - The shifted phase mask.
    """
```

**Default functions in the *experimental_simulator_2d* class:**

- **def** check_lens_for_alias(focal, x_dom = None, y_dom = None, use_prints = True):

```
        """
        Based on the parameters in numeric_parameters.py and the desired focal length,
            ↪ the lens phase mask is checked for alias. The check is done in normal and
            ↪ fourier space.
        Inputs:
            focal = float; real - The focal length of the lens
            x_dom = 1-dimensional array; float - The transversal spatial domain on the X
                ↪ axis.
            y_dom = 1_dimensional array; float - The transversal spatial domain on the Y
                ↪ axis.
            use_prints = boolean - If True, all the print calls in the function are
                ↪ activated. Used for debugging.
        Outputs:
            output = string - Returns one of the 4 possible scenarios:
                1) 'pass' - if no alias is found in both normal and Fourier space
                2) 'n' - if alias is found in normal space
                3) 'f' - if alias is found in Fourier space
```

```
        Note: Based on my experience, scenarios 2 and 3 are the most common. Scenario 1
            ↪ is expected for small values of internal_imports.p.N_x variable.
        """
```

- **def** check_lenses_in_system_for_alias(system, use_prints = True):

```
        """
        Identifies all the aliases for each lens in an optical system by using
            ↪ check_lens_for_alias function. Based on the alias information, the lens
            ↪ phase maskes are computed in the space where no alias is found.
        Inputs:
            system = list - The optical system. Check class documentation for more
                ↪ information on how the optical system is defined.
            use_prints = boolean - If True, all the print calls in the function are
                ↪ activated. Used for debugging.
        Outputs:
            system_warnings = list - A list that speficies the space in which the
                ↪ computation is made.

        Note: The output format is related to the optical system. Consider the optical
            ↪ system

            optical_system = [100, ['l', 10, 0, 0], 110, ['l', 100, 0, 0], 100]

        and consider that the first lens has an alias in normal space, while the second
            ↪ lens has an alias in fourier space. The output in this case is

            system_warnings = [['normal', 'fourier'], 'n', ['fourier', 'normal'], 'f', ['
                ↪ normal', 'normal']].

        See solver.numeric_fourier_solver_1d() variables fft_input and fft_output, and
            ↪ experimental_simulator_1d.propagate() function for more information.
        """
```

- **def** propagate(f0, system, forward = True, norm = True, output_full = True, print_output =
  ↪ False, use_prints = True):

```
        """
        Propagates the beam through the optical system.
        Inputs:
            f0 = 2-dimensional array; complex - The input beam profile.
            system = list - The optical system.
            forward = boolean - If False, the propagation direction through the optical
                ↪ system is reversed.
            norm = boolean - If True, the beam profile is normalized at each step on the
                ↪ propagation axis.
            output_full = boolean - If True, the function returns the solution for each
                ↪ iteration. If False, only the last iteration is returned.
            print_output = boolean - If True, the solution is ploted. Depends on
                ↪ output_full variable.
            use_prints = boolean - If True, all the print calls in the function are
                ↪ activated. Used for debugging.
        Outputs:
            CONDITION:
                if output_full = True
                    f_total = 3-dimensional array; complex - The beam profile for the
                        ↪ entire optical system.
                if output_full = False
                    f_final = 2-dimensional array; complex - The beam profile at the end
                        ↪ of the optical system.
        """
```

- **def** monte_carlo_precision_test(f_in, system, system_errors, number_of_samples):

```
        """
        Simulate the propagation of a beam considering errors regarding the positioning
            ↪ of optical elements. Based of the errors, various scenarios are considered
            ↪ and simulated in order to check how errors influence the optical system.
```

```
        The sampling assumes a uniform distribution inside the intervals.

        Inputs:
            f_in = 2-dimensional array; complex - The input beam profile.
            system = list - The optical system.
            system_errors = list - The errors related to the optical elements.
            number_of_samples = float - Number of parameters sampled and of simulations
                ↪ done.
        Outputs:
            f_out = 2-dimensional array; complex - The beam profile at the end of the
                ↪ optical system.
        """
```

**Default functions in the *mask_generator_2d* class:**

- **def** compute_mask_dual_system(system1, system2, f_in, f_out, check_mask = False,
  ↪ print_output = False, use_prints = True):

```
        """
        Computes the phase mask for a specific optical system that is split in 2
            ↪ subsystems.

        System diagram:
                    Subsystem1                    Subsystem2
        Input[----------------------]MASK[----------------------]Output

        The entire optical system is given by:

            optical_system = system1 + ['mp', MASK] + system2

        Inputs:
            system1 = list - Subsystem1.
            system2 = list - Subsystem2.
            f_in = 2-dimensional array; complex - The initial condition.
            f_out = 2-dimensional array; complex - The desired output beam.
            check_mask = boolean - If True, calls check_output_for_mask function.
            print_output = boolean - If True, the beam profile at the end of the optical
                ↪ system is ploted.
            use_prints = boolean - If True, all the print calls in the function are
                ↪ activated. Used for debugging.
        Outputs:
            mask = 2-dimensional array; real - The phase mask
        """
```

- **def** check_output_for_mask(system, f_in, f_out, output = True, use_prints = True):

```
        """
        Computes the beam profile through an optical system given by

            optical_system = system1 + [['mp', mask]] + system2

        and then compare the computed autput with the desired one (f_out).

        Inputs:
            system1 = list - Subsystem1.
            mask = 2-dimensional array; real - The phase mask.
            system2 = list - Subsystem2.
            f_in = 2-dimensional array; complex - The initial condition.
            f_out = 2-dimensional array; complex - The desired output beam.
            output = boolean - If True, the function returns the convolution of the 2
                ↪ outputs and the shift.
            use_prints = boolean - If True, all the print calls in the function are
                ↪ activated. Used for debugging.
        Outputs:
            CONDITION:
                if output = True
                    corr = 2-dimensional array; complex - The cross-correlation between
                        ↪ the computed output and the desired one
```

```
                shift = int - The difference in position between the 2 outputs,
                    ↪ measured in pixels.
            if output = False
                None
        """
```

- **def** compute_mask_triple_system(system1, system2, system3, f_in, f_out, check_mask = False ↪ , print_output = False, use_prints = True):

```
        """
        Computes the phase mask for a specific optical system that is split in 3
            ↪ subsystems.

        System diagram:
                  Subsystem1                    Subsystem2                    Subsystem3
        Input[----------------------]MASK_1[----------------------]MASK_2[----------------------]
            ↪ Output

        The entire optical system is given by:

            optical_system = system1 + [['mp', MASK_1]] + system2 + [['mp', MASK_2]] +
                ↪ system3

        Inputs:
            system1 = list - Subsystem1.
            system2 = list - Subsystem2.
            system3 = list - Subsystem3.
            f_in = 2-dimensional array; complex - The initial condition.
            f_out = 2-dimensional array; complex - The desired output beam.
            check_mask = boolean - If True, calls check_output_for_mask function.
            print_output = boolean - If True, the whole beam profile is ploted.
            use_prints = boolean - If True, all the print calls in the function are
                ↪ activated. Used for debugging.
        Outputs:
            mask1 = 2-dimensional array; real - The phase mask
            mask2 = 2-dimensional array; real - The phase mask
        """
```

- **def** mask_shift(mask, shift_x, shift_y):

```
        """
        Shift the phase mask on the X and Y directions by an amount given by the user.

        Inputs:
            mask = 2-dimensional array; real - The phase mask.
            shift_x = float; real - Shift on the X axis. Units are the same as in
                ↪ numeric_parameters.py file.
            shift_y = float; real - Shift on the Y axis. Units are the same as in
                ↪ numeric_parameters.py file.
        Outputs:
            mask = 2-dimensional array; real - The shifted phase mask.
        """
```

**Default functions in the *computing_system_estimators* class:**

- **def** memory_estimator(domain_size, system, output_full = True, size_unit = "GB"):

```
        """
        Computes the required minimum RAM needed for storage of the optical system based
            ↪ on the size of the spatial domain and the optical components in the system.

        Note: This function does not estimate the required RAM for the calculations
            ↪ needed during propagation. This implies that the value returned is a lower
            ↪ bound of the RAM requirement.

        Inputs:
            domain_size = array; int - The shape of the spatial domain e.g. (
                ↪ internal_imports.p.N_x) for a 1-dimensional array with internal_imports
```

```
            ↪ .p.N_x elemens, or (internal_imports.p.N_x, internal_imports.p.N_y) for
            ↪  a 2-dimensional array
        system = list; The optical system
        output_full = boolean - If True, the spatial domain for the entire
            ↪ propagation axis is considered.
        size_unit = string - The unit for the memory space required. Possible inputs
            ↪ are: B for bytes, KB for kilobytes, MG for megabytes, and GB for
            ↪ gigabytes.
    Outputs:
        number_of_bytes = float - The minimum memory space required to store the
            ↪ variables.
    """
```

**Default functions in the *plotters* class:**

- **def** full_beam_plot_1d(f, system, coords = ['z','x'], forward = True, x_number_of_ticks = ↪ None, y_number_of_ticks = None, norm = False, which = 'abs'):

```
    """
    Plots the amplitude of a computed 1+1-dimensional beam using the physical spatial
        ↪  scaling specified in numeric_parameters.py.

    Inputs:
        f = 2-dimensional arraay; complex - The computed beam profile.
        system = list - The optical system.
        coords = list - The meaning of the axes. Usually the solver outputs an array
            ↪ with the order of axis ['z', 'x'].
        x_number_of_ticks = int - The number of major ticks on the X axis.
        y_number_of_ticks = int - The number of major ticks on the Y axis.
        norm = boolean - If true, the amplitude profile is normalized such that the
            ↪ maximum value becomes 1 at every step on the propagation axis.
        which = string - Accepts 3 inputs:
            'abs' - plots the amplitude profile
            'phase' - plots the phase profile
            'both' - plots the product between the amplitude profile and the phase
                ↪ profile
    Outputs:
        None
    """
```

### 1.2.4 Numeric parameters

The Numeric parameters submodule's content is given bellow:

```
"""
List of global variables.
Distances are defined by the 'unit' parameter.
All distances should have the same unit of measurement.
"""

unit = 'mm'

#Spacing along the Z propagation axis
dz = 1*10**-1

#Spacing on the X transverse axis
dx = 4*10**-3

#Spacing on the Y transverse axis
dy = 4*10**-3

#Wavelength in vacuum
wavelength = 0.635 * 10**-3

#Number of points on X axis
N_x = 2000
```

```
#Number of points on Y axis
N_y = 2000

#Number of points on Z axis
N_z = 2*10**3

#Shift on the X transverse axis
x0 = 0.

#Shift on the Y transverse axis
y0 = 0.

#Refractive index of the medium
n0 = 1.0
```

Each variable can be changed by updating its value in the particular instance of the imported pyparax module. If a permanent change is desired, then the setup procedure has to be repeated after the desired modification has been made into the original files.

# 2 Module requirements

The PyParax has been tested with Python versions 3.7 and 3.8 under Windows 10 and Ubuntu 18.04 operating systems. The only difference between choosing either one of the Python versions is the desire for including 3-dimensional surface plots using the **mayavi** module which is compatible only with Python 3.7.

With the exception of the **mayavi** module, the requirements independent of the Python version are:

- Numpy - for array, linear algebra, and FFT related operations
- Scipy - for already implemented special functions such as the Airy function, and correlation related functions for validation of phase mask computation
- matplotlib - for basic plotting of various results

# 3 Setup

PyParax is implemented such that it can be used without installation by importing each submodule independently while allowing for the installation of the module such that it can be imported as any other module from an IDE. The installation procedure consists of calling the following command in CMD for Windows or Terminal for Ubuntu

```
>> python3 setup.py install
```

from the same folder or directory the module is stored in. Anaconda users under Windows will require the use of the Anaconda Prompt instead of CMD.

Once installed, the module can be imported in the standard manner

```
>> import pyparax
```

In order to check its functionality a set of python scripts can be found in the folder named **examples** in the package with some basic scenarios for both 1-dimensional and 2-dimensional transverse domains.

### 3.0.1 Illustrative examples

Based on the different functionalities of the module several examples are presented, namely the propagation of a Gaussian beam through an optical system with two lenses and the computation of a phase mask for optical modulation of the output beam profile and its validation, and some Monte-Carlo simulation of a beam propagating though an optical system with the optical elements aligned with some given tolerances.

**Gaussian beam through an optical system - case 1:** A basic application of the PyParax module is to compute the propagation of a beam profile through an optical system, which is showcased below.

```
from pyparax import function_generator as FG
from pyparax import experimental_simulator as ES

system = [100, ["l", 50, 0,0], 150, ["l", 80, 0, 0], 50]
f0 = FG.standard_initial_conditions.generate_gauss_1d(0, 0.2)
f = ES.experimental_simulator_1d.propagate(f0, system, print_output = True)
```

First we import the FG and ES submodules. An optical system is defined with a structure given by the list attributed to the variable **system**, which can be interpreted as having:

1. 100 - free space of 100 units length (millimeters is the implicit unit);
2. ["$l$", 50, 0, 0] - a lens with focal length of 50 units, and offsets of 1 unit on the $x$ axis and 0 units on the $y$ axis;

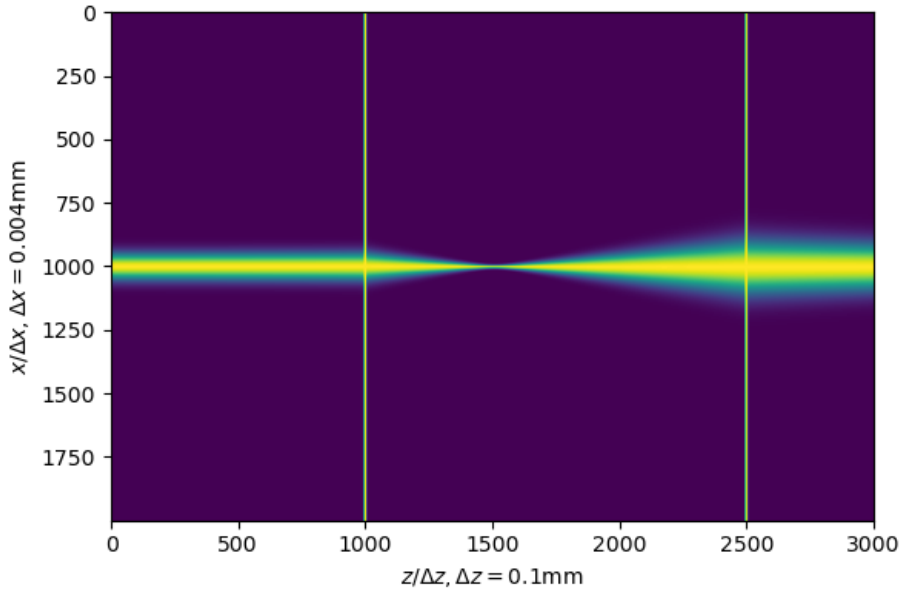


Figure 2: 1+1-dimensional beam profile for a Gaussian beam that propagates through a system of two lenses.

3. 150 - free space of 150 units
4. ["$l$", 80, 0, 0] - a lens with focal length of 80 units, and offsets of 0 units on the $x$ axis and 1 unit on the $y$ axis;
5. 50 - free space of 50 units.

Next the IC is assigned to the **f0** variable which is a Gaussian function of mean 0 units and standard deviation of 0.2 units. Finally the solution is computed and stored in the **f** variable. If the **print_output** variable is *True*, then a plot of the beam profile is made at the end of the computation which is given in figure 2. The 2 vertical lines mark the existence of the lenses at the corresponding position in the optical system.

**Gaussian beam through an optical system - case 2:** In order to highlight the alignment of the optical elements, a modified version of the precious optical system is given below, where the first lens is shifted.

```
from pyparax import function_generator as FG
from pyparax import experimental_simulator as ES

system = [100, ["l", 50, 0.5,0], 150, ["l", 80, 0, 1], 100]
f0 = FG.standard_initial_conditions.generate_gauss_1d(0, 0.2)
f = ES.experimental_simulator_1d.propagate(f0, system, print_output = True)
```

The interpretation of the code sniper is as for the preivous case with the only differences being:

1. ["$l$", 50, 0.5, 0] - a lens with focal length of 50 units, and offsets of 0.5 units on the $x$ axis and 0 units on the $y$ axis;
2. ["$l$", 80, 0, 1] - a lens with focal length of 80 units, and offsets of 0 units on the $x$ axis and 1 unit on the $y$ axis;

The interpretation here is that the shift on the first lens will produce a deviation of the optical beam along the $x$-axis, while the one on the second lens is ignored because the $y$-axis is not used in the propagation for the 1-dimensional case. The second lens does however tilt the beam due to the tilt generated by the first lens but this is idependent of the $y$-axis. This can be checked in figure **??**.

**Monte-Carlo tolerance test:** This example showcases the possibility of computing multiple solutions to the propagation problem, for various values of the tolerance parameters. The following code snippet has been used:

```
from pyparax import function_generator as FG
from pyparax import experimental_simulator as ES

system = [100, ['l', 50, 0.5,0], 150, ['l', 30, 0,0], 100]
system_errors = [0,['l', 0, 0,0], 0, ['l', 0, 0.5, 0], 0]

f0 = FG.standard_initial_conditions.generate_gauss_1d(0, 0.2)
f = ES.experimental_simulator_1d.monte_carlo_precision_test(f0, system, system_errors, 20)
```

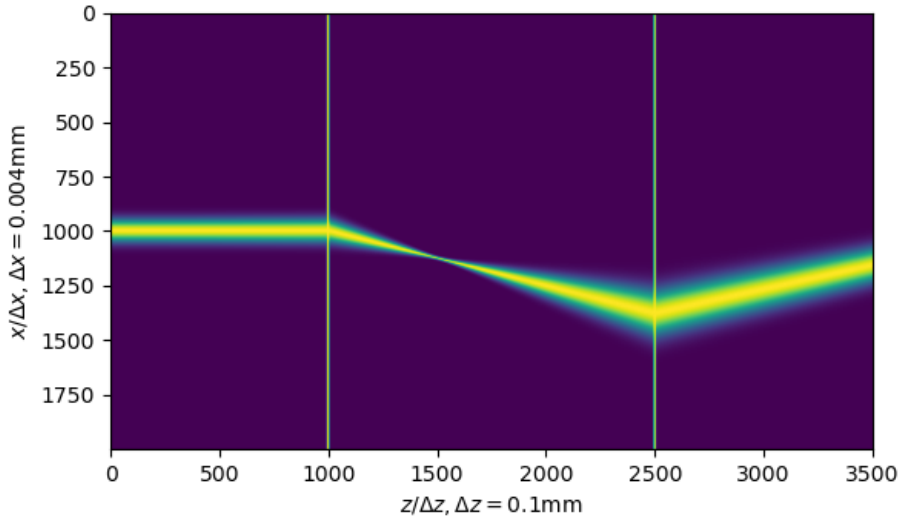


Figure 3: Gaussian beam propagated through a system of two lenses.

As in the previous examples, the main imports are made followed by the initialization of the optical system variable. Then a variable **system_errors** with identical structure as **system** is initialized. It contains the tolerance for the numerical parameters that are used to define each element in **system**. As it is the case in this example, all the optical elements are aligned perfectly with the exception of the second lens which has its position on the $x$-axis centered around 0 (based on the initialization of **system**) with a spread of $\pm 0.5$ units (based on the initialization of **system_errors**). This implies that the position of the second lens along the $x$-axis can be sampled from the interval $[-0.5, 0.5]$ units.

Based on this approach, the function **monte_carlo_precision_test** propagates the initial condition **f0** for a given number of iterations, each time generating an optical system variable with numerical values sampled from their corresponding tolerance intervals. In this example we have considered 20 iterations. For each iteration the amplitude profile of the resulting beam is added to a variable that is returned to initialize **f**.

The plot of all the amplitude profiles is given in figure 4, where as expected, the shift of the second lens changes the position of the spot where the beam focuses on the $x$-axis.

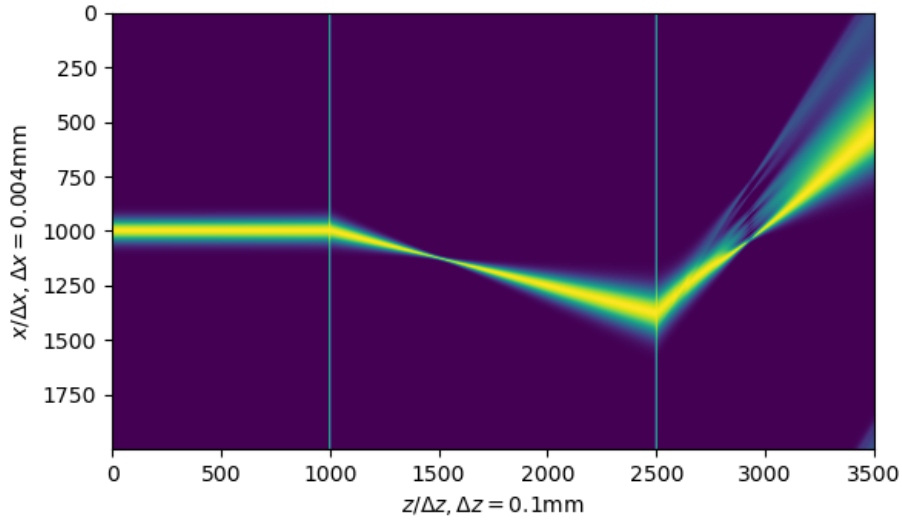


Figure 4: Phase modulated Gaussian beam to generate an Airy beam.

**Phase mask retrieval:** The forth example consists of the computation of a phase mask in order to generate an output beam that resembles the desired one. For this task we use the following code snippet:

```
from pyparax import function_generator as FG
from pyparax import experimental_simulator as ES

system1 = [100, ['l', -50, 0,0], 30]
system2 = [100]
f_in = FG.standard_initial_conditions.generate_gauss_1d(0, 0.6)
```

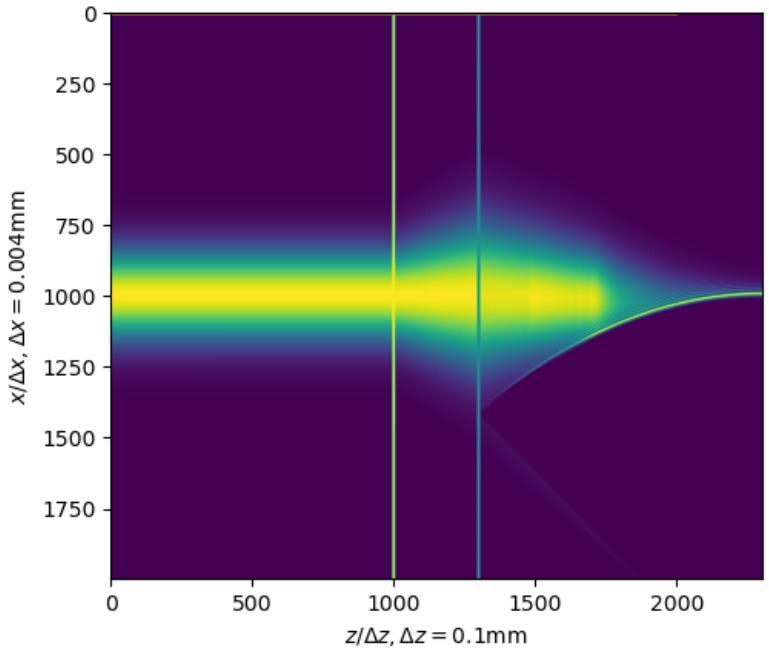


Figure 5: 1+1-dimensional amplitude profile of a Gaussian beam that propagates through a system made of a diverging lens and a phase mask that transforms it into an Airy beam.

```
f_out = FG.standard_initial_conditions.generate_airy_1d(4*10**1, 0, 1*10**-0)
mask = ES.mask_generator_1d.compute_mask_dual_system(system1, system2, f_in, f_out, check_mask = True
    ↪ , print_output = True)
```

In order to compute the phase mask the optical system is split in two parts, namely **system1** which contains all the elements upto the phase mask, and **system2** which contains the elements after the phase mask. The syntax here is similar as in the previous example with the observation that a negative focal length corresponds to a divergent lens. The input beam profile is given by **f_in**, taken as a Gaussian beam, and the desired beam profile is **f_out** which in this case is an Airy function. The parameters of the Airy function generator are in this order **scale** along the transverse axis, **offset** of the main lobe, and **decay** by considering an exponential function that is multiplied with the original function in order to truncate its amplitude profile.

The phase mask is computed using the **compute_mask_dual_system** function inside the class **mask_generator_1d**. This is done by forward propagating the input beam through **system1**, backward propagating the desired output beam through **system2** and computing the phase difference required at the position of the phase mask. The resulting difference is attributed to the **mask** variable. Since not all optical systems are optimal for a given pair of input and desired output profiles, the retrieval is quantitatively described by the maximum of the cross-correlation between the desired output profile and the one that is generated with the retrieved phase mask. This analysis is done if the parameter **check_mask** is *True* and for our choice of parameters the similarity is of approximately 91%. In this case, if **print_output**=*True* a plot of the entire beam profile through the whole system is computed and for this example it is given in figure 5. The second green vertical line represents the position of the phase mask.

**Testing of retrieved phase mask with Monte-Carlo approach:** In this last example a scenario relevant from an experimental perspective is presented. Considering the phase mask from the example above, one could be interested in the manner in which a misalignment of the lens could influence the quality and positioning of the desired output Airy profile. For this purpose we consider the following code snippet:

```
from pyparax import function_generator as FG
from pyparax import experimental_simulator as ES

system1 = [100, ['l', -50, 0,0], 30]
system2 = [100]
f_in = FG.standard_initial_conditions.generate_gauss_1d(0, 0.6)
f_out = FG.standard_initial_conditions.generate_airy_1d(4*10**1, 0, 1*10**-0)
mask = ES.mask_generator_1d.compute_mask_dual_system(system1, system2, f_in, f_out,
    ↪ check_mask = True, print_output = True)

system = system1 + [['mp', mask, 0,0]] + system2
system_errors = [0, ['l', 0, 1, 0], 0, ['mp', 0, 0, 0], 0]
```

```
f = ES.experimental_simulator_1d.monte_carlo_precision_test(f_in, system, system_errors, 20)
```

This is equivalent with the setup from figure 5 with the only difference being that the lens has a tolerance of $\pm 1$ units on the $x$-axis. In order to test the effect of a misalignment comparable to the value of the tolerance in this example, an optical system containing all the optical elements is constructed, the errors associated with each optical element are defined, and then the same function for the Monte-Carlo test is called. The resulting cumulative sum of amplitude profiles is plotted in figure 6 where it can be seen that the profile of the Airy beam is shifted significantly on both the transverse and propagation axis because of the $\pm 1$ units shift of the lens.

This result can be used in order to asses that gravity of the misalignment, while also allowing to rapidly design and test other optical systems that are more resilient to misalignment.

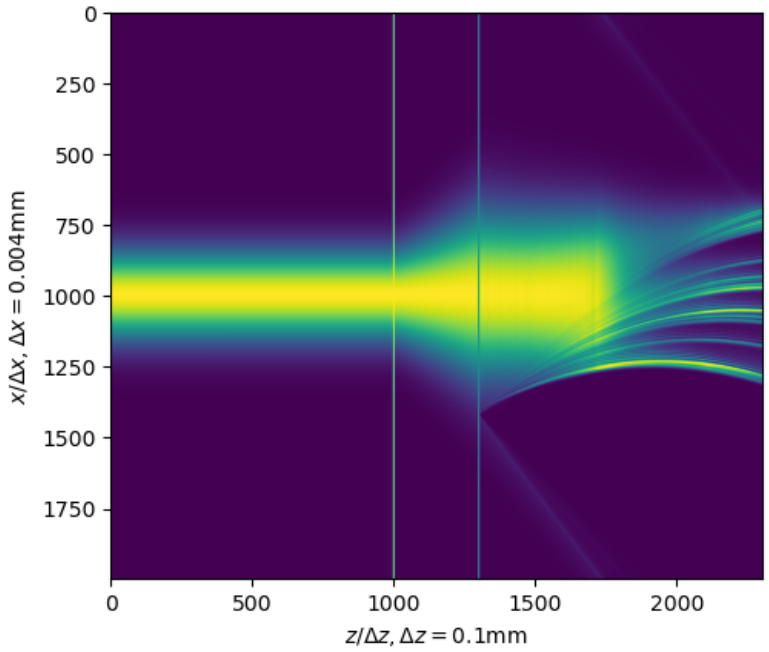


Figure 6: Monte-Carlo test of retrieved phase mask.

Other scenarios can be found in the **examples** folder.