Python diffraction and interference Documentation

Release 0.3.1

Luis Miguel Sanchez Brea

Mar 05, 2024

CONTENTS:

CHAPTER

ONE

PYTHON DIFFRACTION-INTERFERENCE MODULE

- Free software: MIT license
- Documentation: https://diffractio.readthedocs.io/en/latest/



1.1 Features

Diffractio is a Python library for Diffraction and Interference Optics.

It implements Scalar and vector Optics. The main algorithms used are:

- Fast Fourier Transform (FFT).
- Rayleigh Sommerfeld (RS).
- Chirp z-transform (CZT).
- Plane Wave Descomposition (PWD).
- Beam Propagation Method (BPM).
- Wave Propagation Method (WPM).
- Vectorial Rayleigh-Sommerfeld (VRS).
- Vector Fast Fourier Tranform (VFFT).
- Vector Chirp z-transform (VCZT).

When possible, multiprocessing is implemented for a faster computation.

The **scalar** propagation schemes are implemented in modules:

- X - fields are defined in the x axis.
- Z - fields are defined in the z axis (for visualiztion and analysis).

- XZ - fields are defined in the xz plane, being z the propagation direction.
- XY - fields are defined in the xy transversal plane.
- XYZ - fields are defined in the xyz volume.

Each scheme present three modules:

- sources: Generation of light.
- masks: Masks and Diffractive Optical elements.
- fields: Propagation techniques, parameters and general functions.

The **vector** propagation schemes are implemented in modules:

- vector_XY - Ex, Ey, and Ez electric field components are defined, which allows polarization analysis.

There are additional schemes used mainly for representation.

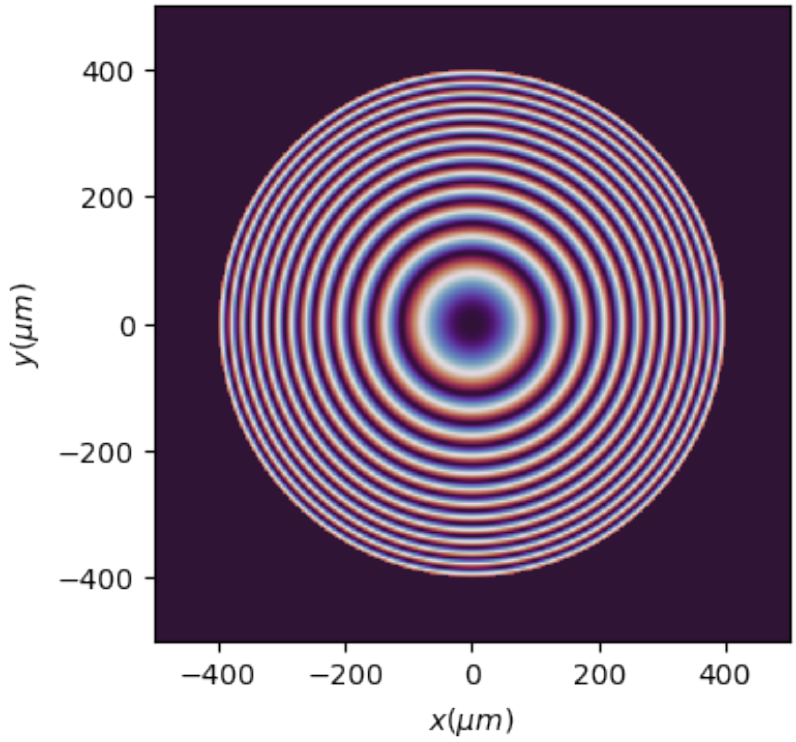
- vector_X - fields are defined in the z axis (for visualiztion and analysis).
- vector_Z - fields are defined in the z axis (for visualiztion and analysis).
- vector_XZ - fields are defined in the xz axis (for visualiztion and analysis).
- vector_XYZ - fields are defined in the xyz axis (for visualiztion and analysis).

For the vector analysis, we also use the py_pol module: https://pypi.org/project/py-pol/

1.1.1 Sources

One main part of this software is the generation of optical fields such as:

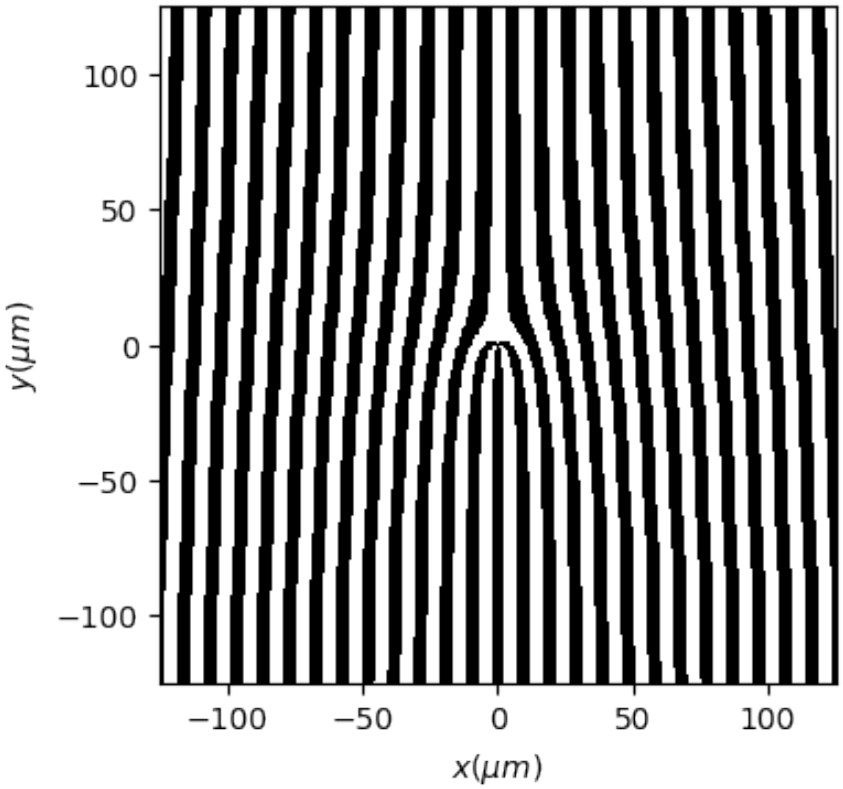
- Plane waves.
- Spherical waves.
- Gaussian beams.
- Bessel beams.
- Vortex beams.
- Laguerre beams.
- Hermite-Gauss beams.
- Zernike beams.



1.1.2 Masks

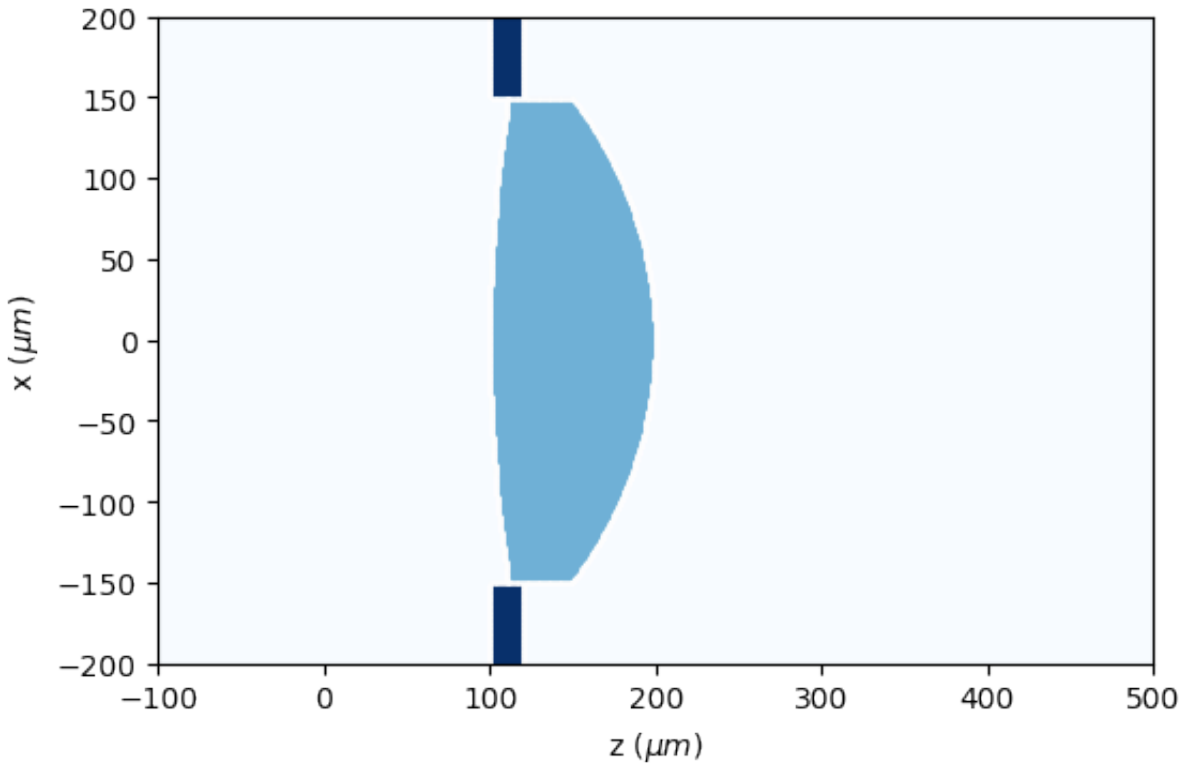
Another important part of Diffractio is the generation of masks and Diffractive Optical Elements such as:

- Slits, double slits, circle, square, ring …
- Lenses, diffractive lenses, aspherical lenses…
- Axicon, prisms, biprism, image, rough surface, gray scale …
- Gratings: Ronchi, phase, radial, angular, sine, forked, blazed, 2D, chess…



In the XZ scheme there is also a number of masks:

- image, sphere, layer, lenses, wedge, prism, probe, gratings…
- Additionally, any X_mask can be extruded to the XZ scheme.
- Masks from functions
- Surfaces can be added to objects



1.1.3 Fields

Several propagation algorithms for propagation of light are implemented in the *fields* modules:

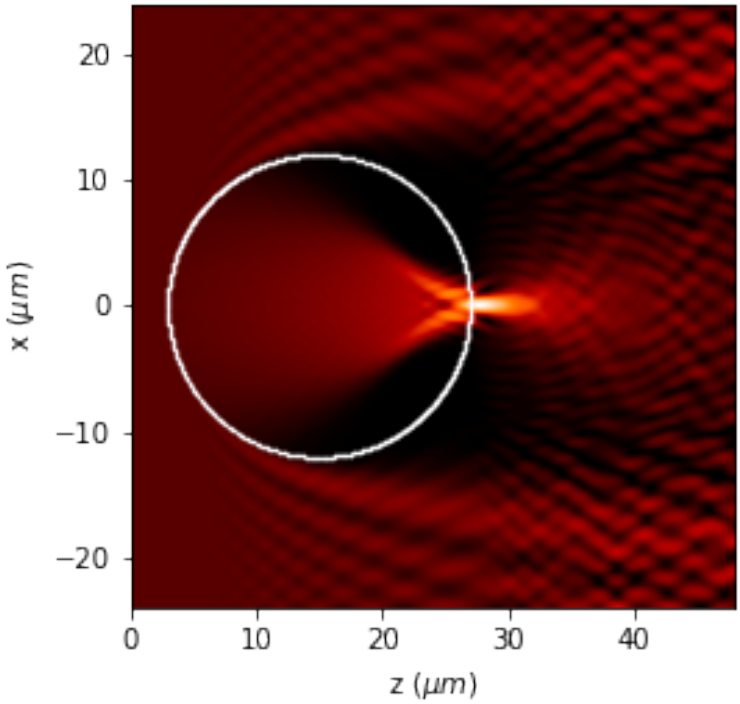
- **Fast Fourier Transform (FFT)** which allows, in a single step, to determine the field at the far field.
- **Rayleigh-Sommerfeld (RS)** [Appl. Opt., 45(6) 1102–1110, (2006)] RS allows, in a single step, to propagate to a near or far observation plane, which allows fast computations. The fields and the masks must be defined in a plane.
- **Beam propagation method (BPM)** [Appl. Opt. 24, 3390-3998 (1978)] which allows to analyze the propation of light in volumetric elements, such as spheres, cylinders and other complex forms, provided that the spatial variations in the refraction index are small. It allows graded index structures. It presents a complexity of O(n) in the two-dimensional and O(n2) in the three-dimensional case. It is computed according to the split-step propagation scheme.
- **Wave Propagation Method (WPM)**. [Appl. Opt. 32, 4984 (1993)] WPM was introduced in order to overcome the major limitations of the beam propagation method (BPM). With the WPM, the range of application can be extended from the simulation of waveguides to simulation of other optical elements like lenses, prisms and gratings. WPM can accurately simulate scalar light propagation in inhomogeneous media at high numerical apertures, and provides valid results for propagation angles up to 85° and that it is not limited to small index variations in the axis of propagation. Fast implementation with discrete number of refraction indexes is also implemented.
- **Chirped Z-Transform (CZT)**. [Light: Science and Applications, 9(1), (2020)] CZT allows, in a single step, to propagate to a near or far observation plane. It present advantages with respecto to RS algorithm, since the region of interest and the sampling numbers can be arbitrarily chosen, endowing the proposed method with superior flexibility. CZT algorithm allows to have a XY mask and compute in XY, Z, XZ, XYZ schemes, simply defining the output arrays.

- **Vector Rayleigh-Sommerfeld (VRS)**. The VRS method [Laser Phys. Lett. 10(6) 065004 (2013)] allows to propagate (Ex,Ey,Ez) fields offering the advantage of significant reduction in computation, from flat diffractive elements (Thin Element Approximation) with full control of polarization. It addresses simultaneously both longitudinal polarization. This approach offers the advantage of significant reduction in computation.
- **Vector Chirped Z-Transform (VCZT)**. [Light: Science and Applications, 9(1), (2020)]. CZT is also implemented in vector fields.

The fields, masks and sources can be stored in files.

Also drawings can be easily obtained, for intensity, phase, fields, etc.

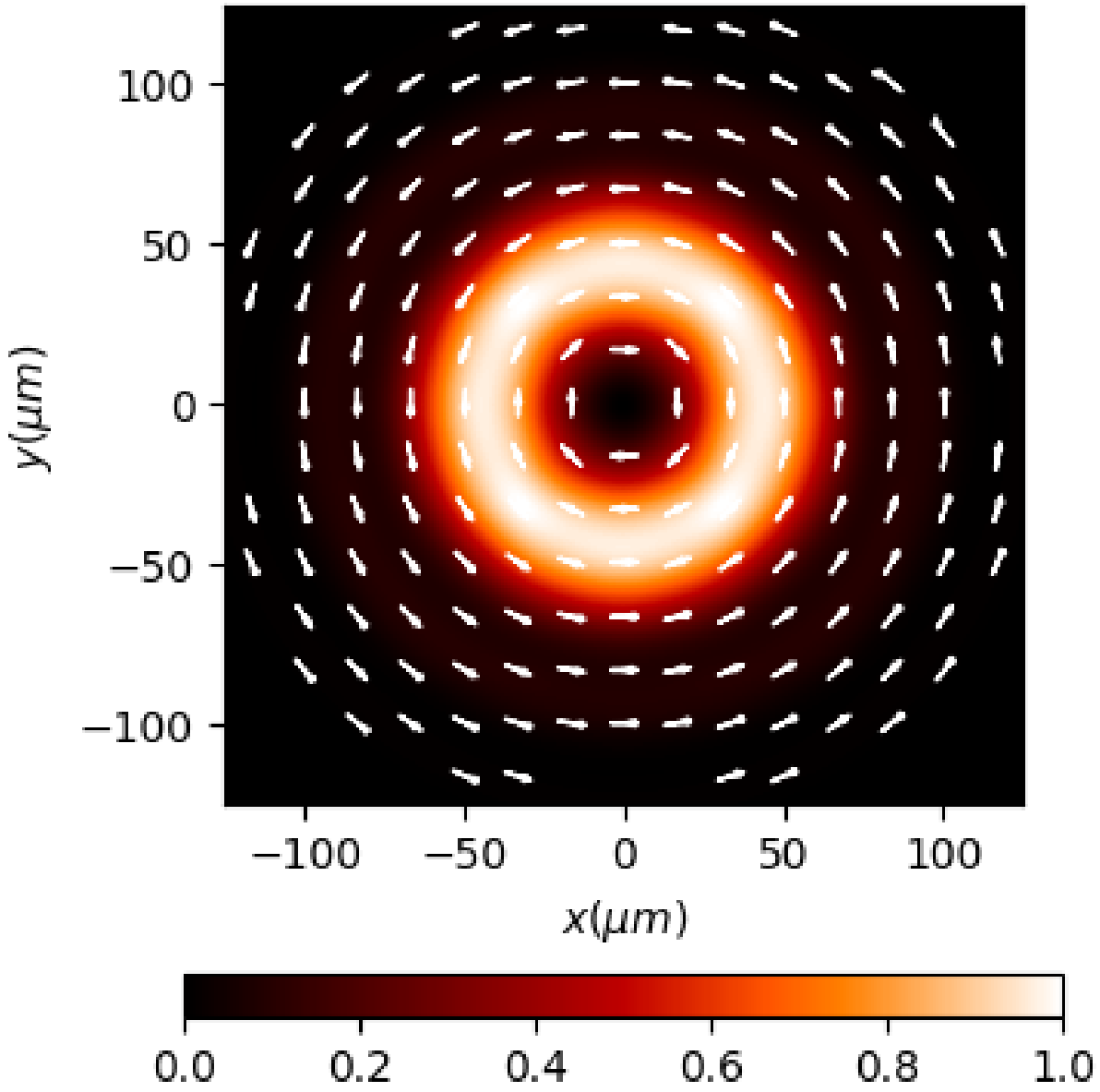
In some modules, videos can be generated for a better analysis of optical fields.



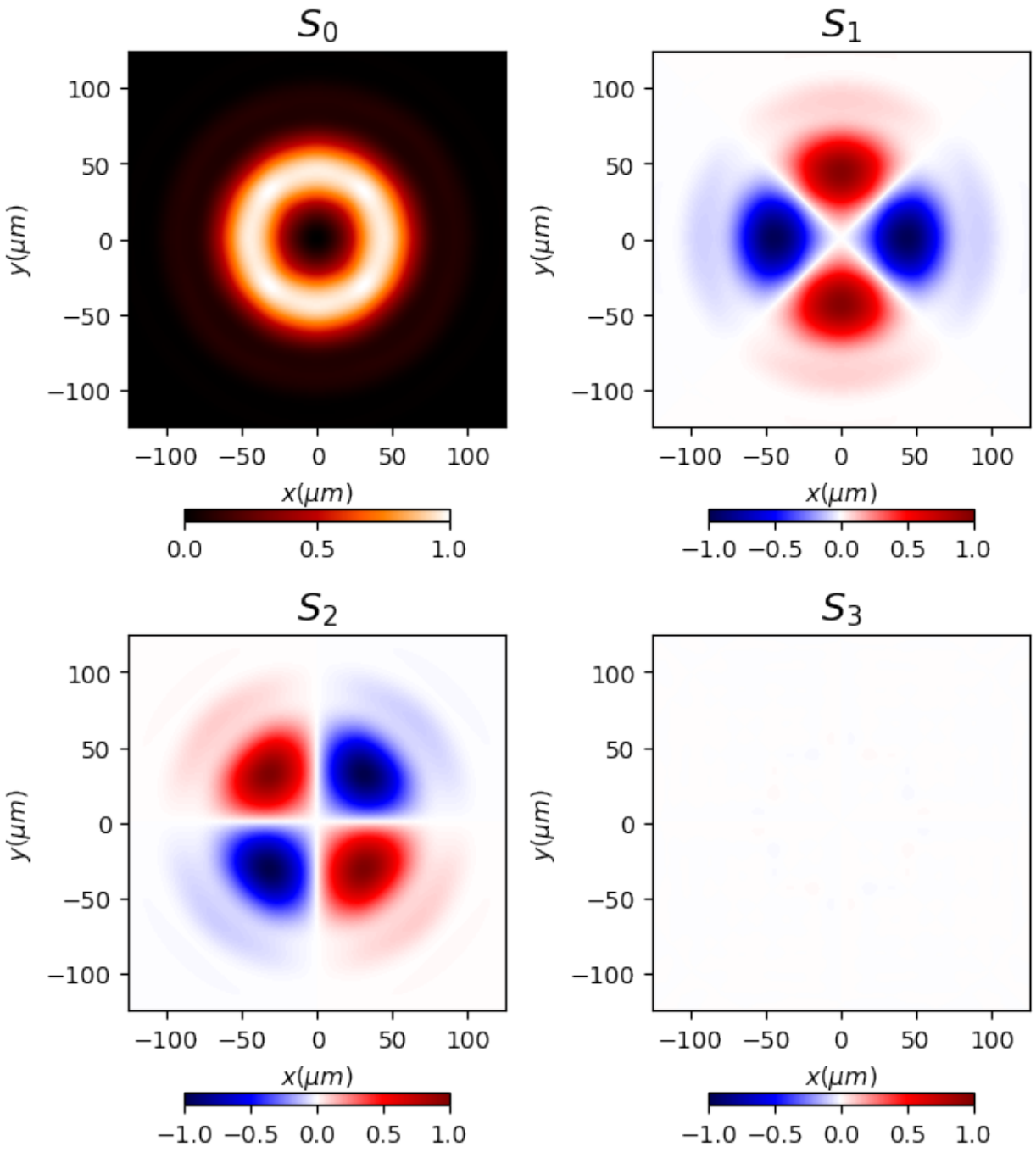
1.1.4 Vector beams

Here, we implement new classes where the E_x, E_y, and E_z fields are generated and propagated using Rayleigh-Sommerfeld and Chirped z-transform algorithms. Also, simple and complex polarizing masks can be created.

Intensity of vector field



Polarization: Stokes parameters


S_0
S_1
S_2
S_3
$y(\mu m)$
$x(\mu m)$
100
50
0
−50
−100
0.0
0.5
1.0
−1.0
−0.5

1.1.5 Other features

- Intensity, MTF and other parameters are obtained from the optical fields.
- Fields can be added simply with the + signe, and interference is produced. Masks can be multiplied, added and substracted in order to make complex structures
- Resampling fields in order to analyze only areas of interest.
- Save and load data for future analysis.
- Rayleigh-Sommerfeld implementation is performed in multiprocessing for fast computation.
- Polychromatic and extended source problems can also be analyzed using multiprocessing.

1.2 Authors

- Luis Miguel Sanchez Brea <optbrea@ucm.es>

  Universidad Complutense de Madrid, Faculty of Physical Sciences, Department of Optics Plaza de las ciencias 1, ES-28040 Madrid (Spain)



1.3 Collaborators

- Ángela Soria Garcia
- Jesús del Hoyo Muñoz
- Francisco Javier Torcal-Milla

1.4 Citing

L.M. Sanchez Brea, “Diffractio, python module for diffraction and interference optics”, https://pypi.org/project/diffractio/ (2019)

1.5 References for algorithms

RS

- Shen, F. & Wang, A. "Fast-Fourier-transform based numerical integration method for the Rayleigh-Sommerfeld diffraction formula. Appl. Opt. 45, 1102–1110 (2006).

WPM

- K.-H. Brenner, W. Singer, "Light propagation through micro lenses: a new simulation method", Appl. Opt., 32(6) 4984-4988 (1993).
- Schmidt, S. et al. "Wave-optical modeling beyond the thin-element-approximation" Opt. Express 24, 30188 (2016).
- Brenner, K.H. "A high-speed version of the wave propagation method applied to micro-optics." 16th Workshop on Information Optics (WIO). IEEE (2017)
- Schmidt, S., et al. "Rotationally symmetric formulation of the wave propagation method-application to the straylight analysis of diffractive lenses" Opt. Lett. 42, 1612 (2017).

VRS

- Ye, H. et al. "Creation of a longitudinally polarized subwavelength hotspot with an ultra-thin planar lens: Vectorial Rayleigh-Sommerfeld method" Laser Phys. Lett. 10, (2013).

CZT

- Bluestein, L., "A linear filtering approach to the computation of the discrete Fourier transform," Northeast Electronics Research and Engineering Meeting Record 10, 218-219 (1968).
- Hu Y. et al. "Efficient full-path optical calculation of scalar and vector diffraction using the Bluestein method" Light: Science & Applications 9(119) (2020)

VCZT

- Leutenegger M. et al. "Fast focus field calculations" Optics Express 14(23) 11277 (2006).
- Hu Y. et al. "Efficient full-path optical calculation of scalar and vector diffraction using the Bluestein method" Light: Science & Applications 9(119) (2020)

VWPM (not implemented yet)

- Fertig, M., Brenner, K.-H. "Vector wave propagation method". J. Opt. Soc. Am. A 27, 709 (2010).
- Fertig, M., Brenner, K.-H. "Vector wave propagation method". PhD thesis (Mannheim, 2010).

1.6 Other References

- J.W. Goodman, "Introduction to Fourier Optics" McGraw-Hill, 1996.
- B.E. Saleh y M. C. Teich, "Fundamentals of photonics" John Wiley & Sons, 2019.
- Z.Qiwen, "Vectorial optical fields: Fundamentals and applications" World scientific, 2013.
- "Numerical Methods in Photonics Lecture Notes". http://ecee.colorado.edu/~mcleod/teaching/nmip/lecturenotes.html.

1.7 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Python diffraction and interference, run this command in your terminal:

```
# Linux:
$ pip3 install diffractio

# Windows:
$ pip install diffractio
```

This is the preferred method to install Python diffraction and interference, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 Additional packages

Diffractio uses also the following non-standard modules:

- py-pol

In some schemes, the following modules are also required:

- numpy, scipy, matplotlib
- Pillow, numexpr, pandas, screeninfo, opencv-python
- ipywidgets, ipympl

They should previously be installed before Diffractio module.

2.3 From sources

The sources for Python diffraction and interference can be downloaded from the Bitbucket repo.

You can either clone the public repository:

```
$ git clone git@bitbucket.org:optbrea/diffractio.git
$ git clone https://optbrea@bitbucket.org/optbrea/diffractio.git
```

Once you have a copy of the source, you can install it with:

```
# Linux:
$ python3 setup.py install

# Windows:
$ python setup.py install
```

CHAPTER

THREE

USAGE

To use Python diffraction and interference in a project:

```
import diffractio
```

3.1 X fields

```
from diffractio import um, nm, mm, np
from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X

x = np.linspace(-500 * um, 500 * um, 4096)
wavelength = .6328 * um

u0 = Scalar_source_X(x, wavelength)
u0.gauss_beam(x0=0, w0=300*um, z0=0)
u0.draw(filename='usage1.png')

t0=Scalar_mask_X(x, wavelength)
t0.slit(x0=0, size=250*um)
t0.draw(filename='usage2.png')

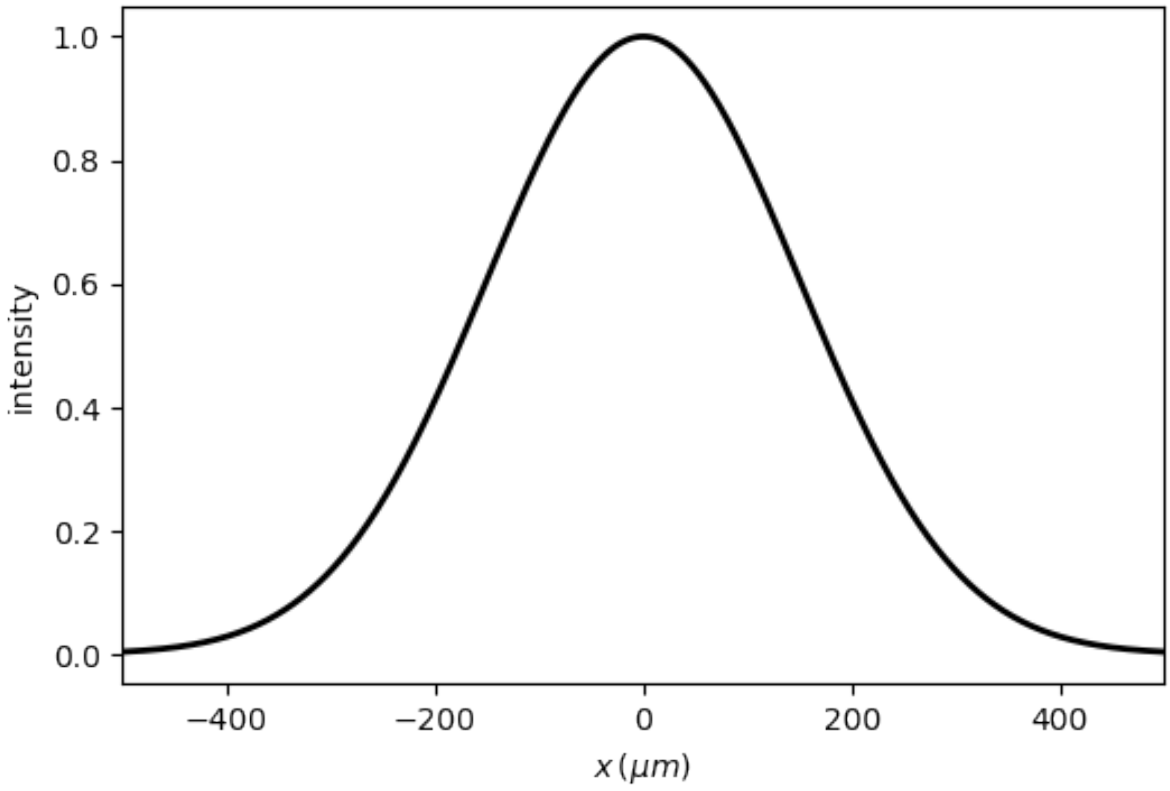
u1=t0*u0
u2 = u1.RS(z=5 * mm, new_field=True)
u2.draw(filename='usage3.png')
```

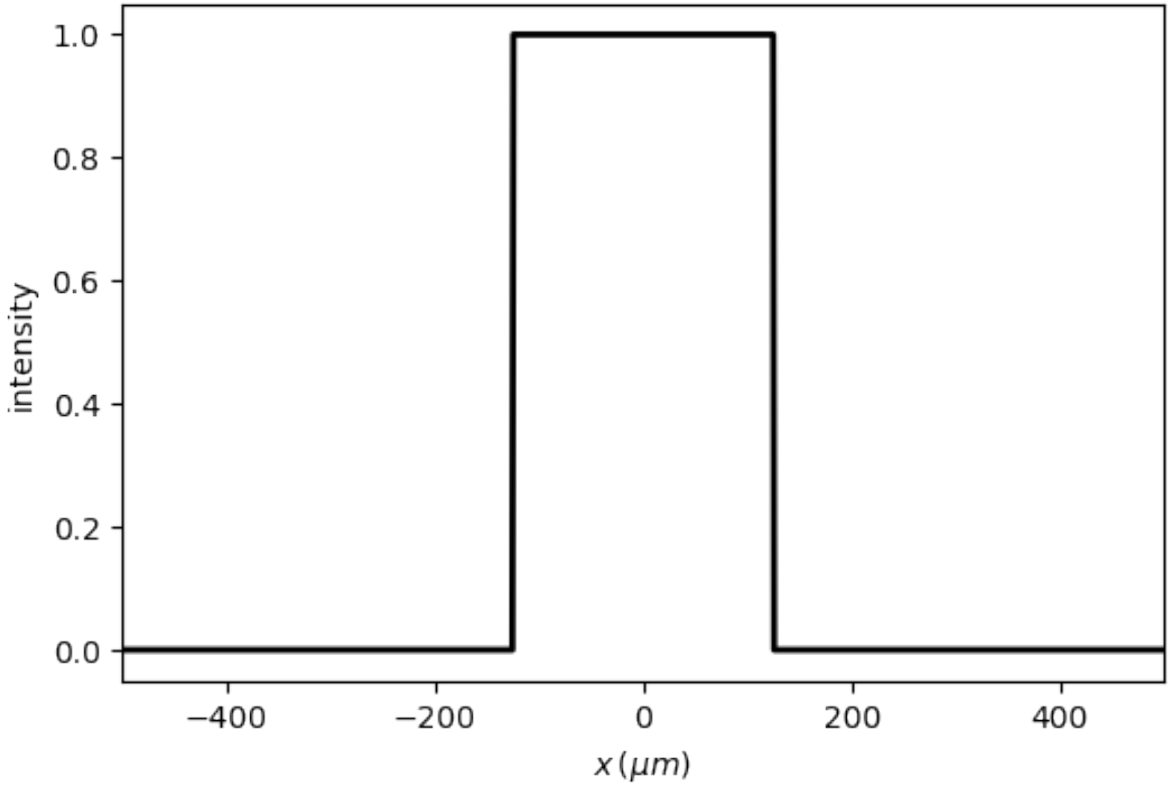
3.2 XZ fields

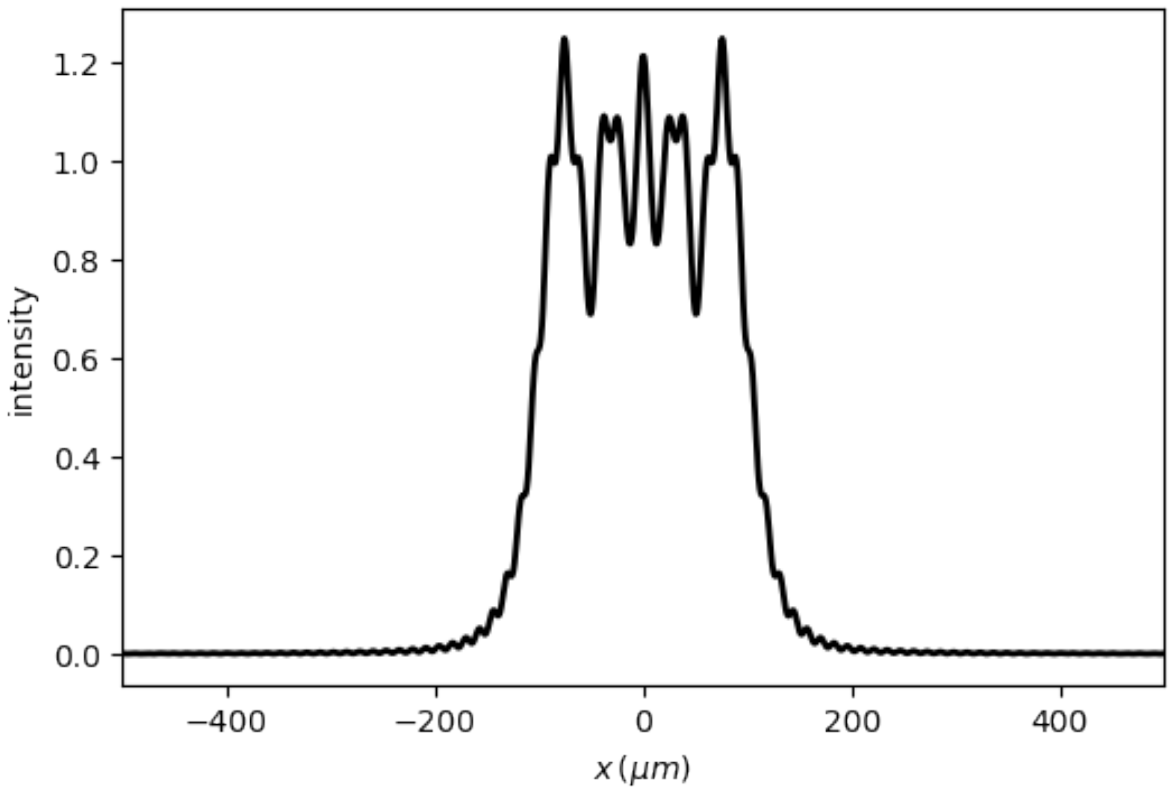
```
from diffractio import um, nm, mm, np,degrees
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ

x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 300 * um, 512)
wavelength = 5 * um
```

(continues on next page)


1.0
0.8
0.6
0.4
0.2
0.0
intensity
−400
−200
0
200
400
x (μm)


1.0
0.8
0.6
0.4
0.2
0.0
intensity
−400
−200
0
200
400
x (μm)

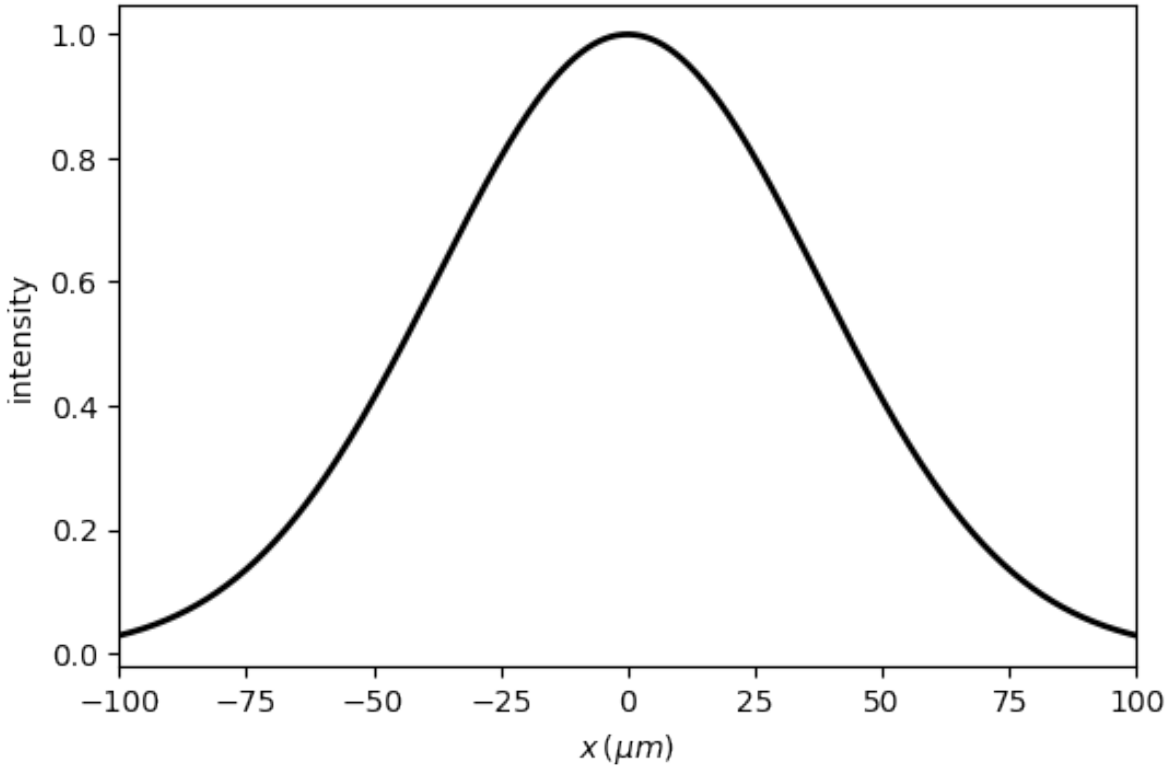


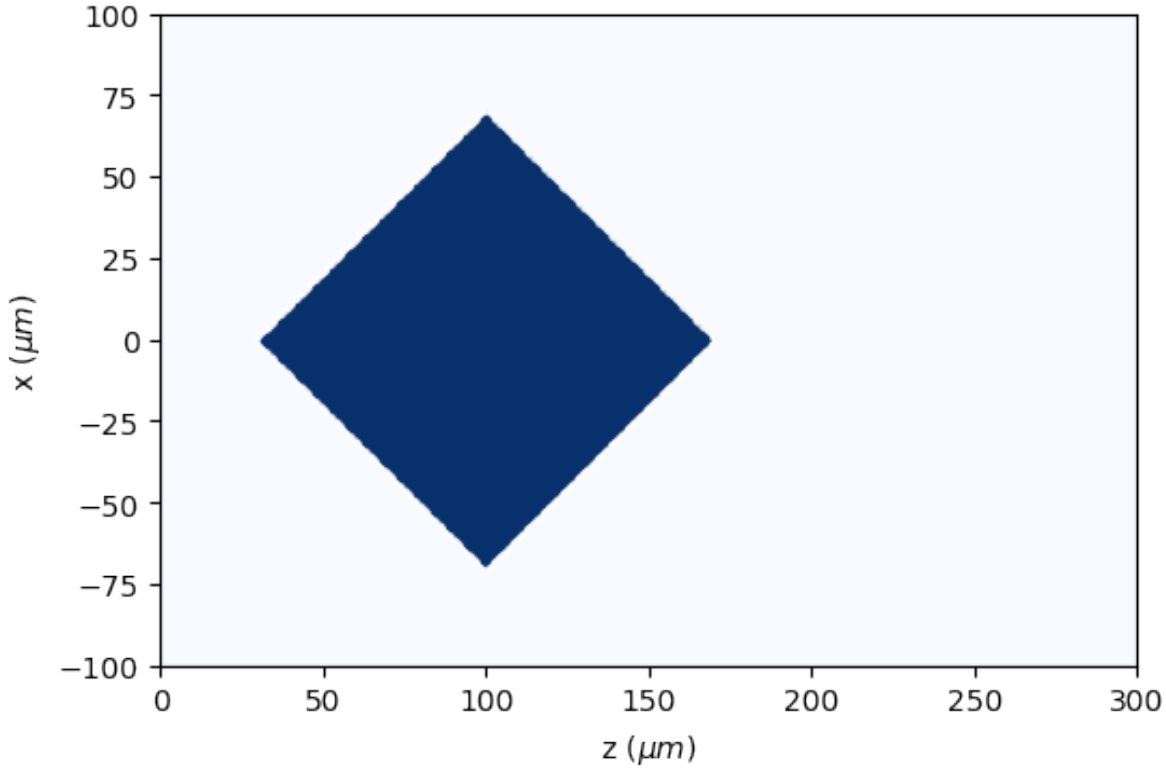
(continued from previous page)

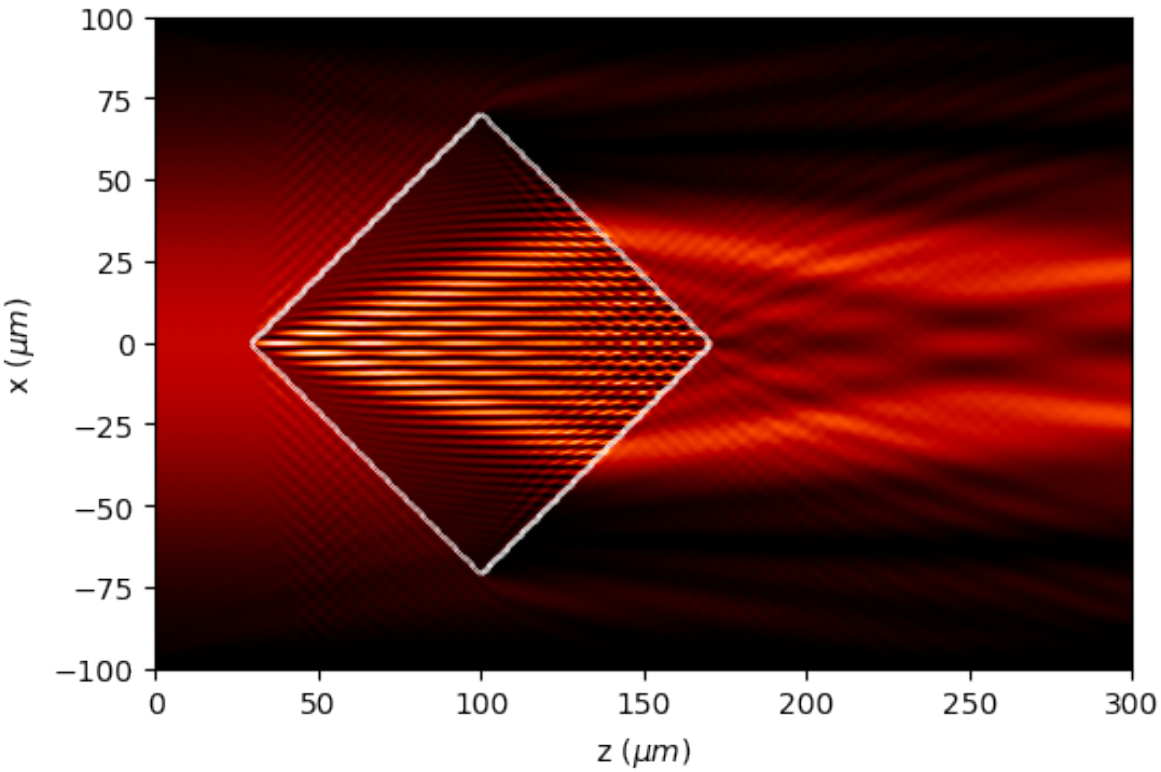
```
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=75 * um, theta=0 * degrees)
u0.draw(kind='field', filename='usage4.png')

u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.rectangle(r0=(0 * um, 100 * um),
  size=(100 * um, 100 * um),
  angle=45 * degrees,
  refraction_index=2)

u1.draw_refraction_index(filename='usage5.png', scale='scaled')
u1.WPM(verbose=False)
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, filename='usage6.png',␣
↪scale='scaled');
```


1.0
0.8
0.6
0.4
0.2
0.0
intensity
−100
−75
−50
−25
0
25
50
75
100
x (μm)


100
75
50
25
0
−25
−50
−75
−100
x (μm)
0
50
100
150
200
250
300
z (μm)



3.3 XY fields

```
from diffractio import um, nm, mm, np, degrees
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY


length = 1000*um
x0 = np.linspace(-length/2, length/2, 512)
y0 = np.linspace(-length/2, length/2, 512)
wavelength = 0.6238 * um

focus=10*mm


u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

u0.zernike_beam(A=1, r0=(0, 0), radius=length / 2, n=[ 0,4,], m=[0,3,], c_nm=[ 1,0.25,])
u0.draw(kind='phase', filename='usage7.png')

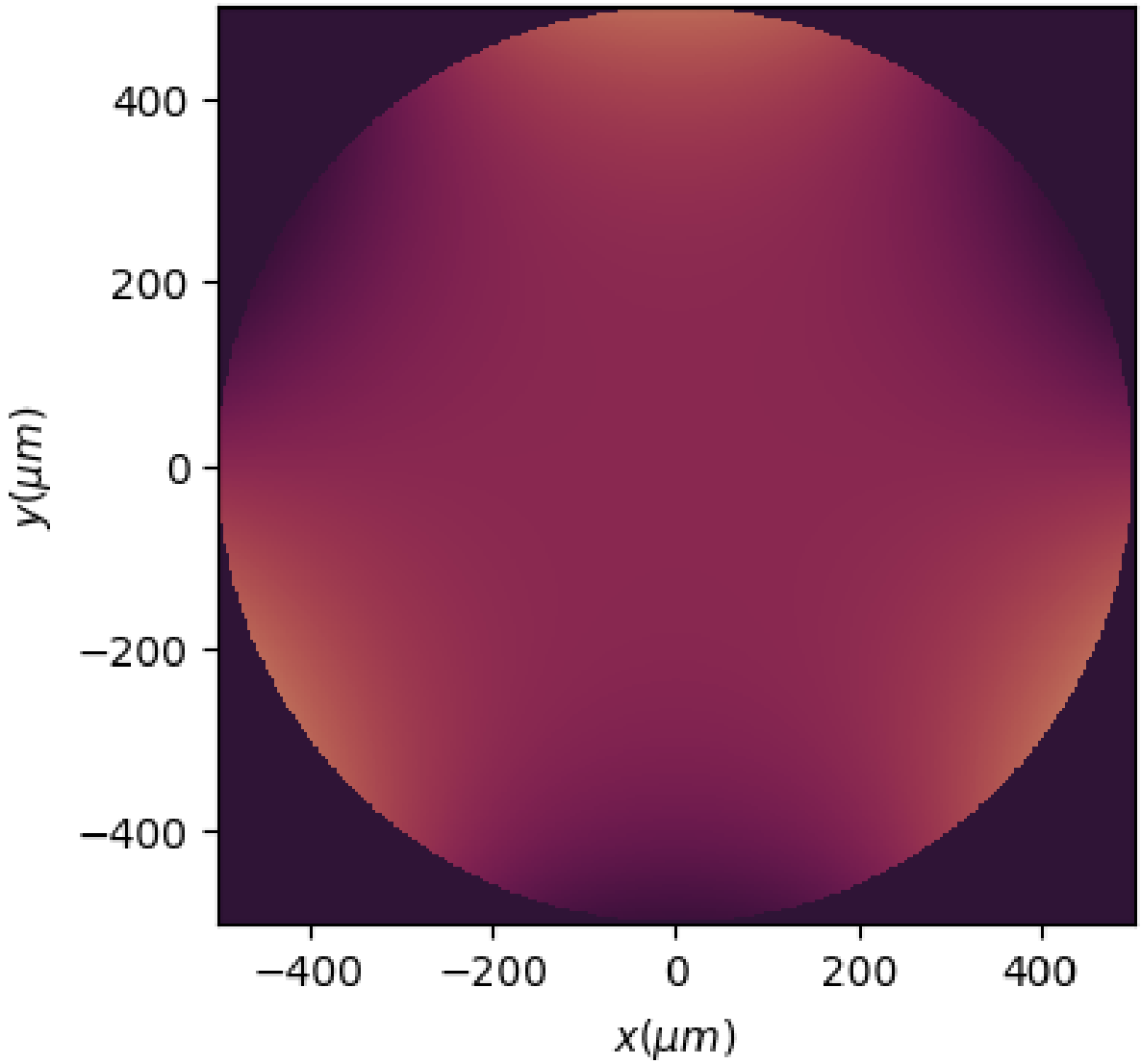
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.fresnel_lens(
  r0=(0 * um, 0 * um),
  radius=(500 * um, 500 * um),
```

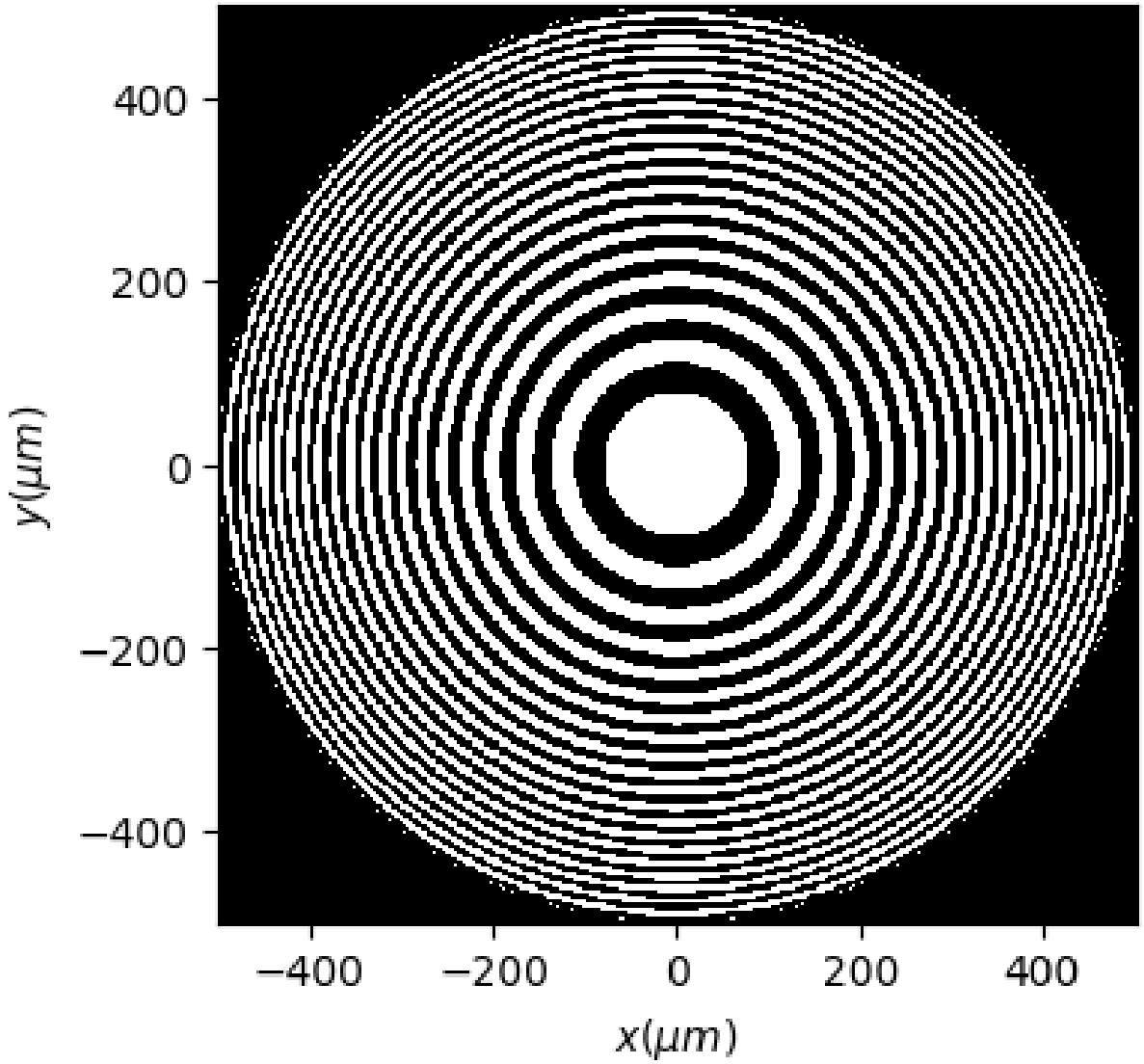
(continues on next page)

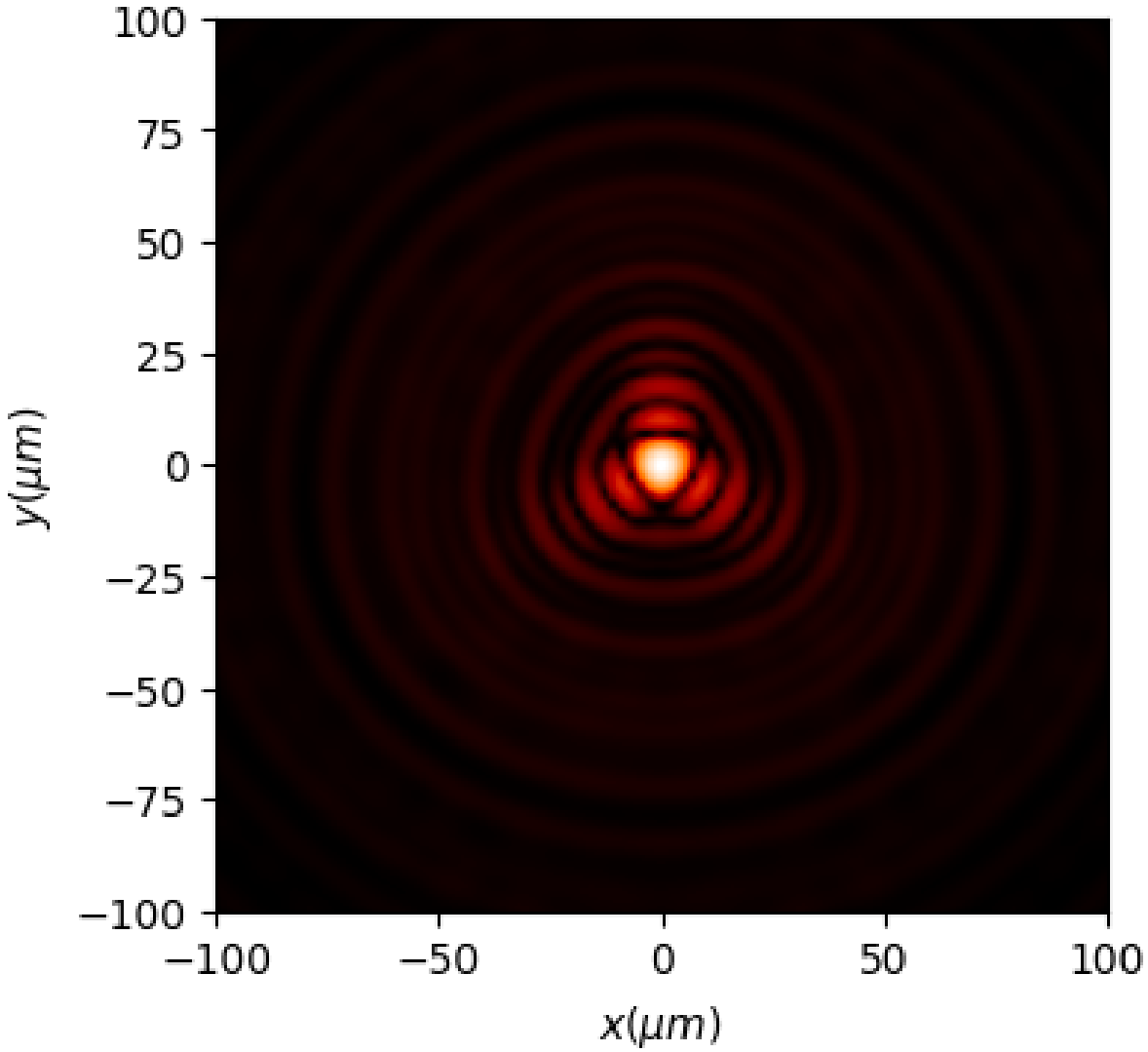
(continued from previous page)

```
  focal=(focus,focus),
  angle=0 * degrees,
  kind='amplitude',
  phase=np.pi)
t0.draw(filename='usage8.png')

u1 = u0 * t0
u2 = u1.RS(z=focus, new_field=True)
u2.cut_resample(
  x_limits=(-100 * um, 100 * um),
  y_limits=(-100 * um, 100 * um),
  num_points=[1024, 1024],
  new_field=False)
u2.draw(kind='intensity', logarithm=True, filename='usage9.png')
```




400
200
0
−200
−400
y(μm)
−400
−200
0
200
400
x(μm)


100
75
50
25
0
−25
−50
−75
−100
y(μm)
−100
−50
0
50
100
x(μm)

3.4 XYZ fields

```
from diffractio import degrees, eps, mm, no_date, np, um
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
from diffractio.scalar_sources_XY import Scalar_source_XY

x0 = np.linspace(-25 * um, 25 * um, 128)
y0 = np.linspace(-25 * um, 25 * um, 128)
z0 = np.linspace(100 * um, 500 * um, 256)
wavelength = .6328 * um

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.circle(
    r0=(0 * um, 0 * um), radius=(10 * um, 10 * um), angle=0 * degrees)
t1.draw(filename='usage10.png')

uxyz = Scalar_mask_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
uxyz.incident_field(u0=t1)

uxyz.RS(verbose=True, num_processors=4)

uxyz.draw_XYZ(y0=0 * mm, logarithm=True, normalize='maximum',filename='xyz_RS.png')
```

3.5 XY Vector fields

```
from diffractio import np, sp, plt
```

from diffractio import nm, um, mm, degrees

from diffractio.scalar_sources_XY import Scalar_source_XY from diffractio.scalar_masks_XY import Scalar_mask_XY from diffractio.scalar_fields_XY import Scalar_field_XY

from diffractio.vector_sources_XY import Vector_source_XY from diffractio.vector_masks_XY import Vector_mask_XY from diffractio.vector_fields_XY import Vector_field_XY

x0 = np.linspace(-125 * um, 125 * um, 256) y0 = np.linspace(-125 * um, 125 * um, 256)

wavelength = 0.6328 * um

u0 = Scalar_source_XY(x0, y0, wavelength) u0.gauss_beam(

r0=(0, 0), w0=(100 * um, 100 * um), z0=0 * um, A=1, theta=0. * degrees, phi=0 * degrees)

EM0 = Vector_source_XY(x0, y0, wavelength) EM0.azimuthal_wave(u=u0, r0=(0, 0), radius=(200, 200)) EM0.draw(kind='ellipses') plt.title('Before mask') plt.savefig('usage12.png')

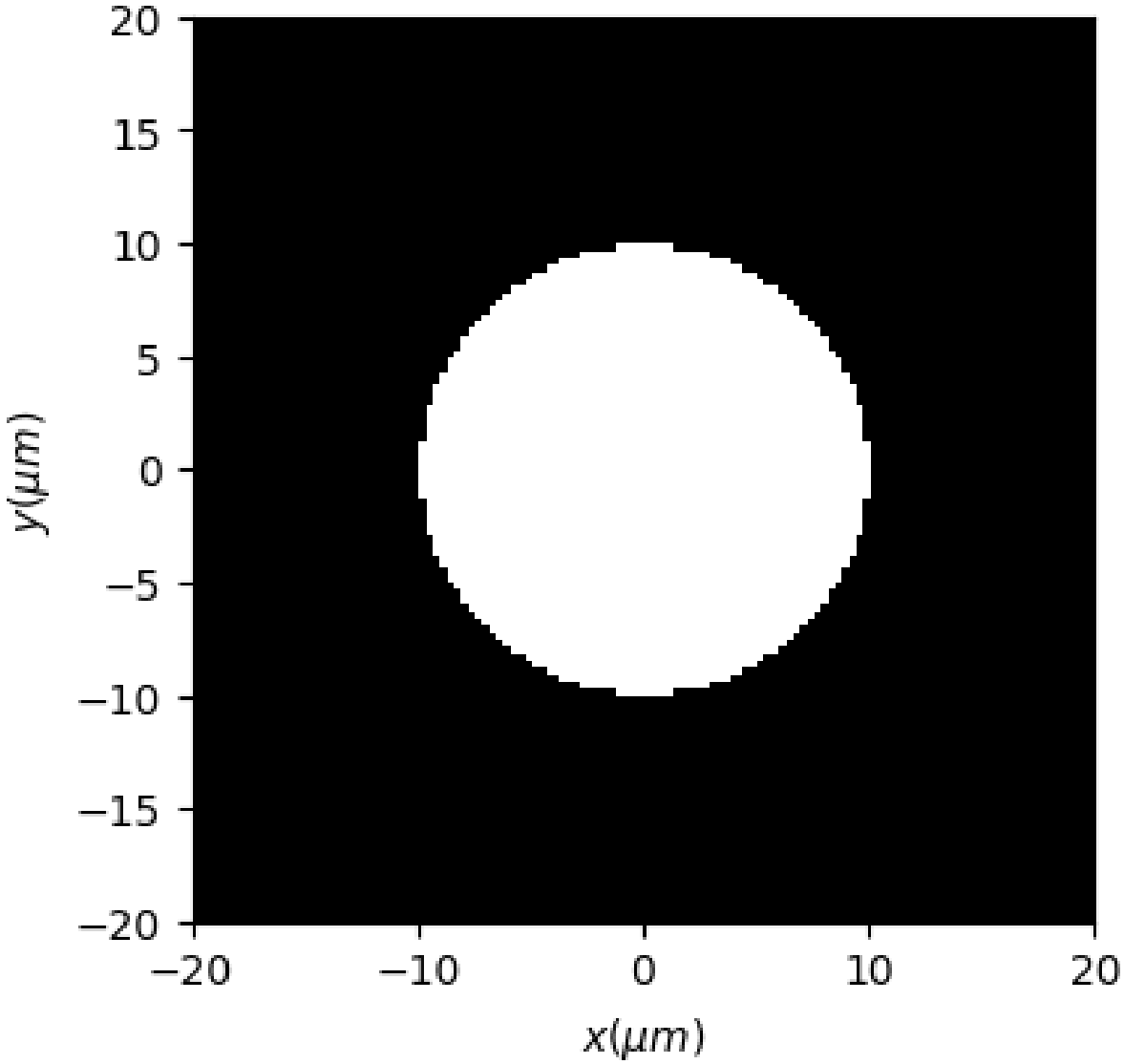
t0 = Scalar_mask_XY(x0, y0, wavelength) t0.two_levels(level1=0, level2=1, x_edge=0, angle=0)

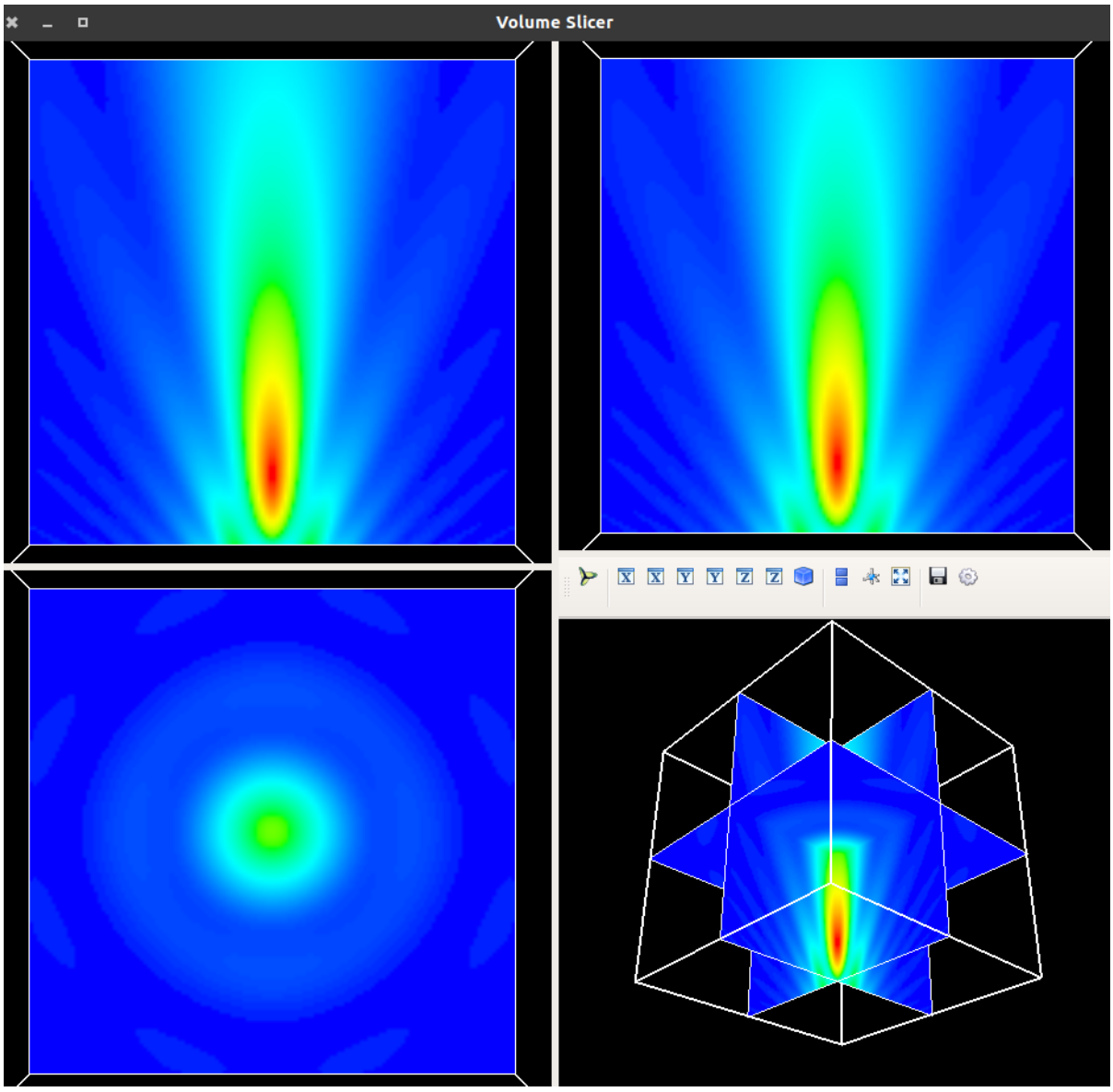
M0 = Vector_mask_XY(x=x0, y=y0, wavelength=wavelength)

state_0 = np.array([[1, 0],[0, 0]]) state_1 = np.array([[0, 0],[0, 1]])

M0.complementary_masks(t0, state_0, state_1)

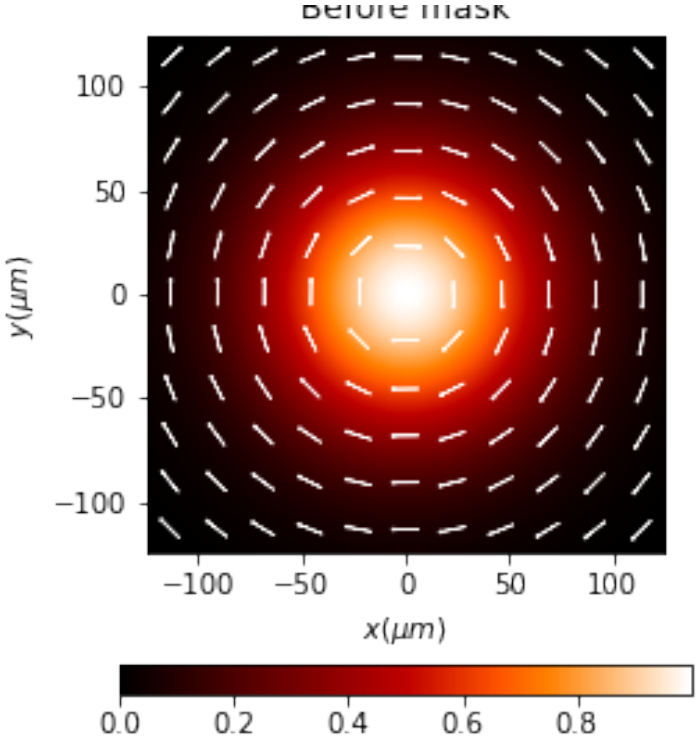
EM1 = EM0 * M0 EM1.draw(kind='ellipses') plt.title('After mask') plt.savefig('usage13.png')

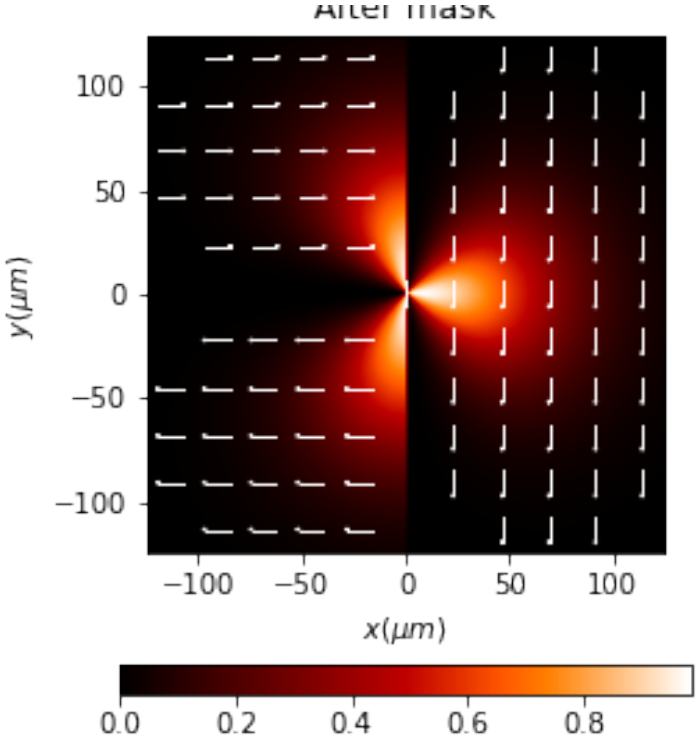

20
15
10
5
0
−5
−10
−15
−20
−20
−10
0
10
20
y(μm)
x(μm)

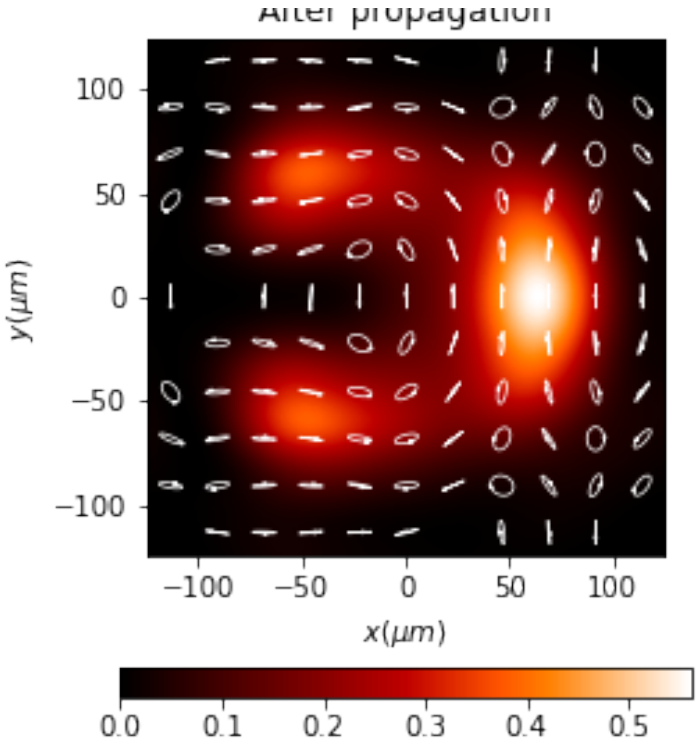

Volume Slicer

EM2 = EM1.RS(z=10 * mm)

EM2.draw(kind='ellipses') plt.title('After propagation') plt.savefig('usage14.png')




After mask
100
50
0
−50
−100
y(μm)
−100
−50
0
50
100
x(μm)
0.0
0.2
0.4
0.6
0.8


After propagation
100
50
0
−50
−100
y(μm)
−100
−50
0
50
100
x(μm)
0.0
0.1
0.2
0.3
0.4
0.5

CHAPTER

FOUR

DIFFRACTIO

4.1 diffractio package

4.1.1 Submodules

4.1.2 diffractio.config module

Configuration file. Standard diffractio units are um: um = 1.

4.1.3 diffractio.scalar_fields_X module

This module generates Scalar_field_X class and several functions for multiprocessing. It is required also for generating masks and fields.

The main atributes are:

- self.x (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n .
- self.wavelength (float): wavelength of the incident field.
- self.u (numpy.array): equal size than x. complex field

There are also some secondary atributes:

- self.quality (float): quality of RS algorithm
- self.info (str): description of data
- self.type (str): Class of the field
- self.date (str): date

Class for unidimensional scalar fields

Definition of a scalar field

- instantiation, duplicate, clear_field, print
- add, substract sources
- multiply masks and sources
- save and load data

Functions for generation of masks

- insert masks, insert_array_masks - insert other masks inside the mask
- filter -

Propagation

- fft, ifft - fourier transform
- RS - Rayleigh Sommerfeld. It allows amplification of the field
- BPM - Beam propagation method

Drawing functions

- draw

Parameters:

- intensity, average intensity
- get_edges_transitions (mainly for pylithography)

Functions

- kernelRS, kernelRSinverse

Multiprocessing

- polychromatic_multiprocessing
- extended_source_multiprocessing
- extended_polychromatic_source

class diffractio.scalar_fields_X.**Scalar_field_X**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1*, *info: str = ''*)

Bases: `object`

Class for unidimensional scalar fields.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **wavelength** (*float*) – wavelength of the incident field
- **n_background** (*float*) – refractive index of background
- **info** (*str*) – String with info about the simulation

`self.`**x**
Linear array with equidistant positions. The number of data is preferibly 2^n.

Type numpy.array

`self.`**wavelength**
Wavelength of the incident field.

Type float

`self.`**u**
Complex field. The size is equal to self.x.

Type numpy.array

`self.`**quality**
Quality of RS algorithm.

Type float

`self.info`
Description of data.

Type str

`self.type`
Class of the field.

Type str

`self.date`
Date when performed.

Type str

`conjugate`(*new_field: bool = True*)
Conjugates the field

`duplicate`(*clear: bool = False*)
Duplicates the instance

Parameters `clear` (*bool*) – If True, clears the self.u argument, else it keeps the same than in self.

Returns new_instance (scalar_field_X): duplicated instance.

`reduce_to_1`()
All the values greater than 1 pass to 1. This is used for Scalar_masks when we add two masks.

`clear_field`()
Removes the field so that self.u = 0.

`save_data`(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- `filename` (*str*) – filename
- `add_name=` (*str*) – sufix to the name, if 'date' includes a date
- `description` (*str*) – text to be stored in the dictionary to save.
- `verbose` (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

`load_data`(*filename: str*, *verbose: bool = False*)

Load data from a file to a Scalar_field_X. The methods included are: npz, matlab

Parameters

- `filename` (*str*) – filename
- `verbose` (*bool*) – shows data process by screen

`cut_resample`(*x_limits: numpy.ndarray[Any, numpy.dtype[numpy.float64]] | str = ''*, *num_points: int = []*, *new_field: bool = False*, *interp_kind: str = 'linear'*)
Cuts the field to the range (x0,x1). If one of this x0,x1 positions is out of the self.x range it does nothing. It is also valid for resampling the field, just write x0,x1 as the limits of self.x

Parameters

- **x_limits** (*numpy.array*) – (x0,x1) - starting and final points to cut, if '' - takes the current limit x[0] and x[-1]
- **num_points** (*int*) – it resamples x, and u [],'',0,None -> it leave the points as it is
- **new_field** (*bool*) – if True it returns a new Scalar_field_X
- **interp_kind** (*str*) – 'linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic'

Returns if new_field is True

Return type (*Scalar_field_X*)

incident_field(*u0*)
Incident field for the experiment. It takes a Scalar_source_X field.

Parameters **u0** (Scalar_source_X) – field produced by Scalar_source_X (or a X field)

inverse_amplitude(*new_field: bool = False*)
Inverts the amplitude of the mask, phase is equal as initial

Parameters **new_field** (*bool*) – If True it returns a Scalar_mask_X object, else, it modifies the existing object.

Returns If new_field is True, it returns a Scalar_mask_X object.

Return type *Scalar_mask_X*

inverse_phase(*new_field: bool = False*)
Inverts the phase of the mask, amplitude is equal as initial

Parameters **new_field** (*bool*) – If True it returns a Scalar_mask_X object, else, it modifies the existing object.

Returns If new_field is True, it returns a Scalar_mask_X object.

Return type *Scalar_mask_X*

filter(*size: int = 0*)

insert_mask(*t1*, *x0_mask1: float*, *clean: bool = True*, *kind_position: str = 'left'*)
Insert mask t1 in mask self. It is performed using interpolation.

Parameters

- **t1** (Scalar_field_X) – field X (shorter)
- **x0_mask1** (*float*) – location of starting point at t0 of init point of t1.
- **clean** (*bool*) – if True remove previous fields, else substitute
- **kind_position** (*str*) – 'left' 'center'

pupil(*x0*, *radius*)
Place a pupil in the field.

Parameters

- **x0** (*float*) – center of pupil.
- **radius** (*float*) – radius of pupil.

insert_array_masks(*t1*, *x_pos*, *clean=True*, *kind_position='left'*)
Insert several identical masks t1 in self.u according to positions x_pos

Parameters

- **t1** (Scalar_field_X) – mask to insert.
- **x_pos** (*numpy.array*) – array with locations.
- **clean** (*bool*) – elliminate preview fields in self.
- **kind_position** (*str*) – 'left', 'center': positions are at left or center.

repeat_structure(*num_repetitions*, *position='center'*, *new_field=True*)
Repeat the structure n times.

Parameters

- **num_repetitions** (*int*) – Number of repetitions of the mask
- **position** (*string or number*) – 'center', 'previous' or initial position. Initial x
- **new_field** (*bool*) – If True, a new mask is produced, else, the mask is modified.

fft(*z: Optional[float] = None*, *shift: bool = True*, *remove0: bool = False*, *matrix: bool = False*, *new_field: bool = False*, *verbose: bool = False*)
Far field diffraction pattern using Fast Fourier Transform (FFT).

Parameters

- **z** (*float*) – distance to the observation plane or focal of lens. If None the exit is in radians
- **shift** (*bool*) – if True, fftshift is performed
- **remove0** (*bool*) – if True, central point is removed
- **matrix** (*bool*) – if True only matrix is returned. If False, returns Scalar_field_X
- **new_field** (*bool*) – if True returns Scalar_field_X, else it puts in self
- **verbose** (*bool*) – if True, prints info

Returns FFT of the input field

Return type (array or *Scalar_field_X* or None)

ifft(*z: Optional[float] = None*, *shift: bool = True*, *remove0: bool = False*, *matrix: bool = False*, *new_field: bool = False*, *verbose: bool = False*)
Inverse Fast Fourier Transform (ifft) of the field.

Parameters

- **z** (*float*) – distance to the observation plane or focal of lens
- **shift** (*bool*) – if True, fftshift is performed
- **remove0** (*bool*) – if True, central point is removed
- **matrix** (*bool*) – if True only matrix is returned. If False, returns Scalar_field_X
- **new_field** (*bool*) – if True returns Scalar_field_X, else it puts in self
- **verbose** (*bool*) – if True, prints info

Returns FFT of the input field

Return type (array or *Scalar_field_X* or None)

RS(*z: float*, *amplification: int = 1*, *n: float = 1.0*, *new_field: bool = True*, *matrix: bool = False*, *xout: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *fast: bool = False*, *kind: str = 'z'*, *verbose: bool = True*)
Fast-Fourier-Transform method for numerical integration of diffraction Rayleigh-Sommerfeld formula. Is we have a field of size N*M, the result of propagation is also a field N*M. Nevertheless, there is a parameter

amplification which allows us to determine the field in greater observation planes (jN)x(jM).

Parameters

- **z** (*float*) – Distance to observation plane. if z<0 inverse propagation is executed
- **n** (*float*) – refractive index
- **new_field** (*bool*) – if False the computation goes to self.u, if True a new instance is produced
- **matrix** (*bool*) – If True, the result of the function is a numpy.array
- **fast** (*bool*) – Instead of using Hankle function for RS kernel uses expotential
- **amplification** (*int*) – number of frames in x direction
- **kind** (*str*) – 'z'. In some circunstamces the function is used for other integrals
- **verbose** (*bool*) – if True it writes to shell

Returns Scalar_field_X, If matrix is True: numpy.array() Else: None

Return type If New_field is True

Info: This approach a quality parameter: If self.quality>1, propagation is right.

CZT(*z: float | numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *xout: float | numpy.ndarray[Any, numpy.dtype[numpy.float64]] | None*, *verbose: float = False*)
Chirped z-transform.

Parameters

- **z** (*float or np.array*) – diffraction distance
- **xout** (*float or np.array*) – x array with positions of the output plane
- **verbose** (*bool*) – If True it prints information.

Returns Complex amplitude of the outgoing light beam

Return type u_out

References

[Light: Science and Applications, 9(1), (2020)]

WPM(*fn*, *zs: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *num_sampling: Optional[list[int]] = None*, *ROI: Optional[list[numpy.ndarray[typing.Any, numpy.dtype[+ _ScalarType_co]]]] = None*, *x_pos: Optional[float] = None*, *z_pos: Optional[float] = None*, *get_u_max: bool = False*, *has_edges: bool = True*, *pow_edge: int = 80*, *verbose: bool = False*)

WPM method used for very dense sampling. It does not storages the intensity distribution at propagation, but only s

- global view with a desired sampling given by num_sampling.
- intensity at the last plane.
- Intensity at plane with maximum intensity (focus of a lens, for example)
- Region of interest given by ROI.

Parameters

- **fn** (*function*) – Function that returns the refractive index at a plane in a numpy.array
- **zs** (*np.array*) – linspace with positions z where to evaluate the field.
- **num_sampling** (*int, int*) – If None, it does not storage intermediate data. Otherwise, it stores a small XZ matrix which size in num_sampling.
- **ROI** (*np.array, np.array*) – x and z arrays with positions and sampling to store. It is used when we need to store with a high density a small area of the total field.
- **x_pos** (*float or None*) – If not None, a high density longitudinal array with the field at x_pos position is stored.
- **z_pos** (*float or None*) – If not None, a high density transversal array with the field at z_pos is stored.
- **get_u_max** (*bool*) – If True, returns field at z position with maximum intensity.
- **has_edges** (*bool*) – If True absorbing edges are used.
- **pow_edge** (*float*) – If has_edges, power of the supergaussian.
- **verbose** (*bool*) – If True prints information.

Returns final field u_out_gv (Scalar_field_XZ): global view u_out_roi (Scalar_field_XZ): field at ROI positions u_axis_x (Scalar_field_X): field at position z_pos u_axis_z (Scalar_field_Z): field at position x_pos u_max (Scalar_field_X): field at position with maximum intensity z_max (float): position with maximum intensity

Return type u_iter (*Scalar_field_X*)

References

1. M.W. Fertig and K.-H. Brenner, "Vector wave propagation method," J. Opt. Soc. Am. A, vol. 27, no. 4, p. 709, 2010.
2. S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

normalize(*new_field: bool = False*)

Normalizes the field so that intensity.max()=1.

Parameters **new_field** (*bool*) – If False the computation goes to self.u. If True a new instance is produced.

Returns u (numpy.array): normalized optical field

MTF(*kind: str = 'mm', has_draw: bool = True*)

Computes the MTF of a field,.

Parameters

- **kind** (*str*) – 'mm', 'degrees'
- **has_draw** (*bool*) – If True draws the MTF

Returns frequencies in lines/mm (numpy.array) mtf_norm: normalizd MTF

Return type (numpy.array) fx

intensity()

Intensity.

Returns Intensity

Return type (numpy.array)

average_intensity(*verbose: bool = False*)
Returns the average intensity as: (np.abs(self.u)**2).sum() / num_data

Parameters **verbose** (*bool*) – If True it prints the value of the average intensity.

Returns average intensity.

Return type (float)

get_edges(*kind_transition: str = 'amplitude'*, *min_step: int = 0*, *verbose: bool = False*, *filename: str = ''*)
Determine locations of edges for a binary mask.

Parameters

- **kind_transition** – 'amplitude' 'phase' if we see the amplitude or phase of the field
- **min_step** – minimum step for consider a transition

Returns array with +1, -1 with rasing or falling edges pos_transition: positions x of transitions raising: positions of raising falling: positions of falling

Return type type_transition

get_RS_minimum_z(*n: float = 1.0*, *quality: int = 1*, *verbose: bool = True*)
Determines the minimum available distance for RS algorithm. If higher or lower quality parameters is required you can add as a parameter

Parameters

- **n** (*float*) – refractive index of the surrounding medium.
- **quality** (*int, optional*) – quality. Defaults to 1.
- **verbose** (*bool, optional*) – prints info. Defaults to True.

Returns z_min for quality_factor>quality

Return type z_min (float)

draw(*kind='intensity'*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *cut_value: Optional[numpy.float64] = None*, *filename: str = ''*, *scale: str = ''*)
Draws X field. There are several data from the field that are extracted, depending of 'kind' parameter.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'field', 'phase', 'fill', 'fft'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **cut_value** (*float*) – If not None, cuts the maximum intensity to this value
- **filename** (*str*) – if not '' stores drawing in file,
- **scale** (*str*) – '', 'scaled', 'equal', scales the XY drawing

diffractio.scalar_fields_X.**kernelRS**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *wavelength: float*, *z: float*, *n: float = 1.0*, *kind: str = 'z'*, *fast: bool = False*)
Kernel for RS propagation. It uses the hankel tansform.

There is a 'fast' version based on $hk_1 = \sqrt{2/(\pi\, k\, R)} e^{i(k\, R - 3\pi/4)}$ which approximates the result.

Parameters

- **x** (*numpy.array*) – positions x
- **wavelength** (*float*) – wavelength of incident fields
- **z** (*float*) – distance for propagation
- **n** (*float*) – refractive index of background
- **kind** (*str*) – 'z', 'x', '0': for simplifying vector propagation
- **fast** (*bool*) – If True The approximation is much faster. According to https://dlmf.nist.gov/10.2#E5

Returns kernel

Return type (complex array)

References

- https://dlmf.nist.gov/10.2#E5
- F. Shen and A. Wang, "Fast-Fourier-transform based numerical integration method for the Rayleigh-Sommerfeld diffraction formula," Appl. Opt., vol. 45, no. 6, pp. 1102–1110, 2006.

diffractio.scalar_fields_X.**kernelRSinverse**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *wavelength: float*, *z: float*, *n: float = 1.0*, *kind: str = 'z'*, *fast: bool = False*)

Kernel for inverse RS propagation. See also kernelRS

Parameters

- **x** (*numpy.array*) – positions x
- **wavelength** (*float*) – wavelength of incident fields
- **z** (*float*) – distance for propagation
- **n** (*float*) – refractive index of background
- **kind** (*str*) – 'z', 'x', '0': for simplifying vector propagation
- **fast** (*bool*) – If True The approximation is much faster. According to https://dlmf.nist.gov/10.2#E5

Returns kernel

Return type complex array

diffractio.scalar_fields_X.**PWD_kernel**(*u: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *n: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *k0: float*, *k_perp2: numpy.ndarray[Any, numpy.dtype[Any]]*, *dz: float*)

Step for scalar (TE) Plane wave decomposition (PWD) algorithm.

Parameters

- **u** (*np.array*) – fields
- **n** (*np.array*) – refractive index
- **k0** (*float*) – wavenumber
- **k_perp2** (*np.array*) – transversal k**2
- **dz** (*float*) – increment in distances

Returns Field at at distance dz from the incident field

Return type numpy.array()

References

1. Schmidt, S. et al. "Wave-optical modeling beyond the thin-element-approximation". 'Opt. Express' 24, 30188 (2016).

diffractio.scalar_fields_X.**WPM_schmidt_kernel**(*u*, *n: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *k0: float*, *k_perp2: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *dz: float*)

Kernel for fast propagation of WPM method

Parameters

- **u** (*np.array*) – fields
- **n** (*np.array*) – refractive index
- **k0** (*float*) – wavenumber
- **k_perp2** (*np.array*) – transversal k**2
- **dz** (*float*) – increment in distances

References

1. M. W. Fertig and K.-H. Brenner, "Vector wave propagation method," J. Opt. Soc. Am. A, vol. 27, no. 4, p. 709, 2010.
2. S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

diffractio.scalar_fields_X.**polychromatic_multiprocessing**(*function_process*, *wavelengths: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *spectrum: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *num_processors: int = 8*, *verbose: bool = False*)

It performs an analysis of polychromatic light. It needs a function with only one input parameter: wavelength. It determines the intensity for each wavelength and the final results is the summation of the intensities.

Parameters

- **function_process** (*function*) – function with accepts params as input parameters:
- **wavelengths** (*array*) – wavelengths in the spectrum
- **spectrum** (*array*) – weights for the spectrum. if None: uniform spectrum, else array with the same dimension as wavelengths
- **num_processors** (*int*) – number of processors for the computation
- **verbose** (*bool*) – if True send information to shell

Returns intensity = intensity + spectrum[i] * np.abs(u_s[i].u)**2 u_s (Scalar_field_X): fields for each wavelength time_proc (float): time interval in the processing

Return type intensity (array, complex)

diffractio.scalar_fields_X.**extended_source_multiprocessing**(*function_process*, *x0s: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *num_processors: int = 8*, *verbose: bool = False*)

It performs an analysis of extendes source light. It needs a function with only an input parameter, that is x0s positions of sources. It determines the intensity for each wavelength and it is added.

Parameters

- **function_process** (*function*) – function with accepts params as input Parameters:
- **x0s** (*array*) – positions of sources
- **num_processors** (*int*) – number of processors for the computation
- **verbose** (*bool*) – if True send information to shell

Returns intensity = intensity + spectrum[i] * np.abs(u_s[i].u)**2 - u_s (Scalar_field_X): fields for each wavelength - time_proc (float): time interval in the processing

Return type

- intensity (array, complex)

diffractio.scalar_fields_X.**extended_polychromatic_source**(*function_process*, *x0s: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *wavelengths: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *spectrum: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *num_processors: int = 8*, *verbose: bool = False*)

It performs an analysis of extendes source light. It needs a function with only an input parameter, that is x0s positions of sources. It determines the intensity for each wavelength and it is added.

Parameters

- **function_process** (*function*) – function with accepts params as input Parameters:
- **x0s** (*array*) – positions of sources
- **wavelengths** (*array*) – wavelengths in the spectrum
- **spectrum** (*array*) – weights for the spectrum. If None: uniform spectrum, else array with the same dimension as wavelengths
- **num_processors** (*int*) – number of processors for the computation
- **verbose** – if True send information to shell

diffractio.scalar_fields_X.**quality_factor**(*range_x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *num_x: int*, *z: float*, *wavelength: float*, *n: float = 1.0*, *verbose: bool = False*)

Determine the quality factor for RS algorithm

Parameters

- **x** (*np.array*) – x array with positions
- **z** (*float*) – observation distance
- **wavelength** (*float*) – wavelength)

- **n** (*float*) – refractive index

Returns _description_

Return type _type_

diffractio.scalar_fields_X.**get_RS_minimum_z**(*range_x: float*, *num_x: int*, *wavelength: float*, *n: float = 1.0*, *quality=1.0*, *verbose: bool = True*)

summary

Parameters

- **range_x** (*float*) – range_x
- **num_x** (*int*) – num_x
- **z** (*float*) – z
- **wavelength** (*float*) – wavelength
- **quality** (*int, optional*) – quality. Defaults to 1.
- **verbose** (*bool, optional*) – prints info. Defaults to True.

Returns _description_

Return type _type_

4.1.4 diffractio.scalar_fields_XY module

This module generates Scalar_field_XY class.

It can be considered an extension of Scalar_field_X for visualizing XY fields

For the case of Rayleigh sommefeld it is not necessary to compute all z positions but the final.

Nevertheless, for BPM method, intermediate computations are required. In this class, intermediate results are stored.

It is required also for generating masks and fields.

The main atributes are:

- self.x - x positions of the field
- self.y - y positions of the field
- self.wavelength - wavdelength of the incident field. The field is monochromatic
- self.u (numpy.array): equal size to x * y. complex field
- self.X (numpy.array): equal size to x * y. complex field
- self.Y (numpy.array): equal size to x * y. complex field
- self.quality (float): quality of RS algorithm
- self.info (str): description of data
- self.type (str): Class of the field
- self.date (str): date of execution

The magnitude is related to microns: *micron = 1.*

Class for XY scalar fields

Definition of a scalar field

- instatiation, duplicate

- save, load data
- cut_resample, binarize, discretize
- get_phase, get_amplitude, remove_amplitude, remove_phase, amplitude2phase, phase2amplitude

Propagation

- fft, ifft, RS, RS_simple, RS_amplificacion

Drawing functions

- draw, draw_profile,
- video, progresion

Parameters

- profile

Functions outside the class

- draw_several_fields
- draw2D
- several_propagations
- kernelRS, kernelRSinverse, kernelFresnel, WPM_schmidt_kernel

class diffractio.scalar_fields_XY.**Scalar_field_XY**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *y: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *info: str = ''*)

Bases: `object`

Class for working with XY scalar fields.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **y** (*numpy.array*) – linear array wit equidistant positions for y values
- **wavelength** (*float*) – wavelength of the incident field
- **info** (*str*) – String with info about the simulation

`self.x`
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

`self.y`
linear array wit equidistant positions for y values

Type numpy.array

`self.wavelength`
wavelength of the incident field.

Type float

`self.u`
(x,z) complex field

Type numpy.array

`self.info`
String with info about the simulation

Type str

`conjugate`(*new_field: bool = True*)
Conjugates the field

`duplicate`(*clear: bool = False*)
Duplicates the instance

`reduce_to_1`()
All the values greater than 1 pass to 1. This is used for Scalar_masks when we add two masks.

`add`(*other*, *kind: str = 'standard'*)
adds two Scalar_field_xy. For example two light sources or two masks

Parameters

- **other** (`Scalar_field_X`) – 2 field to add
- **kind** (*str*) – instruction how to add the fields: - 'maximum1': mainly for masks. If t3=t1+t2>1 then t3= 1. - 'standard': add fields u3=u1+u2 and does nothing.

Returns *u3 = u1 + u2*

Return type *Scalar_field_X*

`rotate`(*angle: float*, *position: Optional[list[float, float]] = None*, *new_field: bool = False*)
Rotation of X,Y with respect to position. If position is not given, rotation is with respect to the center of the image

Parameters

- **angle** (*float*) – angle to rotate, in radians
- **position** (*float, float*) – position of center of rotation

`apodization`(*power: int = 10*)
Multiply field by an apodizer. The apodizer is a super_gauss function.

Parameters **power** (*int, optional*) – size of apodization. Defaults to 10.

`clear_field`()
Removes the field: self.u=0.

`save_data`(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **add_name=** (*str*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

`load_data`(*filename: str*, *verbose: bool = False*)

Load data from a file to a Scalar_field_X. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **verbose** (*bool*) – shows data process by screen

save_mask(*filename: str = ''*, *kind: str = 'amplitude'*, *binarize: bool = False*, *cmap: str = 'gray'*, *info: str = ''*)
Create a mask in a file, for example, ablation or litography engraver

Parameters

- **filename** (*str*) – file name
- **kind** (*str*) – save amplitude, phase or intensity. If intensity it is normalized to (0,1)
- **binarize** (*bool*) – If True convert the mask in (0,1) levels
- **cmap** (*str*) – colormap. (gray)
- **info** (*str*) – info of the mask

Returns area (in um**2)

Return type float

cut_resample(*x_limits: list[float, float] = ''*, *y_limits: list[float, float] = ''*, *num_points: int = []*, *new_field: bool = False*, *interp_kind: list[int, int] = (3, 1)*)
it cut the field to the range (x0,x1). If one of this x0,x1 positions is out of the self.x range it do nothing. It is also valid for resampling the field, just write x0,x1 as the limits of self.x

Parameters

- **x_limits** (*float,float*) – (x0,x1) starting and final points to cut. if '' - takes the current limit x[0] and x[-1]
- **y_limits** (*float,float*) – (y0,y1) - starting and final points to cut. if '' - takes the current limit y[0] and y[-1]
- **num_points** (*int*) – it resamples x, y and u. [],",,None -> it leave the points as it is
- **new_field** (*bool*) – it returns a new Scalar_field_XY
- **interp_kind** – numbers between 1 and 5

incident_field(*u0*)
Incident field for the experiment. It takes a Scalar_source_X field.

Parameters **u0** (Scalar_source_X) – field produced by Scalar_source_X (or a X field)

pupil(*r0: Optional[list[float, float]] = None*, *radius: Optional[float] = None*, *angle: float = 0.0*)
place a pupil in the field. If r0 or radius are None, they are computed using the x,y parameters.

Parameters

- **r0** (*float, float*) – center of circle/ellipse
- **radius** (*float, float) or (float*) – radius of circle/ellipse
- **angle** (*float*) – angle of rotation in radians

Example

pupil(r0=(0 * um, 0 * um), radius=(250 * um, 125 * um), angle=0 * degrees)

fft(*z: float = 0*, *shift: bool = True*, *remove0: bool = True*, *matrix: bool = False*, *new_field: bool = False*)
Fast Fourier Transform (FFT) of the field.

Parameters

- **z** (*float*) – distance to the observation plane or focal of lens if z==0, no x,y scaled is produced
- **shift** (*bool*) – if True, fftshift is performed
- **remove0** (*bool*) – if True, central point is removed
- **matrix** (*bool*) – if True only matrix is returned. if False, returns Scalar_field_X
- **new_field** (*bool*) – if True returns Scalar_field_X, else it puts in self

Returns FFT of the input field

Return type (np.array or *Scalar_field_X* or None)

ifft_proposal(*z: float = 0.0*, *shift: bool = True*, *remove0: bool = True*, *matrix: bool = False*, *new_field: bool = False*)
Fast Fourier Transform (fft) of the field.

Parameters

- **z** (*float*) – distance to the observation plane or focal of lens
- **shift** (*bool*) – if True, fftshift is performed
- **remove0** (*bool*) – if True, central point is removed
- **matrix** (*bool*) – if True only matrix is returned. If False, returns Scalar_field_X
- **new_field** (*bool*) – if True returns Scalar_field_X, else puts in self

Returns FFT of the input field

Return type (np.array or *Scalar_field_X* or None)

ifft(*z: float = 0.0*, *shift: bool = True*, *remove0: bool = True*, *matrix: bool = False*, *new_field: bool = False*)
Fast Fourier Transform (fft) of the field.

Parameters

- **z** (*float*) – distance to the observation plane or focal of lens
- **shift** (*bool*) – if True, fftshift is performed
- **remove0** (*bool*) – if True, central point is removed
- **matrix** (*bool*) – if True only matrix is returned. If False, returns Scalar_field_X
- **new_field** (*bool*) – if True returns Scalar_field_X, else puts in self

Returns FFT of the input field

Return type (np.array or *Scalar_field_X* or None)

RS(*z: float*, *amplification: list[int, int] = (1, 1)*, *n: float = 1.0*, *new_field: bool = True*, *matrix: bool = False*, *xout: Optional[float] = None*, *yout: Optional[float] = None*, *kind: str = 'z'*, *verbose: bool = False*)
Fast-Fourier-Transform method for numerical integration of diffraction Rayleigh-Sommerfeld formula. Is we have a field of size N*M, the result of propagation is also a field N*M. Nevertheless, there is a parameter *amplification* which allows us to determine the field in greater observation planes (jN)x(jM).

Parameters

- **amplification** (*int, int*) – number of frames in x and y direction
- **z** (*float*) – distance to observation plane. if z<0 inverse propagation is executed
- **n** (*float*) – refractive index
- **new_field** (*bool*) – if False the computation goes to self.u, if True a new instance is produced.
- **matrix** (*Bool*) – if True returns a matrix, else a Scalar_field_XY
- **xout** (*float*) – If not None, the sampling area is moved. This is the left position
- **yout** (*float*) – If not None, the sampling area y moved. This is the lower position.
- **kind** (*str*) –
- **verbose** (*bool*) – if True it writes to shell

Returns Scalar_field_X, else None.

Return type if New_field is True

Note: One advantage of this approach is that it returns a quality parameter: if self.quality>1, propagation is right.

References

F. Shen and A. Wang, "Fast-Fourier-transform based numerical integration method for the Rayleigh-Sommerfeld diffraction formula," Appl. Opt., vol. 45, no. 6, pp. 1102–1110, 2006.

WPM(*fn*, *zs: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *num_sampling: list = (512, 512)*, *ROI: list[numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]] | None = (None, None)*, *r_pos: Optional[float] = None*, *z_pos: Optional[float] = None*, *get_u_max: bool = False*, *has_edges: bool = True*, *pow_edge: int = 80*, *matrix: bool = False*, *verbose: bool = False*)

WPM method used for very dense sampling. It does not storages the intensity distribution at propagation, but only s

- global view with a desired sampling given by num_sampling.
- intensity at the last plane.
- Intensity at plane with maximum intensity (focus of a lens, for example)
- Region of interest given by ROI.

Parameters

- **fn** (*function*) – Function that returns the refractive index at a plane in a numpy.array
- **zs** (*np.array*) – linspace with positions z where to evaluate the field.
- **num_sampling** (*int, int*) – If None, it does not storage intermediate data. Otherwise, it stores a small XZ matrix which size in num_sampling.
- **ROI** (*np.array, np.array*) – x and z arrays with positions and sampling to store. It is used when we need to store with a high density a small area of the total field.

- **r_pos** (*float or None*) – If not None, a high density longitudinal array with the field at x_pos position is stored.
- **z_pos** (*float or None*) – If not None, a high density transversal array with the field at z_pos is stored.
- **get_u_max** (*bool*) – If True, returns field at z position with maximum intensity.
- **has_edges** (*bool*) – If True absorbing edges are used.
- **pow_edge** (*float*) – If has_edges, power of the supergaussian.
- **verbose** (*bool*) – If True prints information.

References

1. M.W. Fertig and K.-H. Brenner, "Vector wave propagation method," J. Opt. Soc. Am. A, vol. 27, no. 4, p. 709, 2010.
2. S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

CZT_backup(*z: float*, *xout: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *yout: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *verbose: bool = False*)

Chirped Z Transform algorithm for XY Scheme. z, xout, and yout parameters can be numbers or arrays. The output Scheme depends on this input parameters.

Parameters

- **z** (*float*) – diffraction distance
- **xout** (*np.array*) – x array with positions of the output plane
- **yout** (*np.array*) – y array with positions of the output plane
- **verbose** (*bool*) – If True, it prints some information

Returns Scalar_field_** depending of the input scheme. When all the parameters are numbers, it returns the complex field at that point.

Return type u_out

References

[Light: Science and Applications, 9(1), (2020)]

CZT(*z: float*, *xout: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *yout: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *verbose: bool = False*)

Chirped Z Transform algorithm for XY Scheme. z, xout, and yout parameters can be numbers or arrays. The output Scheme depends on this input parameters.

Parameters

- **z** (*float*) – diffraction distance
- **xout** (*np.array*) – x array with positions of the output plane
- **yout** (*np.array*) – y array with positions of the output plane
- **verbose** (*bool*) – If True, it prints some information

Returns Scalar_field_** depending of the input scheme. When all the parameters are numbers, it returns the complex field at that point.

Return type u_out

References

[Light: Science and Applications, 9(1), (2020)]

profile(*point1: list[float, float] | str = ''*, *point2: list[float, float] | str = ''*, *npixels: Optional[int] = None*, *kind: str = 'intensity'*, *order: int = 2*)
Determine profile in image. If points are not given, then image is shown and points are obtained clicking.

Parameters

- **point1** (*float, float*) – initial point. if '' get from click
- **point2** (*float, float*) – final point. if '' get from click
- **npixels** (*int*) – number of pixels for interpolation
- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase'
- **order** (*int*) – order for interpolation

Returns profile numpy.array: z values for profile (float, float): point1 (float, float): point2

Return type numpy.array

draw_profile(*point1: list[float, float] | str = ''*, *point2: list[float, float] | str = ''*, *npixels: Optional[int] = None*, *kind: str = 'intensity'*, *order: int = 2*)
Draws profile in image. If points are not given, then image is shown and points are obtained clicking.

Parameters

- **point1** (*float*) – initial point. if '' get from click
- **point2** (*float*) – final point. if '' get from click
- **npixels** (*int*) – number of pixels for interpolation
- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase'
- **order** (*int*) – order for interpolation

Returns profile numpy.array: z values for profile (float, float): point1 (float, float): point2

Return type numpy.array

get_edges(*kind_transition: str = 'amplitude'*, *min_step: float = 0*, *verbose: bool = False*, *filename: str = ''*)
Determine locations of edges for a binary mask. Valid for litography engraving of gratings.

Parameters

- **kind_transition** – 'amplitude' 'phase'.
- **min_step** – minimum step for consider a transition

Returns array with +1, -1 with rasing or falling edges pos_transition: positions x of transitions raising: positions of raising falling: positions of falling

Return type type_transition

search_focus(*kind: str = 'moments'*, *verbose: bool = True*)
Search for location of .

Parameters

- **kind** (*str*) – 'moments' or 'maximum'

- **verbose** (*bool*) – If True prints information.

Returns positions of focus

Return type (x,y)

MTF(*kind: str = 'mm'*, *has_draw: bool = True*, *is_matrix: bool = True*)
Computes the MTF of a field, If this field is near to focal point, the MTF will be wide

Parameters

- **kind** (*str*) – 'mm', 'degrees'
- **has_draw** (*bool*) – If True draws the MTF

Returns frequencies in lines/mm (numpy.array) mtf_norm: normalizd MTF

Return type (numpy.array) fx

beam_width_4s(*has_draw: bool = True*)
Returns the beam width parameters according to ISO11146.

Parameters **has_draw** (*bool*) – If True, it draws

Returns dx width x (float): dy width y (float): principal_axis, angle (str): (x_mean, y_mean, x2_mean, y2_mean, xy_mean), Moments

Return type (float)

References

- https://en.wikipedia.org/wiki/Beam_diameter
- http://www.auniontech.com/ueditor/file/20170921/1505982360689799.pdf

intensity()
Returns intensity.

average_intensity(*verbose: bool = False*)
Returns average intensity as: (np.abs(self.u)**2).sum() / num_data.

Parameters **verbose** (*bool*) – If True prints data.

send_image_screen(*id_screen: int*, *kind: str = 'amplitude'*)
Takes the images and sends the images to a screen in full size.

Parameters

- **id_screen** (*hdl*) – handle to screen
- **kind** (*'str'*) – 'amplitude', 'intensity', 'phase'

get_amplitude(*matrix: bool = False*, *new_field: bool = False*)
Gets the amplitude of the field.

Parameters

- **matrix** (*bool*) – if True numpy.matrix is returned
- **new_field** (*bool*) – if True it returns a new Scalar_field_XY

Returns Scalar_field_X if matrix is True: numpy.array

Return type if New_field is True

get_phase(*matrix: bool = False*, *new_field: bool = False*)
Gets the phase of the field.

Parameters

- **matrix** (*bool*) – if True numpy.matrix is returned
- **new_field** (*bool*) – if True it returns a new Scalar_field_XY

Returns Scalar_field_X. if Matrix is True: numpy.array.

Return type if New_field is True

remove_phase(*sign: bool = False*, *matrix: bool = False*, *new_field: bool = False*)
Removes the phase of the field. Amplitude is kept.

Parameters

- **sign** (*bool*) – If True, sign is kept, else, it is removed
- **matrix** (*bool*) – if True numpy.matrix is returned
- **new_field** (*bool*) – if True it returns a new Scalar_field_XY

Returns Scalar_field_X. if Matrix is True: numpy.array.

Return type if New_field is True

binarize(*kind: str = 'amplitude'*, *bin_level: Optional[float] = None*, *level0: Optional[float] = None*, *level1: Optional[float] = None*, *new_field: bool = False*, *matrix: bool = False*)
Changes the number of points in field, mantaining the area.

Parameters

- **kind** (*str*) – 'amplitude' or 'phase'
- **bin_level** (*float*) – value of cut. If None, the cut is in the mean value
- **level0** (*float*) – minimum value. If None, minimum value of field
- **level1** (*float*) – maximum value. If None, maximum value of field
- **new_field** (*bool*) – if True returns new field
- **matrix** (*bool*) – if True it returs a matrix

Returns if new_field is True returns Scalar_field_XY

Return type *Scalar_field_XY*

TODO: Check and pass to utils

discretize(*kind: str = 'amplitude'*, *num_levels: int = 2*, *factor: numpy.float64 = 1*, *phaseInicial: numpy.float64 = 0*, *new_field: bool = True*, *matrix: bool = False*)
Discretize in a number of levels equal to num_levels.

Parameters

- **kind** (*str*) – "amplitude" o "phase"
- **num_levels** (*int*) – number of levels for the discretization
- **factor** (*float*) – from the level, how area is binarized. if 1 everything is binarized,
- **phaseInicial** (*float*) –

  –
- **new_field** (*bool*) – if True returns new field

- **matrix** (*bool*) – if True it returs a matrix

Returns if new_field is True returns Scalar_field_XY

Return type *Scalar_field_XY*

TODO: Check and pass to utils

normalize(*new_field: bool = False*)

Normalizes the field so that intensity.max()=1.

Parameters **new_field** (*bool*) – If False the computation goes to self.u. If True a new instance is produced

Returns u (numpy.array): normalized optical field

get_RS_minimum_z(*n: float = 1.0*, *quality: float = 1.0*, *verbose=True*)

Determines the minimum available distance for RS algorithm. If higher or lower quality parameters is required you can add as a parameter

Parameters

- **n** (*float*) – refractive index of the surrounding medium.
- **quality** (*int, optional*) – quality. Defaults to 1.
- **verbose** (*bool, optional*) – prints info. Defaults to True.

Returns z_min for quality_factor>quality

Return type z_min (float)

draw(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *title: str = ''*, *filename: str = ''*, *cut_value: Optional[numpy.float64] = None*, *has_colorbar: str = ''*, *colormap_kind: str = ''*, *reduce_matrix: str = 'standard'*, *percentage_intensity: Optional[numpy.float64] = None*, ***kwargs*)

Draws XY field.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', ' 'field', 'real_field', 'contour'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'area', 'intensity'
- **title** (*str*) – title for the drawing
- **filename** (*str*) – if not '' stores drawing in file,
- **cut_value** (*float*) – if provided, maximum value to show
- **has_colorbar** (*bool*) – if True draws the colorbar
- **percentage_intensity** (*None or number*) – If None it takes from CONF_DRAWING['percentage_intensity'], else uses this value
- **reduce_matrix** (*str*) – 'standard'

video(*kind: str*, *zs: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *time_video: numpy.float64 = 10.0*, *frames_reduction: int = 1*, *filename: str = 'video.avi'*, *dpi: int = 100*)

Makes a video TODO

Parameters **kind** (*str*) – 'intensity', 'phase', 'amplitude'

diffractio.scalar_fields_XY.**kernelRS**(*X: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *Y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *wavelength: numpy.float64*, *z: numpy.float64*, *n: float = 1.0*, *kind: str = 'z'*)

Kernel for RS propagation.

Parameters

- **X** (*numpy.array*) – positions x
- **Y** (*numpy.array*) – positions y
- **wavelength** (*float*) – wavelength of incident fields
- **z** (*float*) – distance for propagation
- **n** (*float*) – refractive index of background
- **kind** (*str*) – 'z', 'x', '0': for simplifying vector propagation

Returns kernel

Return type complex np.array

diffractio.scalar_fields_XY.**kernelRSinverse**(*X: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *Y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *wavelength: numpy.float64*, *z: numpy.float64*, *n: numpy.float64 = 1.0*, *kind: str = 'z'*)

Kernel for inverse RS propagation

Parameters

- **X** (*numpy.array*) – positions x
- **Y** (*numpy.array*) – positions y
- **wavelength** (*float*) – wavelength of incident fields
- **z** (*float*) – distance for propagation
- **n** (*float*) – refractive index of background
- **kind** (*str*) – 'z', 'x', '0': for simplifying vector propagation

Returns kernel

Return type complex np.array

diffractio.scalar_fields_XY.**kernelFresnel**(*X: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *Y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *wavelength: numpy.float64*, *z: numpy.float64*, *n: numpy.float64 = 1.0*)

Kernel for Fresnel propagation.

Parameters

- **X** (*numpy.array*) – positions x
- **Y** (*numpy.array*) – positions y
- **wavelength** (*float*) – wavelength of incident fields
- **z** (*float*) – distance for propagation
- **n** (*float*) – refractive index of background

Returns kernel

Return type complex np.array

diffractio.scalar_fields_XY.**PWD_kernel**(*u: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *n: numpy.float64*, *k0: numpy.float64*, *k_perp2: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *dz: numpy.float64*)

Step for scalar(TE) Plane wave decomposition(PWD) algorithm.

Parameters

- **u** (*np.array*) – field
- **n** (*np.array*) – refractive index
- **k0** (*float*) – wavenumber
- **k_perp** (*np.array*) – transversal k
- **dz** (*float*) – increment in distances

Returns Field at at distance dz from the incident field

Return type (numpy.array)

References

1. Schmidt, S. et al. Wave - optical modeling beyond the thin - element - approximation. Opt. Express 24, 30188 (2016).

diffractio.scalar_fields_XY.**WPM_schmidt_kernel**(*u: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *n: numpy.float64*, *k0: numpy.float64*, *k_perp2: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *dz: numpy.float64*)

Kernel for fast propagation of WPM method

Parameters

- **u** (*np.array*) – fields
- **n** (*np.array*) – refractive index
- **k0** (*float*) – wavenumber
- **k_perp2** (*np.array*) – transversal k**2
- **dz** (*float*) – increment in distances

References

1. M. W. Fertig and K.-H. Brenner, "Vector wave propagation method," J. Opt. Soc. Am. A, vol. 27, no. 4, p. 709, 2010.
2. S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

diffractio.scalar_fields_XY.**get_RS_minimum_z**(*range_x: numpy.float64*, *range_y: numpy.float64*, *num_x: int*, *num_y: int*, *wavelength: numpy.float64*, *n: float = 1.0*, *quality: numpy.float64 = 1*, *verbose: bool = True*)

summary

Parameters

- **range_x** (*float*) – range_x
- **range_y** (*float*) – range_y

- **num_x** (*int*) – num_x
- **num_y** (*int*) – num_y
- **z** (*float*) – z
- **wavelength** (*float*) – wavelength
- **n** (*float*) – refractive index
- **quality** (*int, optional*) – quality. Defaults to 1.
- **verbose** (*bool, optional*) – prints info. Defaults to True.

Returns z_min for quality_factor>quality

Return type z_min (float)

diffractio.scalar_fields_XY.**quality_factor**(*range_x: numpy.float64*, *range_y: numpy.float64*, *num_x: int*, *num_y: int*, *z: numpy.float64*, *wavelength: numpy.float64*, *n: float = 1.0*, *verbose: bool = False*)

Determine the quality factor for RS algorithm

Parameters

- **range_x** (*float*) – range_x
- **range_y** (*float*) – range_y
- **num_x** (*int*) – num_x
- **num_y** (*int*) – num_y
- **z** (*float*) – z
- **wavelength** (*float*) – wavelength
- **n** (*float*) – refractive index
- **verbose** (*bool, optional*) – prints info. Defaults to True.

Returns _description_

Return type _type_

4.1.5 diffractio.scalar_fields_XYZ module

This module generates Scalar_field_XYZ class and several functions for multiprocessing. It is required also for generating masks and fields. The main atributes are:

- self.u - field XYZ
- self.x - x positions of the field
- self.y - y positions of the fieldalgorithm
- self.z - z positions of the field
- self.wavelength - wavelength of the incident field. The field is monochromatic
- self.u0 - initial field at z=z0
- self.n_background - background refractive index
- self.n - refractive index

The magnitude is related to microns: *micron = 1.*

Class for unidimensional scalar fields

Definition of a scalar field

- instantiation, duplicate
- load and save data
- to_Scalar_field_XY
- xy_2_xyz
- cut_function
- __rotate__
- __rotate_axis__

Propagation

- RS - Rayleigh Sommerfeld
- RS_amplification
- BPM - Beam Propagation method

Drawing functions

- draw_XYZ
- draw_XY
- draw_XZ
- draw_volume
- draw_refractive_index
- video

class diffractio.scalar_fields_XYZ.**Scalar_field_XYZ**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *y: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *z: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: `object`

Class for 3D scalar fields.

Parameters

- **u0** (`Scalar_field_XY`) – Initial scalar field. wavelength, and x, y arrays are obtained from this field.
- **z** (*numpy.array*) – linear array with equidistant positions.
- **n_background** (*float*) – refractive index of background
- **info** (*str*) – String with info about the simulation

`self.x`
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

`self.y`
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

`self.z`
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

`self.u`
equal size than X. complex field

Type numpy.array

`self.wavelength`
wavelength of the incident field.

Type float

`self.u0`
Initial XY field

Type *Scalar_field_XY*

`self.n_background`
background refractive index.

Type float

`self.n`
refractive index. Same dimensions than self.u.

Type numpy.array

`xy_2_xyz`(*u0_XY*, *z: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*)
Similar to Init. send a Scalarfield_XY and passes to XYZ.

Parameters

- **u0_XY** (`Scalar_field_XY`) – init field
- **z** (*numpy.array*) – array with z positions

`conjugate`(*new_field: bool = True*)
Conjugates the field

`normalize`(*new_field: bool = False*)
Normalizes the field so that intensity.max()=1.

Parameters **new_field** (*bool*) – If False the computation goes to self.u. If True a new instance is produced

Returns u (numpy.array): normalized optical field

`duplicate`(*clear: bool = False*)
Duplicates the instance

`reduce_to_1`()
All the values greater than 1 pass to 1. This is used for Scalar_masks when we add two masks.

`clear_field`()
clear field.

clear_refractive_index()
clear refractive index n(x,z)=n_background.

save_data(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **add_name=** (*str*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

load_data(*filename: str*, *verbose: bool = False*)

Load data from a file to a Scalar_field_XZ. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **verbose** (*bool*) – shows data process by screen

intensity()
Returns intensity.

cut_resample(*x_limits: list[float, float] | str = ''*, *y_limits: list[float, float] | str = ''*, *z_limits: list[float, float] | str = ''*, *num_points: Optional[int] = None*, *new_field: bool = False*, *interp_kind=(3, 1)*)
it cut the field to the range (x0,x1). If one of this x0,x1 positions is out of the self.x range it do nothing. It is also valid for resampling the field, just write x0,x1 as the limits of self.x

Parameters

- **x_limits** (*float,float*) – (x0,x1) starting and final points to cut, if '' - takes the current limit x[0] and x[-1]
- **y_limits** (*float,float*) – (y0,y1) - starting and final points to cut, if '' - takes the current limit y[0] and z[-1]
- **num_points** (*int*) – it resamples x, y and u, where [],'',0,None -> it leave the points as it is new_field (bool): it returns a new Scalar_field_XY
- **interp_kind** – numbers between 1 and 5

incident_field(*u0*, *z0: Optional[float] = None*)
Incident field for the experiment. It takes a Scalar_source_XYZ field.

Parameters

- **u0** (Scalar_source_X) – field produced by Scalar_source_XYZ (or a XYZ field)
- **z0** (*float*) – position of the incident field. if None, '', [], is at the beginning

final_field()
Returns the final field as a Scalar_field_XYZ.

RS(*verbose: bool = False*, *num_processors=6*)
Rayleigh Sommerfeld propagation algorithm

Parameters

- **verbose** (*bool*) – shows the quality of algorithm (>1 good)
- **num_processors** (*int*) – number of processors for multiprocessing

Returns time in the processing

RS_amplification(*amplification: list[int, int] = (1, 1)*)
Rayleigh Sommerfeld propagation algorithm. it performs amplification

Parameters **amplification** (*int,int*) – number of fields

Returns

Return type *Scalar_field_XY*

BPM(*has_edges: bool = True*, *pow_edge: int = 80*, *verbose: bool = False*)
3D Beam propagation method (BPM).

Parameters

- **has_edges** (*bool*) – If True absorbing edges are used.
- **pow_edge** (*float*) – If has_edges, power of the supergaussian
- **verbose** (*bool*) – shows data process by screen

References

Algorithm in "Engineering optics with matlab" pag 119

PWD(*n: Optional[float] = None*, *matrix: bool = False*, *verbose: bool = False*)
Plane wave decompostion algorithm.

Parameters

- **n** (*np. array*) – refractive index, If None, it is n_background
- **verbose** (*bool*) – If True prints state of algorithm

Returns Field at at distance dz from the incident field

Return type numpy.array()

References

1. Schmidt, S. et al. Wave-optical modeling beyond the thin-element-approximation. Opt. Express 24, 30188 (2016).

WPM(*has_edges: bool = True*, *pow_edge: int = 80*, *verbose: bool = False*)
WPM Methods. 'schmidt method is very fast, only needs discrete number of refractive indexes'

Parameters

- **has_edges** (*bool*) – If True absorbing edges are used.
- **pow_edge** (*float*) – If has_edges, power of the supergaussian
- **verbose** (*bool*) – If True prints information

References

1. M.W. Fertig and K.-H. Brenner, "Vector wave propagation method," J. Opt. Soc. Am. A, vol. 27, no. 4, p. 709, 2010.
2. S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

to_Scalar_field_XY(*iz0: Optional[int] = None*, *z0: Optional[float] = None*, *is_class: bool = True*, *matrix: bool = False*)

pass results to Scalar_field_XY. Only one of the first two variables (iz0,z0) should be used

Parameters

- **iz0** (*int*) – position i of z data in array
- **z0** (*float*) – position z to extract
- **is_class** (*bool*) – If True it returns a class
- **matrix** (*bool*) – If True it returns a matrix

to_Scalar_field_XZ(*iy0: Optional[int] = None*, *y0: Optional[float] = None*, *is_class: bool = True*, *matrix: bool = False*)

pass results to Scalar_field_XZ. Only one of the first two variables (iy0,y0) should be used

Parameters

- **iy0** (*int*) – position i of y data in array
- **y0** (*float*) – position y to extract
- **class** (*bool*) – If True it returns a class
- **matrix** (*bool*) – If True it returns a matrix

to_Scalar_field_YZ(*ix0: Optional[int] = None*, *x0: Optional[float] = None*, *is_class: bool = True*, *matrix: bool = False*)

pass results to Scalar_field_XZ. Only one of the first two variables (iy0,y0) should be used

Parameters

- **ix0** (*int*) – position i of x data in array
- **x0** (*float*) – position x to extract
- **class** (*bool*) – If True it returns a class
- **matrix** (*bool*) – If True it returns a matrix

to_Z(*kind: str = 'amplitude'*, *x0: Optional[float] = None*, *y0: Optional[float] = None*, *has_draw: bool = True*, *z_scale='mm'*)

pass results to u(z). Only one of the first two variables (iy0,y0) and (ix0,x0) should be used.

Parameters

- **kind** (*str*) – 'amplitude', 'intensity', 'phase'
- **x0** (*float*) – position x to extract
- **y0** (*float*) – position y to extract
- **has_draw** (*bool*) – draw the field
- **z_scale** (*str*) – 'mm', 'um'

Returns array with z field (numpy.array): amplitude, intensity or phase of the field

Return type z (numpy.array)

average_intensity(*has_draw: bool = False*)
Returns average intensity as: (np.abs(self.u)**2).mean()

Parameters **verbose** (*bool*) – If True prints data.

Returns z array with average intensity at each plane z.

Return type intensity_mean (np.array)

beam_widths(*kind: str = 'FWHM2D'*, *has_draw: list[bool] = [True, False]*, *percentage: float = 0.5*, *remove_background: Optional[bool] = None*, *verbose: bool = False*)
Determines the widths of the beam

Parameters

- **kind** (*str*) – kind of algorithm: 'sigma4', 'FWHM2D'
- **has_draw** (*bool, bool*) – First for complete analysis, second for all FWHM2D computations
- **percentage** – TODO
- **remove_background** – TODO
- **verbose** (*bool*) – prints info

Returns beam_width_x (np.array) beam_width_y (np.array) principal_axis_z (np.array)

surface_detection(*mode: int = 1*, *min_incr: float = 0.01*, *has_draw: bool = False*)
detect edges of variation in refractive index

Parameters

- **mode** (*int*) – TODO
- **min_incr** (*float*) – minimum incremental variation to detect.
- **has_draw** (*bool*) – if True, it draws the surface.

draw_proposal(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0*, *normalize='maximum'*, *draw_borders: bool = False*, *filename: str = ''*, *scale: str = ''*, *min_incr: float = 0.0005*, *reduce_matrix: str = 'standard'*, *colorbar_kind: bool | str = False*, *colormap_kind: str = 'gist_heat'*)
Draws XYZ field.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', 'real'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **draw_borders** (*bool*) – If True draw edges of objects
- **filename** (*str*) – if not '' stores drawing in file,
- **scale** (*str*) – '', 'scaled', 'equal', scales the XY drawing
- **min_incr** – incrimum increment in refractive index for detecting edges
- **reduce_matrix** (*int, int*) – when matrix is enormous, we can reduce it only for drawing purposes. If True, reduction factor

draw_XY(*z0: float = 5000.0*, *kind: str = 'intensity'*, *logarithm: numpy.float64 = 0*, *normalize: str = 'maximum'*, *title: str = ''*, *filename: str = ''*, *cut_value: Optional[float] = None*, *has_colorbar='False'*, *reduce_matrix=''*)

longitudinal profile XY at a given z value

Parameters

- **z0** (*float*) – value of z for interpolation
- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', ' 'field', 'real_field', 'contour'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'area', 'intensity'
- **title** (*str*) – title for the drawing
- **filename** (*str*) – if not '' stores drawing in file,
- **cut_value** (*float*) – if provided, maximum value to show
- **has_colorbar** (*bool*) – if True draws the colorbar
- **()** (*reduce_matrix*) –

draw_XZ(*kind: str = 'intensity'*, *y0: float = 0.0*, *logarithm: numpy.float64 = 0*, *normalize: str = ''*, *draw_borders: bool = False*, *filename: str = ''*, ***kwargs*)

Longitudinal profile XZ at a given x0 value.

Parameters

- **y0** (*float*) – value of y for interpolation
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'intensity', 'area'
- **draw_borders** (*bool*) – check
- **filename** (*str*) – filename to save

draw_YZ(*x0: float = 0.0*, *logarithm: numpy.float64 = 0*, *normalize: str = ''*, *draw_borders: bool = False*, *filename: str = ''*)

Longitudinal profile YZ at a given x0 value.

Parameters

- **x0** (*float*) – value of x for interpolation
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'intensity', 'area'
- **draw_borders** (*bool*) – check
- **filename** (*str*) – filename to save

draw_XYZ_deprecated(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0.0*, *normalize: str = ''*, *pixel_size: list[int, int, int] = (128, 128, 128)*)

Draws XZ field.

Parameters

- **kind** (*str*) – type of drawing: 'intensity', 'phase', 'real_field'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm

- **normalize** (*bool*) – If True, max(intensity)=1
- **pixel_size** (*float, float, float*) – pixels for drawing

draw_volume_deprecated(*logarithm: numpy.float64 = 0*, *normalize: str = ''*, *maxintensity: Optional[float] = None*)

Draws XYZ field with mlab

Parameters

- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **maxintensity** (*float*) – maximum value of intensity

TODO: Simplify, drawing include kind and other parameters of draw

draw_refractive_index_deprecated(*kind: str = 'real'*)

Draws XYZ refractive index with slicer

Parameters **kind** (*str*) – 'real', 'imag', 'abs'

video(*filename: str = ''*, *kind: str = 'intensity'*, *fps: int = 15*, *frame: bool = True*, *axis: str = 'auto'*, *verbose: bool = False*, *directory_name: str = 'new_video'*)

Makes a video in the range given by self.z.

Parameters

- **filename** (*str*) – filename (.avi, .mp4)
- **kind** (*str*) – type of drawing: 'intensity', 'phase'.
- **fps** (*int*) – frames per second
- **frame** (*bool*) – figure with or without axis.
- **verbose** (*bool*) – If True prints

TODO: include logarithm and normalize

4.1.6 diffractio.scalar_fields_XZ module

This module generates Scalar_field_XZ class.

It includes multiprocessing for RS and BPM polychromatic

It can be considered an extension of Scalar_field_X for visualizing XZ fields

For the case of Rayleigh sommefeld it is not necessary to compute all z positions but the final.

Nevertheless, for BPM method, intermediate computations are required. In this class, intermediate results are stored.

X,Z fields are defined using ndgrid (not with meshgrid, it is different).

It is required also for generating masks and fields. The main atributes are:

- self.x - x positions of the field
- self.z - z positions of the field
- self.u - field XZ
- self.n - refractive index XZ
- self.wavelength - wavelength of the incident field. The field is monochromatic

The magnitude is related to microns: *micron = 1.*

Class for XZ scalar fields

Definition of a scalar field

- instatiation, duplicate, clean_refractive_index
- save, load data
- rotate_field, cut_resample,

Illumination

- incident_field

Operations

- surface_detection
- search focus

Propagation

- RS, RS_polychormatic,
- BPM, BPM_poychromatic, BPM_inverse, BPM_back_propagation

Drawing functions

- draw
- draw_refractive_index
- draw_incident_field
- draw_profiles_interactive
- video

Parameters

- final_field
- profile_longitudinal
- profile_transversal

class diffractio.scalar_fields_XZ.**Scalar_field_XZ**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *z: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: `object`

Class for working with XZ scalar fields.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **z** (*numpy.array*) – linear array wit equidistant positions for z values
- **wavelength** (*float*) – wavelength of the incident field
- **n_background** (*float*) – refractive index of background

- **info** (*str*) – String with info about the simulation

self.**x**
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

self.**z**
linear array wit equidistant positions for z values

Type numpy.array

self.**wavelength**
wavelength of the incident field.

Type float

self.**u0**
(x) size x - field at the last z position

Type numpy.array

self.**u**
(x,z) complex field

Type numpy.array

self.**n_background**
(x,z) refractive index

Type numpy.array

self.**fast**
if True fast algoritm (approx. Hankle function)

Type bool

self.**info**
String with info about the simulation

Type str

reduce_to_1()
All the values greater than 1 pass to 1. This is used for Scalar_masks when we add two masks.

duplicate(*clear: bool = False*)
Duplicates the instance

refractive_index_from_scalar_mask_XY(*mask_XY*, *refractive_index_max: numpy.float64*)

Transforms XY mask into XZ mask.

- Areas with value 0 pass to n_background.
- When transmittance of mask_XY is 1, pass to refractive_index_max.

Parameters

- **mask_XY** (*diffractio.Scalar_mask_XY*) – mask
- **refractive_index_max** (*float*) – real and imaginary part of maximum refractive index.

Returns _description_

Return type _type_

rotate_field(*angle: numpy.float64*, *center_rotation: list[numpy.float64]*, *kind: str = 'all'*, *n_background: numpy.float64 = 1.0*)

Rotate all the image a certain angle

Parameters

- **angle** (*float*) – angle to rotate, in radians
- **center_rotation** (*float, float*) – (z,x) position for rotation
- **kind** (*str*) – 'all', 'n', 'field'
- **n_background** (*float*) – refractive index of zone incoming

clear_field()

clear field

clear_refractive_index()

clear refractive index n(x,z)=n_background

normalize(*new_field: bool = False*)

Normalizes the field so that intensity.max()=1.

Parameters **new_field** (*bool*) – If False the computation goes to self.u. If True a new instance is produced

Returns u (numpy.array): normalized optical field

mask_field(*size_edge: numpy.float64 = 0*)

mask the incident field at the edges, each edge is masked size_edge

Parameters **size_edge** (*float*) – size of edges

smooth_refractive_index(*type_filter: int = 2*, *pixels_filtering: int = 10*, *max_diff_filter: numpy.float64 = 0.1*, *draw_check: bool = False*)

Technique to remove artifacts in BPM propagation.

Parameters

- **type_filter** (*int*) – 1 - 2D, 2 - 1D z (selective), 3 - 1D x (selective)
- **pixels_filtering** (*int*) – num_pixels used for filtering
- **max_diff_filter** (*float*) – maximum difference of n in profile between two adjancted pixels to use selective filtering
- **draw_check** (*bool*) – draw the differences.

Returns percentage_filtered (np.array): lineas_filtradas

Return type (float)

References

Robert McLeod "Numerical Methods in Photonics Lecture Notes" University of Colorado at Boulder, pag 204 (15/54)

save_data(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **add_name=** (*str*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

load_data(*filename: str*, *verbose: bool = False*)

Load data from a file to a Scalar_field_XZ. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **verbose** (*bool*) – shows data process by screen

cut_resample(*x_limits: list[float, float] = ''*, *z_limits: list[float, float] = ''*, *num_points: Optional[int] = None*, *new_field: bool = False*, *interp_kind: list[int] = (3, 1)*)
it cut the field to the range (x0,x1). if one of this x0,x1 positions is out of the self.x range it do nothing. It is also valid for resampling the field, just write x0,x1 as the limits of self.x

Parameters

- **x_limits** (*float,float*) – (x0,x1) starting and final points to cut. if '' - takes the current limit x[0] and x[-1]
- **z_limits** (*float,float*) – (z0,z1) - starting and final points to cut. if '' - takes the current limit z[0] and z[-1]
- **num_points** (*int*) – it resamples x, z and u. ([],'',0,None) -> it leave the points as it is
- **new_field** (*bool*) – it returns a new Scalar_field_XZ
- **interp_kind** – numbers between 1 and 5

incident_field(*u0*, *z0: Optional[numpy.float64] = None*)
Incident field for the experiment. It takes a Scalar_source_X field

Parameters

- **u0** (Scalar_source_X) – field produced by Scalar_source_X (or a X field)
- **z0** (*float*) – position of the incident field. if None, '', [], is at the beginning

final_field()
Returns the final field as a Scalar_field_X.

BPM(*has_edges: bool = True*, *pow_edge: int = 80*, *division: bool = False*, *matrix: bool = False*, *verbose: bool = False*)
Beam propagation method (BPM).

Parameters

- **has_edges** (*bool or np.array*) – If True absorbing edges are used. If np.array, they are 0 or 1 depending if at this z position filtering is performed.
- **pow_edge** (*float*) – If has_edges, power of the supergaussian
- **division** (*False, int*) – If False nothing, else divides the BPM algorithm in several different executions. To avoid RAM problems
- **matrix** (*bool*) – if True returns a matrix else
- **verbose** (*bool*) – shows data process by screen

References: Algorithm in "Engineering optics with matlab" pag 119.

BPM_inverse(*verbose: bool = False*)
Beam propagation method (BPM) in inverse mode.

Parameters **verbose** (*bool*) – shows data process by screen

References

Algorithm in "Engineering optics with matlab" pag 119

BPM_back_propagation(*verbose: bool = False*)
Beam propagation method (BPM). The field that generates the final field is obtained.

Parameters **verbose** (*bool*) – shows data process by screen

References

Algorithm in "Engineering optics with matlab" pag 119

RS(*xout: Optional[float] = None*, *verbose: bool = False*, *num_processors: int = 8*)
Rayleigh Sommerfeld propagation algorithm

Parameters

- **xout** (*float*) – init position of output position
- **verbose** (*bool*) – shows the quality of algorithm (>1 good)
- **num_processors** (*int*) – number of processors for multiprocessing

Returns Processing time

PWD(*n: Optional[numpy.float64] = None*, *matrix: bool = False*, *verbose: bool = False*)
Plane wave decomposition algorithm (PWD).

Parameters

- **n** (*np. array*) – refractive index, If None, it is n_background
- **matrix** (*bool*) – if True returns a matrix else
- **verbose** (*bool*) – If True prints state of algorithm

Returns Field at at distance dz from the incident field

Return type numpy.array()

References

1. Schmidt, S. et al. Wave-optical modeling beyond the thin-element-approximation. Opt. Express 24, 30188 (2016).

WPM(*kind: str = 'schmidt'*, *has_edges: bool = True*, *pow_edge: int = 80*, *matrix: bool = False*, *verbose: bool = False*)

WPM Method. 'schmidt method is very fast, only needs discrete number of refractive indexes'

Parameters

- **kind** (*str*) – 'schmidt', 'scalar'
- **has_edges** (*bool*) – If True absorbing edges are used.
- **pow_edge** (*float*) – If has_edges, power of the supergaussian
- **matrix** (*bool*) – if True returns a matrix else
- **verbose** (*bool*) – If True prints information

References

1. M.W. Fertig and K.-H. Brenner, "Vector wave propagation method," J. Opt. Soc. Am. A, vol. 27, no. 4, p. 709, 2010.
2. S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

RS_polychromatic(*initial_field*, *wavelengths: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *spectrum: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *verbose: bool = False*, *num_processors: int = 8*)

Rayleigh Sommerfeld propagation algorithm for polychromatic light.

Parameters

- **initial_field** (Scalar_field_X) – function with only input variable wavelength
- **wavelengths** (*numpy.array*) – array with wavelengths
- **spectrum** (*numpy.array*) – array with spectrum. if '' then uniform_spectrum
- **verbose** (*bool*) – shows the quality of algorithm (>1 good)
- **num_processors** (*int*) – number of processors for multiprocessing

Returns self.u=sqrt(Intensities) - no phase is stored, only intensity

Return type *Scalar_field_XZ*

BPM_polychromatic(*initial_field*, *wavelengths: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *spectrum: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *verbose: bool = False*, *num_processors: int = 4*)

Rayleigh Sommerfeld propagation algorithm for polychromatic light

Parameters

- **initial_field** (Scalar_field_X) – function with only input variable wavelength
- **wavelengths** (*numpy.array*) – array with wavelengths
- **spectrum** (*numpy.array*) – array with spectrum. if '' then uniform_spectrum
- **verbose** (*bool*) – shows the quality of algorithm (>1 good)

- **num_processors** (*int*) – number of processors for multiprocessing

Returns self.u=sqrt(Intensities) - no phase is stored, only intensity

Return type *Scalar_field_XZ*

fast_propagation(*mask_xz*, *num_pixels_slice: int = 1024*, *verbose: bool = False*)
combines RS and BPM"" to generate the final field

Parameters

- **mask_xz** (Scalar_mask_XZ) – function that returns Scalar_mask_XZ
- **num_pixels_slice** (*int*) – num of slices for each BPM propagation
- **verbose** (*bool*) – If True prints info.

Returns u_current, fields_BPM, transitions

intensity()
Returns the intensity of the field

Returns intensity of the field.

Return type (np.array)

average_intensity(*has_draw: bool = False*)
Returns average intensity as: (np.abs(self.u)**2).mean()

Parameters **has_draw** (*bool*) – If True draws data.

Returns z array with average intensity at each plane z.

Return type intensity_mean (np.array)

check_intensity(*has_draw: bool = True*, *normalized: bool = True*)
Checks that intensity distribution is not lost by edges. It can be executed after a RS or BPM propagation.

Parameters

- **has_draw** (*bool*) – Draws the intensity
- **normalized** (*bool*) – Draws it normalized

Returns array with intensity I(z)

Return type (np.array)

detect_index_variations(*n_edge: numpy.float64*, *incr_n: numpy.float64 = 0.1*)
In a XZ masks, detects refractive index variations.

Parameteres: n_edge (float): incr_n (float): refractive index variation to detect

Returns x for left edge. h_lens_l (np.array): h for left edge. x_lens_r (np.array): x for right edge. h_lens_r (np.array): h for right edge.

Return type x_lens_l (np.array)

surface_detection(*mode: int = 1*, *min_incr: numpy.float64 = 0.1*, *reduce_matrix: str = 'standard'*, *has_draw: bool = False*)
detect edges of variation in refractive index.

Parameters

- **mode** (*int*) – 1 or 2, algorithms for surface detection: 1-gradient, 2-diff
- **min_incr** (*float*) – minimum incremental variation to detect

- **reduce_matrix** (*int, int*) – when matrix is enormous, we can reduce it only for drawing purposes. If True, reduction factor
- **has_draw** (*bool*) – If True draw.

draw(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0*, *normalize: bool = False*, *draw_borders: bool = False*, *filename: str = ''*, *scale: str = ''*, *min_incr: numpy.float64 = 0.0005*, *reduce_matrix: str = 'standard'*, *colorbar_kind: Optional[str] = None*, *colormap_kind: str = ''*, *z_scale: str = 'um'*, *edge_matrix: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *interpolation: str = 'spline36'*, *percentage_intensity: Optional[numpy.float64] = None*, ***kwargs*)

Draws XZ field.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', 'real'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **draw_borders** (*bool*) – If True draw edges of objects
- **filename** (*str*) – if not '' stores drawing in file,
- **scale** (*str*) – '', 'scaled', 'equal', scales the XY drawing
- **min_incr** – incrimum increment in refractive index for detecting edges
- **reduce_matrix** (*int, int*) – when matrix is enormous, we can reduce it only for drawing purposes. If True, reduction factor
- **z_scale** (*str*) – 'mm', 'um'
- **edge_matrix** (*numpy.array*) – positions of borders
- **interpolation** (*str*) – methods = [None, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos']
- **percentage_intensity** (*None or number*) – If None it takes from CONF_DRAWING['percentage_intensity'], else uses this value

draw_refractive_index(*kind: str = 'all'*, *draw_borders: bool = True*, *title: str = ''*, *filename: str = ''*, *scale: str = ''*, *min_incr: numpy.float64 = 0.01*, *reduce_matrix: str = 'standard'*, *colorbar_kind: Optional[str] = None*, *colormap_kind: str = <matplotlib.colors.LinearSegmentedColormap object>*, *edge_matrix: Optional[numpy.ndarray[Any*, *numpy.dtype[numpy.float64]]] = None*)

Draws refractive index.

Parameters

- **kind** (*str*) – 'all', 'real', 'imag'
- **draw_borders** (*bool*) – If True draw edges of objects
- **filename** (*str*) – if not '' stores drawing in file,
- **title** (*str*) – title of drawing
- **scale** (*str*) – '', 'scaled', 'equal', scales the XY drawing
- **min_incr** – minimum increment in refractive index for detecting edges
- **reduce_matrix** (*int, int*) – when matrix is enormous, we can reduce it only for drawing purposes. If True, reduction factor

- **edge_matrix** (*numpy.array*) – positions of borders

draw_incident_field(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *filename: str = ''*)

Draws incident field self.u0

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'field', 'phase', 'fill', 'fft'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **filename** (*str*) – if not '' stores drawing in file,

profile_longitudinal(*kind: str = 'intensity'*, *x0: numpy.float64 = 0.0*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *z_scale: str = 'um'*, *has_draw: bool = True*, *filename: str = ''*)

Determine and draws longitudinal profile

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', 'refractive_index'
- **x0** (*float*) – profile that passes through x=x0
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'intensity', 'area'
- **has_draw** (*bool*) – If True, draws, False only returns profile
- **filename** (*str*) – if not '' stores drawing in file

Returns profile

Return type numpy.array

profile_transversal(*kind: str = 'intensity'*, *z0: numpy.float64 = 0.0*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *has_draw: bool = True*, *filename: str = ''*)

Determine and draws transversal profile.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', 'refractive_index'
- **z0** (*float*) – profile that passes through z=z0
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'intensity', 'area'
- **draw** (*bool*) – If True, draws, False only returns profile
- **filename** (*str*) – if not '' stores drawing in file,

Returns profile

Return type numpy.array

TODO: Include interpolation

search_focus(*verbose=True*)

Search for location of maximum.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'phase', 'refractive_index'

- **x0** (*float*) – profile that passes through x=x0
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*str*) – False, 'maximum', 'intensity', 'area'
- **draw** (*bool*) – If True, draws, False only returns profile
- **filename** (*str*) – if not '' stores drawing in file,

Returns positions of focus

Return type (x,z)

beam_widths(*kind='FWHM1D'*, *has_draw: list[bool] = [True, False]*, *z_scale: str = 'um'*, *percentage: numpy.float64 = 0.5*, *remove_background: Optional[str] = None*, *verbose: bool = False*)

Computes the beam width for all the distances z. :param kind: kind of algorithm: 'sigma4', 'FWHM1D' :type kind: str :param has_draw: First for complete analysis, second for all FWHM2D computations :type has_draw: bool, bool :param percentage: (0-1) percentage for the beam detection :type percentage: float :param remove_background=None: :param : :param verbose: prints info :type verbose: bool

Returns for each distance z (numpy.array) positions_center: positions of centers for each z

Return type (numpy.array) widths

video(*kind: str = 'intensity'*, *z_min: Optional[numpy.float64] = None*, *z_max: Optional[numpy.float64] = None*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *time_video: numpy.float64 = 10.0*, *frames_reduction: int = 5*, *filename: str = 'video.avi'*, *dpi: int = 100*)

Generates a video in the z dimension.

Parameters

- **kind** (*str*) –
- **z_min** (*float*) –
- **z_max** (*float*) –
- **logarithm** (*bool*) –
- **normalize** (*bool*) –
- **time_video** (*float*) –
- **frames_reduction** (*int*) –
- **filename** (*str*) –
- **dpi** (*int*) –

draw_profiles_interactive(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*)

Draws profiles interactivey. Only transversal

Parameters

- **kind** (*str*) – 'intensity', 'amplitude', 'phase'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1

4.1.7 diffractio.scalar_fields_Z module

This module generates Scalar_field_Z class

The main atributes are:

- self.z (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n .
- self.wavelength (float): wavelength of the incident field.
- self.u (numpy.array): equal size than x. complex field

There are also some secondary atributes:

- self.quality (float): quality of RS algorithm
- self.info (str): description of data
- self.type (str): Class of the field
- self.date (str): date

Class for unidimensional scalar fields

Definition of a scalar field

- instantiation, duplicate, clear_field, print
- save and load data

Drawing functions

- draw

Parameters:

- intensity, average intensity
- get_edges_transitions (mainly for pylithography)

class diffractio.scalar_fields_Z.**Scalar_field_Z**(*z: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: `object`

Class for unidimensional scalar fields in z axis.

Parameters

- **z** (*numpy.array*) – linear array with equidistant positions.
- **wavelength** (*float*) – wavelength of the incident field
- **n_background** (*float*) – refractive index of background
- **info** (*str*) – String with info about the simulation

`self.z`
Linear array with equidistant positions. The number of data is preferibly 2^n.

Type numpy.array

`self.wavelength`
Wavelength of the incident field.

Type float

`self.u`
Complex field. The size is equal to self.z.

Type numpy.array

`self.quality`
Quality of RS algorithm.

Type float

`self.info`
Description of data.

Type str

`self.type`
Class of the field.

Type str

`self.date`
Date when performed.

Type str

`duplicate`(*clear: bool = False*)
Duplicates the instance

`conjugate`(*new_field: bool = True*)
Conjugates the field

`clear_field`()
Removes the field so that self.u = 0.

`save_data`(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **add_name=** (*str*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

`load_data`(*filename: str*, *verbose: bool = False*)

Load data from a file to a Scalar_field_X. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **verbose** (*bool*) – shows data process by screen

`cut_resample`(*z_limits: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *num_points: Optional[int] = None*, *new_field: bool = False*, *interp_kind: str = 'linear'*)
Cuts the field to the range (z0,z1). If one of this z0,z1 positions is out of the self.z range it does nothing. It is also valid for resampling the field, just write z0,z1 as the limits of self.z

Parameters

- **z_limits** (*numpy.array*) – (z0,z1) - starting and final points to cut, if '' - takes the current limit z[0] and z[-1]
- **num_points** (*int*) – it resamples z, and u [],'',0,None -> it leave the points as it is
- **new_field** (*bool*) – if True it returns a new Scalar_field_z
- **interp_kind** (*str*) – 'linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic'

Returns if new_field is True

Return type (*Scalar_field_Z*)

normalize(*new_field: bool = False*)
Normalizes the field so that intensity.max()=1.

Parameters **new_field** (*bool*) – If False the computation goes to self.u. If True a new instance is produced

Returns u (numpy.array): normalized optical field

intensity()
Intensity.

Returns Intensity

Return type (numpy.array)

average_intensity(*verbose: bool = False*)
Returns the average intensity as: (np.abs(self.u)**2).sum() / num_data

Parameters **verbose** (*bool*) – If True it prints the value of the average intensity.

Returns average intensity.

Return type (float)

FWHM1D(*percentage: numpy.float64 = 0.5*, *remove_background: Optional[bool] = None*, *has_draw: bool = False*)

Parameters

- **percentage** (*float*) – value between 0 and 1. 0.5 means that the width is computed at half maximum.
- **remove_background** (*str*) – 'min', 'mean', None
- **has_draw** (*bool*) – If true it draws

Returns width, in this z case: DOF

Return type width (float)

DOF(*percentage: numpy.float64 = 0.5*, *remove_background: Optional[str] = None*, *has_draw: bool = False*)
Determines Depth-of_focus (DOF) in terms of the width at different distances

Parameters

- **percentage** (*float*) – value between 0 and 1. 0.5 means that the width is computed at half maximum.
- **remove_background** (*str*) – 'min', 'mean', None
- **has_draw** (*bool*) – If true it draws

Returns width, in this z case: D

Return type width (float)

References

B. E.A. Saleh and M. C. Teich, Fundamentals of photonics. john Wiley & sons, 2nd ed. 2007. Eqs (3.1-18) (3.1-22) page 79

Returns Depth of focus (float): beam waist (float, float, float): postions (z_min, z_0, z_max) of the depth of focus

Return type (float)

draw(*kind: str = 'intensity'*, *logarithm: numpy.float64 = 0.0*, *normalize: bool = False*, *cut_value: Optional[numpy.float64] = None*, *z_scale: str = 'um'*, *unwrap: bool = False*, *filename: str = ''*)

Draws z field. There are several data from the field that are extracted, depending of 'kind' parameter.

Parameters

- **kind** (*str*) – type of drawing: 'amplitude', 'intensity', 'field', 'phase'
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **cut_value** (*float*) – If not None, cuts the maximum intensity to this value
- **unwrap** (*bool*) – If True, unwraps the phase.
- **filename** (*str*) – if not '' stores drawing in file,

4.1.8 diffractio.scalar_masks_X module

This module generates Scalar_mask_X class for definingn masks. Its parent is Scalar_field_X.

The main atributes are:

- self.u - field
- self.x - x positions of the field
- self.wavelength - wavelength of the incident field. The field is monocromatic

Class for unidimensional scalar masks

Functions

- mask_from_function, mask_from_array
- slit, double slit
- two_levels, gray_scale
- prism, biprism_fresnel, biprism_fresnel_nh
- lens, lens_spherical, aspheric, fresnel_lens
- roughness, dust, dust_different_sizes
- sine_grating, ronchi_grating, binary_grating, blazed_grating
- chriped_grating, chirped_grating_p, chirped_grating_q

- I0s

class diffractio.scalar_masks_X.**Scalar_mask_X**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: *diffractio.scalar_fields_X.Scalar_field_X*

Class for unidimensional scalar masks.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **wavelength** (*float*) – wavelength of the incident field
- **n_background** (*float*) – refractive index of background
- **info** (*str*) – String with info about the simulation

self.**x**
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

self.**wavelength**
wavelength of the incident field.

Type float

self.**u**
equal size than x. complex field

Type numpy.array

self.**quality**
quality of RS algorithm

Type float

self.**info**
description of data

Type str

self.**type**
Class of the field

Type str

self.**date**
date when performed

Type str

filter(*mask*, *new_field: bool = True*, *binarize: bool = False*, *normalize: bool = False*)
Widens a field using a mask.

Parameters

- **mask** (*diffractio.Scalar_mask_X*) – filter
- **new_field** (*bool*) – If True, develope new Field
- **binarize** (*bool, float*) – If False nothing, else binarize in level

- **normalize** (*bool*) – If True divides the mask by sum:

mask_from_function(*index: float = 1.5, f1: float = 0, f2: float = 0, v_globals: dict = {}, x0: float = 0.0, radius: float = 0.0*)

Phase mask defined between two surfaces f_1 and f_2: $h(x,y) = f_2(x,y) - f_1(x,y)$, $t(x) = mask(x)e^{i\,k\,(n-1)(f_2-f_1)}$

Parameters

- **index** (*float*) – refractive index of the mask
- **f1** (*str*) – first surface
- **f2** (*str*) – second surface
- **v_globals** (*dict*) – variable definitions
- **mask** (*bool*) – True if a mask is defined to block areas
- **x0** (*float*) – center of mask
- **radius** (*float*) – radius of the mask

mask_from_array(*index: float = 1.5, array1: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None, array2: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None, interp_kind: str = 'quadratic', radius: float = 0.0, x0: float = 0.0*)

Phase mask defined between two surfaces defined by arrays: array1 and array2, $t(x) = mask(x)e^{i\,k\,(n-1)(array2(x,z)-array1(x,z))}$

Parameters

- **index** (*float*) – refractive index of the mask
- **array1** (*numpy.array*) – array of data (x,z) for the first surface
- **array2** (*numpy.array*) – array of data (x,z) for the second surface
- **interp_kind** (*str*) – ‘linear’, ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’
- **mask** (*bool*) – True if a mask is defined to block areas
- **x0** (*float*) – center of mask
- **radius** (*float*) – radius of the mask

dots(*x0: float | numpy.ndarray[Any, numpy.dtype[numpy.float64]]*)

Generates 1 or several point masks at positions x0

Parameters **np.array** (*x0 float or*) – x point or points where mask is 1.

slit(*x0: float, size: float*)

Slit: 1 inside, 0 outside

Parameters

- **x0** (*float*) – center of slit
- **size** (*float*) – size of slit

double_slit(*x0: float, size: float, separation: float*)

double slit: 1 inside, 0 outside

Parameters

- **x0** (*float*) – center of slit
- **size** (*float*) – size of slit

- **separation** (*float*) – separation between slit centers

two_levels(*level1: float = 0.0*, *level2: float = 1.0*, *x_edge: float = 0.0*)
Divides the image in two levels.

Parameters

- **level1** (*float*) – value of level1
- **level2** (*float*) – value of level2
- **x_edge** (*float*) – position of separation of levels

gray_scale(*num_levels: int*, *levelMin: float = 0*, *levelMax: float = 1*)
Divides the mask in n, vertical levels.

Parameters

- **num_levels** (*int*) – number of levels
- **levelMin** (*float*) – minimum value of levels
- **levelMax** (*float*) – maximum value of levels

prism(*x0: float*, *n: float*, *angle: float*)
Prism.

Parameters

- **x0** (*float*) – vertex of prism
- **n** (*float*) – refractive_index
- **angle** (*float*) – angle of prism

biprism_fresnel(*angle: float*, *x0: float*, *radius: float = 0*)
Fresnel biprism.

Parameters

- **angle** (*float*) – angle of the fresnel biprism
- **x0** (*float*) – central position of fresnel biprism
- **radius** (*float*) – radius of the fresnel biprism, if mask is True

biprism_fresnel_nh(*x0: float*, *width: float*, *height: float*, *n: float*)
Fresnel biprism, uses height and refractive index.

Parameters

- **x0** (*float*) – vertex of biprism
- **width** (*float*) – size of biprism
- **height** (*float*) – height of biprism
- **n** (*float*) – refractive_index

lens(*x0: float*, *focal: float*, *radius: float = 0*)
Paraxial lens.

Parameters

- **x0** (*float*) – center of lens
- **focal** (*float*) – focal length of lens

lens_spherical(*x0: float*, *radius: float*, *focal: float*, *refractive_index: float = 1.5*)
Spherical lens, without paraxial approximation. The focal distance and the refractive index are used for the definition. When the refractive index decreases, the radius of curvature decrases and less paraxial.

Parameters

- **r0** (*float, float*) – (x0,y0) - center of lens
- **radius** (*float, float) or (float*) – radius of lens mask
- **focal** (*float, float) or (float*) – focal length of lens
- **index** (*refractive*) – refractive index of the lens

aspheric(*x0: float*, *c: float*, *k: float*, *a: list*, *n0: float*, *n1: float*, *radius: float*)
Asferic surface. $z = rac{c r^2}{1+sqrt{1-(1+k) c^2 r^2 }}+sum{a_i r^{2i}}$

Parameters: x0 (float): position of center c (float): base curvature at vertex, inverse of radius k (float): conic constant a (list): order aspheric coefficients: A4, A6, A8, … n0 (float): refractive index of first medium n1 (float): refractive index of second medium radius (float): radius of aspheric surface

Conic Constant Surface Type k = 0 spherical k = -1 Paraboloid k < -1 Hyperboloid -1 < k < 0 Ellipsoid k > 0 Oblate eliposid

References: https://www.edmundoptics.com/knowledge-center/application-notes/optics/all-about-aspheric-lenses/

fresnel_lens(*x0: float*, *focal: float*, *kind: str = 'phase'*, *binary: bool = False*, *phase: float = 3.141592653589793*, *radius: float = 0.0*)
Fresnel lens. Amplitude phase, continuous or binary.

Parameters

- **x0** (*float*) – center of lens
- **focal** (*float*) – focal length of lens
- **kind** (*str*) – 'amplitude' or phase
- **binary** (*bool*) – binary or profile
- **phase** (*float*) – if kind=='phase' -> maximum phase
- **mask** (*bool*) – if True, mask with size radius
- **radius** (*float*) – radius of lens mask

Returns heights [0,1] of lens.

Return type h (np.array)

roughness(*t: float*, *s: float*)
Rough surface, phase

According to movile average (Ogilvy p.224). very good in time for long arrays

Parameters

- **t** (*float*) – correlation lens
- **s** (*float*) – std of roughness

Returns topography maps in microns

Return type numpy.array

dust_different_sizes(*percentage: float*, *size: float*, *std: float = 0.0*)
Mask with dust particles of different sizes.

Parameters

- **percentage** (*float*) – percentage of area afected by noise
- **size** (*float*) – mean size of dust
- **std** (*float*) – std for size of dust

Returns positions - positions of dust numpy.array: sizes - size of dust float: percentage_real - real percentage of dust

Return type numpy.array

dust(*percentage: float*, *size: float = 0*)
Mask with dust particles of equal sizes.

Parameters

- **percentage** (*float*) – percentage of area afected by noise
- **size** (*float*) – size of dust
- **value** (*float*) – value included when there is noise

Returns positions - positions of dust numpy.array: sizes - size of dust float: percentage_real - real percentage of dust

Return type numpy.array

sine_grating(*x0: float*, *period: float*, *amp_min: float = 0*, *amp_max: float = 1*)
Sinusoidal grating

Parameters

- **x0** (*float*) – shift of grating
- **period** (*float*) – period of the grating
- **amp_min** (*float*) – minimum amplitude
- **amp_max** (*float*) – maximum amplitude

ronchi_grating(*x0: float*, *period: float*, *fill_factor: float = 0.5*)
Amplitude binary grating, fill-factor can be defined. It is obtained as a sine_grating that after is binarized. Fill factor is determined as y0=cos(pi*fill_factor)

Parameters

- **x0** (*float*) – shift of the grating
- **period** (*float*) – period of the grating
- **fill_factor** (*float*) – (0,1) - fill factor of grating

binary_grating(*x0: float*, *period: float*, *fill_factor: float*, *amin: float*, *amax: float*, *phase: float*)
binary grating amplitude and/or phase

Parameters

- **x0** (*float*) – shift of the grating
- **period** (*float*) – period of the grating
- **fill_factor** (*float*) – (0,1) - fill factor of grating
- **amin** (*float*) – minimum amplitude

- **amax** (*float*) – maximum amplitude
- **phase** (*float*) – phase shift (radians)

blazed_grating(*x0: float*, *period: float*, *phase_max: float*)

Phase, blazed grating. The phase shift is determined by heigth and refractive index.

Parameters

- **x0** (*float*) – shift of the grating
- **period** (*float*) – period of the grating
- **phase_max** (*float*) – maximum_phase_differences

Returns phase for each position

Return type phase (np.array)

chirped_grating_p(*kind: str*, *p0: float*, *p1: float*, *amp_min: float*, *amp_max: float*, *phase_max: float*, *delta_x: float = 0*, *x0: Optional[float] = None*, *length: float = 0*, *x_center: float = 0*)

Chirped grating with linear p(x) variation.

Parameters

- **kind** (*str*) – 'amplitude', 'phase', 'amplitude_binary', 'phase_binary'
- **p0** (*float*) – initial period of the grating
- **p1** (*float*) – final period of the grating
- **amp_min** (*float*) – minimum transmittance
- **amp_max** (*float*) – maximum transmittance
- **phase_max** (*float*) – maximum modulation for phase gratings
- **delta_x** (*float*) – x shifting for movement of grating
- **x0** (*float*) –

  –
- **length** (*float*) – length of the grating. 0: length is equal to size of x l=(x[-1]-x[0]), <l: it can be shorter than l
- **x_center** (*float*) – x-position of center of grating

Returns px

Return type numpy.array

chirped_grating_q(*kind: str*, *p0: float*, *p1: float*, *amp_min: float*, *amp_max: float*, *phase_max: float*, *delta_x: float = 0*, *length: float = 0*, *x_center: float = 0*)

Chirped grating with linear q(x) variation. The transmitance is: t = np.cos(np.pi*q*(x-x0) + np.pi*q0*(x-x0))

Parameters

- **kind** (*str*) – 'amplitude', 'phase', 'amplitude_binary', 'phase_binary'
- **p0** (*float*) – initial period of the grating
- **p1** (*float*) – final period of the grating
- **amp_min** (*float*) – minimum transmittance
- **amp_max** (*float*) – maximum transmittance

- **phase_max** (*float*) – maximum modulation for phase gratings
- **delta_x** (*float*) – x shifting for movement of grating
- **length** (*float*) – length of the grating, 0: length is equal to size of x l=(x[-1]-x[0]). <l: it can be shorter than l
- **x_center** (*float*) – x-position of center of grating

Returns qx

Return type numpy.array

chirped_grating(*kind: str*, *p_x: float*, *x0: float*, *amp_min: float*, *amp_max: float*, *phase_max: float*, *delta_x: float*, *length: float = 0*)

General chirped grating with variation given by function p(x).

Parameters

- **kind** (*str*) – 'amplitude', 'phase', 'amplitude_binary', 'phase_binary'
- **p_x** (*str*) – function with variation of periods
- **amp_min** (*float*) – minimum transmittance
- **amp_max** (*float*) – maximum transmittance
- **phase_max** (*float*) – maximum modulation for phase gratings
- **delta_x** (*float*) – x shifting for movement of grating
- **length** (*float*) – length of the grating. 0: length is equal to size of x l=(x[-1]-x[0]). <l: it can be shorter than l

Returns p(x)

Return type numpy.array

binary_code_positions(*x_transitions: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *start: str = 'down'*, *has_draw: bool = True*)

Genenerates a binary code, using the positions given in x_transitions

Parameters

- **x_transitions** (*numpy.array*) – positions where transitions are placed
- **start** (*str*) – How the binary code starts:'down' (starts in 0) or 'up' (starts in 1)
- **has_draw** (*bool*) – If True, draws the code

binary_code(*x0: float = 0.0*, *kind: str = 'standard'*, *code: list[int] = [1, 1, 0, 0, 1, 0, 1]*, *bit_width: float = 20.0*)

Binary code in form of 1's and 0's.

Parameters

- **kind** (*str*) – there are serveral types of codes 'standard' - normal 'abs_fag' - used in some abs encoders
- **code** (*numpy.array*) – array with values of code
- **bit_width** (*float*) – size of each data of code
- **x0** (*float*) – Initial position

4.1.9 diffractio.scalar_masks_XY module

4.1.10 diffractio.scalar_masks_XYZ module

This module generates Scalar_mask_XYZ class for defininging masks. Its parent is scalar_fields_XYZ.

The main atributes are:

- self.x - x positions of the field
- self.y - y positions of the field
- self.z - z positions of the field
- self.u - field XYZ
- self.n - refractive index XYZ
- self.wavelength - wavelength of the incident field. The field is monochromatic

The magnitude is related to microns: *micron = 1.*

Class for unidimensional scalar masks

Functions

- object_by_surfaces
- sphere
- square
- cylinder

class diffractio.scalar_masks_XYZ.**Scalar_mask_XYZ**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *y: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *z: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: *diffractio.scalar_fields_XYZ.Scalar_field_XYZ*

object_by_surfaces(*r0: list[numpy.float64]*, *refractive_index: numpy.float64*, *Fs*, *angles*, *v_globals={}*)

TODO Mask defined by n surfaces given in array Fs={f1, f2, h(x,y,z)=f1(x,y,z)*f2(x,y,z)*....*fn(x,y,z)

Parameters

- **rotation_point** (*float, float, float*) – location of the mask
- **refractive_index** (*float, str*) – can be a number or a function n(x, y,z)
- **Fs** (*list*) – condtions as str that will be computed using eval
- **array1** (*numpy.array*) – array (x,y,z) that delimits the second surface
- **angle** (*float*) – angle of rotation (radians)
- **v_globals** (*dict*) – dict with global variables
- **verbose** (*bool*) – shows data if true

sphere(*r0: list[numpy.float64]*, *radius: list[numpy.float64]*, *refractive_index: numpy.float64*, *angles*)

Insert a sphere in background. If it is something else previous, it is removed.

Parameters

- **r0** – (x0, y0, z0) Location of sphere, for example (0 * um, 0*um, 0 * um)
- **radius** – (rx, ry, rz) Radius of sphere. It can be a ellipsoid. If radius is a number, then it is a sphere
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j

square(*r0: list[numpy.float64]*, *length: list[numpy.float64]*, *refractive_index: numpy.float64*, *angles=None*, *rotation_point: Optional[list[numpy.float64]] = None*)

Insert a rectangle in background. If something previous, is removed.

Parameters

- **r0** (*float, float, float*) – (x0, y0,z0) Location of the rectangle, for example (0*um, 0*um, 0*um)
- **size** (*float, float, float*) – x,y,z size of the rectangle
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float, float*) –

cylinder(*r0: list[numpy.float64]*, *radius: list[numpy.float64]*, *length: float*, *refractive_index: float*, *axis: list[numpy.float64]*, *angle: numpy.float64*)

Insert a cylinder in background. If something previous, is removed.

Parameters

- **r0** (*float, float, float*) – (x0, y0,z0) Location of the rectangle, for example (0*um, 0*um, 0*um)
- **radius** (*float,float*) – x,y, size of the circular part of cylinder
- **length** (*float*) – length of cylidner
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **axis** (*float float, float*) – axis direction
- **angle** (*float*) – angle of rotation of the semi-plane, in radians

4.1.11 diffractio.scalar_masks_XZ module

Generates Scalar_mask_XZ class for definingn masks. Its parent is Scalar_field_XZ.

The main atributes are:

- self.x - x positions of the field
- self.z - z positions of the field
- self.u - field XZ
- self.n - refractive index XZ
- self.wavelength - wavelength of the incident field. The field is monochromatic

The magnitude is related to microns: *micron = 1.*

Class for unidimensional scalar masks

Functions

- extrude_mask, mask_from_function, mask_from_array, object_by_surfaces

- image
- semi_plane, layer, rectangle, slit, sphere, semi_sphere
- wedge, prism, biprism
- ronchi_grating, sine_grating
- probe
- lens_plane_convergent, lens_convergent, lens_plane_divergent, lens_divergent
- roughness

class diffractio.scalar_masks_XZ.**Scalar_mask_XZ**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *z: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: *diffractio.scalar_fields_XZ.Scalar_field_XZ*

Class for working with XZ scalar masks.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **z** (*numpy.array*) – linear array wit equidistant positions for z values
- **wavelength** (*float*) – wavelength of the incident field
- **n_background** (*float*) – refractive index of background
- **info** (*str*) – String with info about the simulation

self.**x**
linear array with equidistant positions. The number of data is preferibly 2^n.

Type numpy.array

self.**z**
linear array wit equidistant positions for z values

Type numpy.array

self.**wavelength**
wavelength of the incident field.

Type float

self.**u0**
(x) size x - field at the last z position

Type numpy.array

self.**u**
(x,z) complex field

Type numpy.array

self.**n_background**
(x,z) refractive index

Type numpy.array

`self.`**info**
String with info about the simulation

Type str

extrude_mask(*t*, *z0: numpy.float64*, *z1: numpy.float64*, *refractive_index: numpy.float64*, *angle: numpy.float64 = 0.0*, *v_globals: dict = {}*)
Converts a Scalar_mask_X in volumetric between z0 and z1 by growing between these two planes :param t: an amplitude mask of type Scalar_mask_X. :type t: Scalar_mask_X :param z0: initial position of mask :type z0: float :param z1: final position of mask :type z1: float :param refractive_index: can be a number or a function n(x,z) :type refractive_index: float, str

mask_from_function(*r0: list[numpy.float64]*, *refractive_index: numpy.float64 | str*, *f1*, *f2*, *z_sides: list[numpy.float64]*, *angle: numpy.float64*, *v_globals: dict = {}*)
Phase mask defined between two surfaces f1 and f1: h(x,z)=f2(x,z)-f1(x,z)

Parameters

- **r0** (*float, float*) – location of the mask
- **refractive_index** (*float, str*) – can be a number or a function n(x,z)
- **f1** (*str*) – function that delimits the first surface
- **f2** (*str*) – function that delimits the second surface
- **z_sides** (*float, float*) – limiting upper and lower values in z,
- **angle** (*float*) – angle of rotation (radians)
- **v_globals** (*dict*) – dict with global variables

mask_from_array(*r0=(0.0, 0.0)*, *refractive_index=1.5*, *array1: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *array2: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *x_sides: Optional[list[float, float]] = None*, *angle: numpy.float64 = 0.0*, *v_globals: dict = {}*, *interp_kind: str = 'quadratic'*, *has_draw: bool = False*)
Mask defined between two surfaces given by arrays (x,z): h(x,z)=f2(x,z)-f1(x,z). For the definion of f1 and f2 from arrays is performed an interpolation

Parameters

- **r0** (*float, float*) – location of the mask
- **refractive_index** (*float, str*) – can be a number or a function n(x,z)
- **array1** (*numpy.array*) – array (x,z) that delimits the first surface
- **array2** (*numpy.array*) – array (x,z) that delimits the second surface
- **x_sides** (*float, float*) – limiting upper and lower values in x,
- **angle** (*float*) – angle of rotation (radians): TODO -> not working
- **v_globals** (*dict*) – dict with global variables -> TODO perphaps it is not necessary
- **interp_kind** – ‘linear’, ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’

mask_from_array_proposal(*r0: list[float, float] = (0.0, 0.0)*, *refractive_index_substrate: float | numpy.float64 = 1.5*, *refractive_index_mask: Optional[Union[float, numpy.float64]] = None*, *array1: Optional[Union[numpy.ndarray[Any, numpy.dtype[numpy.float64]], float]] = None*, *array2: Optional[Union[numpy.ndarray[Any, numpy.dtype[numpy.float64]], float]] = None*, *x_sides: Optional[list[float, float]] = None*, *angle: numpy.float64 = 0.0*, *v_globals: dict = {}*, *interp_kind: str = 'quadratic'*, *has_draw: bool = False*)

Mask defined between two surfaces given by arrays (x,z): h(x,z)=f2(x,z)-f1(x,z). For the definion of f1 and f2 from arrays is performed an interpolation

Parameters

- **r0** (*float, float*) – location of the mask
- **refractive_index_mask** (*float, str*) – can be a number or a function n(x,z)
- **refractive_index_substrate** (*float, str*) – can be a number or a function n(x,z)
- **array1** (*numpy.array*) – array (x,z) that delimits the first surface
- **array2** (*numpy.array*) – array (x,z) that delimits the second surface
- **x_sides** (*float, float*) – limiting upper and lower values in x,
- **angle** (*float*) – angle of rotation (radians): TODO -> not working
- **v_globals** (*dict*) – dict with global variables -> TODO perphaps it is not necessary
- **interp_kind** – 'linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic'

object_by_surfaces(*rotation_point: list[float, float]*, *refractive_index: float | str*, *Fs: list*, *angle: numpy.float64*, *v_globals: dict = {}*, *verbose: bool = False*)

Mask defined by n surfaces given in array Fs={f1, f2, ...}. h(x,z)=f1(x,z)*f2(x,z)*....*fn(x,z)

Parameters

- **rotation_point** (*float, float*) – location of the mask
- **refractive_index** (*float, str*) – can be a number or a function n(x,z)
- **Fs** (*list*) – condtions as str that will be computed using eval
- **angle** (*float*) – angle of rotation (radians)
- **v_globals** (*dict*) – dict with global variables -> TODO perphaps it is not necessary
- **verbose** (*bool*) – shows data if true

add_surfaces(*fx*, *x_sides*, *refractive_index*, *min_incr=0.1*, *angle=0.0*)

A topography fx is added to one of the faces of object u (self.n).

Parameters

- **u** (Scalar_mask_XZ) – topography
- **fx** (*numpy.array, numpy.array*) – [x1, fx1], [x2, fx2] array with topography to add
- **x_sides** (*float, float*) – positions of edges
- **refractive_index** (*float, str*) – refractive index: number of string
- **min_incr** (*float*) – minimum variation of refractive index to detect edge.
- **angle** (*float (float, float)*) – angle and optative rotation angle.

discretize_refractive_index(*num_layers=None*, *n_layers=None*, *new_field: bool = False*)

Takes a refractive index an discretize it according refractive indexes.

Parameters

- **num_layers** (*(int,int), optional*) – number of layers (for real and imaginary parts), without counting background. Defaults to None. By default, both parameters are None, but one of then must be filled. If both are present, num_layers is considered
- **n_layers** (*np.array, optional*) – array with refractive indexes to discretize. Defaults to None.
- **new_field** (*bool*) – If True, it generates a new field.

Returns refractive indexes selected.

Return type (np.array)

image(*filename*, *n_max*, *n_min*, *angle=0*, *invert=False*)

Converts an image file in an xz-refractive index matrix. If the image is gray-scale the refractive index is gradual betwee n_min and n_max. If the image is color, we get the first Red frame

Parameters

- **filename** (*str*) – filename of the image
- **n_max** (*float*) – maximum refractive index
- **n_min** (*float*) – minimum refractive index
- **angle** (*float*) – angle to rotate the image in radians
- **invert** (*bool*) – if True the image is inverted

TODO: Now it is only possible that image size is equal to XZ, change using interpolation Rotation position

dots(*positions*, *refractive_index=1*)

Generates 1 or several point masks at positions r0

Parameters

- **positions** (*float, float) or (np.array, np.array*) – (x,z) point or points where mask is 1
- **refractive_index** (*float*) – refractive index

semi_plane(*r0*, *refractive_index*, *angle=0*, *rotation_point=None*)

Inserts a semi-sphere in background (x>x0). If something else previous, it is removed.

Parameters

- **r0=** (*x0,z0) (float,float*) – Location of the same plane.
- **refractive_index** (*float, str*) – refractive index.
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

layer(*r0*, *depth*, *refractive_index*, *angle*, *rotation_point=None*)

Insert a layer. If it is something else previous, it is removed.

Parameters: r0 (float, float): (x0,z0) Location of the same plane, for example (0 * um, 20 * um) depth (float): depth of the layer refractive_index (float, str): refractive index , for example: 1.5 + 1.0j angle (float): angle of rotation of the semi-plane, in radians rotation_point (float, float). Rotation point

rectangle(*r0*, *size*, *refractive_index*, *angle=0.0*, *rotation_point=None*)

Insert a rectangle in background. Something previous, is removed.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **size** (*float, float*) – x,z size of the rectangle
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

slit(*r0*, *aperture*, *depth*, *refractive_index*, *refractive_index_center=''*, *angle=0*, *rotation_point=None*)
Insert a slit in background.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **aperture** (*float*) – length of the opened part of the slit
- **depth** (*float*) – depth of the slit
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **refractive_index_center** (*float, str?*) – refractive index of center if refractive_index_center='', [], 0 then we copy what it was previously at aperture
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

sphere(*r0*, *radius*, *refractive_index*, *angle=0*, *rotation_point=None*)
Insert a sphere in background.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **radius** (*float, float*) – radius x,y of the sphere (ellipsoid)
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

semi_sphere(*r0*, *radius*, *refractive_index*, *angle=0*, *rotation_point=None*)
Insert a semi_sphere in background.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **radius** (*float, float*) – radius x,y of the sphere (ellipsoid)
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

lens_plane_convergent(*r0*, *aperture*, *radius*, *thickness*, *refractive_index*, *angle=0*, *rotation_point=None*, *mask=0*)
Insert a plane-convergent lens in background-

Parameters

- **r0** (*float, float*) – (x0,z0) position of the center of lens, for example (0 * um, 20 * um) for plane-convergent z0 is the location of the plane for convergent-plane (angle =180*degrees) the thickness has to be added to z0
- **aperture** (*float*) – aperture of the lens. If it is 0, then it is not applied
- **radius** (*float*) – radius of the curved surface
- **thickness** (*float*) – thickness at the center of the lens
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **mask** (*array, str*) – (mask_depth, refractive_index) or False. It masks the field outer the lens using a slit with depth = mask_depth
- **rotation_point** (*float, float*) –

Returns geometrical focal distance (numpy.array): ipasa, indexes [iz,ix] of lens

Return type (float)

lens_convergent(*r0*, *aperture*, *radius*, *thickness*, *refractive_index*, *angle=0*, *rotation_point=None*, *mask=0*)

Inserts a convergent lens in background.

Parameters

- **r0** (*float, float*) – (x0,z0) position of the center of lens, for example (0 * um, 20 * um) for plane-convergent z0 is the location of the plane for convergent-plane (angle =180*degrees) the thickness has to be added to z0
- **aperture** (*float*) – aperture of the lens. If it is 0, then it is not applied
- **radius** (*float, float*) – (radius1,radius2) radius of curvature (with sign)
- **thickness** (*float*) – thickness at the center of the lens
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) – rotation point.
- **mask** (*array, str*) – (mask_depth, refractive_index) or False. It masks the field outer the lens using a slit with depth = mask_depth

Returns geometrical focal distance (numpy.array): ipasa, indexes [iz,ix] of lens

Return type (float)

lens_plane_divergent(*r0*, *aperture*, *radius*, *thickness*, *refractive_index*, *angle=0*, *rotation_point=None*, *mask=False*)

Insert a plane-divergent lens in background.

Parameters

- **r0** (*float, float*) – (x0,z0) position of the center of lens, for example (0 * um, 20 * um) for plane-convergent z0 is the location of the plane for convergent-plane (angle =180*degrees) the thickness has to be added to z0
- **aperture** (*float*) – aperture of the lens. If it is 0, then it is not applied
- **radius** (*float*) – radius of curvature (with sign)
- **thickness** (*float*) – thickness at the center of the lens
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j

- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **mask** (*array, str*) – (mask_depth, refractive_index) or False. It masks the field outer the lens using a slit with depth = mask_depth

Returns geometrical focal distance (numpy.array): ipasa, indexes [iz,ix] of lens

Return type (float)

lens_divergent(*r0*, *aperture*, *radius*, *thickness*, *refractive_index*, *angle=0*, *rotation_point=None*, *mask=0*)
Insert a divergent lens in background.

Parameters

- **r0** (*float, float*) – (x0,z0) position of the center of lens, for example (0 * um, 20 * um) for plane-convergent z0 is the location of the plane for convergent-plane (angle =180*degrees) the thickness has to be added to z0
- **aperture** (*float*) – aperture of the lens. If it is 0, then it is not applied
- **radius** (*float, float*) – (radius1, radius2) radius of curvature (with sign)
- **thickness** (*float*) – thickness at the center of the lens
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) – rotation point
- **mask** (*array, str*) – (mask_depth, refractive_index) or False. It masks the field outer the lens using a slit with depth = mask_depth

Returns geometrical focal distance (numpy.array): ipasa, indexes [iz,ix] of lens

Return type (float)

aspheric_surface_z(*r0*, *refractive_index*, *cx*, *Qx*, *a2*, *a3*, *a4*, *side*, *angle*)
Define an aspheric surface

Parameters

- **r0** (*float, float*) – (x0,z0) position of apex
- **refractive_index** (*float, str*) – refractive index
- **cx** (*float*) – curvature
- **Qx** (*float*) – Conic constant
- **side** (*str*) – 'left', 'right'

Returns Bool array with positions inside the surface

Return type numpy.array

aspheric_lens(*r0*, *angle*, *refractive_index*, *cx*, *Qx*, *depth*, *size*, *a2=(0, 0)*, *a3=(0, 0)*, *a4=(0, 0)*, *a5=(0, 0)*, *a6=(0, 0)*, *a7=(0, 0)*)
Define an aspheric surface as defined in Gomez-Pedrero.

Parameters

- **r0** (*float, float*) – position x,z of lens
- **angle** (*float*) – rotation angle of lens + r0_rot
- **cx** (*float, float*) – curvature
- **Qx** (*float, float*) – Conic constant

- **depth** (*float, float*) – distance of the apex
- **size** (*float*) – diameter of lens

Returns Bool array with positions inside the surface

Return type numpy.array

wedge(*r0*, *length*, *refractive_index*, *angle_wedge*, *angle=0*, *rotation_point=None*)
Insert a wedge pointing towards the light beam

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **length** (*float*) – length of the long part (z direction)
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle_wedge** (*float*) –
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

prism(*r0*, *length*, *refractive_index*, *angle_prism*, *angle=0*, *rotation_point=None*)
Similar to wedge but the use is different. Also the angle is usually different. One of the sides is paralel to x=x0

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **length** (*float*) – length of the long part (z direction)
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle_prism** (*float*) –
- **angle** (*float*) – angle of rotation of the semi-plane, in radians
- **rotation_point** (*float, float*) –

biprism(*r0*, *length*, *height*, *refractive_index*, *angle=0*)
Fresnel biprism.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **length** (*float*) – length of the long part (z direction)
- **height** (*float*) – height of biprism
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians

ronchi_grating(*r0*, *period*, *fill_factor*, *length*, *height*, *Dx*, *refractive_index*, *heigth_substrate*, *refractive_index_substrate*, *angle*)
Insert a ronchi grating in background.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **period** (*float*) – period of the grating
- **fill_factor** (*float*) – [0,1], fill factor of the grating

- **length** (*float*) – length of the grating
- **height** (*float*) – height of the grating
- **Dx** (*float*) – displacement of grating with respect x=0
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **heigth_substrate** (*float*) – height of the substrate
- **refractive_index_substrate** (*float, str*) – refractive index of substrate, 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians

sine_grating(*period*, *heigth_sine*, *heigth_substrate*, *r0*, *length*, *Dx*, *refractive_index*, *angle=0*)
Insert a sine grating in background.

Parameters

- **period** (*float*) – period of the grating
- **fill_factor** (*float*) – [0,1], fill factor of the grating
- **heigth_sine** (*float*) – height of the grating
- **heigth_substrate** (*float*) – height of the substrate
- **r0** (*float, float*) – (x0,z0) Location of the rectangle, for example (0 * um, 20 * um)
- **length** (*float*) – length of the grating
- **Dx** (*float*) – displacement of grating with respect x=0
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians

probe(*r0*, *base*, *length*, *refractive_index*, *angle*)
Probe with a sinusoidal shape.

Parameters

- **r0** (*float, float*) – (x0,z0) position of the center of base, for example (0 * um, 20 * um)
- **base** (*float*) – base of the probe
- **length** (*float*) – length of the graprobeing
- **Dx** (*float*) – displacement of grating with respect x=0
- **refractive_index** (*float, str*) – refractive index , for example: 1.5 + 1.0j
- **angle** (*float*) – angle of rotation of the semi-plane, in radians

rough_sheet(*r0*, *size*, *t*, *s*, *refractive_index*, *angle*, *rotation_point=None*)
Sheet with one of the surface rough.

Parameters

- **r0** (*float, float*) – (x0,z0) Location of sphere, for example (0 * um, 20 * um)
- **size** (*float, float*) – (sizex, sizez) size of the sheet
- **s** (*float*) – std roughness
- **t** (*float*) – correlation length of roughness
- **refractive_index** (*float, str*) – refractive index

- **angle** (*float*) – angle
- **rotation_point** (*float, float*) – rotation point

Returns ipasa, indexes [iz,ix] of lens

Return type (numpy.array)

References

According to Ogilvy p.224

4.1.12 diffractio.scalar_sources_X module

This module generates Scalar_field_X class for defining sources. Its parent is Scalar_field_X.

The main atributes are:

- self.u - field
- self.x - x positions of the field
- self.wavelength - wavelength of the incident field. The field is monocromatic

The magnitude is related to microns: *mifcron = 1.*

Class for unidimensional scalar masks

Functions

- plane_wave
- gauss_beam
- spherical_wave
- plane_waves_dict
- plane_waves_several_inclined
- gauss_beams_several_parallel
- gauss_beams_several_inclined

Also

- Polychromatic and extendes sources are defined in scalar_fields_X.py for multiprocessing purposes.

class diffractio.scalar_sources_X.**Scalar_source_X**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None, wavelength: float = 0, n_background: float = 1.0, info: str = ''*)

Bases: *diffractio.scalar_fields_X.Scalar_field_X*

Class for unidimensional scalar sources.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **wavelength** (*float*) – wavelength of the incident field
- **n_background** (*float*) – refractive index of background

- **info** (*str*) – String with info about the simulation

self.**x**
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

self.**wavelength**
wavelength of the incident field.

Type float

self.**u**
equal size than x. complex field

Type numpy.array

self.**quality**
quality of RS algorithm

Type float

self.**info**
description of data

Type str

self.**type**
Class of the field

Type str

self.**date**
date when performed

Type str

plane_wave(*A=1*, *theta=0.0*, *z0=0.0*)
Plane wave.

Parameters

- **A** (*float*) – maximum amplitude
- **theta** (*float*) – angle in radians
- **z0** (*float*) – constant value for phase shift

gauss_beam(*x0: float*, *w0: float*, *z0: float*, *A: float = 1*, *theta: float = 0.0*)
Gauss Beam.

Parameters

- **x0** (*float*) – x position of center
- **w0** (*float*) – minimum beam width
- **z0** (*float*) – position of beam width
- **A** (*float*) – maximum amplitude
- **theta** (*float*) – angle in radians

spherical_wave(*A: float*, *x0: float*, *z0: float*, *normalize: bool = False*)
Spherical wave. self.u = amplitude * A * exp(-1.j * sign(z0) * k * Rz) / Rz

Parameters

- **A** (*float*) – maximum amplitude
- **x0** (*float*) – x position of source
- **z0** (*float*) – z position of source
- **mask** (*bool*) – If true, masks the spherical wave with radius
- **normalize** (*bool*) – If True, maximum of field is 1

plane_waves_dict(*params: dict*)

Several plane waves with parameters defined in dictionary

Parameters **params** – list with a dictionary: A (float): maximum amplitude theta (float): angle in radians z0 (float): constant value for phase shift

plane_waves_several_inclined(*A: float*, *num_beams: int*, *max_angle: float*)

Several paralel plane waves.

Parameters

- **A** (*float*) – maximum amplitude
- **num_beams** (*int*) – number of ints
- **max_angle** (*float*) – maximum angle for beams

gauss_beams_several_parallel(*A: float*, *num_beams: int*, *w0: float*, *z0: float*, *x_central: float*, *x_range: float*, *theta: float = 0.0*)

Several parallel gauss beams

Parameters

- **A** (*float*) – maximum amplitude
- **num_beams** (*int*) – number of gaussian beams (equidistintan)
- **w0** (*float*) – beam width of the bemas
- **z0** (*float*) – constant value for phase shift
- **x_central** (*float*) – central position of rays
- **x_range** (*float*) – range of rays
- **theta** (*float*) – angle of the parallel beams

gauss_beams_several_inclined(*A: float*, *num_beams: int*, *w0: float*, *x0: float*, *z0: float*, *max_angle: float*)

Several inclined gauss beams

Parameters

- **A** (*float*) – maximum amplitude
- **num_beams** (*int*) – number of gaussian beams (equidistintan)
- **w0** (*float*) – beam width of the bemas
- **x0** (*fl(float*) – maximum amplitude
- **num_beams** – number of ints
- **maoat)** – initial position of gauss beam at x
- **z0** (*float*) – constant value for phase shift
- **max_angle** (*float*) – maximum angle for beams

4.1.13 diffractio.scalar_sources_XY module

This module generates Scalar_source_XY class for defining sources. Its parent is Scalar_field_XY.

The main atributes are:

- self.x - x positions of the field
- self.y - y positions of the field
- self.u - field XY
- self.wavelength - wavelength of the incident field. The field is monocromatic

The magnitude is related to microns: *micron = 1.*

Class for unidimensional scalar masks

Functions

- plane_wave
- gauss_beam
- spherical_wave
- vortex_beam
- laguerre_beam
- hermite_gauss_beam
- zernike_beam
- bessel_beam
- plane_waves_dict
- plane_waves_several_inclined
- gauss_beams_several_parallel
- gauss_beams_several_inclined

Also

- laguerre_polynomial_nk
- fZernike
- delta_kronecker

class diffractio.scalar_sources_XY.**Scalar_source_XY**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *y: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: *diffractio.scalar_fields_XY.Scalar_field_XY*

Class for XY scalar sources.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2^n .
- **y** (*numpy.array*) – linear array wit equidistant positions for y values

- **wavelength** (*float*) – wavelength of the incident field
- **info** (*str*) – String with info about the simulation

`self.x`
linear array with equidistant positions. The number of data is preferibly 2^n .

Type numpy.array

`self.y`
linear array wit equidistant positions for y values

Type numpy.array

`self.wavelength`
wavelength of the incident field.

Type float

`self.u`
(x,z) complex field

Type numpy.array

`self.info`
String with info about the simulation

Type str

plane_wave(*A=1*, *theta=0.0*, *phi=0.0*, *z0=0.0*)

**Plane wave. self.u = A * exp(1.j * k *

(self.X * sin(theta) * cos(phi) + self.Y * sin(theta) * sin(phi) + z0 * cos(theta)))

According to https://en.wikipedia.org/wiki/Spherical_coordinate_system: physics (ISO 80000-2:2019 convention)

Parameters

- **A** (*float*) – maximum amplitude
- **theta** (*float*) – angle in radians
- **phi** (*float*) – angle in radians
- **z0** (*float*) – constant value for phase shift

gauss_beam(*r0*, *w0*, *z0*, *alpha=0.0*, *beta=0.0*, *A=1*, *theta=0.0*, *phi=0.0*)
Gauss Beam.

Parameters

- **r0** (*float, float*) – (x,y) position of center
- **w0** (*float, float*) – (wx,wy) minimum beam width
- **z0** (*float*) – (z0x, z0y) position of beam width for each axis (could be different)
- **alpha** (*float*) – rotation angle of the axis of the elliptical gaussian amplitude
- **beta** (*float*) – rotation angle of the axis of the main directions of the wavefront (it can be different from alpha)
- **A** (*float*) – maximum amplitude
- **theta** (*float*) – angle in radians (angle of k with respect to z))

References:

spherical_wave(*r0*, *z0*, *A=1*, *radius=0*, *normalize: bool = False*)
Spherical wave.

Parameters

- **A** (*float*) – maximum amplitude
- **r0** (*float, float*) – (x,y) position of source
- **z0** (*float) or (float, float*) – z0 or (zx0, zy0) position of source. When two parameters, the source is stigmatic
- **radius** (*float*) – radius for mask
- **normalize** (*bool*) – If True, maximum of field is 1

vortex_beam(*A*, *r0*, *w0*, *m*)
Vortex beam.

Parameters

- **A** (*float*) – Amplitude
- **r0** (*float, float*) – (x,y) position of source
- **w0** (*float*) – width of the vortex beam
- **m** (*int*) – order of the vortex beam

Example

vortex_beam(r0=(0 * um, 0 * um), w0=100 * um, m=1)

hermite_gauss_beam(*r0*, *A*, *w0*, *n*, *m*, *z*, *z0*)
Hermite Gauss beam.

Parameters

- **A** (*float*) – amplitude of the Hermite Gauss beam.
- **r0** (*float, float*) – (x,y) position of the beam center.
- **w0** (*float, float*) – Gaussian waist.
- **n** (*int*) – order in x.
- **m** (*int*) – order in y.
- **z** (*float*) – Propagation distance.
- **z0** (*float, float*) – Beam waist position at each dimension

Example

hermite_gauss_beam(A=1, r0=(0,0), w0=(100*um, 50*um), n=2, m=3, z=0)

laguerre_beam(*r0*, *A*, *w0*, *n*, *l*, *z*, *z0*)
Laguerre beam.

Parameters

- **A** (*float*) – amplitude of the Hermite Gauss beam.
- **r0** (*float, float*) – (x,y) position of the beam center.
- **w0** (*float*) – Gaussian waist.
- **n** (*int*) – radial order.
- **l** (*int*) – angular order.
- **z** (*float*) – Propagation distance.
- **z0** (*float*) – Beam waist position.

Example

laguerre_beam(A=1, r0=(0 * um, 0 * um), w0=1 * um, p=0, l=0, z=0)

zernike_beam(*A*, *r0*, *radius*, *n*, *m*, *c_nm*)
Zernike beam.

Parameters

- **A** (*float*) – amplitude of the Zernike beam beam
- **r0** (*float, float*) – (x,y) position of source
- **radius** (*float*) – width of the beam
- **n** (*list*) – list of integers with orders
- **m** (*list*) – list of integers with orders
- **c_nm** (*list*) – list of coefficients, in radians

Example

zernike_beam(A=1, r0=(0,0), radius=5 * mm, n=[1, 3, 3, 5, 5, 5], m=[1, 1, 3, 1, 3, 5], c_nm=[.25, 1, 1, 1, 1, 1])

bessel_beam(*A*, *r0*, *alpha*, *n*, *theta=0.0*, *phi=0.0*, *z0=0*)
Bessel beam produced by an axicon. Bessel-beams are generated using 2D axicons.

Parameters

- **A** (*float*) – amplitude of the Bessel beam
- **r0** (*float, float*) – (x,y) position of source
- **alpha** (*float*) – angle of the beam generator
- **n** (*int*) – order of the beam
- **theta** (*float*) – angle in radians
- **phi** (*float*) – angle in radians

- **z0** (*float*) – constant value for phase shift

References

J. Durnin, J. Miceli, and J. H. Eberly, Phys. Rev. Lett. 58, 1499 (1987). F. Courvoisier, et al. "Surface nanoprocessing with nondiffracting femtosecond Bessel beams" Optics Letters Vol. 34, No. 20 3163 (2009)

plane_waves_dict(*params*)
Several plane waves with parameters defined in dictionary

Parameters **params** – list with a dictionary: A (float): maximum amplitude theta (float): angle in radians phi (float): angle in radians z0 (float): constant value for phase shift

plane_waves_several_inclined(*A*, *num_beams*, *max_angle*, *z0=0*)
Several paralel plane waves

Parameters

- **A** (*float*) – maximum amplitude
- **num_beams** (*int, int*) – number of beams in the x and y directions
- **max_angle** (*float, float*) – maximum angle of the beams
- **z0** (*float*) – position of the beams

gauss_beams_several_parallel(*r0*, *A*, *num_beams*, *w0*, *z0*, *r_range*, *theta=0.0*, *phi=0.0*)
Several parallel gauss beams

Parameters

- **A** (*float*) – maximum amplitude
- **num_beams** (*int, int*) – number of gaussian beams (equidistintan) in x and y direction.
- **w0** (*float*) – beam width of the bemas
- **z0** (*float*) – constant value for phase shift
- **r0** (*float, float*) – central position of rays (x_c, y_c)
- **r_range** (*float, float*) – range of rays x, y
- **theta** (*float*) – angle
- **phi** (*float*) – angle

gauss_beams_several_inclined(*A*, *num_beams*, *w0*, *r0*, *z0*, *max_angle*)
Several inclined gauss beams

Parameters

- **A** (*float*) – maximum amplitude
- **num_beams** (*int, int*) – number of gaussian beams (equidistintan) in x and y direction.
- **w0** (*float*) – beam width
- **r0** (*float, float*) – central position of rays (x_c, y_c)
- **z0** (*float*) – constant value for phase shift
- **max_angle** (*float, float*) – maximum angles

4.1.14 diffractio.utils_common module

Common functions to classes

diffractio.utils_common.**computer_purcearameters**(*verbose: bool = False*) → list

Determine several computer parameters:

- number of processors
- available memory
- total memory
- max frequency

Parameters **verbose** (*bool*) – If True prints data

Returns number of processors info_memory (int) : Gbytes memory_available (int): % available memory freq_max (int): Maximum frequency (GHz)

Return type num_max_processors (int)

diffractio.utils_common.**clear_all**()
clear all variables

diffractio.utils_common.**several_propagations**(*source*, *masks*, *distances: list[float]*)
performs RS propagation through several masks

Parameters

- **source** (Scalar_source_XY) – illumination
- **masks** (*list*) – list with several (Scalar_masks_XY)
- **distances** (*list*) – list with seera distances

Returns u0 field at the last plane given by distances Scalar_field_XY: u1 field just at the plane of the last mask

Return type *Scalar_field_XY*

diffractio.utils_common.**get_date**()
gets current date and hour.

Returns date in text

Return type (str)

diffractio.utils_common.**save_data_common**(*cls*, *filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*) → str
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **=** (*add_name*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

diffractio.utils_common.**load_data_common**(*cls*, *filename: str*, *verbose: bool = False*)

Common load data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **cls** (*class*) – class X, XY, XZ, XYZ, etc..
- **filename** (*str*) – filename
- **verbose** (*bool*) – If True prints data

diffractio.utils_common.**print_axis_info**(*cls*, *axis*)
Prints info about axis

Parameters

- **cls** (*class*) – class of the modulus.
- **axis()** – axis x, y, z... etc.

diffractio.utils_common.**date_in_name**(*filename: str*) → str
introduces a date in the filename.

Parameters **filename** (*str*) – filename

Returns filename with current date

Return type (str)

4.1.15 diffractio.utils_drawing module

Functions for drawing

diffractio.utils_drawing.**view_image**(*filename: str*)
reproduces image

Parameters **filename** (*str*) – filename

diffractio.utils_drawing.**concatenate_drawings**(*kind1: str = 'png'*, *kind2: str = 'png'*, *nx: int = 5*, *ny: int = 3*, *geometria_x: int = 256*, *geometria_y: int = 256*, *raiz: str = 'fig4_nsensors_1'*, *filename: str = 'figura2.png'*, *directory: str = ''*)

diffractio.utils_drawing.**draw2D**(*image*, *x*, *y*, *xlabel: str = '$x (\\mu m)$'*, *ylabel: str = '$y (\\mu m)$'*, *title: str = ''*, *color: str = 'YlGnBu'*, *interpolation: str = 'nearest'*, *scale: str = 'scaled'*, *reduce_matrix: str = 'standard'*, *range_scale: str = 'um'*, *verbose: bool = False*) → tuple
makes a drawing of XY

Parameters

- **image** (*numpy.array*) – image to draw
- **x** (*numpy.array*) – positions x
- **y** (*numpy.array*) – positions y
- **xlabel** (*str*) – label for x
- **ytlabel** (*str*) – label for y
- **title** (*str*) – title

- **color** (*str*) – color
- **interpolation** (*str*) – 'bilinear', 'nearest'
- **scale** (*str*) – kind of axis (None, 'equal', 'scaled', etc.)
- **range_scale** (*str*) – 'um' o 'mm'
- **verbose** (*bool*) – if True prints information

Returns handle of figure IDax: handle of axis IDimage: handle of image

Return type id_fig

diffractio.utils_drawing.**draw_several_fields**(*fields*, *titles: str = ''*, *title: str = ''*, *figsize: Optional[tuple[float, float]] = None*, *kinds: str = ''*, *logarithm: list | float | bool = False*, *normalize: bool = False*)

Draws several fields in subplots

Parameters

- **fields** (*list*) – list with several scalar_fields_XY
- **titles** (*list*) – list with titles
- **title** (*str*) – suptitle
- **kinds** (*list*) – list with kinds of figures (amplitude', 'intensity', 'phase', 'real_field', 'contour')
- **logarithm** (*bool*) – If True, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1

diffractio.utils_drawing.**change_image_size**(*image_name: str*, *length: str = '800x600'*, *final_filename: str = 'prueba.png'*, *dpi: int = 100*)

change the size with imageMagick

Parameters

- **image_name** (*str*) – name of file
- **length** (*str*) – size of image
- **final_filename** (*str*) – final filename
- **dpi** (*int*) – dpi

Examples:

convert image_name -resize '1000' -units 300 final_filename.png

- anchura 1000 - mantiene forma

convert image_name -resize 'x200' final_filename.png

- height 200 - mantiene forma

convert image_name -resize '100x200>' final_filename.png

- mantiene forma, lo que sea mayor

convert image_name -resize '100x200<' final_filename.png

- mantiene forma, lo que sea menor

convert image_name -resize '@1000000' final_filename.png

- mantiene la forma, con 1Mpixel

convert image_name -resize '100x200!' final_filename.png

- obliga a tener el tamaño, no mantiene escala

diffractio.utils_drawing.**extract_image_from_video**(*nombre_video: Optional[str] = None*, *num_frame: str = '[0,]'*, *final_filename: str = 'prueba.png'*)

Extract images form a video using imageMagick.

convert 'animacion.avi[15,]' animacion_frame.png. Extracts frame 15 (ony 15) convert 'animacion.avi[15]' animacion_frame.png. Extracts the first 15 convert 'animacion.avi[5,10]' animacion_frame.png. Extracts frame 5 and 10

diffractio.utils_drawing.**normalize_draw**(*u*, *logarithm: numpy.float64 | bool = False*, *normalize: bool = False*, *cut_value: Optional[int] = None*)

Gets a field and changes its caracteristics for drawing

Parameters

- **u** (*field*) – field
- **logarithm** (*float*) – logarithm to image: np.log(logarithm*u + 1)
- **normalize** (*str or bool*) – False, 'mean', 'intensity'

diffractio.utils_drawing.**prepare_drawing**(*u*, *kind: str = 'intensity'*, *logarithm: numpy.float64 | bool = False*, *normalize: bool = False*)

It is necessary that figure is previously defined: plt.figure()

Parameters

- **field** (*u -*) –
- **'intensity'** (*kind -*) –
- **'amplitude'** –
- **'phase'** –
- **False** (*logarithm - True or*) –
- **normalize** – False, 'maximum', 'intensity', 'area'

Returns I_drawing for direct plotting

Return type returns (numpy.array)

diffractio.utils_drawing.**prepare_video**(*fps: int = 15*, *title: str = ''*, *artist: str = ''*, *comment: str = ''*)

diffractio.utils_drawing.**make_video_from_file**(*self*, *files: list*, *filename: str = ''*)

make a video from file

Parameters

- **files** (*list*) – files to add
- **filename** (*bool, optional*) – final filename. Defaults to "".

diffractio.utils_drawing.**reduce_matrix_size**(*reduce_matrix: str | list[int]*, *x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *image: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *verbose: bool = False*)

Reduces the size of matrix for drawing purposes. If the matrix is very big, the drawing process is slow.

Parameters

- **reduce_matrix** (*str or (int, int)*) – if str: 'standard', if (int, int) reduction_factor.
- **x** (*np.array*) – array with x.
- **y** (*np.array*) – array with y or z
- **image** (*np.array*) – image to reduce the size.
- **verbose** (*bool*) – if True, prints info

Returns reduced image

Return type (np.array)

4.1.16 diffractio.utils_math module

Common functions to classes

diffractio.utils_math.**nextpow2**(*x*)

Exponent of next higher power of 2. It returns the exponents for the smallest powers of two that satisfy 2^pA for each element in A. By convention, nextpow2(0) returns zero.

Parameters **x** (*float*) – value

Returns Exponent of next higher power of 2

Return type integer

diffractio.utils_math.**Bluestein_dft_x**(*x*, *f1*, *f2*, *fs*, *mout*)

Bluestein dft

Parameters

- **x** (*_type_*) – _description_
- **f1** (*_type_*) – _description_
- **f2** (*_type_*) – _description_
- **fs** (*_type_*) – _description_
- **mout** (*_type_*) – _description_

Reference:

- Hu, Y., Wang, Z., Wang, X., Ji, S., Zhang, C., Li, J., Zhu, W., Wu, D., Chu, J. (2020). "Efficient full-path optical calculation of scalar and vector diffraction using the Bluestein method". Light: Science and Applications, 9(1). https://doi.org/10.1038/s41377-020-00362-z

diffractio.utils_math.**Bluestein_dft_xy**(*x*, *f1*, *f2*, *fs*, *mout*)

Bluestein dft

Parameters

- **x** (*_type_*) – _description_
- **f1** (*_type_*) – _description_
- **f2** (*_type_*) – _description_
- **fs** (*_type_*) – _description_
- **mout** (*_type_*) – _description_

diffractio.utils_math.**find_local_extrema**(*kind: str*, *y*, *x*, *pixels_interpolation=0*, *pixels_filter=0*)

Determine local minima in a numpy array.

Parameters

- **kind** (*str*) – 'maxima', 'minima'
- **y** (*numpy.ndarray*) – variable with local minima.
- **x** (*numpy.ndarray*) – x position

Returns i positions of local minima.

Return type (numpy.ndarray)

diffractio.utils_math.**reduce_to_1**(*class_diffractio*)
All the values greater than 1 pass to 1. This is used for Scalar_masks when we add two masks. :param class: Scalar_field_X, XY ,.... :type class: class

diffractio.utils_math.**distance**(*x1*, *x2*)
Compute distance between two vectors.

Parameters

- **x1** (*numpy.array*) – vector 1
- **x2** (*numpy.array*) – vector 2

Returns distance between vectors.

Return type (float)

diffractio.utils_math.**nearest**(*vector*, *number*)
Computes the nearest element in vector to number.

Parameters

- **vector** (*numpy.array*) – array with numbers
- **number** (*float*) – number to determine position

Returns index - index of vector which is closest to number. (float): value - value of vector[index]. (float): distance - difference between number and chosen element.

Return type (int)

diffractio.utils_math.**nearest2**(*vector*, *numbers*)
Computes the nearest element in vector to numbers.

Parameters

- **vector** (*numpy.array*) – array with numbers
- **number** (*numpy.array*) – numbers to determine position

Returns index - indexes of vector which is closest to number. (numpy.array): value - values of vector[indexes]. (numpy.array): distance - difference between numbers and chosen elements.

Return type (numpy.array)

diffractio.utils_math.**find_extrema**(*array2D*, *x*, *y*, *kind='max'*, *verbose: bool = False*)
In a 2D-array, formed by vectors x, and y, the maxima or minima are found

Parameters

- **array2D** (*np. array 2D*) – 2D array with variable
- **x** (*np.array 1D*) – 1D array with x axis
- **y** (*np.array 1D*) – 1D array with y axis
- **kind** (*str*) – 'min' or 'max': detects minima or maxima

- **verbose** (*bool*) – If True prints data.

Returns indexes of the position xy_ext (float, float): position of maximum extrema (float): value of maximum

Return type indexes (int,int)

diffractio.utils_math.**ndgrid_deprecated**(**args*, ***kwargs*)
n-dimensional gridding like Matlab's NDGRID.

Parameters

- **sequences** (*The input *args are an arbitrary number of numerical*) –
- **lists** (*e.g.*) –
- **arrays** –
- **tuples.** (*or*) –
- **argument** (*The i-th dimension of the i-th output*) –
- **argument.** (*has copies of the i-th input*) –

Example

```
>>> x, y, z = [0, 1], [2, 3, 4], [5, 6, 7, 8]
```

```
>>> X, Y, Z = ndgrid(x, y, z)
    # unpacking the returned ndarray into X, Y, Z
```

Each of X, Y, Z has shape [len(v) for v in x, y, z].

```
>>> X.shape == Y.shape == Z.shape == (2, 3, 4)
    True
```

```
>>> X
    array([[[0, 0, 0, 0],
                    [0, 0, 0, 0],
                    [0, 0, 0, 0]],
            [[1, 1, 1, 1],
                    [1, 1, 1, 1],
                    [1, 1, 1, 1]]])
>>> Y
    array([[[2, 2, 2, 2],
                    [3, 3, 3, 3],
                    [4, 4, 4, 4]],
            [[2, 2, 2, 2],
                    [3, 3, 3, 3],
                    [4, 4, 4, 4]]])
>>> Z
    array([[[5, 6, 7, 8],
                    [5, 6, 7, 8],
                    [5, 6, 7, 8]],
            [[5, 6, 7, 8],
                    [5, 6, 7, 8],
                    [5, 6, 7, 8]]])
```

With an unpacked argument list:

```
>>> V = [[0, 1], [2, 3, 4]]
```

```
>>> ndgrid_deprecated(*V) # an array of two arrays with shape (2, 3)
    array([[[0, 0, 0],
        [1, 1, 1]],
        [[2, 3, 4],
        [2, 3, 4]]])
```

For input vectors of different data kinds, same_dtype=False makes ndgrid_deprecated() return a list of arrays with the respective dtype. >>> ndgrid_deprecated([0, 1], [1.0, 1.1, 1.2], same_dtype=False) [array([[0, 0, 0], [1, 1, 1]]), array([[1. , 1.1, 1.2], [1. , 1.1, 1.2]])]

Default is to return a single array.

```
>>> ndgrid_deprecated([0, 1], [1.0, 1.1, 1.2])
    array([[[ 0. ,  0. ,  0. ], [ 1. ,  1. ,  1. ]],
        [[ 1. ,  1.1,  1.2], [ 1. ,  1.1,  1.2]]])
```

diffractio.utils_math.**get_amplitude**(*u*, *sign: bool = False*)
Gets the amplitude of the field.

Parameters

- **u** (*numpy.array*) – Field.
- **sign** (*bool*) – If True, sign is kept, else, it is removed

Returns numpy.array

Return type (numpy.array)

diffractio.utils_math.**get_phase**(*u*)
Gets the phase of the field.

Parameters **u** (*numpy.array*) – Field.

Returns numpy.array

Return type (numpy.array)

diffractio.utils_math.**amplitude2phase**(*u*)
Passes the amplitude of a complex field to phase. Previous phase is removed. $u = Ae^{i\phi} -> e^(iabs(A))$

Parameters **u** (*numpy.array, dtype=complex*) – complex field

Returns only-phase complex vector.

Return type (numpy.array)

diffractio.utils_math.**phase2amplitude**(*u*)
Passes the phase of a complex field to amplitude.

Parameters **u** (*numpy.array, dtype=complex*) – complex field

Returns amplitude without phase complex vector.

Return type (numpy.array)

diffractio.utils_math.**normalize**(*v*, *order=2*)
Normalize vectors with different L norm (standard is 2).

Parameters

- **v** (*numpy.array*) – vector to normalize
- **order** (*int*) – order for norm

Returns normalized vector.

Return type (numpy.array)

diffractio.utils_math.**binarize**(*vector*, *min_value=0*, *max_value=1*)
Binarizes vector between two levels, min and max. The central value is (min_value+max_value)/2

Parameters

- **vector** – (numpy.array) array with values to binarize
- **min_value** (*float*) – minimum value for binarization
- **max_value** (*float*) – maximum value for binarization

Returns binarized vector.

Return type (numpy.array)

diffractio.utils_math.**discretize**(*u*, *kind: str = 'amplitude'*, *num_levels: int = 2*, *factor=1*, *phase0=0*, *new_field: bool = True*, *matrix: bool = False*)
Discretize in a number of levels equal to num_levels.

Parameters

- **kind** (*str*) – "amplitude" o "phase"
- **num_levels** (*int*) – number of levels for the discretization
- **factor** (*float*) – from the level, how area is binarized. if 1 everything is binarized,
- **phase0** (*float*) –

  –
- **new_field** (*bool*) – if True returns new field
- **matrix** (*bool*) – if True it returs a matrix

Returns if new_field is True returns scalar_fields_XY

Return type scalar_fields_XY

diffractio.utils_math.**delta_kronecker**(*a*, *b*)
Delta kronecker

Parameters

- **a** (*np.float*) – number
- **b** (*np.float*) – number

Returns 1 if a==b and 0 if a<>b

diffractio.utils_math.**vector_product**(*A*, *B*)
Returns the vector product between two vectors.

Parameters

- **A** (*numpy.array*) – 3x1 vector array.
- **B** (*numpy.array*) – 3x1 vector array.

Returns 3x1 vector product array

Return type (numpy.array)

diffractio.utils_math.**dot_product**(*A*, *B*)
Returns the dot product between two vectors.

Parameters

- **A** (*numpy.array*) – 3x1 vector array.
- **B** (*numpy.array*) – 3x1 vector array.

Returns 3x1 dot product

Return type (complex)

diffractio.utils_math.**divergence**(*E*, *r*)
Returns the divergence of a field a given point (x0,y0,z0).

Parameters

- **E** (*numpy.array*) – complex field
- **r** (*numpy.array*) – 3x1 array with position r=(x,y,z).

Returns Divergence of the field at (x0,y0,z0)

Return type (float)

diffractio.utils_math.**curl**(*E*, *r*)
Returns the Curl of a field a given point (x0,y0,z0)

Parameters

- **E** (*numpy.array*) – complex field
- **r** (*numpy.array*) – 3x1 array with position r=(x,y,z).

Returns Curl of the field at (x0,y0,z0)

Return type (numpy.array)

diffractio.utils_math.**get_edges**(*x*, *f*, *kind_transition='amplitude'*, *min_step=0*, *verbose: bool = False*, *filename: str = ''*)
We have a binary mask and we obtain locations of edges. valid for litography engraving of gratings

Parameters

- **x** (*float*) – position x
- **f** (*numpy.array*) – Field. If real function, use 'amplitude' in kind_transition.
- **kind_transition** (*str*) – 'amplitude' 'phase' of the field where to get the transitions.
- **min_step** (*float*) – minimum step for consider a transition
- **verbose** (*bool*) – If True prints information about the process.
- **filename** (*str*) – If not '', saves the data on files. filename is the file name.

Returns array with +1, -1 with rasing or falling edges pos_transition (numpy.array): positions x of transitions raising (numpy.array): positions of raising falling (numpy.array): positions of falling

Return type type_transition (numpy.array)

diffractio.utils_math.**cut_function**(*x*, *y*, *length*, *x_center=''*)
takes values of function inside (x_center+length/2: x_center+length/2)

diffractio.utils_math.**fft_convolution2d**(*x*, *y*)
2D convolution, using FFT

Parameters

- **x** (*numpy.array*) – array 1 to convolve
- **y** (*numpy.array*) – array 2 to convolve

Returns convolved function

diffractio.utils_math.**fft_convolution1d**(*x*, *y*)
1D convolution, using FFT

Parameters

- **x** (*numpy.array*) – array 1 to convolve
- **y** (*numpy.array*) – array 2 to convolve

Returns convolved function

diffractio.utils_math.**fft_filter**(*x*, *y*, *normalize: bool = False*)
1D convolution, using FFT

Parameters

- **x** (*numpy.array*) – array 1 to convolve
- **y** (*numpy.array*) – array 2 to convolve

Returns convolved function

diffractio.utils_math.**fft_correlation1d**(*x*, *y*)
1D correlation, using FFT (fftconvolve)

Parameters

- **x** (*numpy.array*) – array 1 to convolve
- **y** (*numpy.array*) – array 2 to convolve

Returns correlation function

Return type numpy.array

diffractio.utils_math.**fft_correlation2d**(*x*, *y*)

Parameters

- **x** (*numpy.array*) – array 1 to convolve
- **y** (*numpy.array*) – array 2 to convolve

Returns 2d correlation function

Return type numpy.array

diffractio.utils_math.**rotate_image**(*x*, *z*, *img*, *angle*, *pivot_point*)
similar to rotate image, but not from the center but from the given

Parameters

- **img** (*np.array*) – image to rotate
- **angle** (*float*) – angle to rotate
- **pivot_point** (*float, float*) – (z,x) position for rotation

Returns rotated image

Reference: https://stackoverflow.com/questions/25458442/rotate-a-2d-image-around-specified-origin-in-python
point

diffractio.utils_math.**cart2pol**(*x*, *y*)
cartesian to polar coordinate transformation.

Parameters

- **x** (*np.array*) – x coordinate
- **y** (*np.aray*) – y coordinate

Returns rho numpy.array: phi

Return type numpy.array

diffractio.utils_math.**pol2cart**(*rho*, *phi*)
polar to cartesian coordinate transformation

Parameters

- **rho** (*np.aray*) – rho coordinate
- **rho** – rho coordinate

Returns x numpy.array: y

Return type numpy.array

diffractio.utils_math.**fZernike**(*X*, *Y*, *n*, *m*, *radius=5000.0*)
Zernike function for aberration computation.

Note: k>=l

if k is even then l is even. if k is odd then l is odd.

The first polinomial is the real part ant the second de imaginary part.

- n m aberración
- 0 0 piston
- 1 -1 vertical tilt
- 1 1 horizontal tilt
- 2 -2 astigmatismo oblicuo
- 2 0 desenfoque miopía si c>0 o desenfoque hipermetropía si c<0
- 2 2 astigmatismo anormal si c>0 o astigmatismo normal si c<0
- 3 -3 trebol oblicuo
- 3 -1 coma vertical, c>0 empinamiento superior, c<0 emp. inferior
- 3 1 como horizontal
- 3 3 trebol horizontal
- 4 -4 trebol de cuatro hojas oblicuo
- 4 -2 astigmatismo secundario oblicuo
- 4 0 esférica c>0 periferia más miópica que centro, c<0 periferia más hipertrópica que el centro
- 4 2 astigmatismo secundario a favor o en contra de la regla
- 4 4 trebol de cuatro hojas horizontal

Reference:

R. Navarro, J. Arines, R. Rivera "Direct and inverse discrete Zernike transform" Opt. Express 17(26) 24269

diffractio.utils_math.**laguerre_polynomial_nk**(*x*, *n=4*, *k=5*)

Auxiliar laguerre polinomial of orders n and k function y = LaguerreGen(varargin) LaguerreGen calculates the utilsized Laguerre polynomial L{n, alpha} This function computes the utilsized Laguerre polynomial L{n,alpha}. If no alpha is supplied, alpha is set to zero and this function calculates the "normal" Laguerre polynomial.

Parameters: - n = nonnegative integer as degree level - alpha >= -1 real number (input is optional)

The output is formated as a polynomial vector of degree (n+1) corresponding to MatLab norms (that is the highest coefficient is the first element).

Example: - polyval(LaguerreGen(n, alpha), x) evaluates L{n, alpha}(x) - roots(LaguerreGen(n, alpha)) calculates roots of L{n, alpha}

Calculation is done recursively using matrix operations for very fast execution time.

Author: Matthias.Trampisch@rub.de Date: 16.08.2007 Version 1.2

References

Szeg: "Orthogonal Polynomials" 1958, formula (5.1.10)

diffractio.utils_math.**get_k**(*x*, *flavour='-'*)

provides k vector from x vector. With flavour set to "-", the axis will be inverse-fftshifted, thus its DC part being the first index.

Parameters

- **x** (*np.array*) – x array
- **flavour** (*str*) – '-' (for ifftshifted axis)

Returns k vector

Return type kx (np.array)

Fixed by Bob-Swinkels

diffractio.utils_math.**get_k_deprecated**(*x*, *flavour='-'*)

provides k vector from x vector. Two flavours are provided (ordered + or disordered -)

Parameters

- **x** (*np.array*) – x array
- **flavour** (*str*) – '+' or '-'

Returns k vector

Return type kx (np.array)

diffractio.utils_math.**filter_edge_1D**(*x*, *size=1.1*, *exponent=32*)

function 1 at center and reduced at borders. For propagation algorithms

Parameters

- **x** (*np.array*) – position
- **size** (*float*) – related to relative position of x
- **exponent** (*integer*) – related to shape of edges

Returns function for filtering

Return type np.array

diffractio.utils_math.**filter_edge_2D**(*x*, *y*, *size=1.1*, *exponent=32*)
function 1 at center and reduced at borders. For propagation algorithms

Parameters

- **x** (*np.array*) – x position
- **y** (*np.array*) – y position
- **size** (*float*) – related to relative position of x and y
- **exponent** (*integer*) – related to shape of edges

Returns function for filtering

Return type np.array

4.1.17 diffractio.utils_multiprocessing module

diffractio.utils_multiprocessing.**separate_from_iterable**(*iterable*, *shape=0*)
This function does somehow the opposite of the previous one, it takes an iterable made of lists and separates each one in a different variable, reshaped with the desired shape

class diffractio.utils_multiprocessing.**auxiliar_multiprocessing**
Bases: object

execute_multiprocessing(*function*, *var_iterable*, *dict_constants={}*, *Ncores=8*)

method_single_proc(*elem_iterable*)

diffractio.utils_multiprocessing.**execute_multiprocessing**(*__function_process__*, *dict_Parameters*, *num_processors*, *verbose: bool = False*)
Executes multiprocessing reading a dictionary.

Parameters

- **process** (*__function_process__ function tu*) –
- **dictionary** (*it only accepts a*) –
- **dict_Parameters** –
- **Parameters** (*dictionary / array with*) –
- **num_processors** –
- **used** (*if 1 no multiprocessing is*) –
- **verbose** –
- **time** (*prints processing*) –

Returns reults of multiprocessing processing time

Return type data

Examples

def __function_process__(xd): x = xd[‘x’] y = xd[‘y’] # grt = copy.deepcopy(grating) suma = x + y return dict(sumas=suma, ij=xd[‘ij’])

def creation_dictionary_multiprocessing(): # create Parameters: for multiprocessing t1 = time.time() X = np.linspace(1, 2, 10) Y = np.linspace(1, 2, 1000) dict_Parameters = [] ij = 0 for i, x in enumerate(X):

for j, y in enumerate(Y): dict_Parameters.append(dict(x=x, y=y, ij=[ij])) ij += 1

t2 = time.time() print(“time creation dictionary = {}”.format(t2 - t1)) return dict_Parameters

4.1.18 diffractio.utils_optics module

General purpose optics functions

diffractio.utils_optics.**roughness_1D**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *t: numpy.float64*, *s: numpy.float64*, *kind='normal'*)

Rough surface, 1D

Parameters

- **x** (*numpy.array*) – array with x positions
- **t** (*float*) – correlation lens
- **s** (*float*) – std of roughness
- **kind** (*str*) – ‘normal’, ‘uniform’

Returns (numpy.array) Topography of roughnness in microns.

References

JA Oglivy “Theory of wave scattering from random surfaces” Adam Hilger p.224.

diffractio.utils_optics.**roughness_2D**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *y: list[numpy.float64, numpy.float64]*, *t: numpy.float64*, *s: numpy.float64*)

Rough surface, 2D

Parameters

- **x** (*numpy.array*) – x positions
- **y** (*numpy.array*) – y positions
- **t** (*float, float*) – (tx, ty), correlation length of roughness
- **s** (*float*) – std of heights

Returns (numpy.array) Topography of roughnness in microns.

Example

roughness(t=(50 * um, 25 * um), s=1 * um)

References

JA Oglivy "Theory of wave scattering from random surfaces" Adam Hilger p.224.

diffractio.utils_optics.**beam_width_1D**(*u: numpy.ndarray[Any, numpy.dtype[numpy.complex128]]*, *x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *remove_background: bool = False*)

One dimensional beam width, according to D4 or second moment width.

Parameters

- **u** (*np.array*) – field (not intensity).
- **x** (*np.array*) – x

Returns width (float): x_mean

Return type (float)

References

https://en.wikipedia.org/wiki/Beam_diameter

diffractio.utils_optics.**width_percentage**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *percentage: numpy.float64 = 0.5*, *verbose: bool = False*)

beam width (2*sigma) given at a certain height from maximum

Parameters

- **x** (*np.array*) – x
- **y** (*np.array*) – y
- **percentage** (*float*) – percentage of height. For example: 0.5

Returns width, width of at given % (list): x_list: (x[i_left], x[i_max], x[i_right]) (list): x_list: (i_left, i_max, i_right)

Return type (float)

Notes

y=np.exp(-x**2/(s**2)) percentage=1/e -> width = 2*s y=np.exp(-x**2/(s**2)) percentage=1/e**4 -> width = 4*s y=np.exp(-x**2/(2*s**2)) percentage=1/e**2 = -> width = 4*s

diffractio.utils_optics.**beam_width_2D**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *intensity: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *remove_background: bool = False*, *has_draw: bool = False*)

2D beam width, ISO11146 width

Parameters

- **x** (*np.array*) – 1d x

- **y** (*np.array*) – 1d y
- **intensity** (*np.array*) – intensity

Returns dx width x (float): dy width y (float): principal_axis, angle (str): (x_mean, y_mean, x2_mean, y2_mean, xy_mean), Moments

Return type (float)

References

- https://en.wikipedia.org/wiki/Beam_diameter
- http://www.auniontech.com/ueditor/file/20170921/1505982360689799.pdf

diffractio.utils_optics.**refractive_index**(*filename: str*, *wavelength: numpy.float64*, *raw: bool = False*, *has_draw: bool = <class 'bool'>*)

gets refraction index from https://refractiveindex.info .

- Files has to be converted to xlsx format.
- n and k checks has to be activated.

Parameters

- **filename** (*str*) – xlsx file
- **wavelength** (*float*) – wavelength in microns, example, 0.6328.
- **raw** (*bool*) – if True returns all the data in file.
- **has_draw** (*bool*) – draw the data from the file

Returns n, k from each wavelength if raw is True (np.array, np.array): n,k for wavelengths in file

Return type if raw is False (float, float)

diffractio.utils_optics.**FWHM1D**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *intensity: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *percentage: numpy.float64 = 0.5*, *remove_background=None*, *has_draw: bool = False*)

FWHM1D #TODO

remove_background = 'min', 'mean', None

diffractio.utils_optics.**FWHM2D**(*x: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *y: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *intensity: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *percentage: numpy.float64 = 0.5*, *remove_background: bool = False*, *has_draw: bool = False*, *xlim: Optional[list[numpy.float64]] = None*)

TODO: perform profiles at several angles and fit to a ellipse. Get dx, dy, angle, x_center, y_center

diffractio.utils_optics.**DOF**(*z: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *widths: numpy.ndarray[Any, numpy.dtype[numpy.float64]]*, *w_factor: numpy.float64 = 1.4142135623730951*, *w_fixed: numpy.float64 = 0*, *has_draw: bool = False*, *verbose: bool = False*)

Determines Depth-of_focus (DOF) in terms of the width at different distances

Parameters

- **z** (*np.array*) – z positions
- **widths** (*np.array*) – width at positions z

- **w_factor** (*float*) – range to determine z where w = w_factor * w0, being w0 the beam waist
- **w_fixed** (*float*) – If it is not 0, then it is used as w_min
- **has_draw** (*bool*) – if True draws the depth of focus
- **verbose** (*bool*) – if True, prints data

References

B. E. A. Saleh and M. C. Teich, Fundamentals of photonics. john Wiley & sons, 2nd ed. 2007. Eqs (3.1-18) (3.1-22) page 79

Returns Depth of focus (float): beam waist (float, float, float): postions (z_min, z_0, z_max) of the depth of focus

Return type (float)

diffractio.utils_optics.**detect_intensity_range**(*x*, *intensity*, *percentage=0.95*, *has_draw: bool = True*, *logarithm=True*)

Determines positions x_min, x_max where intensity of the beam is percentage

Parameters

- **x** (*np.array*) – x positions
- **intensity** (*np.array*) – Intensity of the 1D beam
- **percentage** (*float*) – value 0-1 representing the percentage of intensity between area
- **has_draw** (*bool*) – if True draws the field an the range
- **logarithm** (*bool*) – when has_draw, draws logarithm or normal intensity

Returns positions (x_min, right) where intensity beam is enclosed at %.

Return type (float, float)

diffractio.utils_optics.**MTF_ideal**(*frequencies*, *wavelength*, *diameter*, *focal*, *kind: str*, *verbose: bool = False*, *has_draw: bool = False*)

Determines the ideal MTF of a lens.

References

https://www.edmundoptics.com/resources/application-notes/optics/introduction-to-modulation-transfer-function/

https://www.optikos.com/wp-content/uploads/2015/10/How-to-Measure-MTF-and-other-Properties-of-Lenses.pdf

Parameters

- **frequencies** (*numpy.array*) – array with frequencies in *lines/mm*
- **wavelength** (*float*) – wavelength of incoming light beam
- **diameter** (*float*) – diameter of lens
- **focal** (*float*) – focal distance of lens
- **kind** (*float*) – '1D', '2D'
- **verbose** (*bool*) – if True displays limit frequency of the lens

Returns Normalized MTF of ideal lens (float) frequency_max: maximum frequency of the lens

Return type (numpy.array) MTF

diffractio.utils_optics.**lines_mm_2_cycles_degree**(*lines_mm*, *focal*)
Pasa líneas por mm a cyclos/grado, más tipico de ojo Infor saca estos cálculos 181022 :param lines_mm: lines_per_mm :type lines_mm: numpy.array or float :param focal: focal of lens :type focal: float

diffractio.utils_optics.**MTF_parameters**(*MTF*, *MTF_ideal*, *lines_mm=50*, *verbose: bool = False*)
MTF parameters: strehl_ratio, mtf_50_ratio, freq_50_real, freq_50_ideal

References

https://www.edmundoptics.com/resources/application-notes/optics/introduction-to-modulation-transfer-function/
strehl_ratio

frequencies of mtf ar given since both MTF can have different steps MTF:

Parameters

- **MTF** (*N,2 numpy.array*) – (freq, MTF) of system in lines/mm
- **MTF_ideal** (*M,2 numpy.array*) – (freq, MTF) of ideal system in lines/mm
- **lines_mm** (*float*) – (0-1) Height of MTF for ratios

Returns strehl_ratio (float): MTF_ratio at freq_obs height (float): frequency at freq_obs of MTF (float): frequency at freq_obs of MTF_ideal

Return type (float)

diffractio.utils_optics.**gauss_spectrum**(*wavelengths*, *w_central*, *Dw*, *normalize=True*)
returns weigths for a gaussian spectrum :param wavelengths: array with wavelengths :param w_central: central wavelength :param Dw: width of the spectrum :param normalize: if True sum of weights is 1

diffractio.utils_optics.**lorentz_spectrum**(*wavelengths*, *w_central*, *Dw*, *normalize=True*)
returns weigths for a gaussian spectrum :param wavelengths: array with wavelengths :param w_central: central wavelength :param Dw: width of the spectrum :param normalize: if True sum of weights is 1

diffractio.utils_optics.**uniform_spectrum**(*wavelengths*, *normalize=True*)
returns weigths for a gaussian spectrum :param wavelengths: array with wavelengths :param w_central: central wavelength :param Dw: width of the spectrum :param normalize: if True sum of weights is 1

diffractio.utils_optics.**normalize_field**(*self*, *new_field: bool = False*)
Normalize the field to maximum intensity.

Parameters

- **(Scalar_field)** – field_*
- **new_field** (*bool*) – If True returns a field, else returns a matrix

Returns normalized field.

Return type (np.array)

diffractio.utils_optics.**field_parameters**(*u*, *has_amplitude_sign=False*)
Determines main parameters of field: amplitude intensity phase. All this parameters have the same dimension as u.

Parameters

- **u** (*numpy.array*) – optical field (comes usually form field.u)

- **has_amplitude_sign** (*bool*) – If True - amplitude = np.sign(u) * np.abs(u), Else: amplitude = np.abs(u)

Returns np.abs(u) intensity (numpy.array): np.abs(u)**2 phase (numpy.array): np.angle(u)

Return type amplitude (numpy.array)

diffractio.utils_optics.**convert_phase2heigths**(*phase*, *wavelength*, *n*, *n_background*)

We have a phase and it is converted to a depth. It is useful to convert Scalar_mask_X to Scalar_mask_XZ

phase(x,z)= k (n-n_0) h(x,z).

Parameters

- **phase** (*np.array*) – Phases
- **wavelength** (*float*) – wavelength
- **n** (*float or complex*) – refraction index of material
- **n_background** (*float*) – refraction index of background

Returns depths related to phases

Return type (np.array)

diffractio.utils_optics.**convert_amplitude2heigths**(*amplitude*, *wavelength*, *kappa*, *n_background*)

We have a phase and it is converted to a depth. It is useful to convert Scalar_mask_X to Scalar_mask_XZ.

Parameters

- **phase** (*np.array*) – Phases
- **wavelength** (*float*) – wavelength
- **kappa** (*float*) – refraction index of material.
- **n_background** (*float*) – refraction index of background

Returns depths related to amplitudes

Return type (np.array)

diffractio.utils_optics.**fresnel_equations_kx**(*kx*, *wavelength*, *n1*, *n2*, *outputs=[True, True, True, True]*, *has_draw: bool = True*, *kind='amplitude_phase'*)

Fresnel_equations where input are kx part of wavevector.

Parameters

- **kx** (*np.array*) – kx
- **wavelength** (*float*) – wavelength
- **n1** (*float*) – refraction index of first materia
- **n2** (*float*) – refraction index of second materia
- **outputs** (*bool,bool,bool,bool*) – Selects the outputs to compute
- **has_draw** (*bool, optional*) – if True, it draw. Defaults to False.
- **kind** (*str*) – It draw 'amplitude_phase' or 'real_imag'

Returns t_TM, t_TE, r_TM, r_TE (TM is parallel and TE is perpendicular)

Return type _type_

diffractio.utils_optics.**transmitances_reflectances_kx**(*kx*, *wavelength*, *n1*, *n2*, *outputs=[True, True, True, True]*, *has_draw: bool = False*)

Transmitances and reflectances, where input are kx part of wavevector.

Parameters

- **kx** (*np.array*) – kx
- **wavelength** (*float*) – wavelength
- **n1** (*float*) – refraction index of first materia
- **n2** (*float*) – refraction index of second materia
- **has_draw** (*bool, optional*) – if True, it draw. Defaults to False.
- **outputs** (*bool,bool,bool,bool*) – Selects the outputs to compute

Returns T_TM, T_TE, R_TM, R_TE (TM is parallel and TE is perpendicular)

Return type _type_

diffractio.utils_optics.**fresnel_equations**(*theta*, *wavelength*, *n1*, *n2*, *outputs=[True, True, True, True]*, *has_draw: bool = False*, *kind='amplitude_phase'*)

Fresnel equations and reflectances, where input are angles of incidence.

Parameters

- **kx** (*np.array*) – kx
- **wavelength** (*float*) – wavelength
- **n1** (*float*) – refraction index of first materia
- **n2** (*float*) – refraction index of second materia
- **kind** (*str*) – It draw 'amplitude_phase' or 'real_imag'
- **has_draw** (*bool, optional*) – if True, it draw. Defaults to False.
- **kind** – It draw 'amplitude_phase' or 'real_imag'

Returns T_TM, T_TE, R_TM, R_TE (TM is parallel and TE is perpendicular)

Return type _type_

diffractio.utils_optics.**transmitances_reflectances**(*theta*, *wavelength*, *n1*, *n2*, *outputs=[True, True, True, True]*, *has_draw: bool = False*)

Transmitances and reflectances, where input are angles of incidence.

Parameters

- **kx** (*np.array*) – kx
- **wavelength** (*float*) – wavelength
- **n1** (*float*) – refraction index of first materia
- **n2** (*float*) – refraction index of second materia
- **has_draw** (*bool, optional*) – if True, it draw. Defaults to False.
- **outputs** (*bool,bool,bool,bool*) – Selects the outputs to compute

Returns T_TM, T_TE, R_TM, R_TE (TM is parallel and TE is perpendicular)

Return type _type_

4.1.19 diffractio.utils_slicer_deprecated module

4.1.20 diffractio.utils_tests module

4.1.21 diffractio.vector_fields_X module

This module generates Vector_field_X class. It is required also for generating masks and fields. The main atributes are:

- self.x - x positions of the field
- self.Ex - x component of electric field
- self.Ey - y component of electric field
- self.Ez - z component of electric field
- self.wavelength - wavelength of the incident field. The field is monocromatic
- self.info (str): description of data
- self.type (str): Class of the field
- self.date (str): date when performed

The magnitude is related to microns: *micron = 1.*

Class for X vector fields

Definition of a scalar field

- add, substract fields
- save, load data, clean, get, normalize
- cut_resample

Vector parameters

- polarization_states

Drawing functions

- draw: intensity, intensities, phases, fields, stokes, param_ellipse, ellipses

class diffractio.vector_fields_X.**Vector_field_X**(*x: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *info: str = ''*)

Bases: `object`

Class for vectorial fields.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2**n.
- **wavelength** (*float*) – wavelength of the incident field
- **info** (*str*) – String with info about the simulation

`self.x`
linear array with equidistant positions. The number of data is preferibly 2**n.

Type numpy.array

self.**wavelength**
wavelength of the incident field.

Type float

self.**Ex**
Electric_x field

Type numpy.array

self.**Ey**
Electric_y field

Type numpy.array

self.**Ez**
Electric_z field

Type numpy.array

save_data(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **add_name=** (*str*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

load_data(*filename: str*, *verbose: bool = False*)

Load data from a file to a Vector_field_X. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **verbose** (*bool*) – shows data process by screen

clear_field()
Removes the fields Ex, Ey, Ez

duplicate(*clear: bool = False*)
Duplicates the instance

get(*kind: str = 'fields'*, *is_matrix=True*)
Takes the vector field and divide in Scalar_field_X.

Parameters **kind** (*str*) – 'fields', 'intensity', 'intensities', 'phases', 'stokes', 'params_ellipse'

Returns (Ex, Ey, Ez),

Return type *Vector_field_X*

apply_mask(*u*)
Multiply field by binary scalar mask: self.Ex = self.Ex * u.u

Parameters **u** (Scalar_mask_X) – mask

intensity()
"Returns intensity.

polarization_states(*matrix: bool = False*)
returns the Stokes parameters

Parameters **Matrix** (*bool*) – if True returns Matrix, else Scalar_field_X

Returns S0,S1,S2,S3 images for Matrix=True S0,S1,S2,S3 for Matrix=False

polarization_ellipse(*pol_state=None*, *matrix: bool = False*)
returns A, B, theta, h polarization parameter of elipses

Parameters

- **pol_state** (*None or (I, Q, U, V)*) – Polarization state previously computed
- **Matrix** (*bool*) – if True returns Matrix, else Scalar_field_X

Returns A, B, theta, h for Matrix=True CA, CB, Ctheta, Ch for Matrix=False

normalize()
Normalizes the field

draw(*kind: str = 'intensity'*, *logarithm: bool | numpy.float64 = 0*, *normalize: bool = False*, *cut_value: Optional[Union[float, bool]] = None*, *filename: str = ''*, *draw: bool = True*, ***kwargs*)
Draws electromagnetic field

Parameters

- **kind** (*str*) – 'intensity', 'intensities', intensities_rz, 'phases', fields', 'stokes'
- **logarithm** (*float*) – If >0, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **cut_value** (*float*) – If not None, cuts the maximum intensity to this value
- **filename** (*str*) – if not '' stores drawing in file,

4.1.22 diffractio.vector_fields_XY module

4.1.23 diffractio.vector_fields_XYZ module

4.1.24 diffractio.vector_fields_XZ module

4.1.25 diffractio.vector_fields_Z module

This module generates Vector_field_Z class. It is required also for generating masks and fields. The main atributes are:

- self.z - z positions of the field
- self.Ex - x component of electric field
- self.Ey - y component of electric field
- self.Ez - z component of electric field
- self.wavelength - wavelength of the incident field. The field is monocromatic
- self.info (str): description of data
- self.type (str): Class of the field

- self.date (str): date when performed

The magnitude is related to microns: *micron = 1.*

Class for X vector fields

Definition of a scalar field

- add, substract fields
- save, load data, clean, get, normalize
- cut_resample

Vector parameters

- polarization_states

Drawing functions

- draw: intensity, intensities, phases, fields, stokes, param_ellipse, ellipses

class diffractio.vector_fields_Z.**Vector_field_Z**(*z: Optional[numpy.ndarray[Any, numpy.dtype[numpy.float64]]] = None*, *wavelength: Optional[float] = None*, *n_background: float = 1.0*, *info: str = ''*)

Bases: `object`

Class for vectorial fields.

Parameters

- **x** (*numpy.array*) – linear array with equidistant positions. The number of data is preferibly 2**n.
- **wavelength** (*float*) – wavelength of the incident field
- **info** (*str*) – String with info about the simulation

`self.z`
linear array with equidistant positions. The number of data is preferibly 2**n.

Type numpy.array

`self.wavelength`
wavelength of the incident field.

Type float

`self.Ex`
Electric_x field

Type numpy.array

`self.Ey`
Electric_y field

Type numpy.array

`self.Ez`
Electric_z field

Type numpy.array

save_data(*filename: str*, *add_name: str = ''*, *description: str = ''*, *verbose: bool = False*)
Common save data function to be used in all the modules. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **add_name=** (*str*) – sufix to the name, if 'date' includes a date
- **description** (*str*) – text to be stored in the dictionary to save.
- **verbose** (*bool*) – If verbose prints filename.

Returns filename. If False, file could not be saved.

Return type (str)

load_data(*filename: str*, *verbose: bool = False*)

Load data from a file to a Vector_field_Z. The methods included are: npz, matlab

Parameters

- **filename** (*str*) – filename
- **verbose** (*bool*) – shows data process by screen

clear_field()
Removes the fields Ex, Ey, Ez

duplicate(*clear: bool = False*)
Duplicates the instance

get(*kind: str = 'fields'*, *is_matrix: bool = True*)
Takes the vector field and divide in Scalar_field_Z.

Parameters **kind** (*str*) – 'fields', 'intensity', 'intensities', 'phases', 'stokes', 'params_ellipse'

Returns (Ex, Ey, Ez),

Return type *Vector_field_Z*

apply_mask(*u*)
Multiply field by binary scalar mask: self.Ex = self.Ex * u.u

Parameters **u** (Scalar_mask_X) – mask

intensity()
"Returns intensity.

polarization_states(*matrix: bool = False*)
returns the Stokes parameters

Parameters **Matrix** (*bool*) – if True returns Matrix, else Scalar_field_Z

Returns S0,S1,S2,S3 images for Matrix=True S0,S1,S2,S3 for Matrix=False

polarization_ellipse(*pol_state: Union[None, list] = None*, *matrix: bool = False*)
returns A, B, theta, h polarization parameter of elipses

Parameters

- **pol_state** (*None or (I, Q, U, V)*) – Polarization state previously computed
- **Matrix** (*bool*) – if True returns Matrix, else Scalar_field_Z

Returns A, B, theta, h for Matrix=True CA, CB, Ctheta, Ch for Matrix=False

normalize(*new_field: bool = False*)
Normalizes the field so that intensity.max()=1.

Parameters **new_field** (*bool*) – If False the computation goes to self.u. If True a new instance is produced

Returns u (numpy.array): normalized optical field

draw(*kind: str = 'intensity'*, *logarithm: bool | numpy.float64 = 0*, *normalize: bool = False*, *cut_value: Optional[numpy.float64] = None*, *filename: str = ''*, *draw: bool = True*, ***kwargs*)
Draws electromagnetic field

Parameters

- **kind** (*str*) – 'intensity', 'intensities', intensities_rz, 'phases', fields', 'stokes'
- **logarithm** (*float*) – If >0, intensity is scaled in logarithm
- **normalize** (*bool*) – If True, max(intensity)=1
- **cut_value** (*float*) – If not None, cuts the maximum intensity to this value
- **filename** (*str*) – if not '' stores drawing in file,

4.1.26 diffractio.vector_masks_XY module

4.1.27 diffractio.vector_sources_XY module

4.1.28 Module contents

Top-level package for Python Scalar and vector diffraction and interference.

CHAPTER

FIVE

NEW FEATURES

5.1 0.2.3 (2023-11-21)

- Improvement:
  - XYZ drawings is removed temporally.
  - utils_slicer.py is deprecated.
  -
- Fix bugs:
  - Bug in XYZ video
  - Blazed grating wrong defined.
  - bug in Scalar_field_XY.kernelRSinverse
- New features:
  - XY masks: new polygon, regular_polygon, star, and superformula functions.
  - XY masks: new circular_sector function.
  - XY masks: new lens_cyliindrical function.
  - X, XY, XYZ fields: new conjugate function.
  - WPM function without storing all the data.
  - X fields: inverse_amplitude, inverse_phase, pupil

5.2 Future

- Implement plotly for drawings
- Some bugs in CZT (y)
- VWPM

CHAPTER

SIX

TUTORIALS

6.1 Algorithms

Propagation algorithms is a fundamental key in Diffractio package. The algorithms implmented are:

Scalar Propagation

- Fast Fourier Transform (FFT).
- Rayleigh Sommerfeld (RS).
- Plane Wave Descomposition (PWD).
- Beam Propagation Method (BPM).
- Wave Propagation Method (WPM).
- Chirp z-transform (CZT).

Vector propagation

- Vector Fast Fourier Tranform (VFFT).
- Vector Rayleigh-Sommerfeld (VRS).
- Vector Chirped z-transform (VCZT)

Fast Fourier Transform (FFT)

It allows, in a single step to determine the field at the far field. The fields and the masks must be defined in a plane.

Rayleigh-Sommerfeld (RS) - Shen, F., and Wang, A. (2006). Fast-Fourier-transform based numerical integration method for the Rayleigh-Sommerfeld diffraction formula. Applied Optics, 45(6), 1102–1110. https://doi.org/10.1364/AO.45.001102

Single step to propagate to a near or far observation plane, which allows fast computations.

Beam propagation method (BPM)

- Feit, M. D., and Fleck, J. A. (1978). Light propagation in graded-index optical fibers.

Propation of light in volumetric elements, provided that the spatial variations in the refraction index are small. It allows graded index structures. It presents a complexity of O(n) in the two-dimensional and O(n2) in the three-dimensional case. Computed according to split-step propagation scheme.

Wave Propagation Method (WPM)

- Brenner, K.-H. H., and Singer, W. (1993). Light propagation through microlenses: a new simulation method. Applied Optics, 32(26), 4984–4988. https://doi.org/10.1364/ao.32.004984
- Brenner, K. H. (2017). A high-speed version of the wave propagation method applied to micro-optics. 2017 16th Workshop on Information Optics, WIO 2017, 1, 2–4. https://doi.org/10.1109/WIO.2017.8038108

- Schmidt, S., Thiele, S., Herkommer, A., Tünnermann, A., and Gross, H. (2017). Rotationally symmetric formulation of the wave propagation method-application to the straylight analysis of diffractive lenses. Optics Letters, 42(8), 1612. https://doi.org/10.1364/ol.42.001612

Solves the major limitations of the beam propagation method (BPM). The wave propagation scheme provides valid results for propagation angles up to 85° and that it is not limited to small index variations in the axis of propagation. Very fast with a discrete number of refraction indexes.

Chirped z Transform (CZT)

- Leutenegger, M., Rao, R., Leitgeb, R. A., and Lasser, T. (2006). Fast focus field calculations. Optics Express, 14(23), 11277–11291. http://lob.epfl.ch/
- Hu, Y., Wang, Z., Wang, X., Ji, S., Zhang, C., Li, J., Zhu, W., Wu, D., and Chu, J. (2020). Efficient full-path optical calculation of scalar and vector diffraction using the Bluestein method. Light: Science and Applications, 9(1). https://doi.org/10.1038/s41377-020-00362-z

Vector Fast Fourier Transform (VFFT)

- todo

Vector Rayleigh-Sommerfeld (VRS)

- Ye, H., Qiu, C. W., Huang, K., Teng, J., Luk'Yanchuk, B., and Yeo, S. P. (2013). Creation of a longitudinally polarized subwavelength hotspot with an ultra-thin planar lens: Vectorial Rayleigh-Sommerfeld method. Laser Physics Letters, 10(6). https://doi.org/10.1088/1612-2011/10/6/065004

The VRS method propagates (Ex,Ey,Ez) fields offering the advantage of significant reduction in computation, from flat diffractive elements (Thin Element Approximation) with full control of polarization. It addresses simultaneously both longitudinal polarization. This approach offers the advantage of significant reduction in computation.

Vector Chirped Z Transform (VCZT)

- Hu, Y., Wang, Z., Wang, X., Ji, S., Zhang, C., Li, J., Zhu, W., Wu, D., and Chu, J. (2020). Efficient full-path optical calculation of scalar and vector diffraction using the Bluestein method. Light: Science and Applications, 9(1). https://doi.org/10.1038/s41377-020-00362-z

6.1.1 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) is used to determine the field at a distance far away from the mask,

$$z \gg \frac{\pi}{\lambda}\left(\xi^2+\eta^2\right)_{\max},$$

where ξ and η are the locations of the mask. The field can be computed using the following integral (in 2D):

$$E(x,y,z) = \frac{e^{ik(z+\frac{x^2+y^2}{2z})}}{i\lambda z}\iint E_0(\xi,\eta)e^{-i\frac{k}{z}(x\xi+y\eta)}d\xi d\eta,$$

which can be computed as a FFT:

$$E(x,y,z) = \frac{e^{ik(z+\frac{x^2+y^2}{2z})}}{i\lambda z} TF\left[E_0(\xi,\eta)\right].$$

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
```

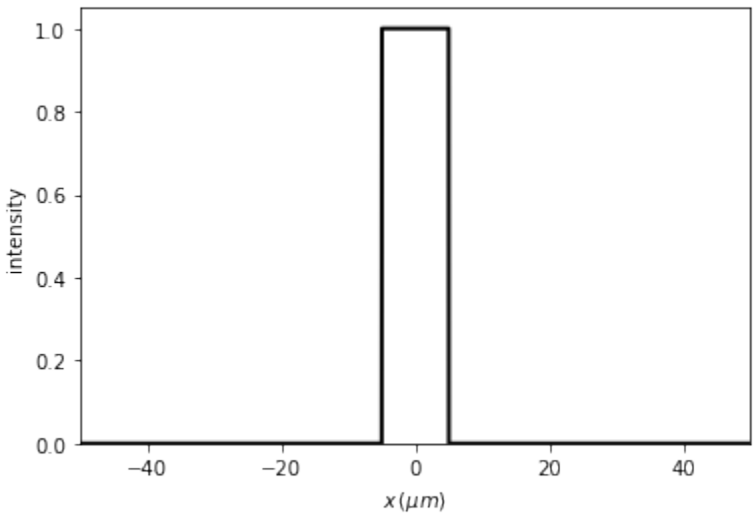
6.1.1.1 X Scheme

[2]:
```
x0 = np.linspace(-50 * um, 50 * um, 1024 * 32)
wavelength = .5 * um
```

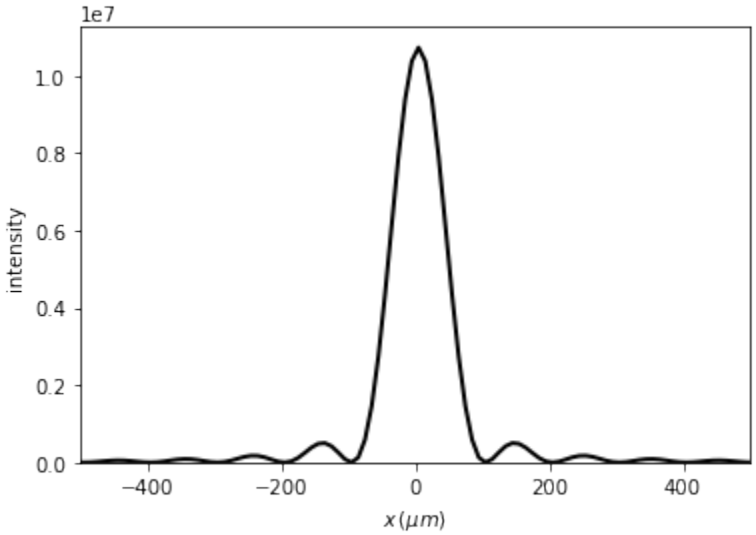
[3]:
```
# plane wave
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0)

# slit
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.slit(x0=0, size=10 * um)
t0.draw()

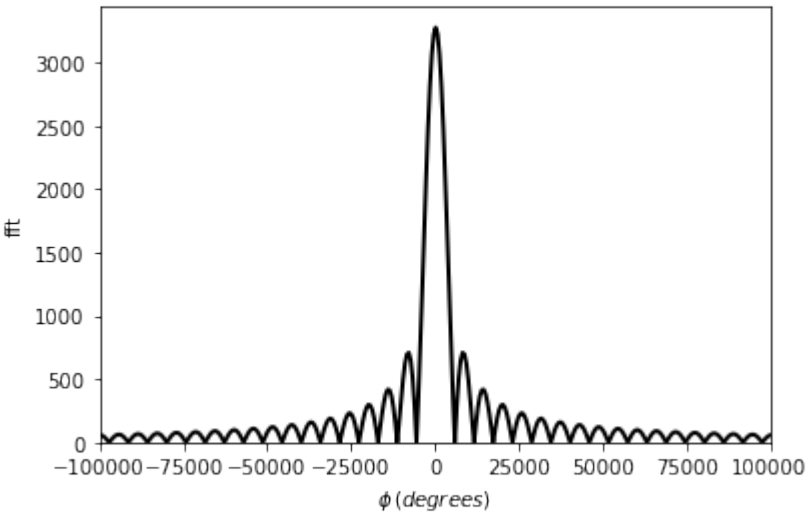
u1 = u0 * t0
```



[4]:
```
u2 = u1.fft(z=2 * mm, remove0=False, new_field=True)
u2.draw(kind='intensity', logarithm=False, normalize=True)
plt.xlim(-500, 500)
plt.ylim(bottom=0)
```



[5]:
```
u3 = u1.fft(z=2 * mm, remove0=False, new_field=True)
u3.draw(kind='fft', logarithm=False, normalize=True)
plt.xlim(-1e5, 1e5)
plt.ylim(bottom=0)
```



Fast Fourier Transform using Rayleigh-Sommerfeld (near field) and a lens

Now, let us determine the field at the Fourier plane of the lens.

[6]:
```
x = np.linspace(-100 * um, 100 * um, 4096)
wavelength = .5 * um

focal = .25 * mm

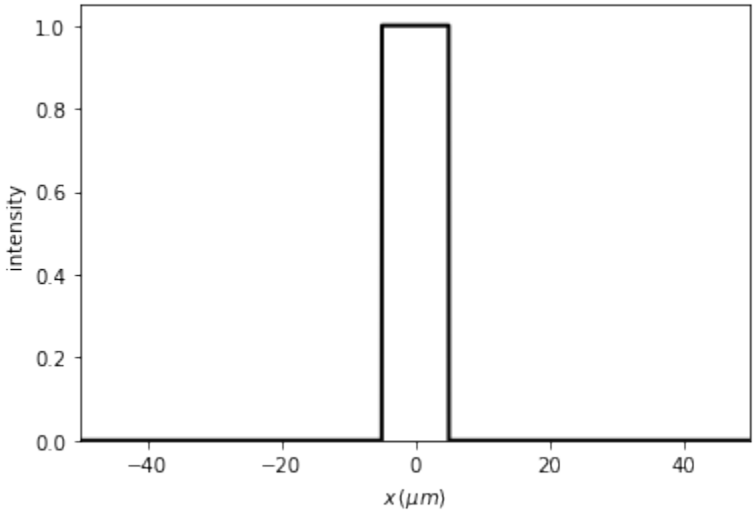
u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1, theta=0 * degrees)
```

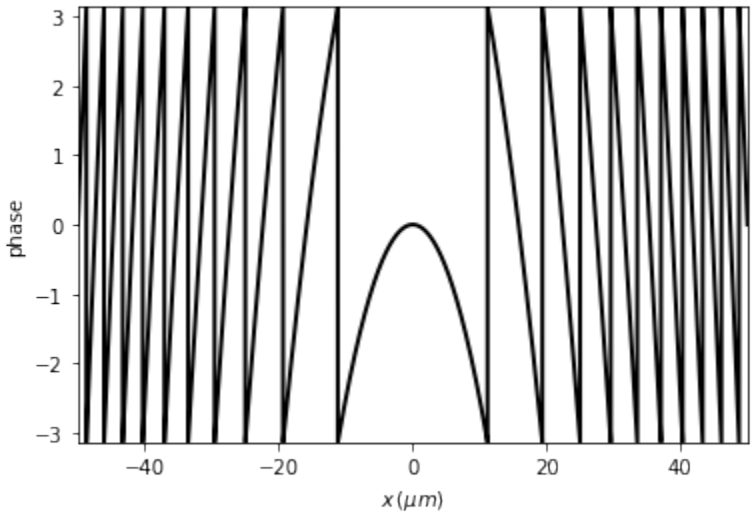
(continues on next page)

(continued from previous page)

```
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.slit(x0=0, size=10 * um)
t0.draw()

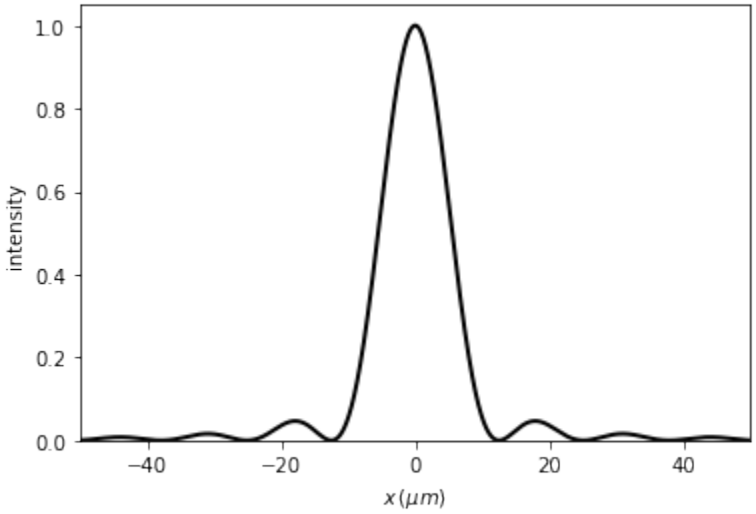
t1 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t1.lens(x0=0, radius=500, focal=focal)
t1.draw(kind='phase')
```





[7]:
```
u1 = t0 * t1 * u0

t2 = u1.RS(z=focal, verbose=False)
t2.normalize()
t2.draw(kind='intensity')
```



6.1.1.2 XY Scheme

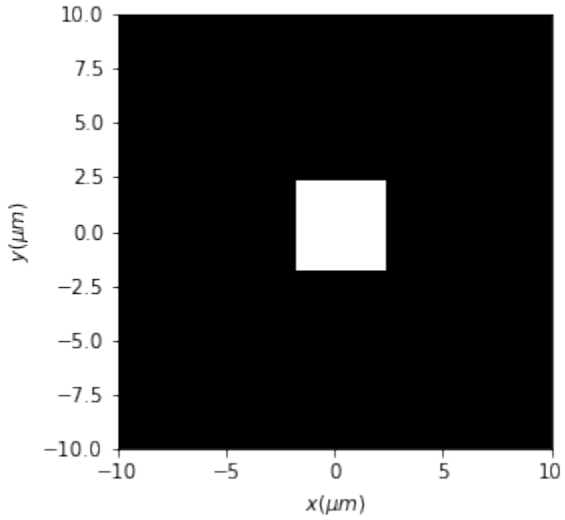
[8]:
```
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

[9]:
```
x0 = np.linspace(-150 * um, 150 * um, 1024)
y0 = np.linspace(-150 * um, 150 * um, 1024)
wavelength = .5 * um
```

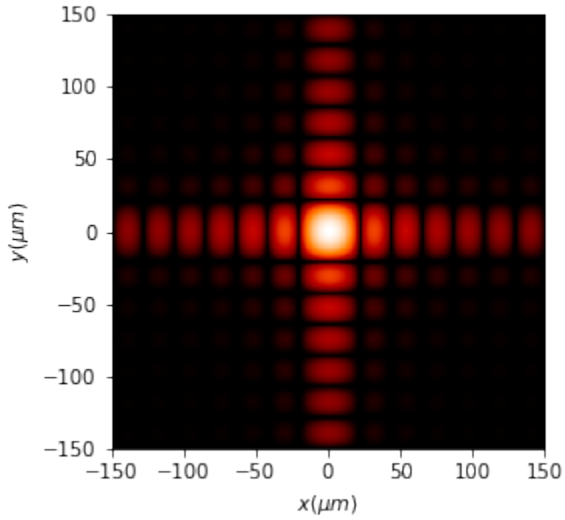
[10]:
```
# plane wave
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0)

# slit
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.square(r0=(0, 0), size=(4 * um, 4 * um), angle=0)
t0.draw()
plt.xlim(-10, 10)
plt.ylim(-10, 10)

u1 = u0 * t0
```



[11]:
```
u2 = u1.fft(remove0=False, new_field=True)
u2.draw(logarithm=1e-1)
```



6.1.2 Rayleigh-Sommerfeld (RS)

According to Rayleigh-Sommerfeld approach, the field at a plane z from the mask, can be computed as

$$E(\mathbf{r},t) = ike^{-i\omega t}\iint E_0(\xi,\eta)cos\theta\frac{e^{ikR}}{R}d\xi d\eta,$$

where $\mathbf{r} = (x,y,z)$, $R = \sqrt{(\xi-x)^2+(\eta-y)^2+z^2}$, and $cos\theta = z/R$.

This integral can be computed as a convolution,

$$E(x,y,z) = E_0(\xi,\eta) * h(\xi,\eta;z),$$

where the convolution kernel is

$$h(\xi, \eta; z) = -\frac{1}{2\pi}\frac{z}{R}\left(ik - \frac{1}{R}\right)\frac{e^{ikR}}{R},$$

with $R = \sqrt{\xi^2 + \eta^2 + z^2}$.

In Diffractio, we have used the algorithm described in:

Shen, F. & Wang, A. Fast-Fourier-transform based numerical integration method for the Rayleigh-Sommerfeld diffraction formula. Appl. Opt. 45, 1102–1110 (2006).

The algorithm can be applied to X, and XY schemes, and also in the propagated schemes: XZ and XYZ.

The main characteristics of the algorithm are: - Propagation ocurrs between two parallel planes, whose distance between them is z. The results are more accurate when de distance between de planes increases. The function to perform Rayleigh-Sommerfeld approach is RS. There is a parameter which stores the quality parameter (self.quality). Also, it can be seen when invoquing to RS function. The parameter verbose, when it is True, shows the quality parameter.

- The algorithm is programmed using two FFT and one IFFT. The size of the matrices is 2N x 2N, where N is the number of data involved in the analysis.
- There is a quality parameter, described in self.quality, which allows us to determine whether the results is acceptable (quality parameter >1).
- The size of the output matrices is equal to that of the input matrices. Nevertheless, higher size of the output fields can be obtained as the output field can be displaced from the center. This is programmed with parameter amplification, which size the amplification of the output field with respect to the input field. The computation time is proportional to the amplification.
- The X and XZ schemes are accurate when the fields and masks are simmetrycal with respect to Z axis. That means that the lens in the X and XZ shcemes is a cylindrical lens.

Now, let us show an example of usage of Rayleigh-Sommerfeld approach for the differente schemes:

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
```

6.1.2.1 X scheme

[2]:
```
from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
```

[3]:
```
wavelength = 0.5 * um
x0 = np.linspace(-0.25 * mm, 0.25 * mm, 1024 * 16)

u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1)
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.slit(x0=0, size=300 * um)
t0.draw(kind='intensity')

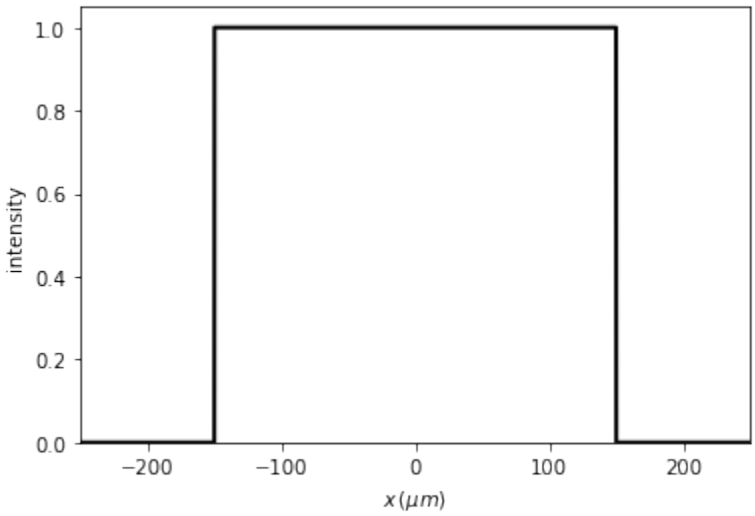
u1 = t0 * u0
```

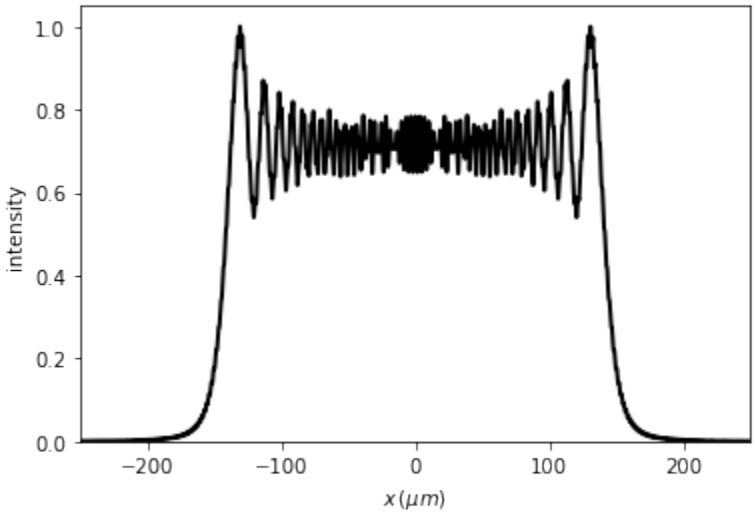
(continues on next page)

(continued from previous page)

```
t2 = u1.RS(z=1 * mm, verbose=True)
t2.normalize()
t2.draw(kind='intensity')
```

```
Good result: factor 33.64
```





[5]:
```
print("quality = {}".format(t2.quality))
```

```
quality = [33.64442563]
```

Bad quality parameter

When the quality parameter is < 1, then the field computed using the RS algorithm produces bad results:

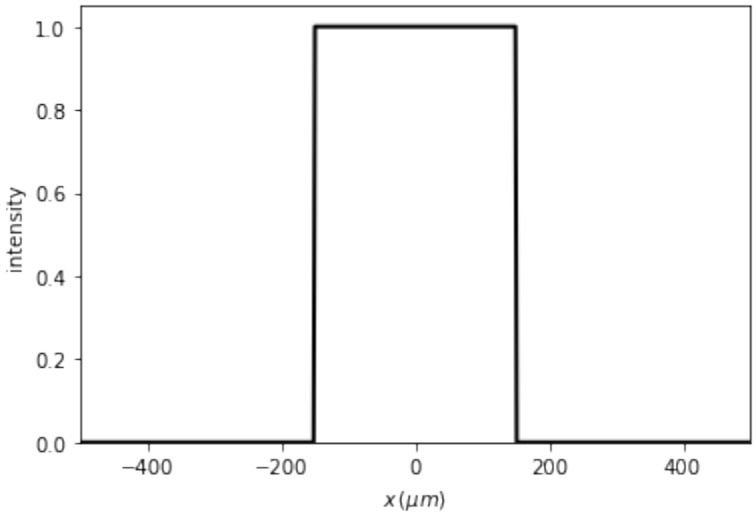
[6]:
```
wavelength = 0.5 * um
x0 = np.linspace(-0.5 * mm, 0.5 * mm, 512)

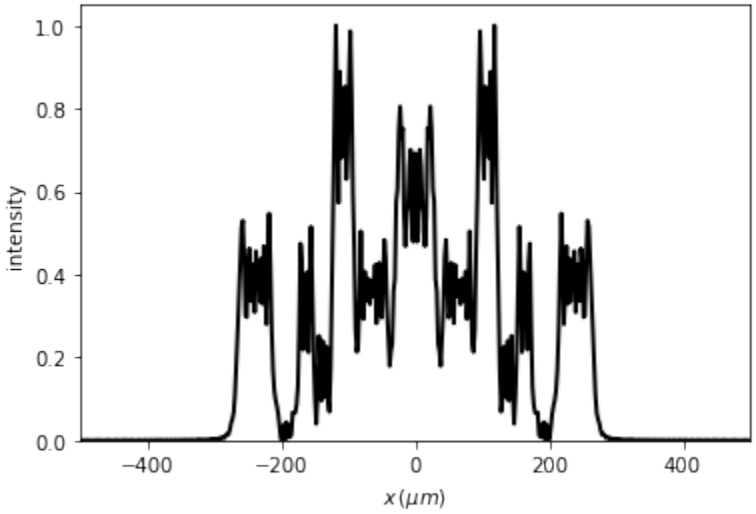
u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1)
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.slit(x0=0, size=300 * um)
t0.draw(kind='intensity')

u1 = t0 * u0

t2 = u1.RS(z=.1 * mm, verbose=True)
t2.normalize()
t2.draw(kind='intensity')
```

```
- Needs denser sampling: factor 0.13
```





Amplification of the field

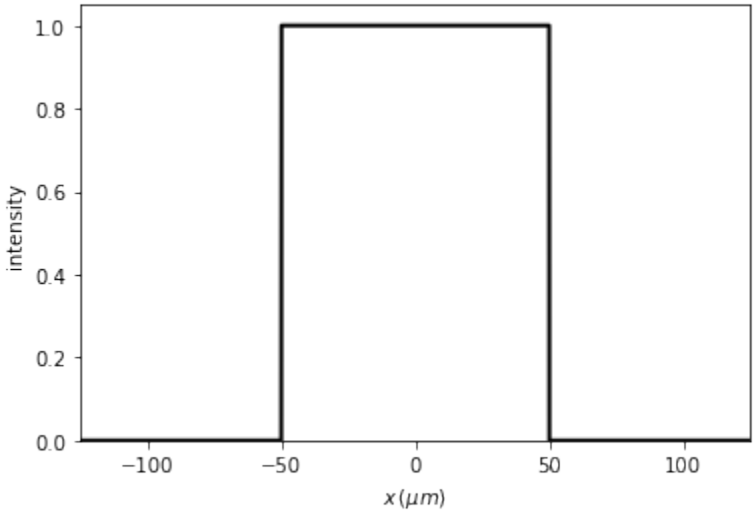
The Rayleigh-Sommefeld implementation allows an amplification of the field since, sometimes, light diverges. Processing time increases approximately linearly with amplification number.

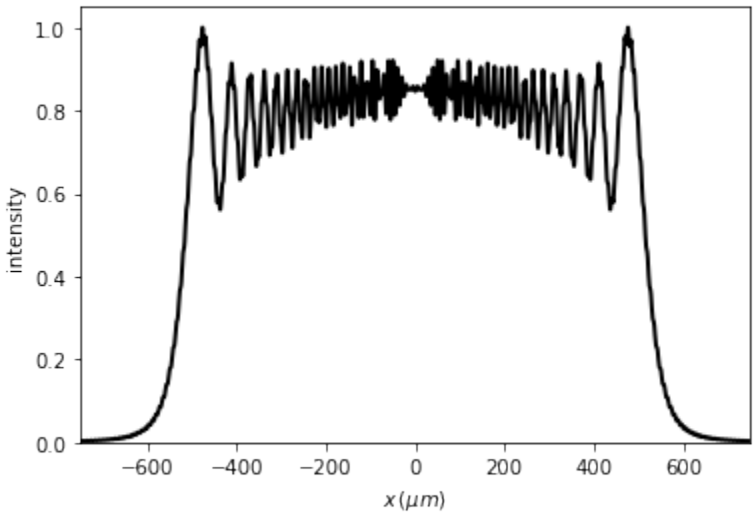
[7]:
```
wavelength = 0.5 * um
x0 = np.linspace(-0.125 * mm, 0.125 * mm, 1024 * 16)

u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.spherical_wave(A=1, x0=0 * um, z0=-.1 * mm)
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.slit(x0=0, size=100 * um)
t0.draw(kind='intensity')

u1 = t0 * u0

u2 = u1.RS(z=1 * mm, amplification=6, verbose=False)
u2.normalize()
u2.draw(kind='intensity')
```





6.1.2.2 XZ scheme

[8]:
```
from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

[9]:
```
wavelength = 0.5 * um
x0 = np.linspace(-100 * um, 100 * um, 1024)
z0 = np.linspace(100 * um, 10 * mm, 512)

u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1)
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.slit(x0=0, size=100 * um)
```

(continues on next page)

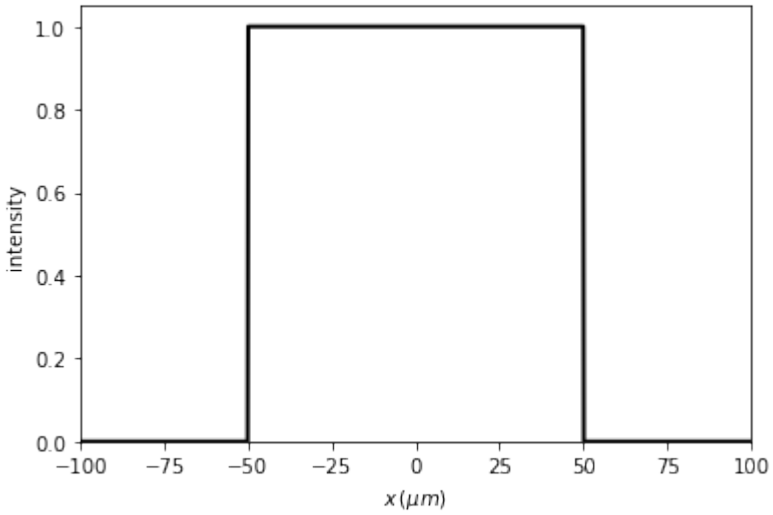
(continued from previous page)

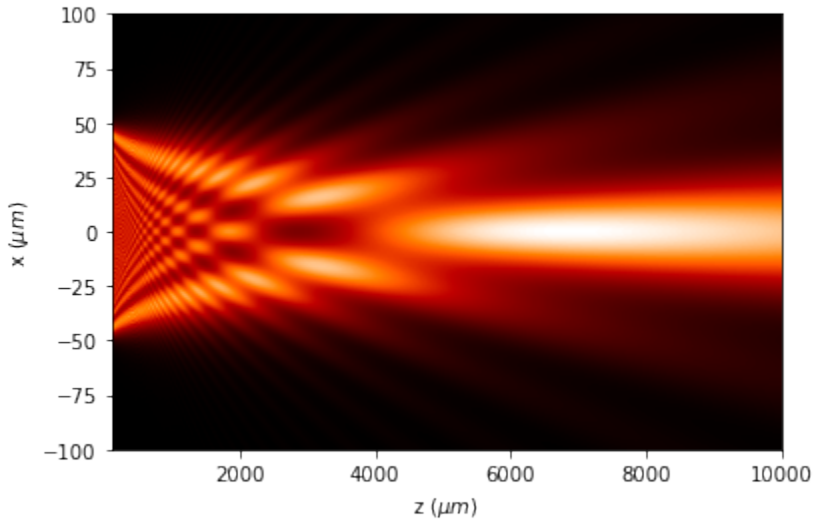
```
u1 = t0 * u0

uz = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, info='t0')
uz.incident_field(u1)

uz.RS(verbose=True)
uz.draw(kind='intensity')
```

```
Good result: factor 3.61
time in RS= 2.6193251609802246. num proc= 8
```





6.1.2.3 XY scheme

[10]:
```
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

[11]:
```
wavelength = 0.5 * um
x0 = np.linspace(-100 * um, 100 * um, 1024)
y0 = np.linspace(-100 * um, 100 * um, 1024)

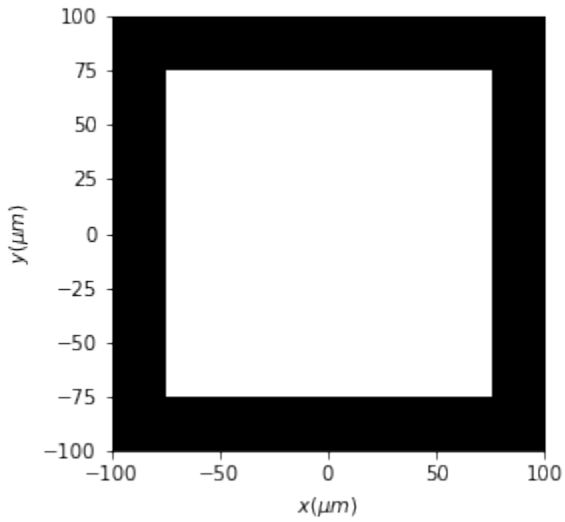
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1)

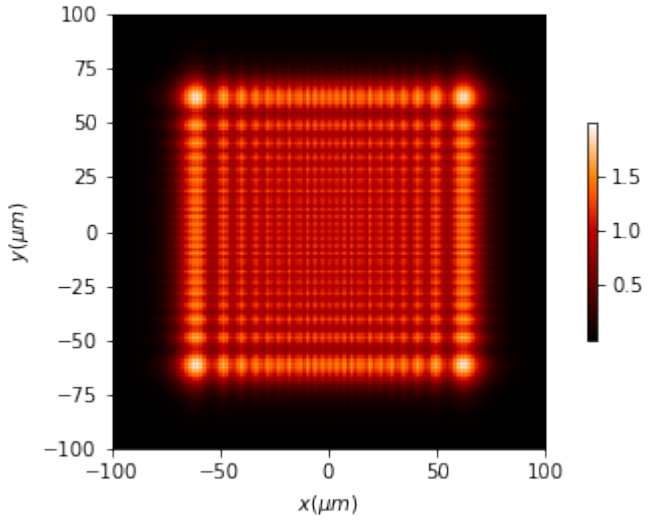
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.square(r0=(0, 0), size=150 * um, angle=0 * degrees)

u1 = t0 * u0

u2 = u1.RS(z=500 * um)

u2.draw(kind='intensity', has_colorbar='vertical')
```





6.1.2.4 XYZ Scheme

We perform the XYZ field distribution by a for loop. Then the computing time and the storing size is proportional to the number of z planes used in the z parameter.

[12]:
```
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ
```

[13]:
```
wavelength = 0.5 * um
x0 = np.linspace(-100 * um, 100 * um, 256)
y0 = np.linspace(-100 * um, 100 * um, 256)
z0 = np.linspace(500 * um, 2 * mm, 64)

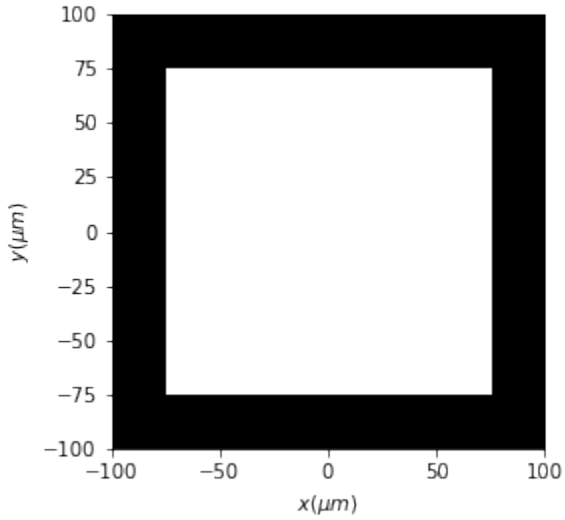
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1)

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.square(r0=(0, 0), size=150 * um, angle=0 * degrees)

u1 = t0 * u0

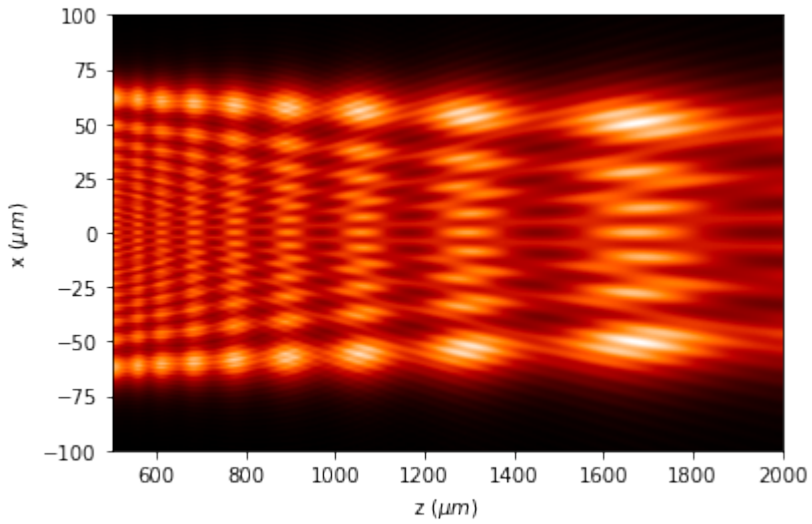
uz = Scalar_field_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
uz.incident_field(u1)
uz.RS(verbose=True)
```

```
time in RS= 20.78619694709778. num proc= 6
```



[14]:
```
uz.draw_XZ(kind='intensity')
```

```
<Figure size 432x288 with 0 Axes>
```



6.1.3 Beam Propagation Method (BPM)

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X

import matplotlib.cm as cm
```

6.1.3.1 XZ scheme

[2]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024 * 4)
z0 = np.linspace(-0.1 * mm, 1.25 * mm, 1024 * 2)
wavelength = 5 * um
```

[3]:
```
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

[4]:
```
lens = Scalar_mask_XZ(x0, z0, wavelength, n_background=1, info='')
ipasa, conds = lens.aspheric_lens(r0=(0 * mm, 0 * mm),
                                  angle=(0 * degrees, (0 * mm, 0 * mm)),
                                  refractive_index=1.5,
                                  cx=(1 / (1 * mm), -1 / (.25 * mm)),
                                  Qx=(0, 0),
                                  a2=(0, 1e-13),
                                  a3=(0, 0),
                                  a4=(0, 0),
                                  depth=.4 * mm,
                                  size=0.8 * mm)

lens.slit(r0=(0, 100 * um),
          aperture=800 * um,
          depth=75 * um,
          refractive_index=1 + 2j)

lens.draw_refractive_index(draw_borders=True,
                           min_incr=0.01,
                           colormap_kind=cm.Blues,
                           colorbar_kind='vertical')
```

[ ]:
```
lens.smooth_refractive_index(type_filter=2, pixels_filtering=25)
```

[ ]:
```
lens.incident_field(u0)
```

[ ]:
```
%%time
lens.clear_field()
lens.BPM(verbose=False)
lens.normalize()
```

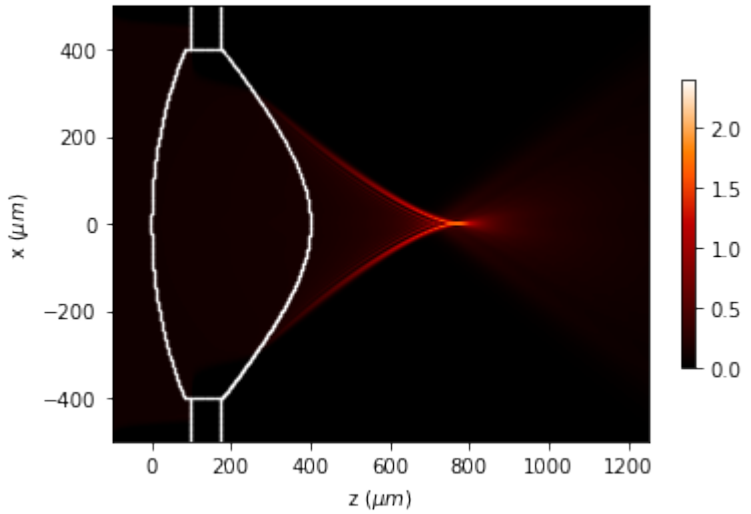
```
CPU times: user 1.48 s, sys: 186 ms, total: 1.66 s
Wall time: 1.66 s
```

[ ]:
```
lens.draw(kind='intensity',
          logarithm=1e1,
          normalize=None,
```

(continues on next page)

(continued from previous page)

```
        draw_borders=True,
        colorbar_kind='vertical')
```



[ ]:
```
Intensity_BPM = lens.intensity()
```

Focusing area

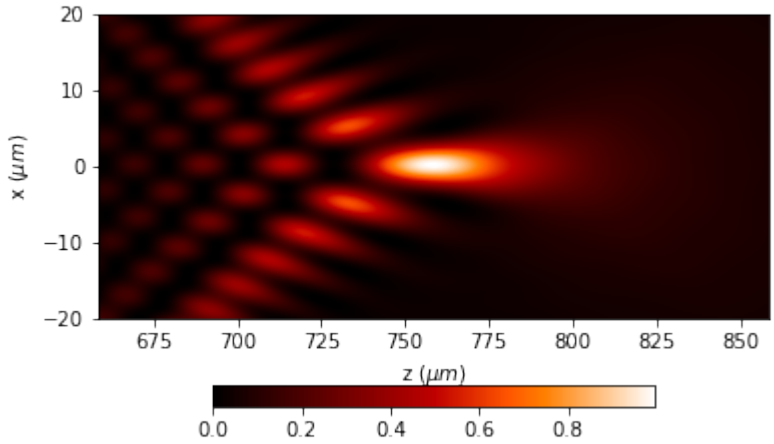
[ ]:
```
x_f_bpm, z_f_bpm = lens.search_focus()
```

```
x = -0.122 um, z = 758.671 um
```

[ ]:
```
ylim_max = 20 * um
zlim_max = 100 * um
```

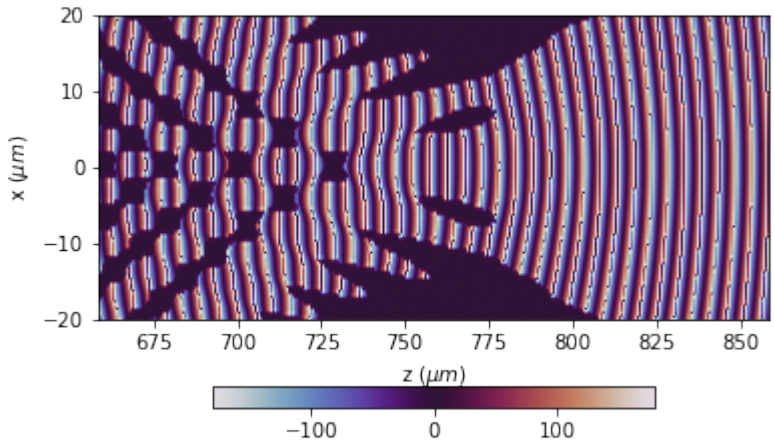
[ ]:
```
lens.draw(kind='intensity', colorbar_kind='horizontal')

plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_bpm - zlim_max, z_f_bpm + zlim_max)
```



[ ]:
```
lens.draw(kind='phase', colorbar_kind='horizontal', percentage_intensity=0.05)

plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_bpm - zlim_max, z_f_bpm + zlim_max)
```



6.1.4 Wave propagation Method (WPM)

[23]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
```

6.1.4.1 XZ Scheme

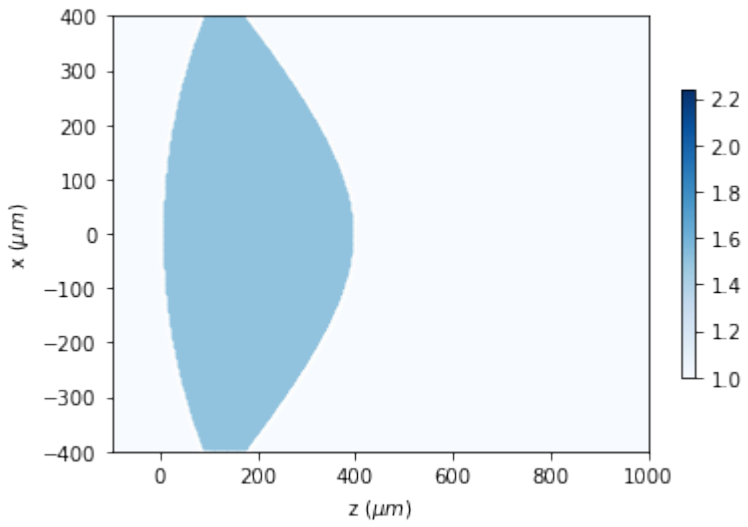
[24]:
```
x0 = np.linspace(-400 * um, 400 * um, 1024 * 2)
z0 = np.linspace(-0.1 * mm, 1 * mm, 1024 * 2)
wavelength = 2 * um
```

[25]:
```
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

[26]:
```
lens = Scalar_mask_XZ(x0, z0, wavelength, n_background=1, info='')
ipasa, conds = lens.aspheric_lens(r0=(0 * mm, 0 * mm),
                                  angle=(0 * degrees, (0 * mm, 0 * mm)),
                                  refractive_index=1.5,
                                  cx=(1 / (1 * mm), -1 / (.25 * mm)),
                                  Qx=(0, 0),
                                  a2=(0, 1e-13),
                                  a3=(0, 0),
                                  a4=(0, 0),
                                  depth=.4 * mm,
                                  size=0.8 * mm)

lens.slit(r0=(0, 100 * um),
          aperture=800 * um,
          depth=75 * um,
          refractive_index=1 + 2j)

lens.draw_refractive_index(draw_borders=True,
                           min_incr=0.01,
                           colorbar_kind='vertical')
```



[27]:
```
lens.smooth_refractive_index(type_filter=2, pixels_filtering=15)
```

Since the computation time of WPM is proportional to the number or refraction indexes at each layer, it is important

to discretize the refraction index, with not to many layers.

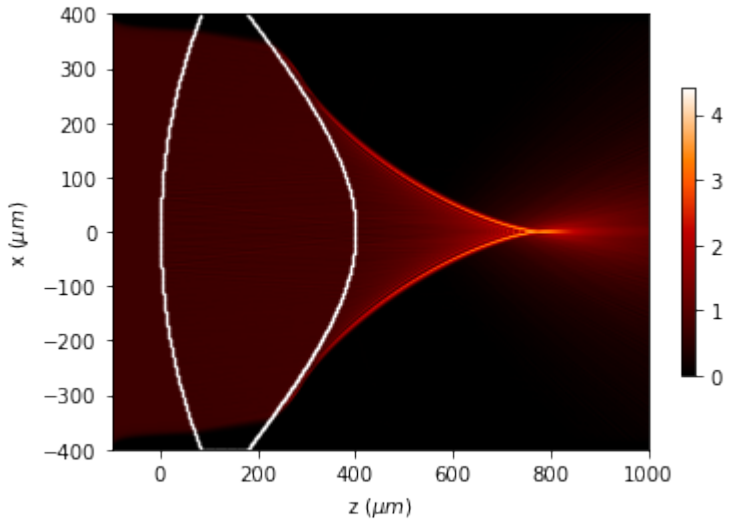
[28]:
```
lens.discretize_refractive_index(num_layers=6)
```

[29]:
```
lens.incident_field(u0)
```

[30]:
```
lens.clear_field()
lens.WPM(has_edges=True, verbose=False)
```

```
get_k
```

[31]:
```
lens.draw(kind='intensity',
          logarithm=1,
          normalize=None,
          draw_borders=True,
          colorbar_kind='vertical')
```



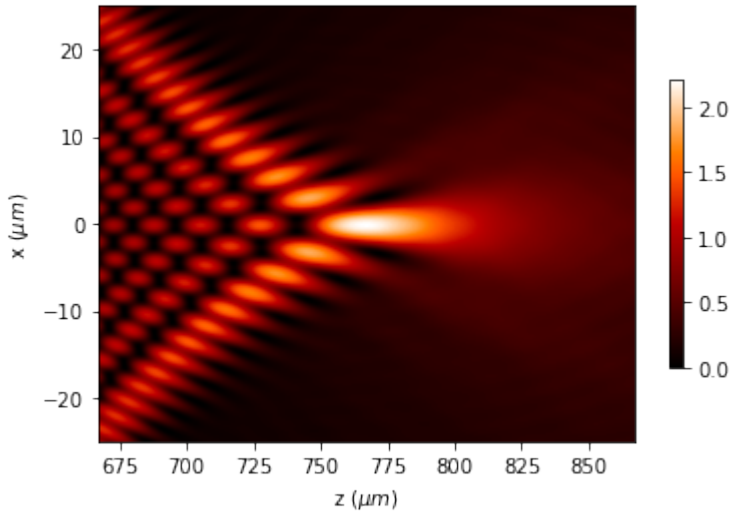
[32]:
```
x_f_wpm, z_f_wpm = lens.search_focus()
```

```
x = -0.195 um, z = 767.318 um
```

Focusing area

[33]:
```
ylim_max = 25 * um
zlim_max = 100 * um
```

[34]:
```
lens.draw(kind='intensity', logarithm=1e-1, colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_wpm - zlim_max, z_f_wpm + zlim_max)
```



6.1.4.2 XYZ Scheme

[35]:
```
from diffractio import np, plt, sp, um, mm, degrees
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ

from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
```

[36]:
```
x = np.linspace(-15 * um, 15 * um, 256)
y = np.linspace(-15 * um, 15 * um, 256)
z = np.linspace(0, 30 * um, 256)
wavelength = 0.6328 * um
```

[37]:
```
t0 = Scalar_mask_XY(x, y, wavelength)
t0.circle(r0=(0, 0), radius=12.5 * um, angle=0)

u0 = Scalar_source_XY(x, y, wavelength)
u0.plane_wave(A=1)
```

[38]:
```
u = Scalar_mask_XYZ(x, y, z, wavelength)
u.sphere(r0=(0, 0, 12.5), radius=10 * um, refractive_index=2, angles=(0, 0, 0))
```

[39]:
```
x = np.linspace(-50 * um, 50 * um, 256)
y = np.linspace(-50 * um, 50 * um, 256)
z = np.linspace(0, 500 * um, 256)
wavelength = .6 * um
```

[40]:
```
t0 = Scalar_mask_XY(x, y, wavelength)
t0.circle(r0=(0, 0), radius=45 * um, angle=0)
```

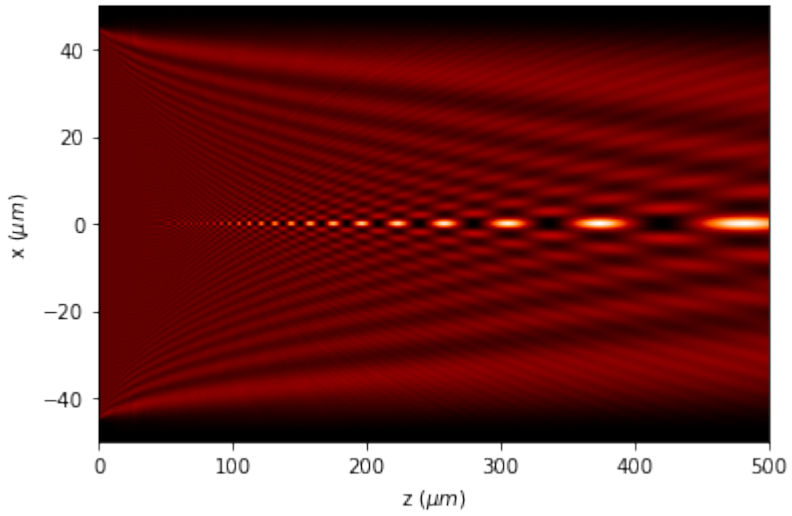
[41]:
```
u = Scalar_mask_XYZ(x, y, z, wavelength)
u.incident_field(u0=t0)
```

[42]:
```
%%time
u.clear_field()
u.WPM(verbose=True, has_edges=True)
```

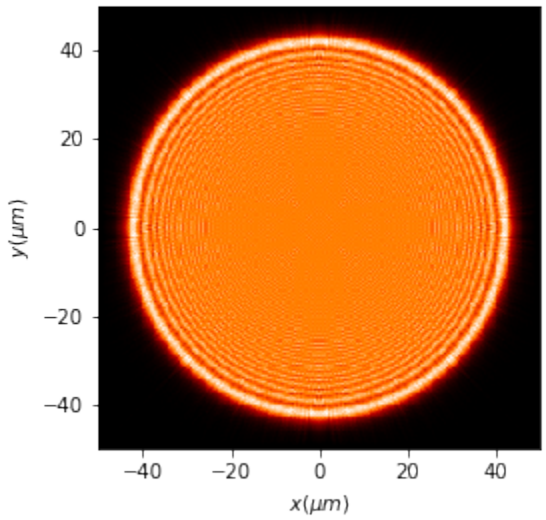
```
get_k
get_k
Time = 6.20 s, time/loop = 24.23 ms
CPU times: user 6.32 s, sys: 156 ms, total: 6.48 s
Wall time: 6.4 s
```

[43]:
```
u.draw_XZ(y0=0, logarithm=False)
```

```
<Figure size 432x288 with 0 Axes>
```



[44]:
```
u.draw_XY(z0=20, logarithm=True)
```



6.1.5 Chirped Z-transform (CZT)

CZT allows, in a single step, to propagate to a near or far observation plane. IThe main advantage of CZT is that the region of interest and the sampling numbers can be arbitrarily chosen, endowing CZT with superior flexibility, and produces much faster results (acceleration > x100 with respect to RS algorithm) for focusing and far field diffraction patterns.

As the sampling area and pixels can be reduced to the desired observation area, the storage is also greatly reduced.

CZT algorithm allows to have a XY mask and compute in XY, Z, XZ, XYZ schemes, simply defining the output arrays.

6.1.5.1 X Scheme

[1]:
```
from diffractio import degrees, mm, um, nm
from diffractio import np, plt, sp

from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_sources_X import Scalar_source_X

from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_fields_Z import Scalar_field_Z
```

[2]:
```
size = 250 * um
xin = np.linspace(-size, size, 4096)
wavelength = 550 * nm
z = 2 * mm
```

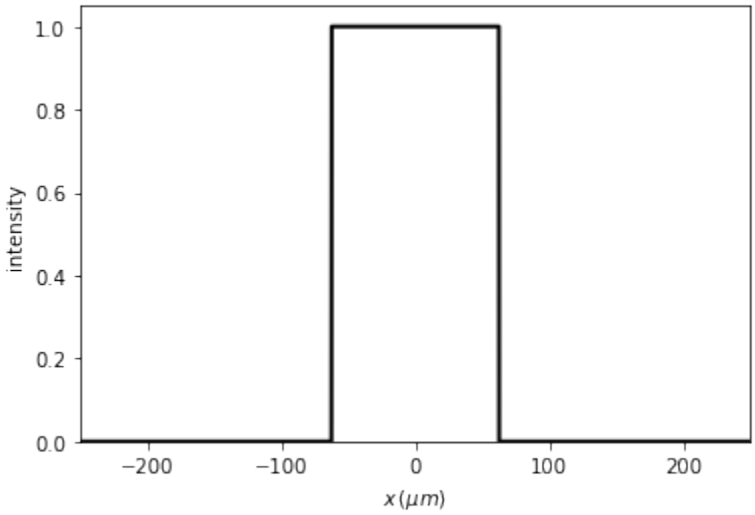
[3]:
```
t0 = Scalar_mask_X(xin, wavelength)
t0.slit(x0=0, size=size / 2)
u0 = Scalar_source_X(xin, wavelength)
```

(continues on next page)

(continued from previous page)

```
u0.plane_wave(A=1)

u1 = t0 * u0
u1.draw()
```



to just one data

[4]:
```
xout = 0.
z = 2 * mm
```

[5]:
```
%%time
u2 = u1.CZT(z, xout)
print(u2)
```

```
[0.63252658+0.60319534j]
CPU times: user 6.8 ms, sys: 0 ns, total: 6.8 ms
Wall time: 5.72 ms
```

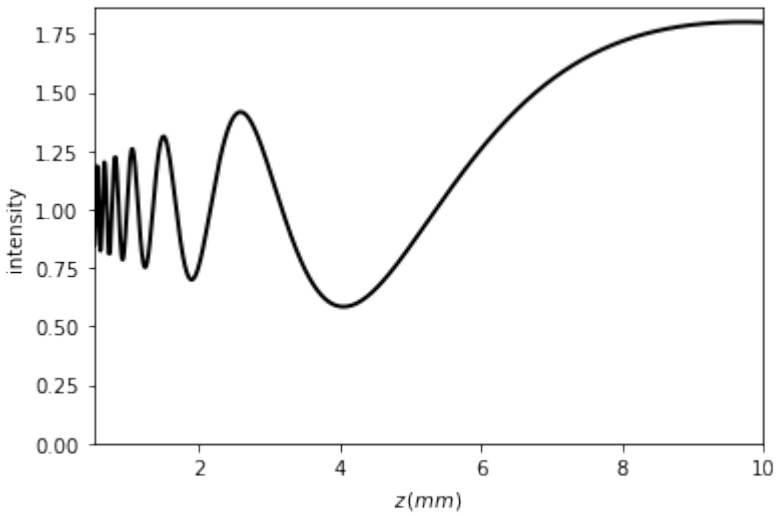
to field_Z

[6]:
```
xout = 0
z = np.linspace(.5 * mm, 10 * mm, 1024)
```

[7]:
```
%%time
u2 = u1.CZT(z, xout, verbose=True)
```

```
CPU times: user 3.81 s, sys: 54 ms, total: 3.86 s
Wall time: 3.81 s
```

[8]:
```
u2.draw(z_scale='mm')
```



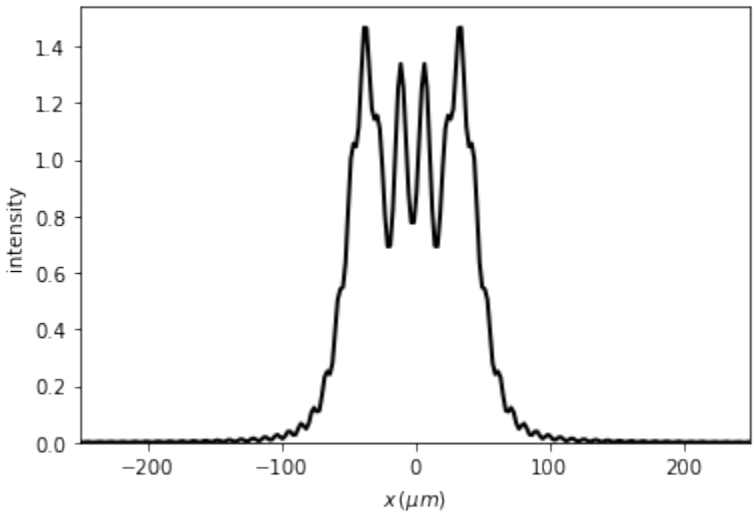
to field_X

[9]:
```
xout = np.linspace(-size, size, 256)
z = 2 * mm
```

[10]:
```
%%time
u2 = u1.CZT(z, xout)
```

```
CPU times: user 9.8 ms, sys: 215 µs, total: 10 ms
Wall time: 9.34 ms
```

[11]:
```
u2.draw()
```



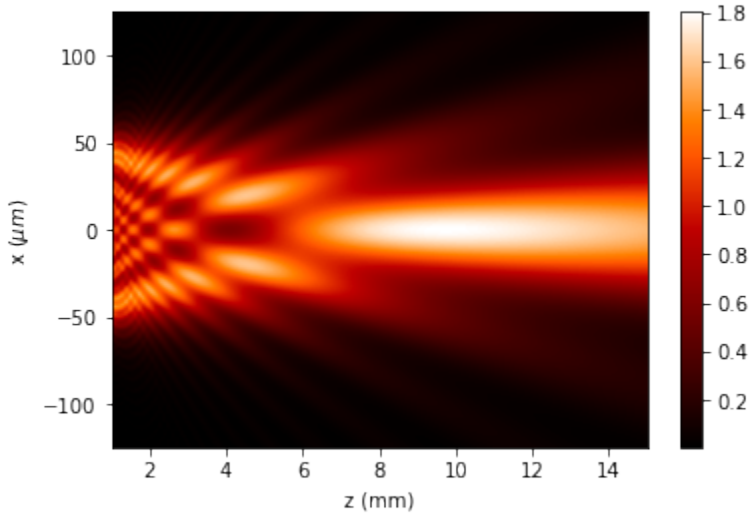
to field_XZ

[12]:
```
xout = np.linspace(-size / 2, size / 2, 2048)
z = np.linspace(1 * mm, 15 * mm, 128)
```

[13]:
```
%%time
u2 = u1.CZT(z, xout, verbose=True)
```

```
CPU times: user 1.13 s, sys: 23.2 ms, total: 1.15 s
Wall time: 1.15 s
```

[14]:
```
u2.draw(logarithm=0, z_scale='mm')
plt.colorbar()
```



to data

[18]:
```
xout = 0
yout = 0.
z = .5 * mm
```

[19]:
```
%%time
u2 = u1.CZT(z, xout, yout)
print("{}".format(np.abs(u2)**2))
```

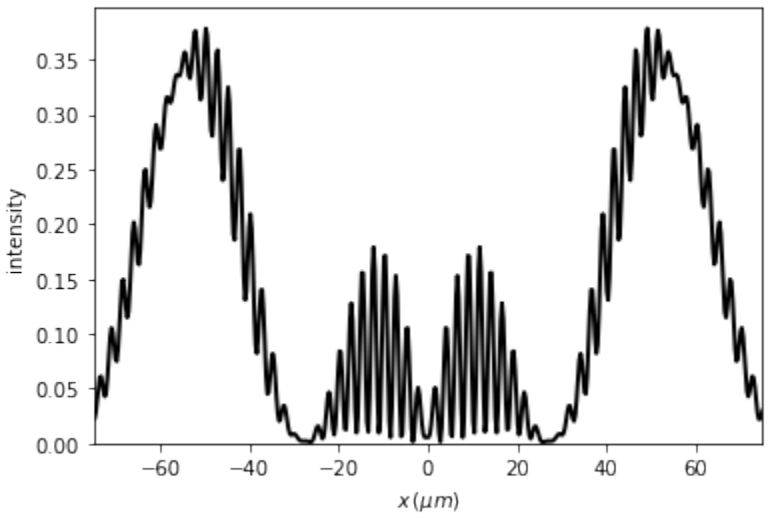
```
0.005578354037388983
CPU times: user 65 ms, sys: 23.6 ms, total: 88.6 ms
Wall time: 88.7 ms
```

to field_X

[20]:
```
xout = np.linspace(-size, size, 512)
yout = 0.
z = .5 * mm
```

[21]:
```
%%time
u2 = u1.CZT(z, xout, yout)
u2.draw()
```

```
CPU times: user 120 ms, sys: 7.92 ms, total: 127 ms
Wall time: 126 ms
```

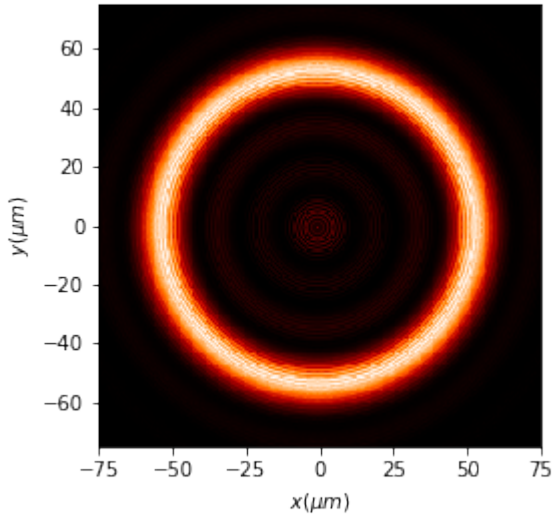


to field_XY

[22]:
```
xout = np.linspace(-size, size, 256)
yout = np.linspace(-size, size, 256)
z = .25 * mm
```

[23]:
```
%%time
u2 = u1.CZT(z, xout, yout)
u2.draw()
```

```
CPU times: user 342 ms, sys: 124 ms, total: 465 ms
Wall time: 256 ms
```



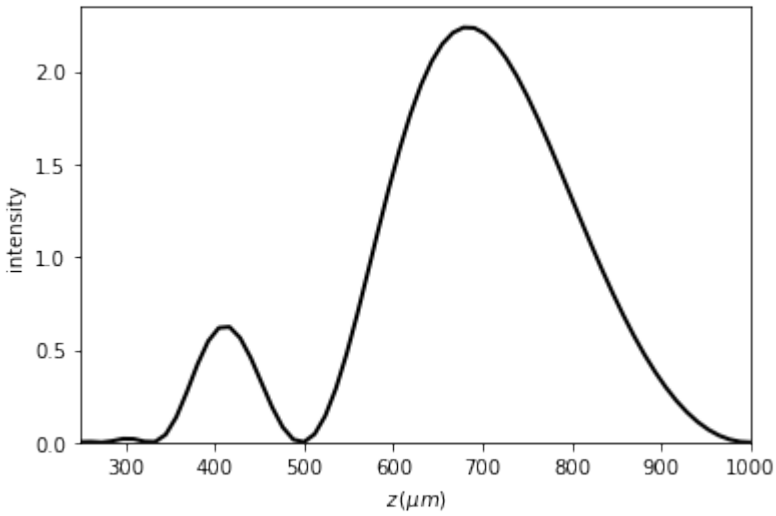
to field_Z

The Z field is computed with a for loop, thus it is a bit slower.

[24]:
```
xout = -1.
yout = 0.
z = np.linspace(0.25 * mm, 1 * mm, 64)
```

[25]:
```
%%time
u2 = u1.CZT(z, xout, yout, verbose=True)
u2.draw()
```

```
num x, num y, num z = 1, 1, 64
CPU times: user 3.75 s, sys: 37.4 ms, total: 3.78 s
Wall time: 3.75 s
```

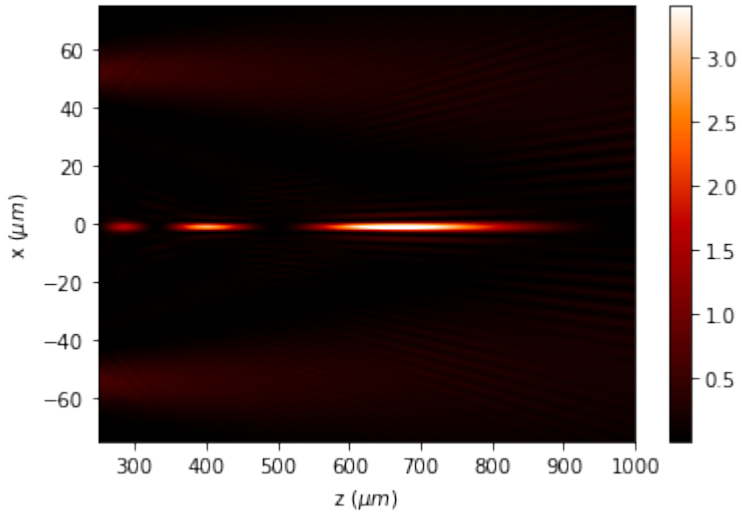


to field_XZ

[26]:
```
xout = np.linspace(-size, size, 128)
yout = 0.
z = np.linspace(0.25 * mm, 1 * mm, 128)
```

[27]:
```
%%time
u2 = u1.CZT(z, xout, yout, verbose=True)
u2.draw()
plt.colorbar()
```

```
num x, num y, num z = 128, 1, 128
CPU times: user 7.31 s, sys: 1.9 s, total: 9.22 s
Wall time: 9.22 s
```



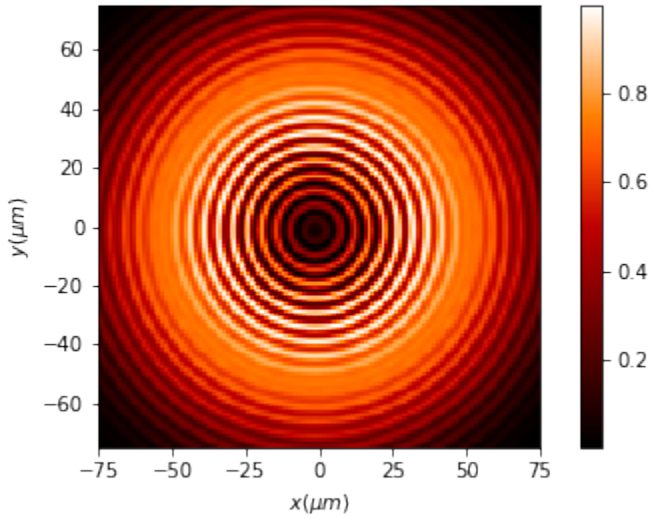
to field_XYZ

[28]:
```
xout = np.linspace(-size, size, 128)
yout = np.linspace(-size, size, 128)
z = np.linspace(0.25 * mm, 6 * mm, 64)
```

[29]:
```
%%time
u2 = u1.CZT(z, xout, yout, verbose=True)
```

```
num x, num y, num z = 128, 128, 64
CPU times: user 3.86 s, sys: 815 ms, total: 4.68 s
Wall time: 4.68 s
```

[30]:
```
u2.draw_XY(z0=1 * mm)
plt.colorbar()
```



6.1.5.2 CZT for reducing the output size

Chirped z-Transform algorithm is specially indicated for cases where the output field is much smaller than the input field, as you can choose the position and sampling of the field. An important example for this is the focusing of a lens.

X scheme

[32]:
```
size = 3 * mm
xin = np.linspace(-size, size, 4096)
wavelength = 550 * nm
focal = 250 * mm
```

[33]:
```
%load_ext autoreload
%autoreload 2
```

[34]:
```
t0 = Scalar_mask_X(xin, wavelength)
t0.lens(x0=0, focal=focal)

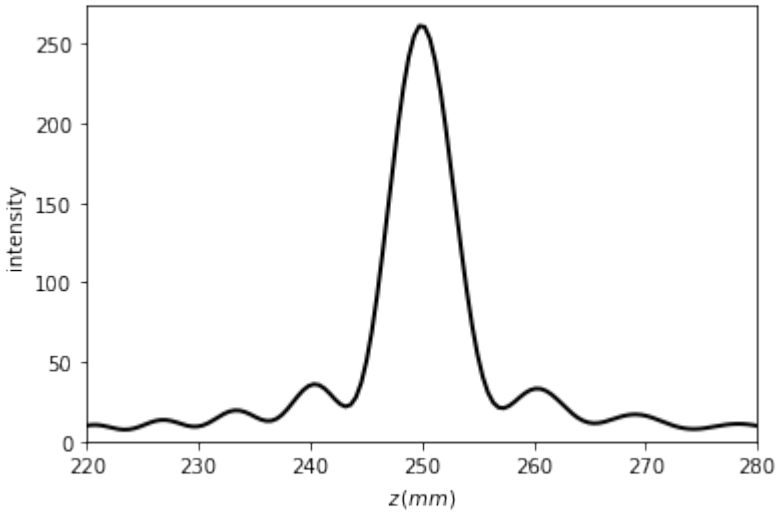
u0 = Scalar_source_X(xin, wavelength)
u0.plane_wave(A=1)

u1 = t0 * u0
```

[35]:
```
xout = 0.
z = np.linspace(focal - 30 * mm, focal + 30 * mm, 128)
```

[36]:
```
%%time
u2 = u1.CZT(z, xout, verbose=True)
u2.draw(z_scale='mm')
```

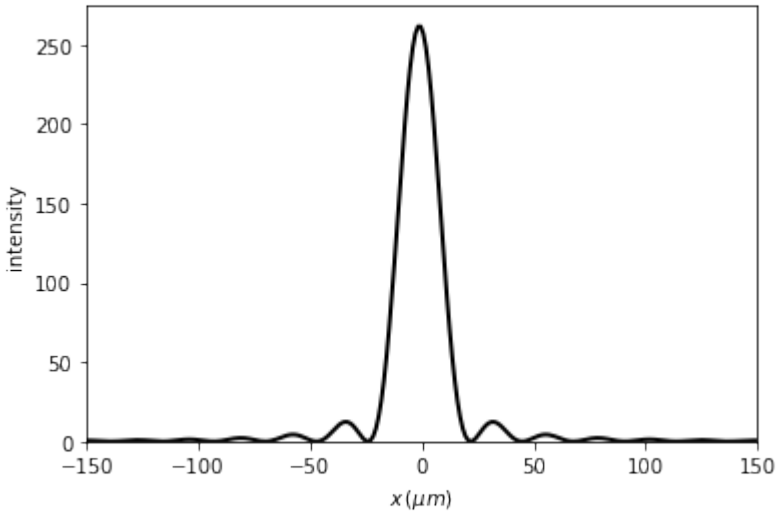
```
CPU times: user 591 ms, sys: 4.43 ms, total: 595 ms
Wall time: 596 ms
```



[37]:
```
xout = np.linspace(-150 * um, 150 * um, 256)
z = focal
```

[38]:
```
%%time
u2 = u1.CZT(focal, xout)
u2.draw()
```

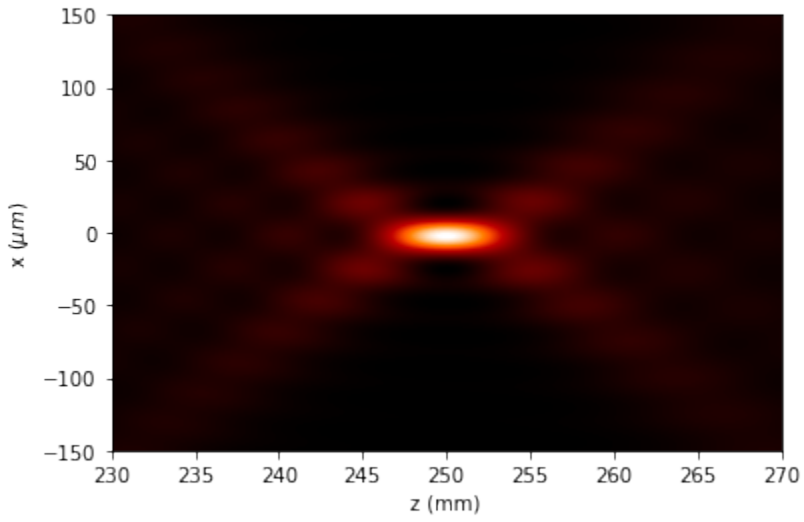
```
CPU times: user 28.9 ms, sys: 168 µs, total: 29.1 ms
Wall time: 28.4 ms
```



[39]:
```
xout = np.linspace(-150 * um, 150 * um, 128)
z = np.linspace(focal - 20 * mm, focal + 20 * mm, 128)
```

[40]:
```
%%time
u2 = u1.CZT(z, xout, verbose=True)
u2.draw(logarithm=0, z_scale='mm')
```

```
CPU times: user 1.17 s, sys: 2.44 ms, total: 1.17 s
Wall time: 1.16 s
```



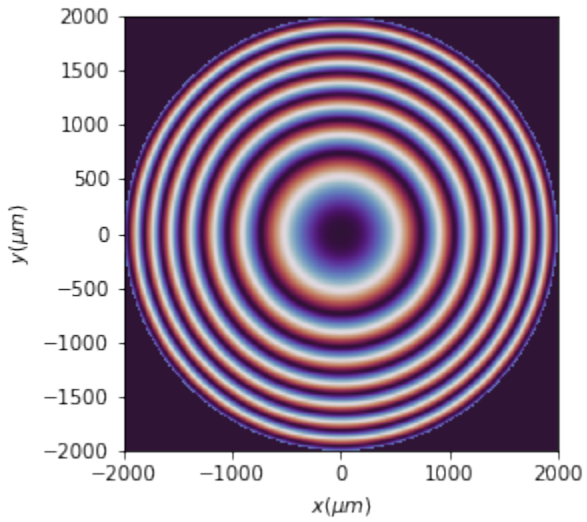
XY scheme

[41]:
```
size = 2 * mm
xin = np.linspace(-size, size, 512)
yin = np.linspace(-size, size, 512)
wavelength = 550 * nm
focal = 500 * mm
```

[42]:
```
t0 = Scalar_mask_XY(xin, yin, wavelength)
t0.lens(r0=(0, 0), focal=focal, radius=0)
t0.pupil()

u0 = Scalar_source_XY(xin, yin, wavelength)
u0.plane_wave(A=1)

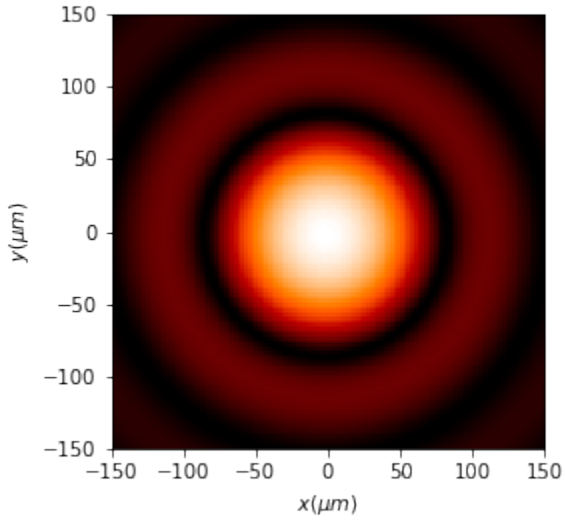
u1 = t0 * u0
```



[43]:
```
xout = np.linspace(-150 * um, 150 * um, 128)
yout = np.linspace(-150 * um, 150 * um, 128)
z = focal
```

[44]:
```
%%time
u2 = u1.CZT(z, xout, yout, verbose=True)
u2.draw(logarithm=1e-1)
```

```
num x, num y, num z = 128, 128, 1
CPU times: user 408 ms, sys: 279 ms, total: 688 ms
Wall time: 336 ms
```



6.1.6 Vector Fast Fourier Transform (VFFT)

The incident field is an vector field (Ex, Ey) and it is propagated and converted to (Ex,Ey,Ez)

[1]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees
```

[2]:
```
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

[3]:
```
from diffractio.vector_fields_XY import Vector_field_XY
from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
```

6.1.6.1 XY scheme

[4]:
```
size = 64 * um
x0 = np.linspace(-size / 2, size / 2, 1024)
y0 = np.linspace(-size / 2, size / 2, 1024)

wavelength = 0.6328 * um
```

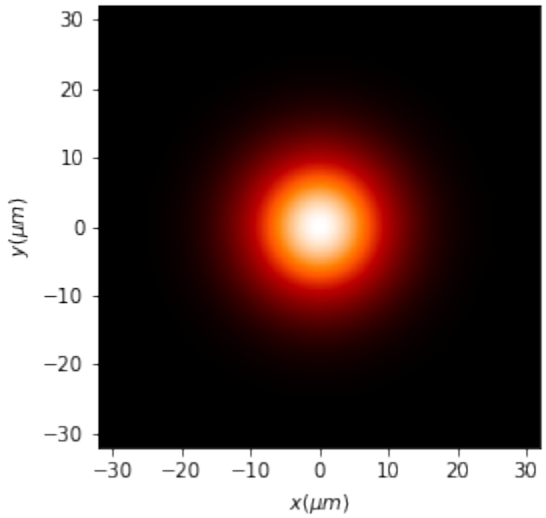
[5]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(16 * um, 16 * um),
              z0=0 * um,
              A=1,
```

(continues on next page)

(continued from previous page)

```
                theta=0. * degrees,
                phi=0 * degrees)

u0.draw()
```

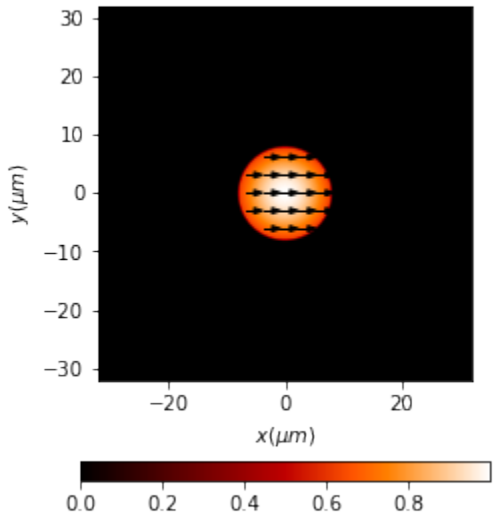


[6]:
```
radius = 8 * um
t = Scalar_mask_XY(x0, y0, wavelength)
t.circle(r0=(0, 0), radius=radius)
```

[7]:
```
u1 = t * u0

EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.constant_polarization(u=u1, v=(1, 0))
```

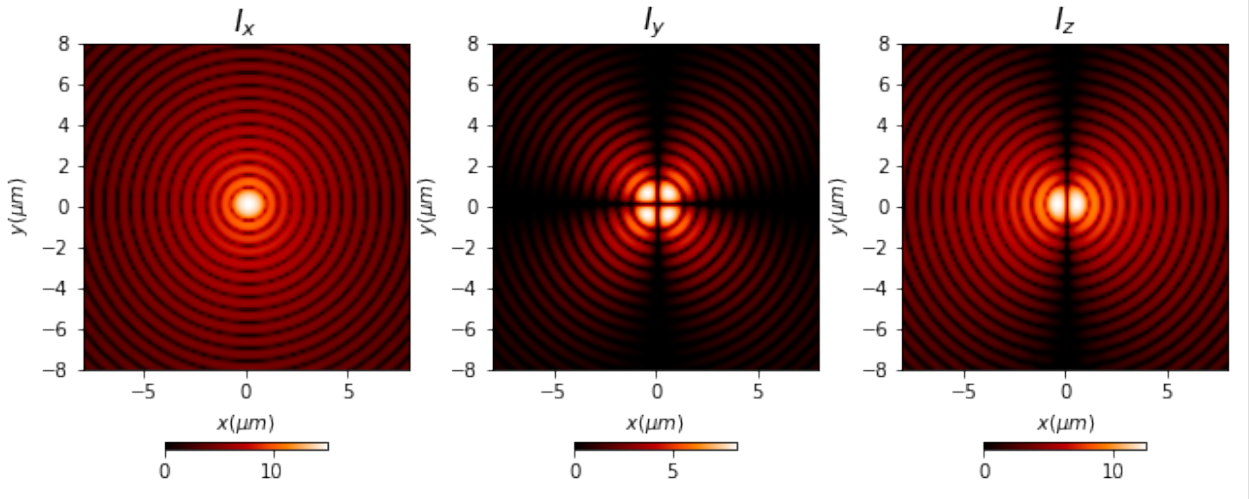
[8]:
```
EM1.draw(kind='ellipses',
         amplification=0.5,
         color_line='k',
         line_width=0.75,
         num_ellipses=(21, 21))
```



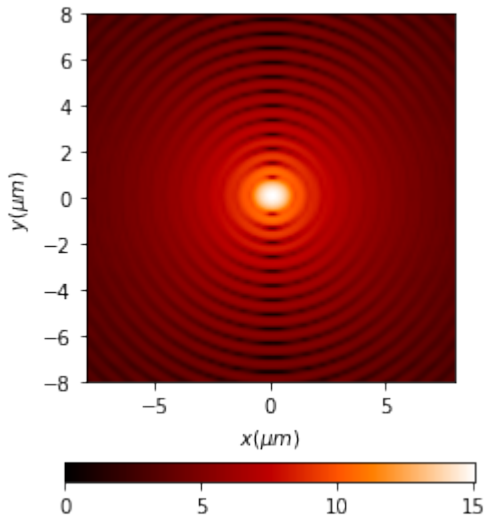
[9]:
```
EM2 = EM1.VFFT(radius=64 * um,
               focal=10 * um,
               remove0=False,
               n=1,
               new_field=True,
               has_draw=False)
```

[10]:
```
x_resample = 8
EM2.cut_resample([-x_resample, x_resample], [-x_resample, x_resample],
                 num_points=(1024, 1024))
```

[11]:
```
EM2.draw('intensities', logarithm=1e-1)
```

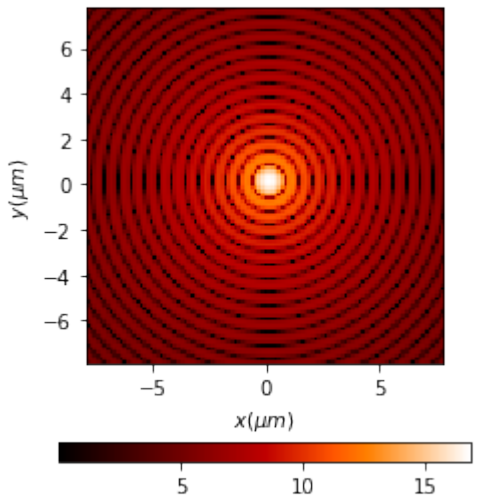


[12]:
```
EM2.draw('intensity', logarithm=1e-1)
```



This result can be compared to the scalar FFT result.

[13]:
```
u_fft = u1.fft(z=15 * um, new_field=True, remove0=False)
u_fft.cut_resample(x_limits=(-x_resample, x_resample),
                   y_limits=(-x_resample, x_resample))
u_fft.draw(logarithm=1e-2, has_colorbar='horizontal')
```



[ ]:

6.1.7 Vector Rayleigh-Sommerfeld method (VRS)

[1]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees
from matplotlib import cm

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

from diffractio.vector_fields_XY import Vector_field_XY
```

6.1.7.1 Radial polarized Gaussian beam through a circular aperture

[2]:
```
size = 6.4 * um
x0 = np.linspace(-size, size, 512)
y0 = np.linspace(-size, size, 512)
```

[3]:
```
wavelength = .640 * um

radius = 10 * wavelength / 2
z_obs = 10 * um
beam_width = 6 * um / (2 * np.sqrt(2))
```

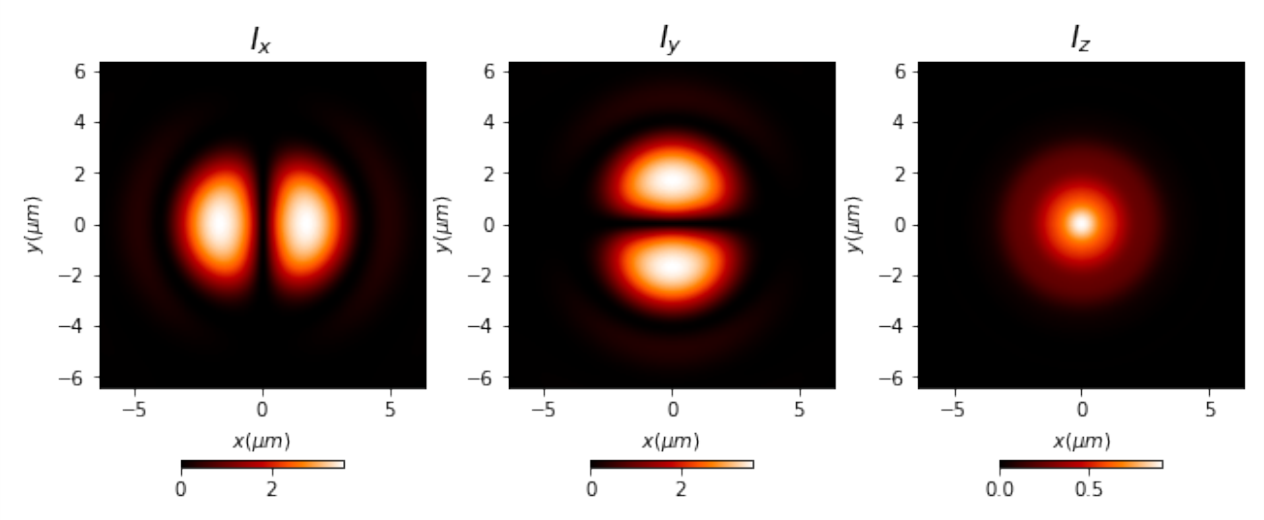
[4]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(A=1, r0=(0, 0), z0=0, w0=beam_width)

t0 = Scalar_mask_XY(x0, y0, wavelength)
t0.circle(r0=(0, 0), radius=radius)

t = u0 * t0
```

[5]:
```
E0 = Vector_source_XY(x0, y0, wavelength)
E0.radial_wave(u=t, r0=(0, 0))
```

[6]:
```
E1 = E0.VRS(z=z_obs, n=1, new_field=True)
E1.draw('intensities', logarithm=1e2)
```



6.1.7.2 Focusing with a lens a beam with a high E_z component

In Figures 4 and 5 a planar lens formed by rings is developed in order to obtain a high E_z component. The data of the radii are shown in Table 1.

[7]:
```
size = 30 * um
x0 = np.linspace(-size, size, 1024)
y0 = np.linspace(-size, size, 1024)
```

[8]:
```
wavelength = .6328 * um
focal = 10 * um
```

[9]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)

t0 = Scalar_mask_XY(x0, y0, wavelength)
t0.lens(r0=(0, 0), focal=focal, radius=20 * um)

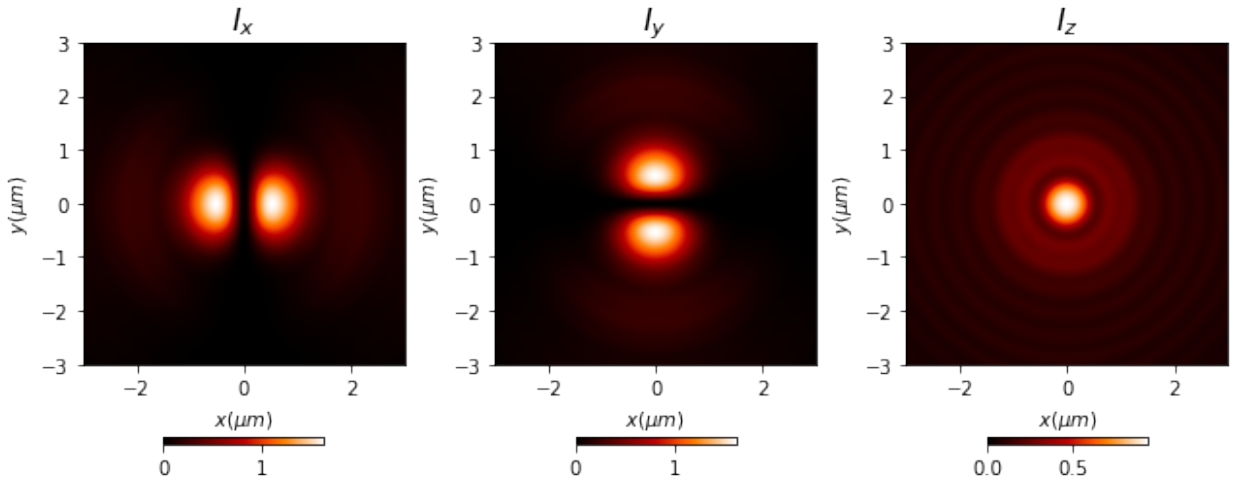
t1 = u0 * t0
```

[11]:
```
E0 = Vector_source_XY(x0, y0, wavelength)
E0.radial_wave(u=t1, r0=(0, 0))
```

[12]:
```
%%time
E1=E0.VRS(z=z_obs, n=1, new_field=True)
E1.cut_resample([-3, 3], [-3, 3], num_points=[512, 512])
```

```
CPU times: user 5.57 s, sys: 336 ms, total: 5.91 s
Wall time: 5.91 s
```

[13]:
```
E1.draw('intensities', logarithm=1e-1)
```



6.1.8 Vector Chirped Z-transform (VCZT)

[1]:
```
%load_ext autoreload
%autoreload 2
```

[2]:
```
from diffractio import degrees, mm, um, nm
from diffractio import np, plt, sp

from diffractio.scalar_fields_XY import Scalar_field_X
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

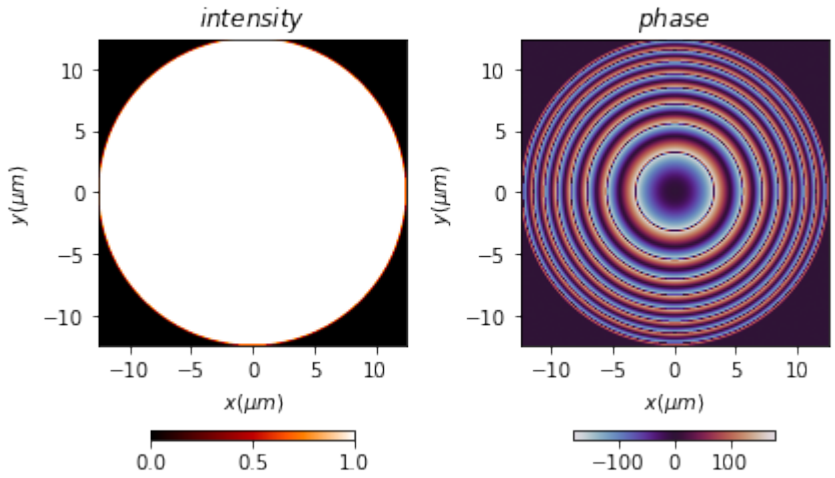
from diffractio.vector_fields_XY import Vector_field_XY
from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
```

[3]:
```
size = 25 * um
x0 = np.linspace(-size / 2, size / 2, 512)
y0 = np.linspace(-size / 2, size / 2, 512)

focal = 16 * um
wavelength = 0.6328 * um
```

[4]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(32 * um, 32 * um),
              z0=0 * um,
              A=1,
              theta=0. * degrees,
              phi=0 * degrees)
```

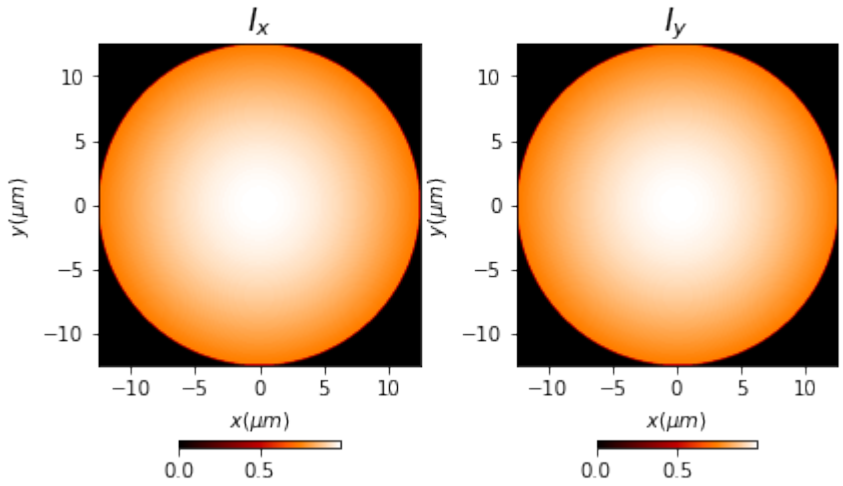
[5]:
```
t = Scalar_mask_XY(x0, y0, wavelength)
t.lens(r0=(0, 0), focal=focal, radius=size / 2)
```



[6]:
```
u1 = t * u0

EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.constant_polarization(u=u1, v=(1, 1j))
```

[8]:
```
EM1.draw('intensities', logarithm=0)
```



6.1.8.1 to value

[9]:
```
x_out = 0.
y_out = 0.
z = focal
```

[10]:
```
%%time
EM2_1D=EM1.CZT(z, x_out, y_out, verbose=True)
print(EM2_1D)
```

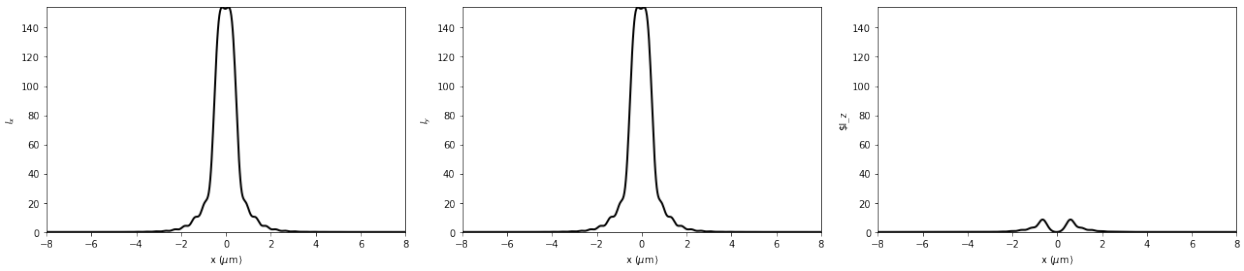
```
((12.01931943234928-2.910555097903726j), (2.910555097903726+12.01931943234928j), (-5.
↪329070518200751e-15+3.2307490016592055e-14j))
CPU times: user 198 ms, sys: 47.6 ms, total: 246 ms
Wall time: 245 ms
```

6.1.8.2 to x_field

[11]:
```
size_out = 16 * um
x_out = np.linspace(-size_out / 2, size_out / 2, 512)
y_out = 0.
z = focal
```

[12]:
```
%%time
EM2_1D=EM1.CZT(z, x_out, y_out)
EM2_1D.draw('intensities');
```

```
CPU times: user 765 ms, sys: 339 ms, total: 1.1 s
Wall time: 689 ms
```



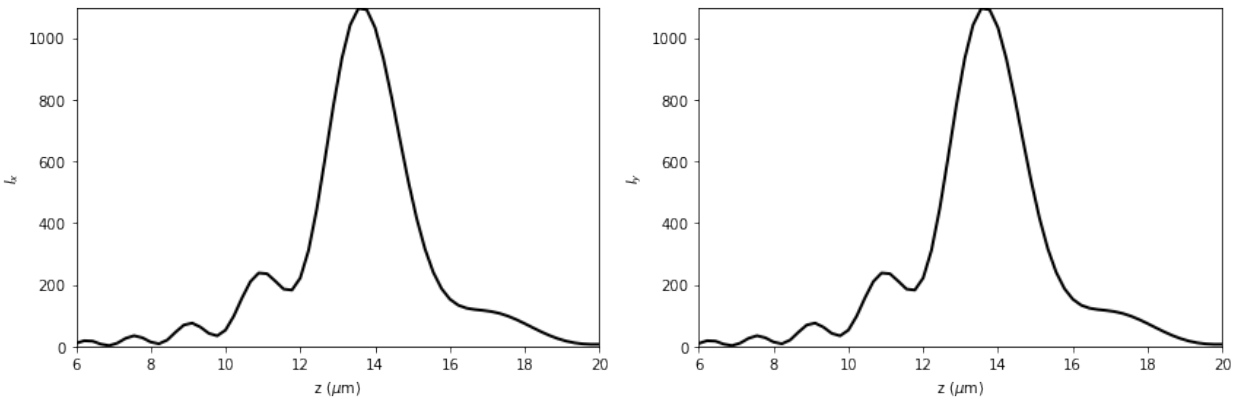
6.1.8.3 to z_field

[13]:
```
x_out = 0.
y_out = 0.
z = np.linspace(focal - 4 * um - 6 * um, focal - 2 * um + 6 * um, 64)
```

[14]:
```
%%time
EM2=EM1.CZT(z, x_out, y_out, verbose=False);
```

```
CPU times: user 11.1 s, sys: 304 ms, total: 11.4 s
Wall time: 11.4 s
```

[15]:
```
EM2.draw('intensities')
```



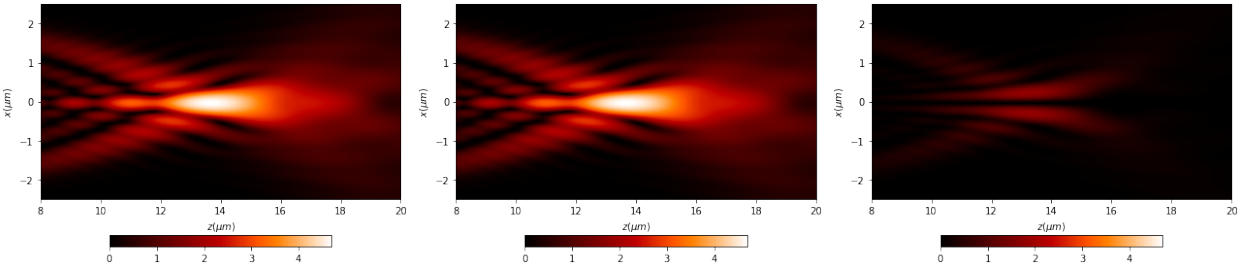
6.1.8.4 to XZ_field

[16]:
```
size_out = 5 * um
x_out = np.linspace(-size_out / 2, size_out / 2, 128)
y_out = 0.
z = np.linspace(focal - 2 * um - 6 * um, focal - 2 * um + 6 * um, 64)
```

[17]:
```
%%time
EM2=EM1.CZT(z, x_out, y_out)
```

```
CPU times: user 10.3 s, sys: 1.38 s, total: 11.6 s
Wall time: 11.7 s
```

[19]:
```
EM2.draw('intensities', logarithm=1e-1)
```



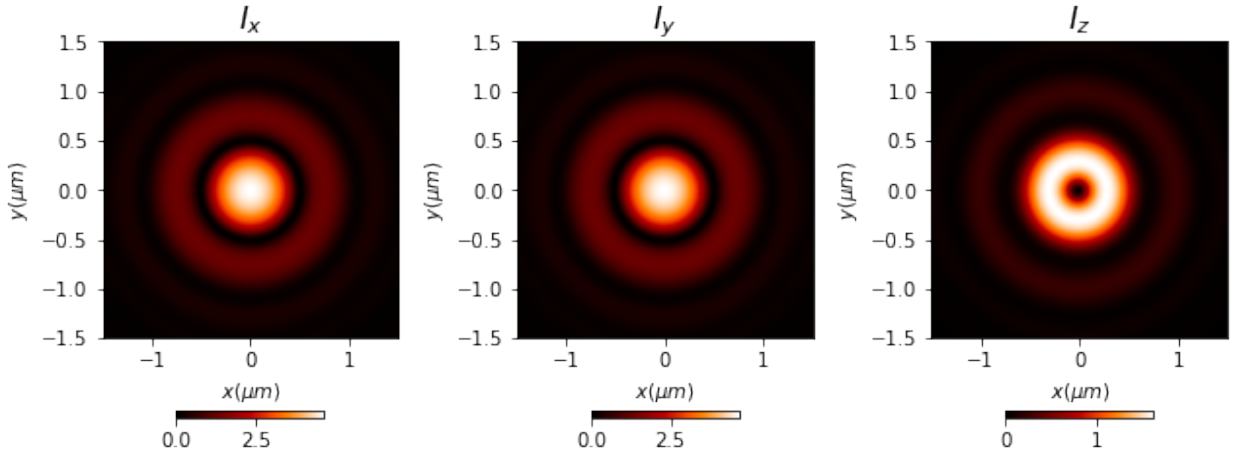
6.1.8.5 to XY_field

[21]:
```
size_out = 3 * um
x_out = np.linspace(-size_out / 2, size_out / 2, 256)
y_out = np.linspace(-size_out / 2, size_out / 2, 256)
z = focal - 2
```

[22]:
```
%%time
EM2=EM1.CZT(z, x_out, y_out)
```

```
CPU times: user 273 ms, sys: 48.1 ms, total: 321 ms
Wall time: 323 ms
```

[23]:
```
EM2.draw('intensities', logarithm=1e-1)
```



6.1.8.6 to XYZ_field

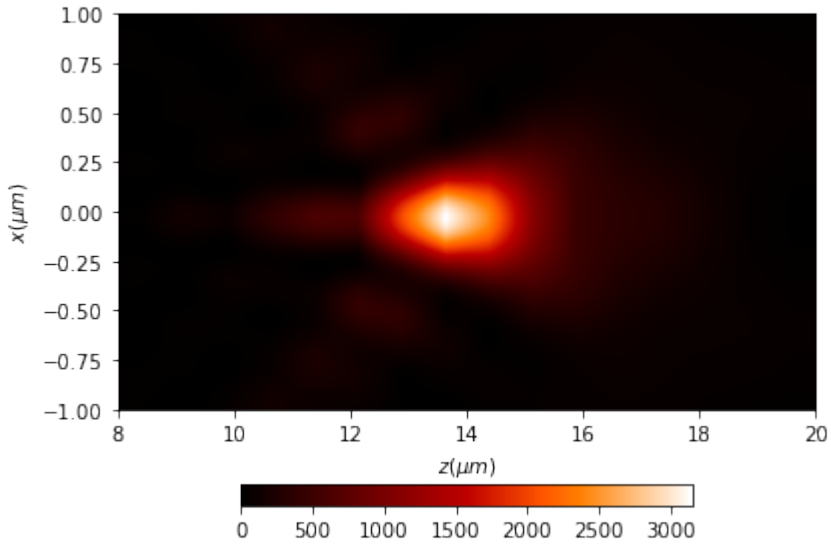
[29]:
```
size_out = 2 * um
x_out = np.linspace(-size_out / 2, size_out / 2, 64)
y_out = np.linspace(-size_out / 2, size_out / 2, 64)
z = np.linspace(focal - 2 * um - 6 * um, focal - 2 * um + 6 * um, 16)
```

[30]:
```
%%time
EM2=EM1.CZT(z, x_out, y_out)
```

```
CPU times: user 2.97 s, sys: 407 ms, total: 3.37 s
Wall time: 3.38 s
```

[31]:
```
EM2.draw_XZ(y0=0, logarithm=False)
```

```
<Figure size 432x288 with 0 Axes>
```



6.1.9 Comparison between Rayleigh-Sommerfeld (RS) and WPM Propagation Method (WPM)

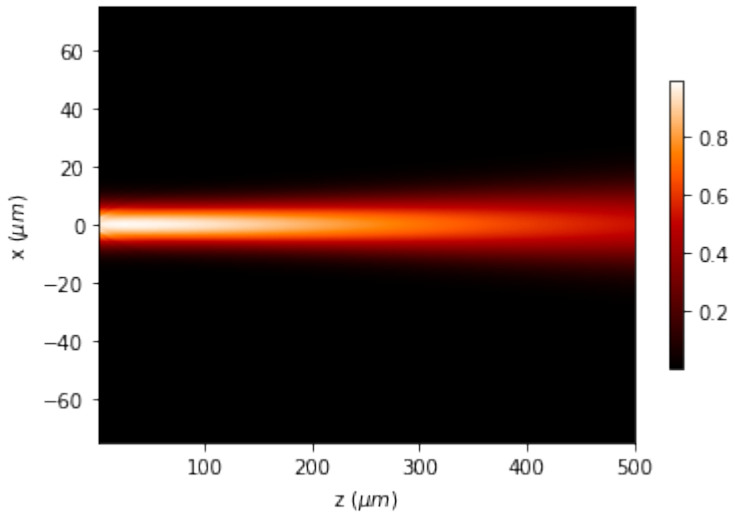
[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
```

RS

[2]:
```
length = 150 * um
wavelength = 1 * um
x0 = np.linspace(-length / 2, length / 2, 1024)
z0 = np.linspace(2 * um, 500 * um, 512)

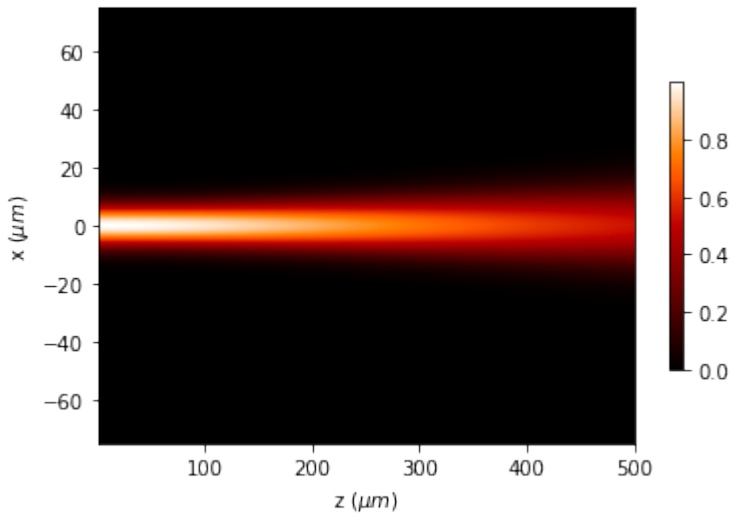
# source
f1 = Scalar_source_X(x0, wavelength)
f1.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=10 * um, theta=0 * degrees)
```

[3]:
```
u_rs = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
u_rs.incident_field(f1)
u_rs.RS()
u_rs.draw(kind='intensity', logarithm=False, colorbar_kind='vertical')
```



WPM

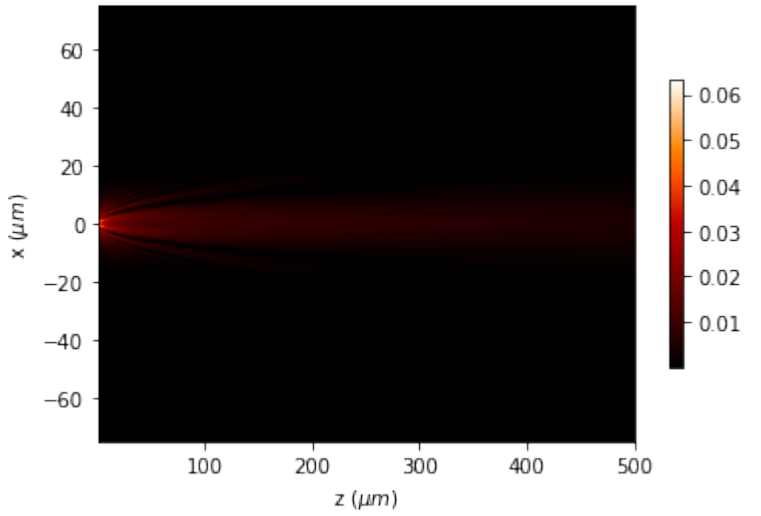
[4]:
```
u_wpm = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
u_wpm.incident_field(f1)
u_wpm.WPM(verbose=False)
u_wpm.draw(kind='intensity',
           logarithm=False,
           normalize='intensity',
           colorbar_kind='vertical')
```



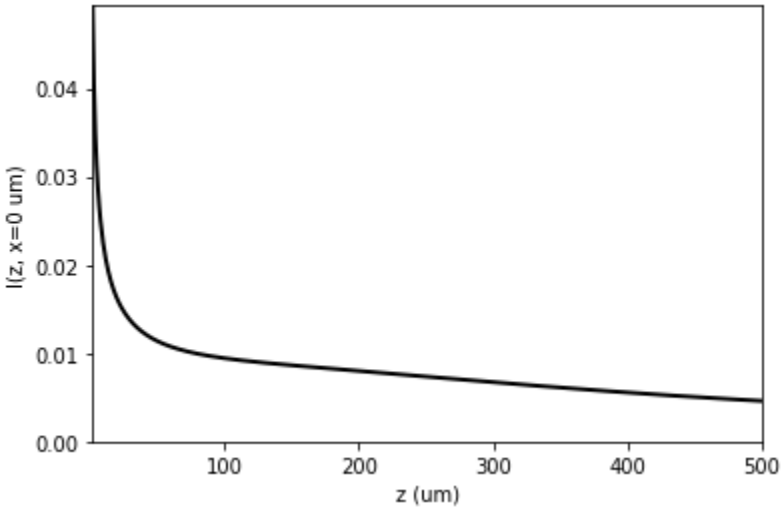
Differences

Here we show the relative difference between BPM and RS

[5]:
```
u_diff = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
u_diff.u = np.sqrt(np.abs(np.abs(u_rs.u)**2 - np.abs(u_wpm.u)**2))
u_diff.draw(kind='intensity', logarithm=False, colorbar_kind='vertical')
```



[6]:
```
profile_error = u_diff.profile_longitudinal()
plt.ylim(ymin=0)
```



[ ]:
```
```

6.1.10 Comparison between Wave propagation Method (WPM) and Beam Propagation Method (BPM)

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X

from matplotlib import cm
```

(continues on next page)

(continued from previous page)

[2]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024 * 4)
z0 = np.linspace(-0.1 * mm, 1.25 * mm, 1024 * 2)
wavelength = 25 * um

logarithm_drawing = 0
```

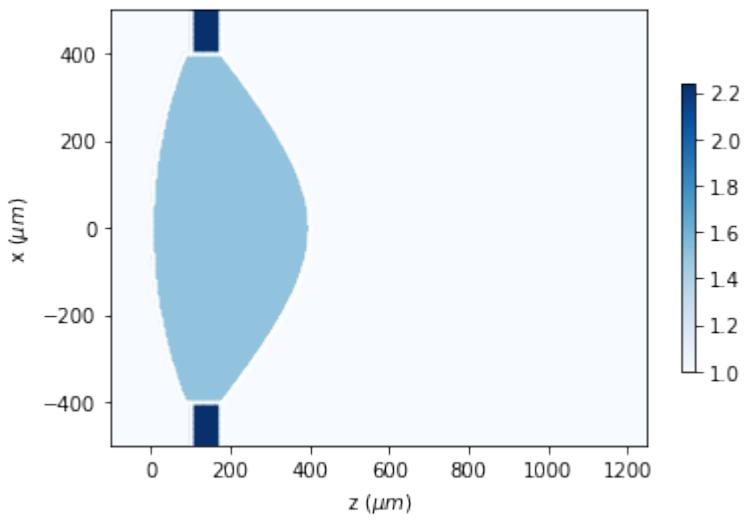
[3]:
```
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

BPM

[4]:
```
lens = Scalar_mask_XZ(x0, z0, wavelength, n_background=1, info='')
ipasa, conds = lens.aspheric_lens(r0=(0 * mm, 0 * mm),
                                  angle=(0 * degrees, (0 * mm, 0 * mm)),
                                  refractive_index=1.5,
                                  cx=(1 / (1 * mm), -1 / (.25 * mm)),
                                  Qx=(0, 0),
                                  a2=(0, 1e-13),
                                  a3=(0, 0),
                                  a4=(0, 0),
                                  depth=.4 * mm,
                                  size=0.8 * mm)

lens.slit(r0=(0, 100 * um),
          aperture=800 * um,
          depth=75 * um,
          refractive_index=1 + 2j)

lens.draw_refractive_index(draw_borders=True,
                           min_incr=0.01,
                           colorbar_kind='vertical')
```



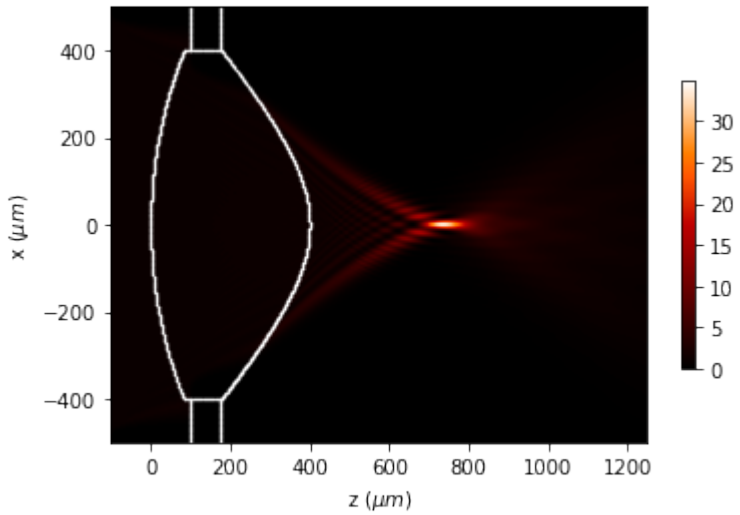
[5]:
```
lens.smooth_refractive_index(type_filter=2, pixels_filtering=25)
```

[6]:
```
lens.incident_field(u0)
```

[7]:
```
%%time
lens.clear_field()
lens.BPM(verbose=False)
```

```
CPU times: user 1.33 s, sys: 72.1 ms, total: 1.41 s
Wall time: 1.4 s
```

[8]:
```
lens.draw(kind='intensity',
          logarithm=logarithm_drawing,
          draw_borders=True,
          colorbar_kind='vertical')
```



[9]:
```
Intensity_BPM = lens.intensity()
```

Focusing area

[10]:
```
x_f_bmp, z_f_bmp = lens.search_focus()
```

```
x = -0.122 um, z = 736.908 um
```

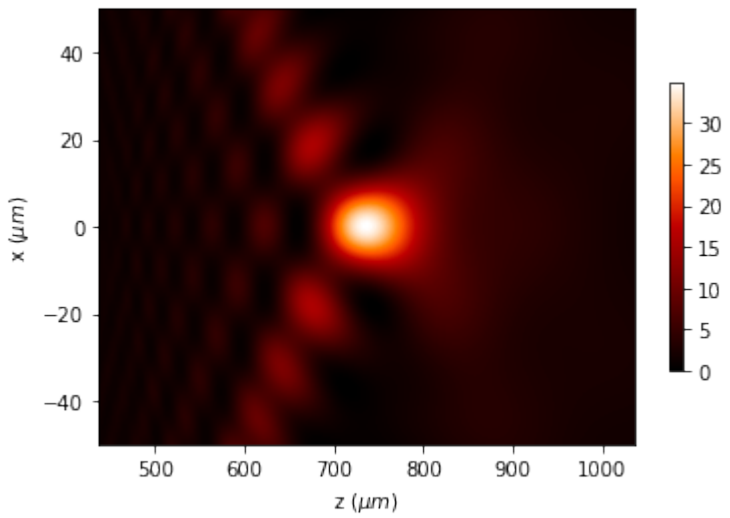
[11]:
```
ylim_max = 50 * um
zlim_max = 300 * um
```

[12]:
```
lens.draw(kind='intensity',
          logarithm=logarithm_drawing,
          colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
```

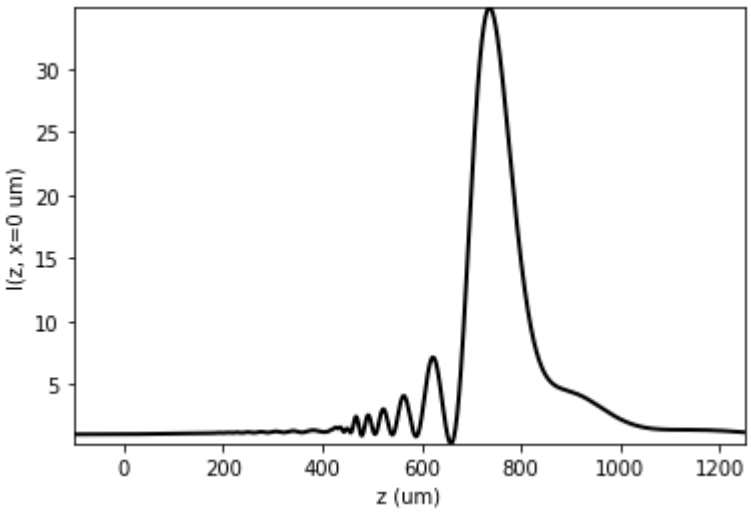
(continues on next page)

(continued from previous page)

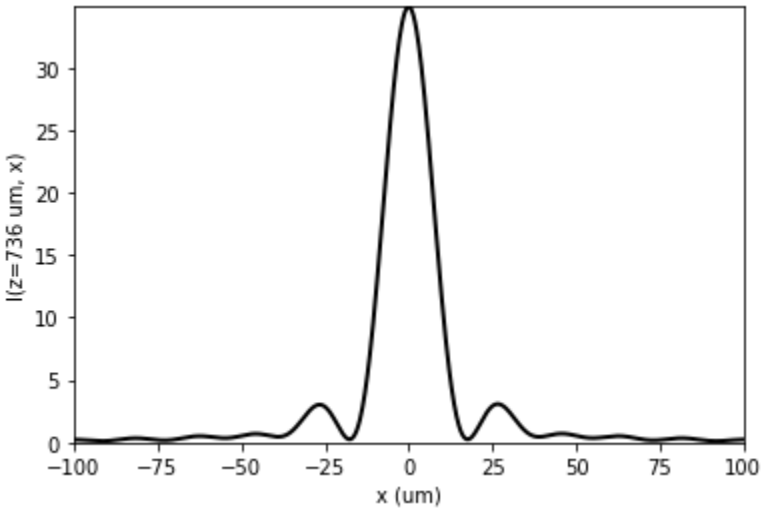
```
plt.xlim(z_f_bmp - zlim_max, z_f_bmp + zlim_max)
```



[13]:
```
lens.profile_longitudinal(kind='intensity', logarithm=logarithm_drawing, x0=0)
```



[14]:
```
lens.profile_transversal(kind='intensity',
                         logarithm=logarithm_drawing,
                         z0=z_f_bmp)
plt.xlim(-100, 100)
```



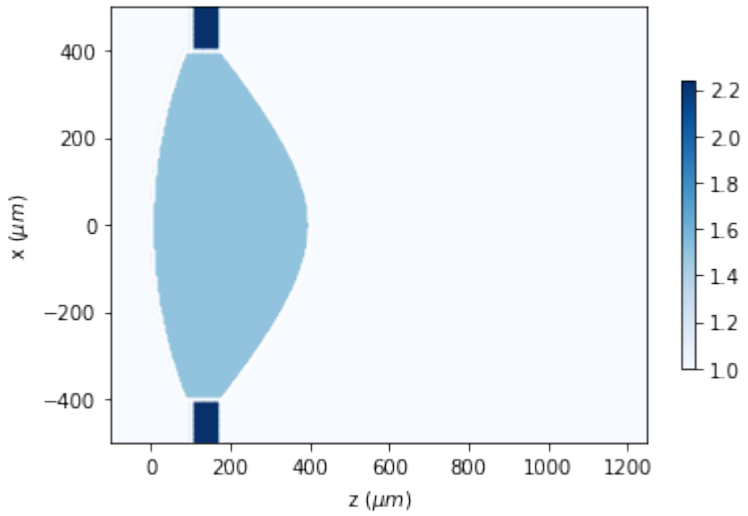
WPM

[15]:
```
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

[16]:
```
lens2 = Scalar_mask_XZ(x0, z0, wavelength, n_background=1, info='')
ipasa, conds = lens2.aspheric_lens(r0=(0 * mm, 0 * mm),
                                   angle=(0 * degrees, (0 * mm, 0 * mm)),
                                   refractive_index=1.5,
                                   cx=(1 / (1 * mm), -1 / (.25 * mm)),
                                   Qx=(0, 0),
                                   a2=(0, 1e-13),
                                   a3=(0, 0),
                                   a4=(0, 0),
                                   depth=.4 * mm,
                                   size=0.8 * mm)

lens2.slit(r0=(0, 100 * um),
           aperture=800 * um,
           depth=75 * um,
           refractive_index=1 + 2j)

lens2.draw_refractive_index(draw_borders=True,
                            min_incr=0.01,
                            colorbar_kind='vertical')
```

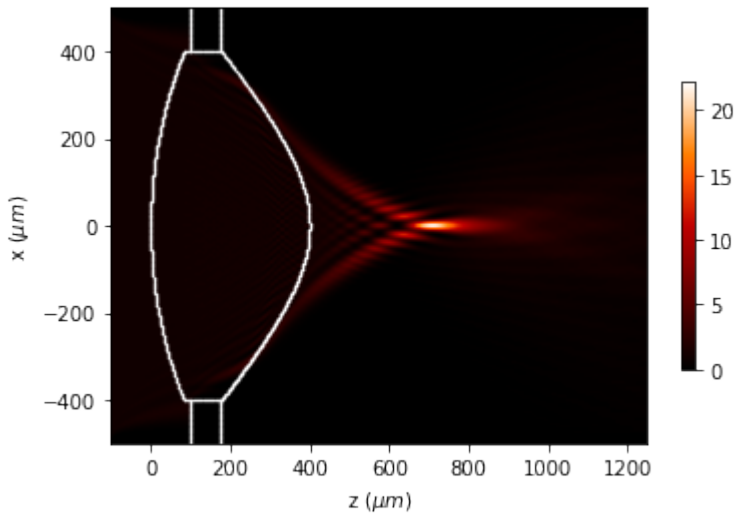


[17]:
```
lens2.smooth_refractive_index(type_filter=2, pixels_filtering=25)
lens2.discretize_refractive_index(num_layers=6)
```

[18]:
```
lens2.incident_field(u0)
```

[19]:
```
lens2.clear_field()
lens2.WPM(verbose=False)
```

[20]:
```
lens2.draw(kind='intensity',
           logarithm=logarithm_drawing,
           normalize=None,
           draw_borders=True,
           colorbar_kind='vertical')
```



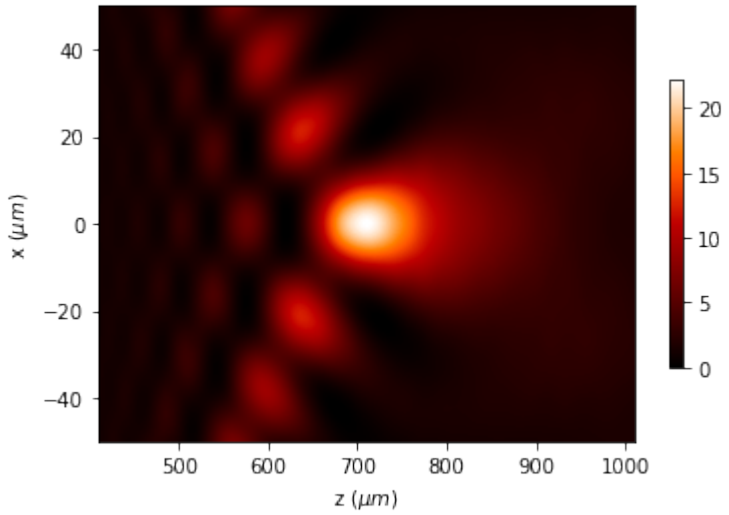
[21]:
```
Intensity_WPM = lens2.intensity()
```

[22]:
```
x_f_wpm, z_f_wpm = lens2.search_focus()
```

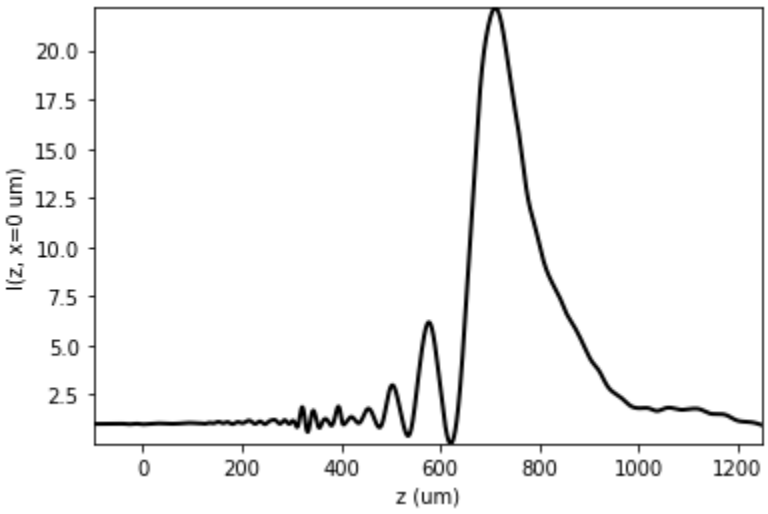
```
x = -0.122 um, z = 710.528 um
```

Focusing area

[23]:
```
lens2.draw(kind='intensity',
           logarithm=logarithm_drawing,
           colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_wpm - zlim_max, z_f_wpm + zlim_max)
```



[24]:
```
lens2.profile_longitudinal(kind='intensity', logarithm=logarithm_drawing, x0=0)
```

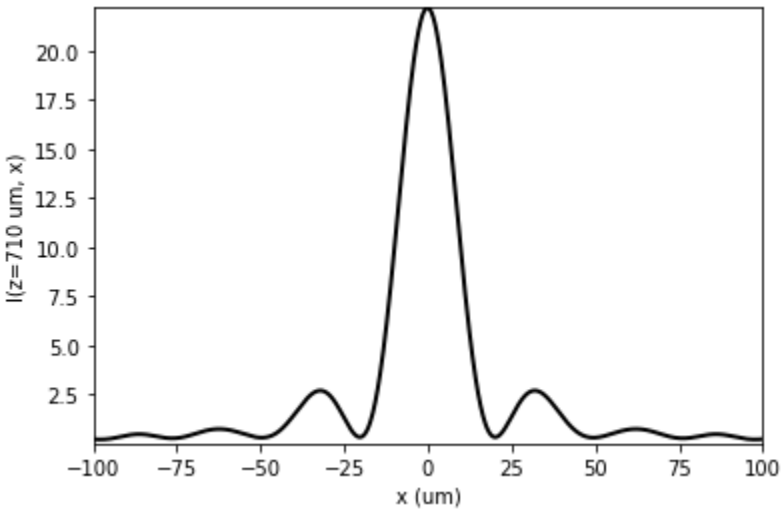


[25]:
```
lens2.profile_transversal(kind='intensity',
                          logarithm=logarithm_drawing,
```

(continues on next page)

(continued from previous page)

```
                              z0=z_f_wpm)
plt.xlim(-100, 100)
```



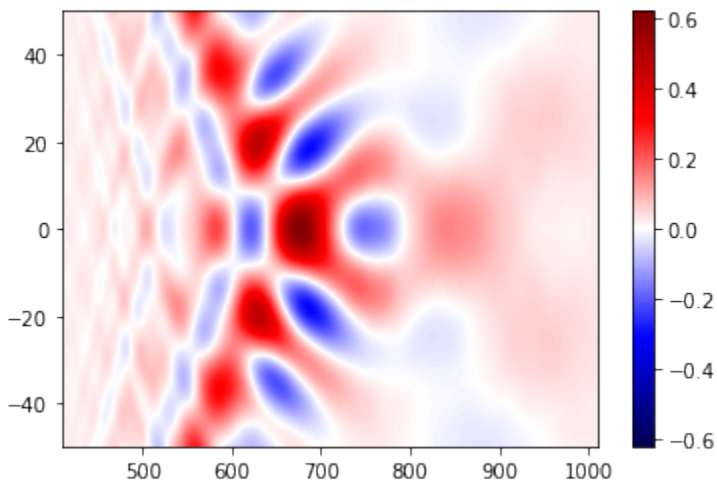
6.1.10.1 Diferences

[26]:
```
Intensity_diff = (Intensity_WPM / Intensity_WPM.max() -
                  Intensity_BPM / Intensity_BPM.max())
```

[27]:
```
extension = [lens2.z[0], lens2.z[-1], lens2.x[0], lens2.x[-1]]

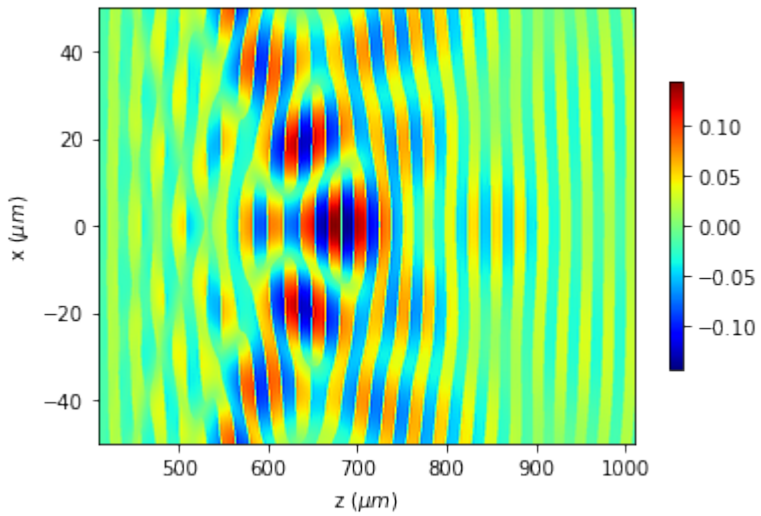
h1 = plt.imshow(Intensity_diff,
                interpolation='bilinear',
                aspect='auto',
                origin='lower',
                extent=extension)
plt.axis(extension)

h1.set_cmap(cm.seismic)  # OrRd # Reds_r gist_heat
h1.set_clim(-Intensity_diff.max(), Intensity_diff.max())
plt.colorbar(orientation='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_wpm - zlim_max, z_f_wpm + zlim_max)
```

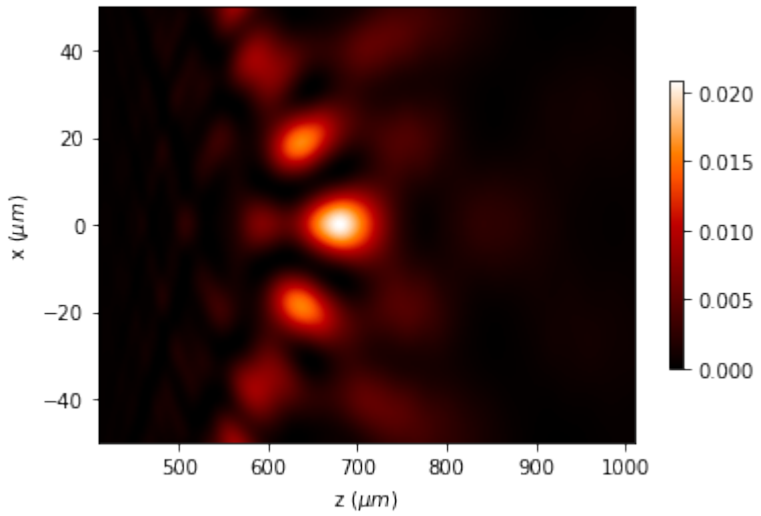


[28]:
```
diff = Scalar_mask_XZ(x0, z0, wavelength, n_background=1, info='')
diff.u = lens2.u / Intensity_WPM.max() - lens.u / Intensity_BPM.max()

diff.draw(kind='amplitude', logarithm=False, colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_wpm - zlim_max, z_f_wpm + zlim_max)
```



[29]:
```
diff.draw(kind='intensity', logarithm=False, colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f_wpm - zlim_max, z_f_wpm + zlim_max)
```



[30]:
```
# focal shift

focal_shift = z_f_wpm - z_f_bmp

print("Dz = {:2.2f} um".format(focal_shift))
```

```
Dz = -26.38 um
```

6.1.11 Comparison between Rayleigh-Sommerfeld (RS) and Chirped Z-transform (CZT)

In this example, we will analyze how CZT algorithm produces better results than in RS when the region of interest in the focus of a lens.

6.1.11.1 X Scheme

[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
```

[2]:
```
focal = 25 * mm
diameter = 4 * mm

# Initial parameters
x0 = np.linspace(-diameter / 2 - 50 * um, diameter / 2 + 50 * um, 4096)
wavelength = 0.6238 * um

# Definition of source
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1)
```

(continues on next page)

(continued from previous page)

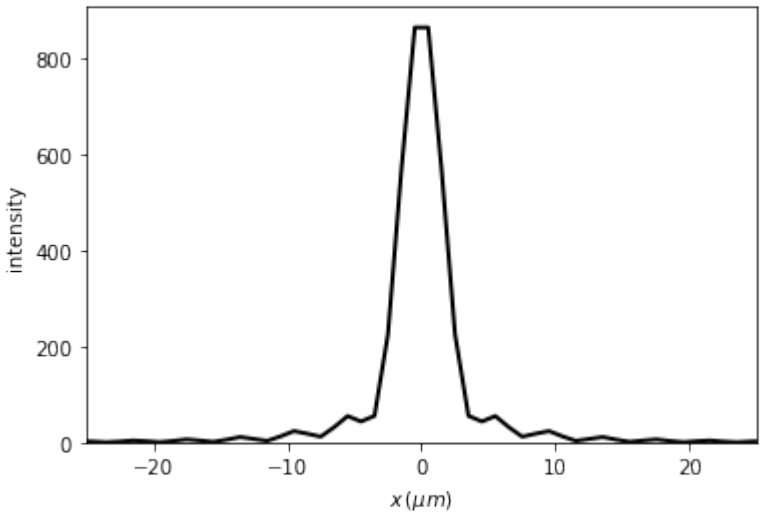
```
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.lens(x0=0.0, focal=focal, radius=diameter / 2)

u1 = t0 * u0
```

RS

[3]:
```
%%time
u2_rs = u1.RS(z=focal, verbose=False)
u2_rs.draw()
plt.xlim(-25, 25)
```

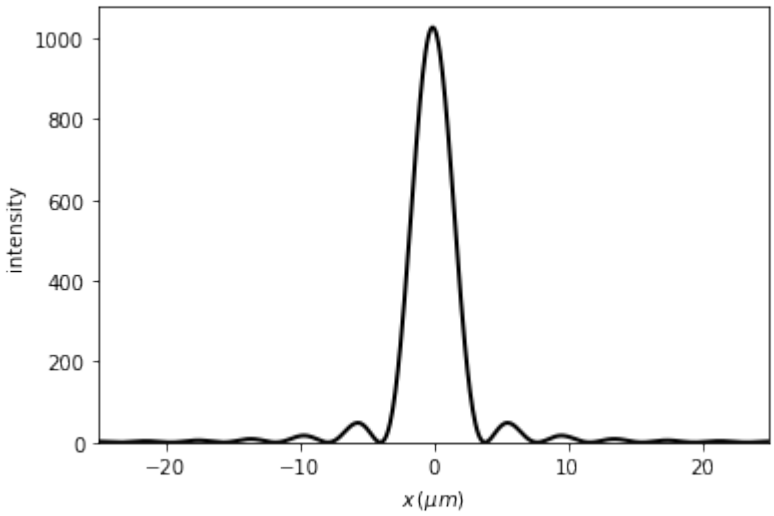
```
CPU times: user 248 ms, sys: 3.67 ms, total: 251 ms
Wall time: 277 ms
```



CZT

[4]:
```
%%time
xout=np.linspace(-25,25,512)
u2_czt = u1.CZT(z=focal, xout=xout)
u2_czt.draw()
```

```
CPU times: user 59.5 ms, sys: 0 ns, total: 59.5 ms
Wall time: 71.4 ms
```



6.1.11.2 XZ scheme

[5]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

[6]:
```
focal = 15 * mm
diameter = 4 * mm

x0 = np.linspace(-diameter / 2, diameter / 2, 1024)
wavelength = 0.6238 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)

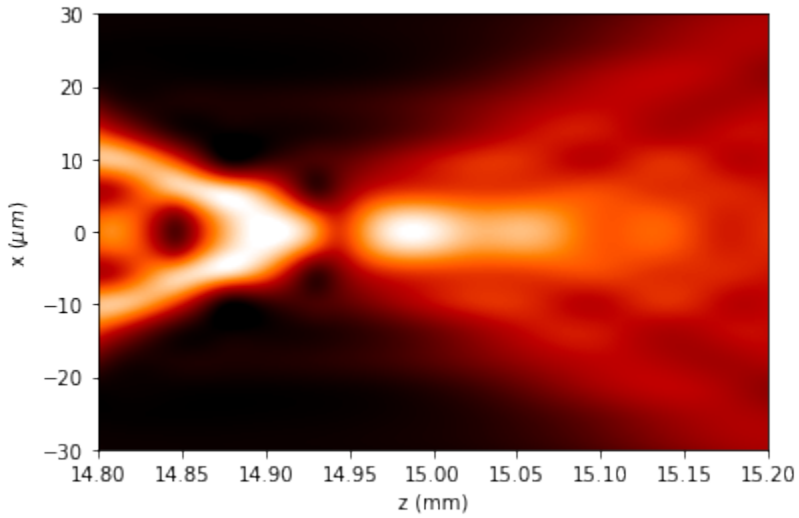
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.lens(x0=0.0, radius=diameter / 2, focal=focal)

u1 = u0 * t0
```

RS

[7]:
```
%%time
z0 = np.linspace(14.8 * mm, 15.2 * mm, 512)
u2_rs = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u2_rs.incident_field(u1)
u2_rs.RS(num_processors=4);
u2_rs.draw(logarithm=1e-1, z_scale='mm')
plt.ylim(-30,30)
```

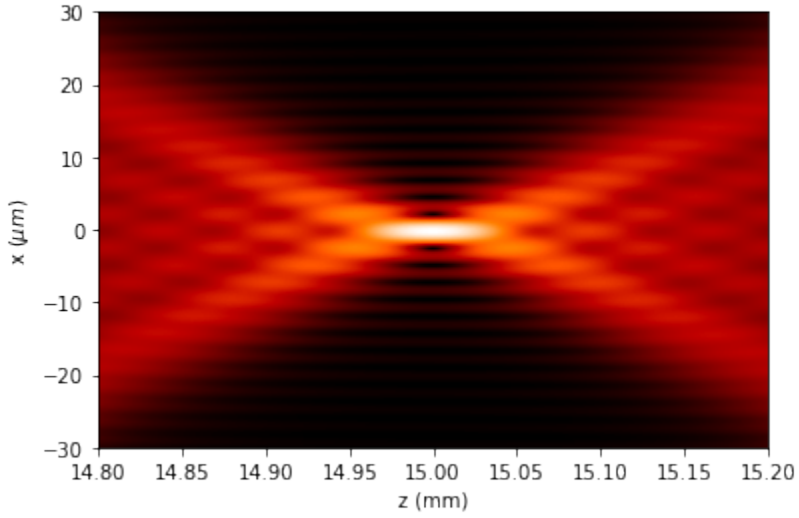
```
CPU times: user 805 ms, sys: 305 ms, total: 1.11 s
Wall time: 2.44 s
```



CZT

[8]:
```
%%time
xout=np.linspace(-30,30,256)
u2_czt = u1.CZT(z=z0, xout=xout)
u2_czt.draw(logarithm=1e-1, z_scale='mm')
```

```
CPU times: user 2.43 s, sys: 7.56 ms, total: 2.44 s
Wall time: 3.9 s
```



XY scheme

[9]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

[10]:
```
diameter = 2 * mm
focal = 25 * mm

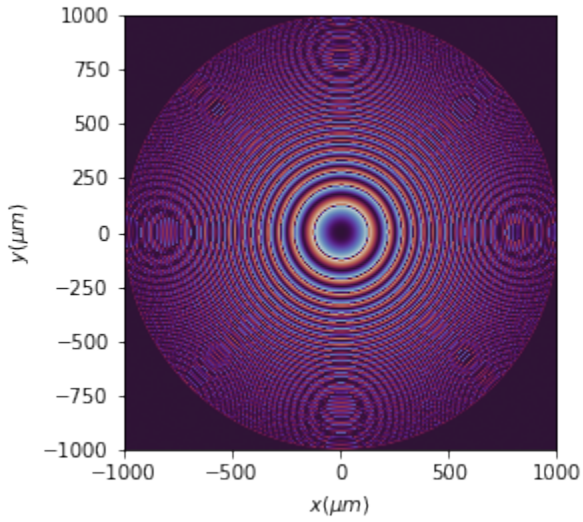
x0 = np.linspace(-diameter / 2, diameter / 2, 1024)
y0 = np.linspace(-diameter / 2, diameter / 2, 1024)
wavelength = 0.6238 * um
```

[11]:
```
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.plane_wave()

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.lens(r0=(0 * um, 0 * um),
        radius=(diameter / 2, diameter / 2),
        focal=(focal, focal))

t0.draw('phase')

u1 = u0 * t0
```



RS

[12]:
```
%%time
u2_rs = u1.RS(z=focal)

u2_rs.cut_resample(x_limits=(-40, 40),
                   y_limits=(-40, 40),
                   num_points=(128, 128),
```

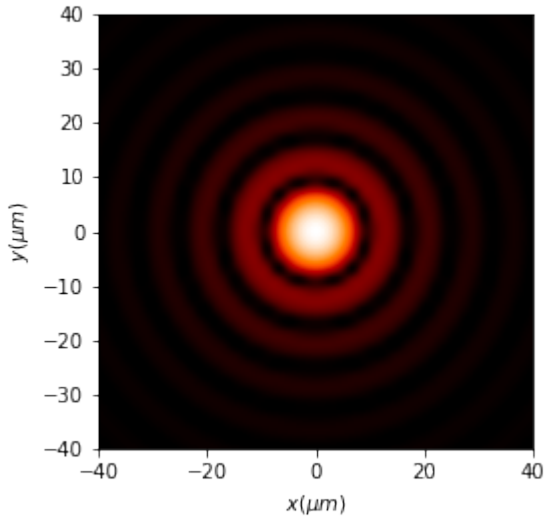
(continues on next page)

(continued from previous page)

```
                    new_field=False,
                    interp_kind=(3, 1))

u2_rs.draw(logarithm=1e-2)
```

```
CPU times: user 3.84 s, sys: 346 ms, total: 4.19 s
Wall time: 4.03 s
```

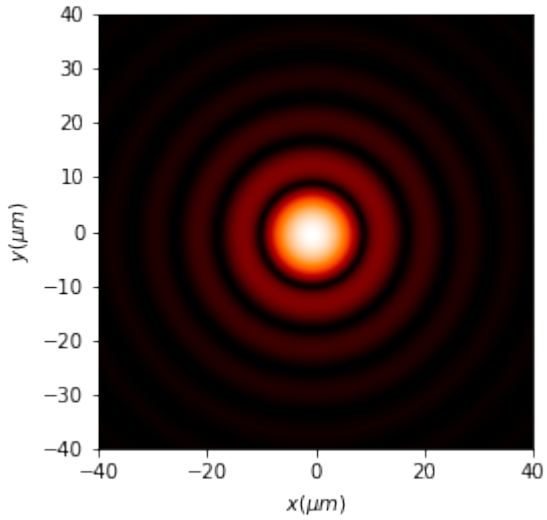


CZT

[13]:
```
%%time
xout=np.linspace(-40,40,128)
yout=np.linspace(-40,40,128)
u2_czt = u1.CZT(z=focal, xout=xout, yout=yout)

u2_czt.draw(logarithm=1e-2)
```

```
CPU times: user 774 ms, sys: 185 ms, total: 959 ms
Wall time: 709 ms
```



6.1.11.3 Vector scheme

[14]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_fields_XY import Vector_field_XY
```

[15]:
```
diameter = 2 * mm
x0 = np.linspace(-diameter / 2, diameter / 2, 512)
y0 = np.linspace(-diameter / 2, diameter / 2, 512)
wavelength = 0.6238 * um
```

[16]:
```
focal = 25 * mm
limit = 80 * um
```

[17]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.lens(r0=(0 * um, 0 * um),
        radius=(diameter / 2, diameter / 2),
        focal=(focal, focal))

u1 = u0 * t0
```

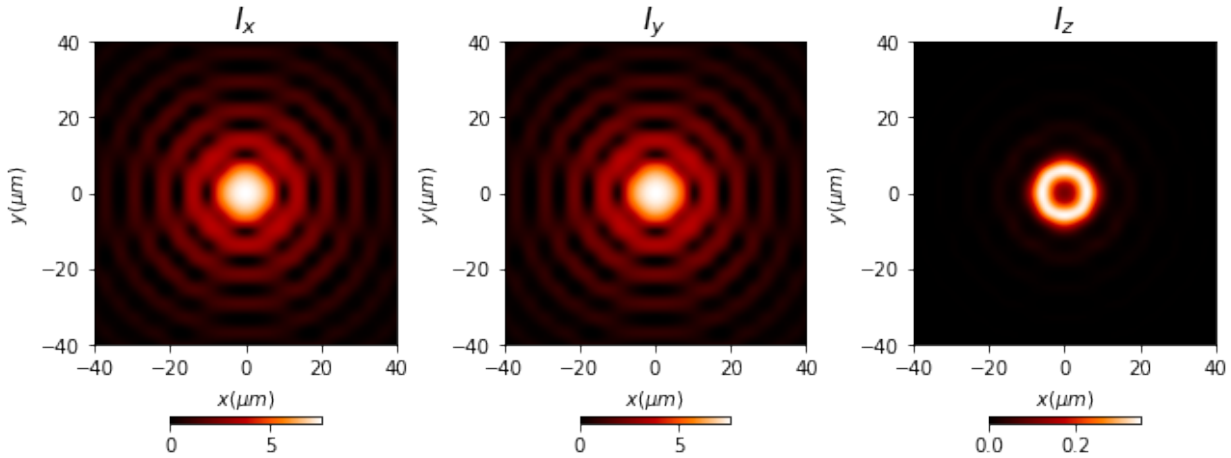
[18]:
```
EM0 = Vector_source_XY(x0, y0, wavelength)
EM0.constant_polarization(u1, v=[(1 - 1j) / 2, (1 + 1j) / 2])
```

RS

[19]:
```
%%time
EM1_rs = EM0.VRS(z=focal)
EM1_rs.cut_resample(x_limits=(-limit / 2, limit / 2),
                    y_limits=(-limit / 2, limit / 2),
                    num_points=(128, 128),
                    new_field=False,
                    interp_kind=(3, 1))

EM1_rs.draw('intensities', logarithm=1e-1)
```

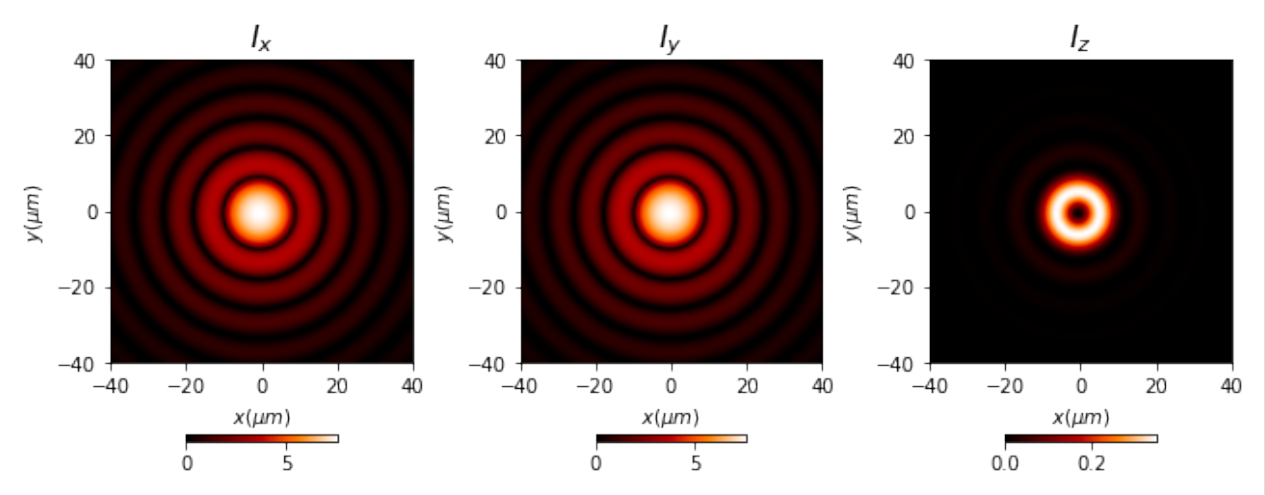
```
CPU times: user 2.61 s, sys: 388 ms, total: 3 s
Wall time: 2.37 s
```



CZT

[20]:
```
%%time
xout = np.linspace(-limit / 2, limit / 2, 128)
yout = np.linspace(-limit / 2, limit / 2, 128)
EM1_czt = EM0.CZT(z=focal, xout=xout, yout=yout)
EM1_czt.draw('intensities', logarithm=1e-1)
```

```
CPU times: user 1.55 s, sys: 372 ms, total: 1.92 s
Wall time: 1.39 s
```



6.2 Scalar_X

6.2.1 Characteristics

Scalar_X is a set of three modules for:

- Generation of 1D (x-axis) light source.
- Generation of 1D (x-axis) masks and diffractive optical elements.
- Propagation of light, determination of parameters, and other functions.
- Drawing sources, masks and fields.

These modules are named: scalar_fields_X.py, scalar_sources_X.py, and scalar_masks_X.py.

Each module present a main class:

- Scalar_field_X
- Scalar_masks_X
- Scalar_sources_X

The main attributes for these classes are the following:

- self.x (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n .
- self.wavelength (float): wavelength of the incident field.
- self.u (numpy.array): equal size than x. complex field.

We can also find these atributes:

- self.quality (float): quality of RS algorithm. Valid for values > 1.
- self.info (str): description of data.
- self.type (str): Class of the field.
- self.date (str): date when performed.

The dimensional magnitudes are related to microns: `micron = 1.`

Creating an instance

[1]:
```
from diffractio import sp, plt, np
from diffractio import nm, mm, degrees, um
from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
```

Creating a light source.

An instance must be created before starting to operate with light sources. The initialization accepts several arguments.

[2]:
```
x0 = np.linspace(-0.5 * mm, 0.5 * mm, 1024 * 32)
wavelength = 0.5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
```

Light sources are defined in the scalar_sources_x.py module. When the field is initialized, the amplitude of the field is zero. There are many methods that can be used to generate a light source:

- **plane_wave**: Generates a plane wave with a given direction and amplitude
- **gauss_beam**: Generates a gauss beam with a given amplitude, direction, beam-waist and position of beam-waist.
- **spherical_wave**: Generates a spherical wave.
- **plane_waves_several_inclined**: Generate several plane waves with different incident angles.
- **plane_waves_dict**: Generate several plane waves with parameters defined in a dictionary.
- **gauss_beams_several_parallel**: Generates a number of parallel gauss beam at different positions.
- **gauss_beams_several_inclined**: Generates a number of gauss beams all placed at the same position but with different incidence angles.

For a more detailed description of each method, refer to the individual documentation of each one.

Example: gauss beam.

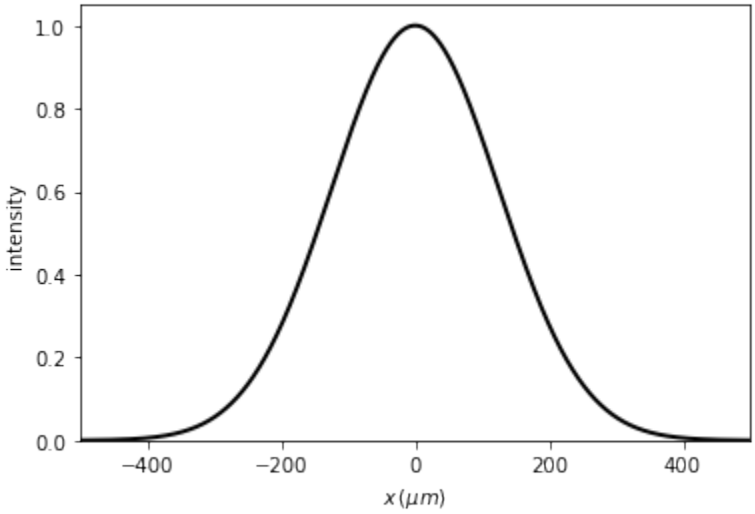
[3]:
```
u0.gauss_beam(A=1, w0=250 * um, x0=0, z0=0 * um, theta=0 * degrees)

print(u0)
```

```
Scalar_source_X
 - x:  (32768,),   u:  (32768,)
 - xmin:       -500.00 um,  xmax:      500.00 um,  Dx:   0.03 um
 - Imin:       0.00,     Imax:      1.00
 - phase_min:  0.00 deg, phase_max: 0.01 deg
 - wavelength: 0.50 um
 - date:       2023-10-07_17_05_24
 - info:       u0
```

Several ways to show the field are defined: 'amplitude', 'intensity', 'field', 'phase', ...

[4]:
```
u0.draw(kind='intensity')
```



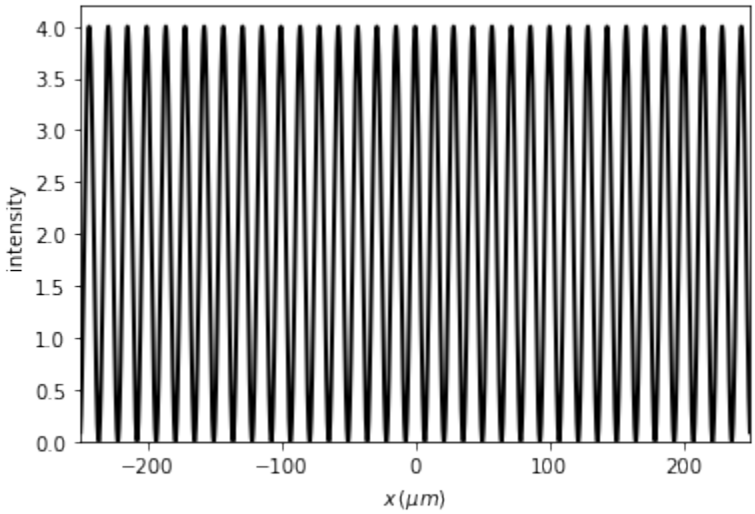
6.2.1.1 Basic operation: add two light sources

[5]:
```
x0 = np.linspace(-0.25 * mm, 0.25 * mm, 1024 * 32)
wavelength = 0.5 * um

u1 = Scalar_source_X(x=x0, wavelength=wavelength, info='u1')
u2 = Scalar_source_X(x=x0, wavelength=wavelength, info='u2')

u1.plane_wave(A=1, theta=-1 * degrees)
u2.plane_wave(A=1, theta=+1 * degrees)

u_sum = u1 + u2
u_sum.draw(kind='intensity')
```



Since we have two plane waves propagating at different angles, an interference effect is produced, shown in the intensity variation of the optical field.

Masks

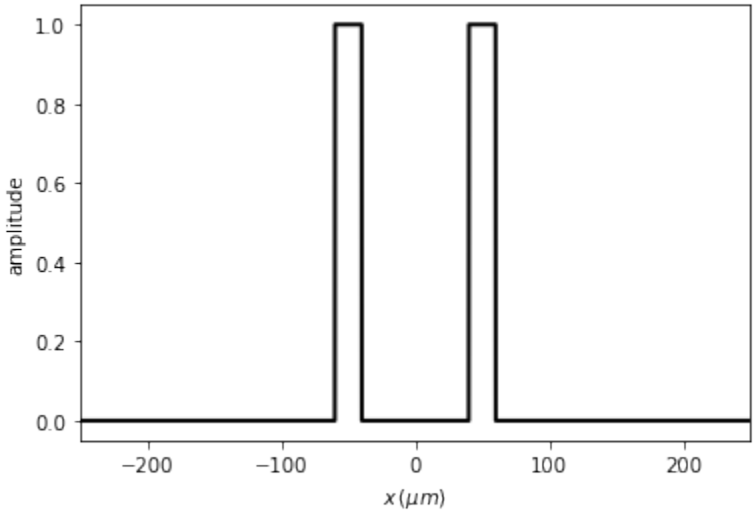
Masks are defined in the scalar_masks_x.py module. There are many methods that can be used to generate masks and diffractive optical elements:

- mask_from_function, mask_from_array
- dots
- slit, double slit
- two_levels, gray_scale
- prism, biprism_fresnel, biprism_fresnel_nh
- lens, aspheric, fresnel_lens
- roughness, dust, dust_different_sizes
- sine_grating, ronchi_grating, binary_grating, blazed_grating
- chriped_grating, chirped_grating_p, chirped_grating_q
- binary_code, binary_code_positions

For a more detailed description of each method, refer to the individual documentation of each one.

Example: double slit.

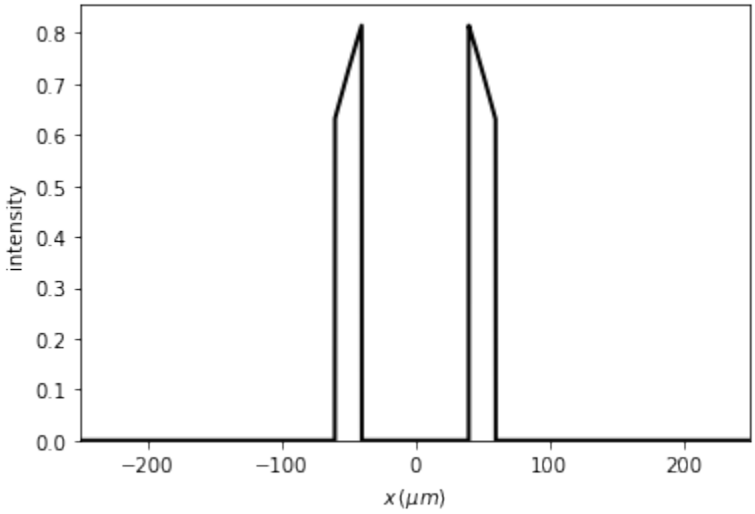
[6]:
```
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.double_slit(x0=0, size=20 * um, separation=100 * um)
t0.draw(kind='amplitude')
```



6.2.1.2 Basic operations: multiplication of masks and fields

When the light u_0 passes through a mask t_0, the field just after the mask, according to the Thin Element Approximation (TEA), is $u_1 = t_0.u_0$. This can be represented in the following way:

[7]:
```
u1 = t0 * u0
u1.draw(kind='intensity')
```



6.2.1.3 Propagation

Propagation and other actions and parameters of the optical fields are defined in the scalar_field_x.py module. There are several methods of determining the field at a given plane after the mask:

- **RS**: Rayleigh-Sommerfeld propagation at a certain distance
- **fft**: Fast Fourier propagation at the far field.
- **ifft**: Inverse Fast Fourier propagation at the far field.

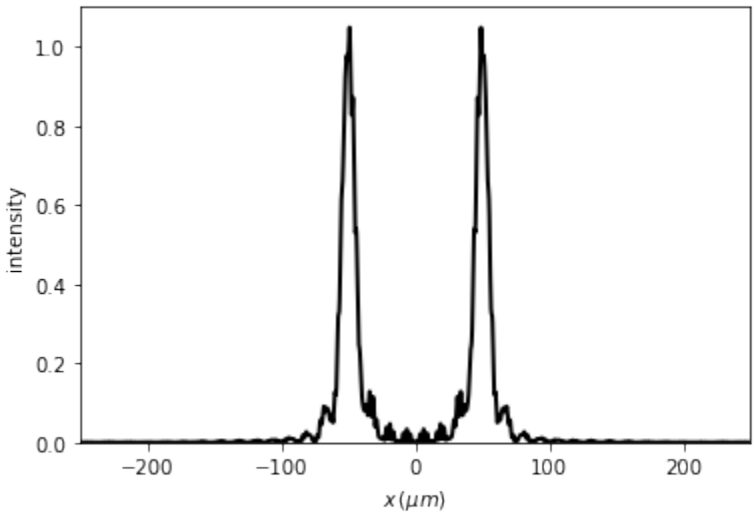
The field can be stored in the same instance, generate a new instance, or generate a numpy.array when fast computation is required.

Other propagation techniques are possible, in the XZ scheme, which is an extension of X scheme.

Near field: Rayleigh-Sommerfeld approach

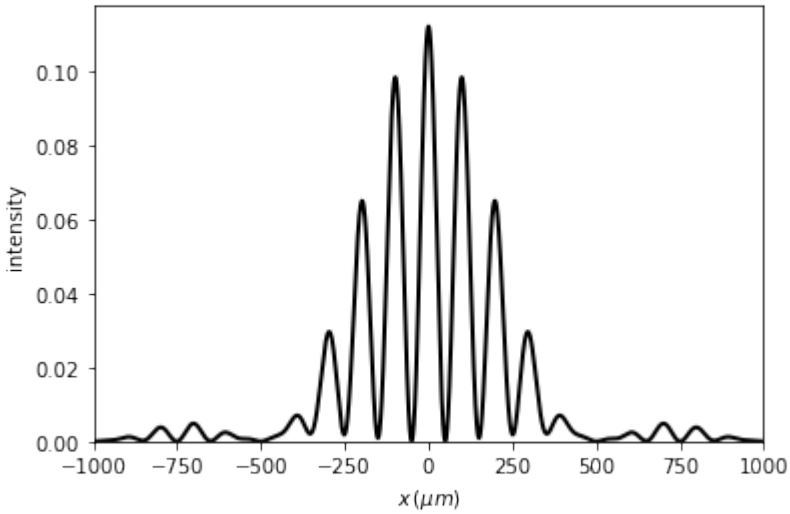
[8]:
```
u2 = u1.RS(z=0.5 * mm, new_field=True, verbose=True)
u2.draw(kind='intensity')
```

```
Good result: factor 36.57
```



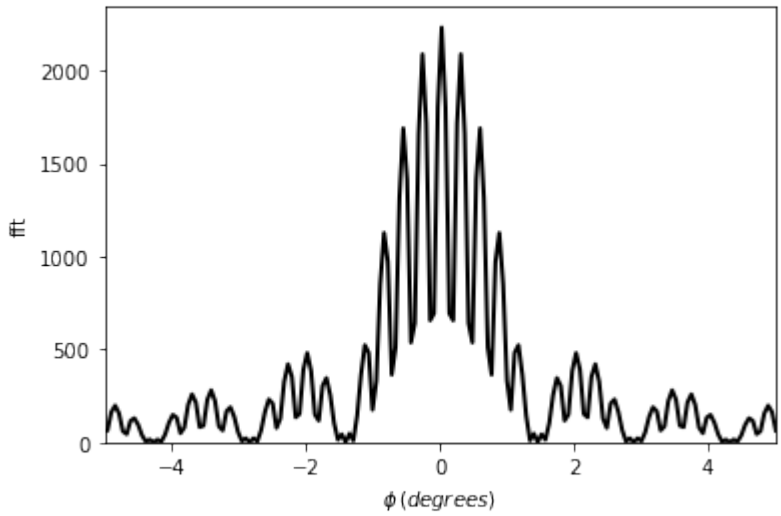
[9]:
```
u2 = u1.RS(z=20 * mm, amplification=4, new_field=True, verbose=True)
u2.draw(kind='intensity')
```

```
Good result: factor 326.46
Good result: factor 642.94
Good result: factor 4633.98
Good result: factor 33409.94
```



Far field: Fast Fourier transform

[10]:
```
u3 = u1.fft(new_field=True, shift=True)
u3.draw(kind='fft', normalize=True)
plt.xlim(-5, 5)
plt.ylim(0)
```



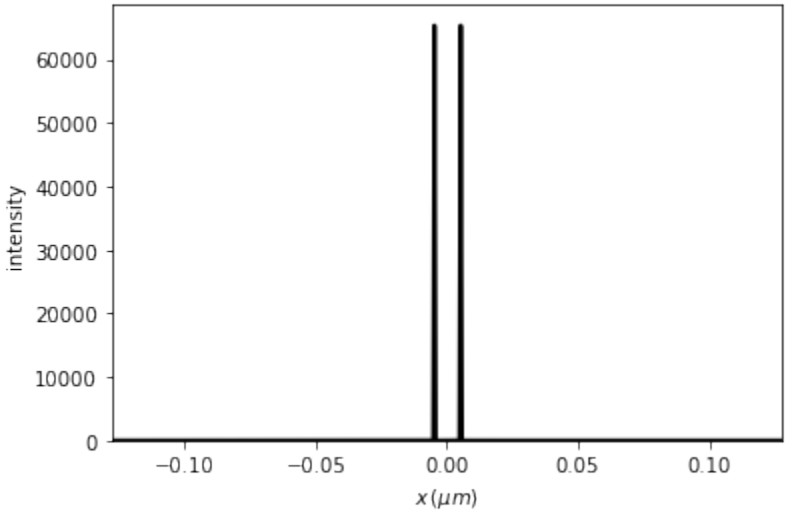
Inverse propagation: IFFT

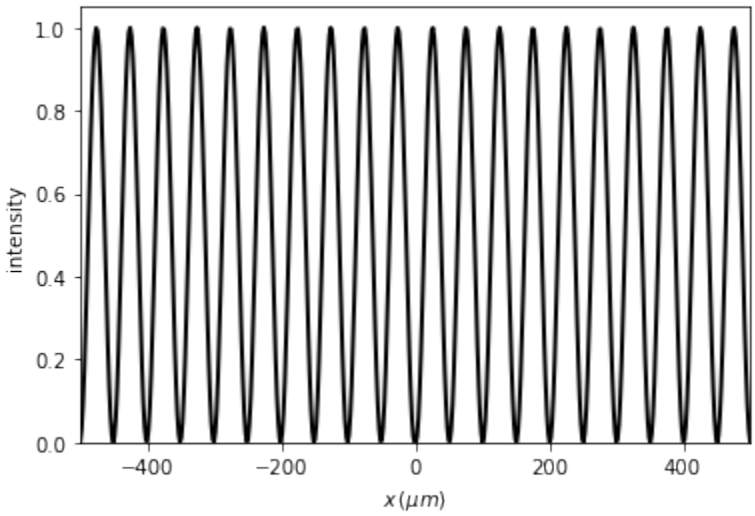
[11]:
```
x = np.linspace(-500 * um, 500 * um, 512)
wavelength = .5 * um

t1 = Scalar_field_X(x, wavelength)
t1.u = np.sin(2 * np.pi * x / 100)

t2 = t1.fft(z=None,
            shift=True,
            remove0=False,
            matrix=False,
            new_field=True,
            verbose=False)
t2.draw()

t3 = t2.ifft(z=None,
             shift=True,
             remove0=False,
             matrix=False,
             new_field=True,
             verbose=False)
t3.draw()
```





Amplification of the field

The Rayleigh-Sommefeld implementation allows an amplification of the field since, sometimes, light diverges. Processing time increases approximately linearly with amplification number.

[12]:
```
wavelength = 0.5 * um
x0 = np.linspace(-0.25 * mm, 0.25 * mm, 1024 * 8)

u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.spherical_wave(A=1, x0=0, z0=-0.25 * mm)

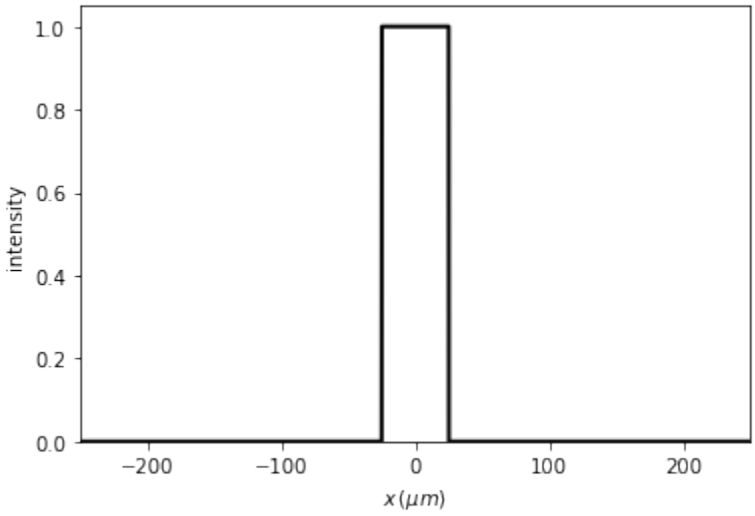
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.slit(x0=0, size=50 * um)
t0.draw(kind='intensity')

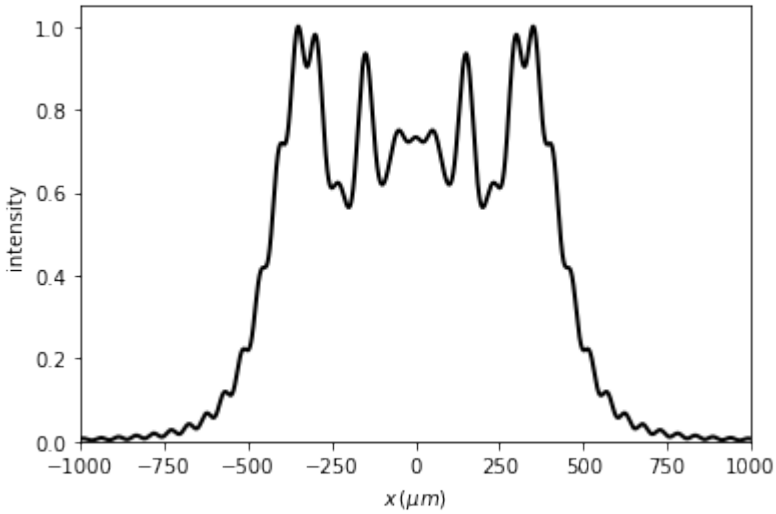
u1 = t0 * u0
```

(continues on next page)

(continued from previous page)

```
t2 = u1.RS(z=5 * mm, amplification=4, verbose=False)
t2.normalize()
t2.draw(kind='intensity')
```





Reduction of the field

There are some cases where we need to analyze the field in a small area, such as focus of a lens.

[13]:
```
focal = 5 * mm

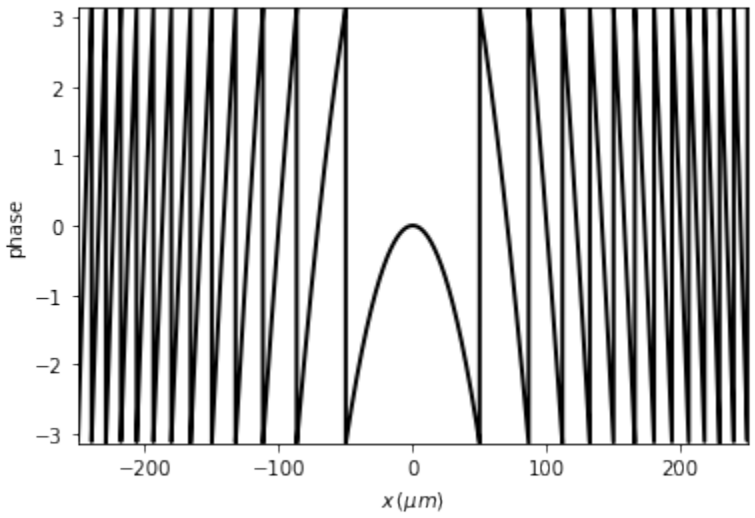
u0 = Scalar_source_X(x=x0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1, theta=0 * degrees)

t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.lens(x0=0, focal=focal, radius=0)
```

(continues on next page)

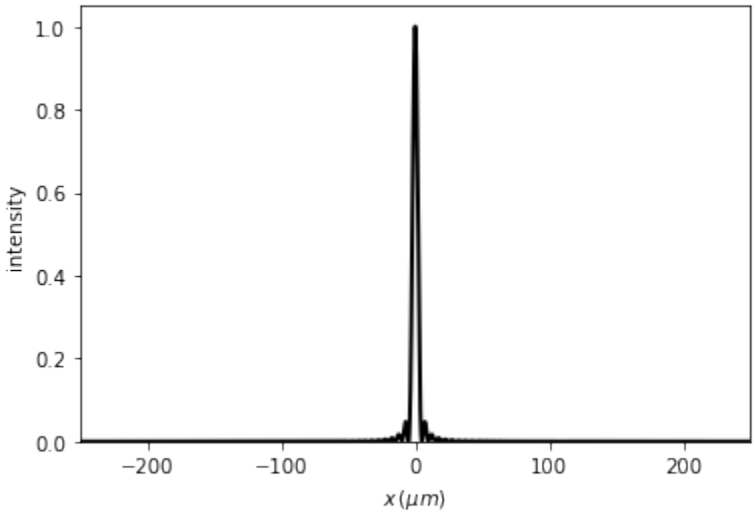
(continued from previous page)

```
t0.draw(kind='phase')
```



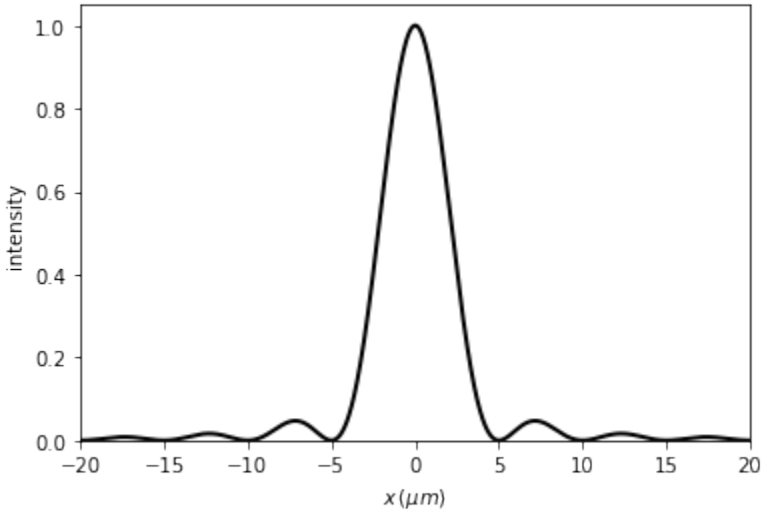
[14]:
```
u1 = t0 * u0

t2 = u1.RS(z=focal, verbose=False)
t2.normalize()
t2.draw(kind='intensity')
```



As we cannot see the details, we can use the cut_resample function in order to amplify the focusing zone.

[15]:
```
t2.cut_resample(x_limits=(-20 * um, 20 * um), num_points=2048, new_field=False)
t2.draw(kind='intensity')
```



6.2.1.4 Other operations

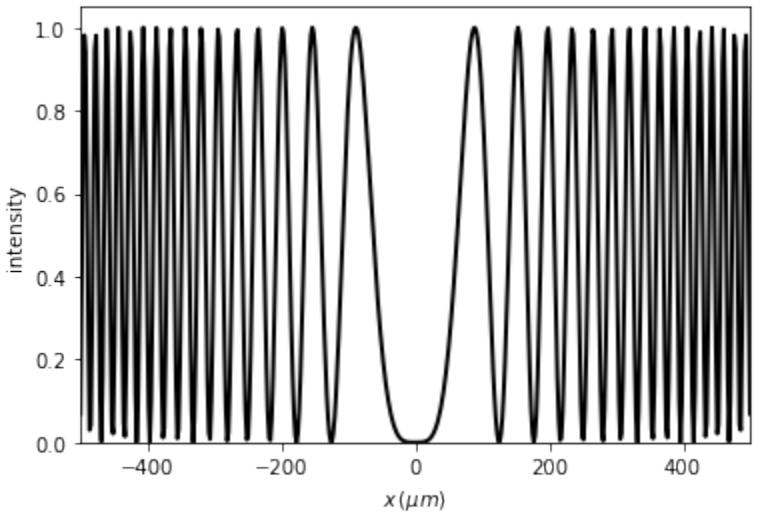
Saving and loading data from instance

saving

[16]:
```
x = np.linspace(-500 * um, 500 * um, 512)
wavelength = .5 * um
filename = 'save_load.npz'

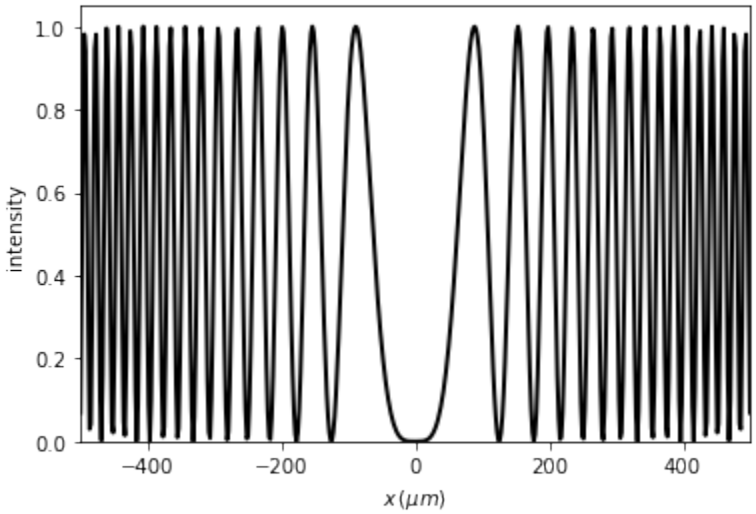
t1 = Scalar_field_X(x, wavelength)
t1.u = np.sin(x**2 / 5000)
t1.draw()

t1.save_data(filename=filename)
```



loading

[17]:
```
t2 = Scalar_field_X(x, wavelength)
t2.load_data(filename=filename)
t2.draw()
```

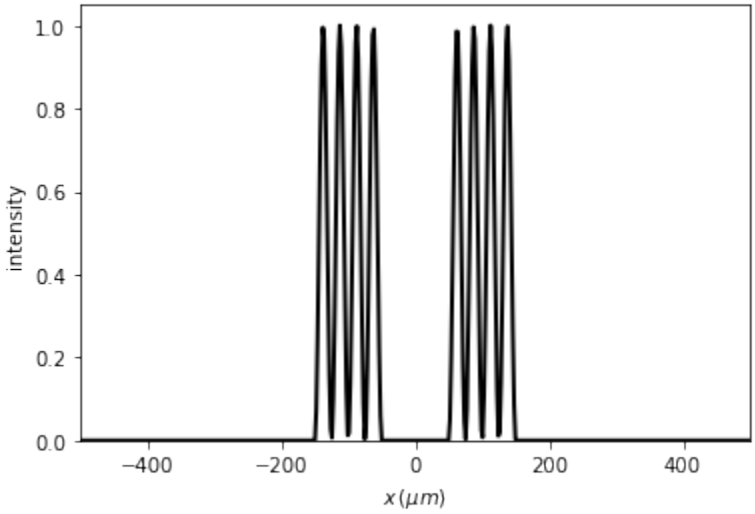


6.2.1.5 Inserting a mask in other instance

[18]:
```
x1 = np.linspace(-500 * um, 500 * um, 512)
wavelength = .5 * um
t1 = Scalar_field_X(x1, wavelength)

x2 = np.linspace(-50 * um, 50 * um, 512)
wavelength = .5 * um
t2 = Scalar_field_X(x2, wavelength)
t2.u = np.sin(2 * np.pi * x2 / 50)

t1.insert_mask(t2, x0_mask1=-100 * um, clean=False, kind_position='center')
t1.insert_mask(t2, x0_mask1=100 * um, clean=False, kind_position='center')
t1.draw()
```



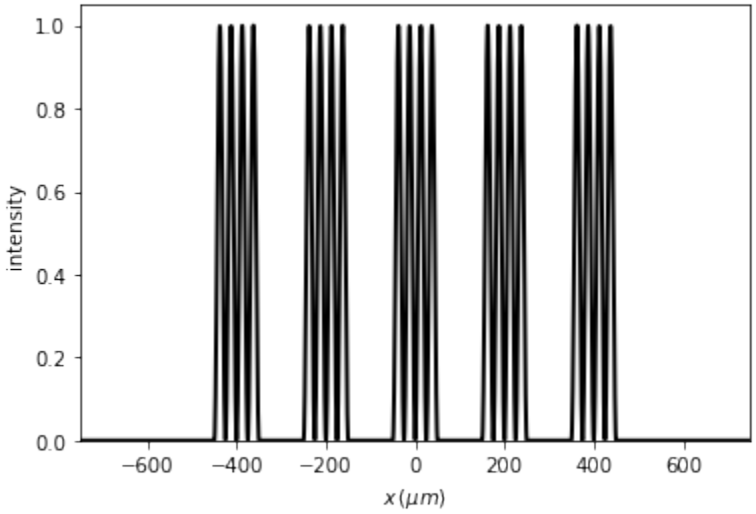
Inserting an array of masks in an instance

[19]:
```
x1 = np.linspace(-750 * um, 750 * um, 2048)
wavelength = .5 * um

t1 = Scalar_field_X(x1, wavelength)

x2 = np.linspace(-50 * um, 50 * um, 2048)
wavelength = .5 * um
t2 = Scalar_field_X(x2, wavelength)
t2.u = np.sin(2 * np.pi * x2 / 50)

t1.insert_array_masks(t2,
                      x_pos=[-400, -200, 0, 200, 400],
                      clean=True,
                      kind_position='center')
t1.draw()
```



6.2.1.6 Multiprocessing

For fast computation, several functions are used for generating polychromatic, and extended-size sources:

- polychromatic_multiprocessing
- extended_source_multiprocessing
- extended_polychromatic_source

When the computer present several cores, a signficative increase in speed is obtained.

[20]:
```
from diffractio.scalar_fields_X import extended_source_multiprocessing
import multiprocessing

num_max_processors = multiprocessing.cpu_count()
```

A function with only the variable to multiprocess is required:

[21]:
```
def __experiment_extended_source__(x0):

    x = np.linspace(-333 * um, 333 * um, 1024 * 2)
    wavelength = 0.850 * um
    z0 = -500 * mm
    period = 50 * um
    focal = 5 * mm

    red = Scalar_mask_X(x, wavelength)
    red.ronchi_grating(x0=0 * um, period=period, fill_factor=0.5)

    lens = Scalar_mask_X(x, wavelength)
    lens.lens(x0=0, focal=focal, radius=30 * mm)

    u1 = Scalar_source_X(x, wavelength)
    u1.spherical_wave(A=1., x0=x0, z0=z0, radius=0, normalize=True)
    u2 = u1 * red * lens
```

(continues on next page)

(continued from previous page)

```
    u2.RS(z=focal, new_field=False, verbose=False)

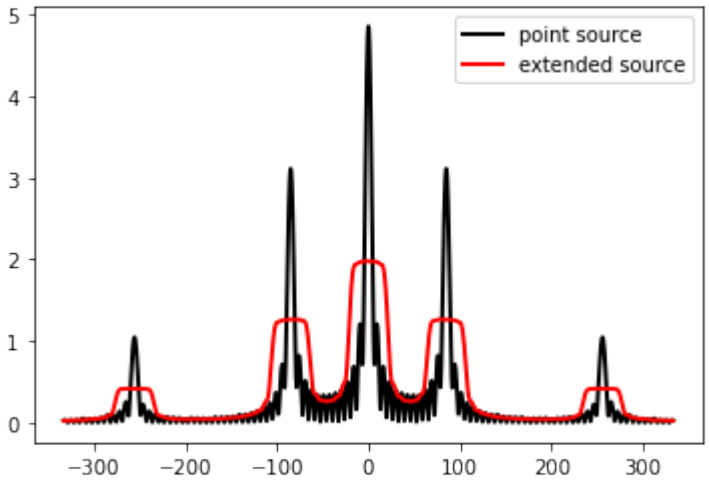
    return u2
```

[22]:
```
x0s = np.linspace(-2 * mm, 2 * mm, 1001)
x0_central = 0 * um

intensity, u_s, time_proc = extended_source_multiprocessing(
    __experiment_extended_source__,
    x0s,
    num_processors=num_max_processors,
    verbose=True)
intensity0, u_s0, time_proc0 = extended_source_multiprocessing(
    __experiment_extended_source__, x0_central, num_processors=1, verbose=True)
```

```
num_proc: 8, time=1.0968291759490967
num_proc: 1, time=0.006218910217285156
```

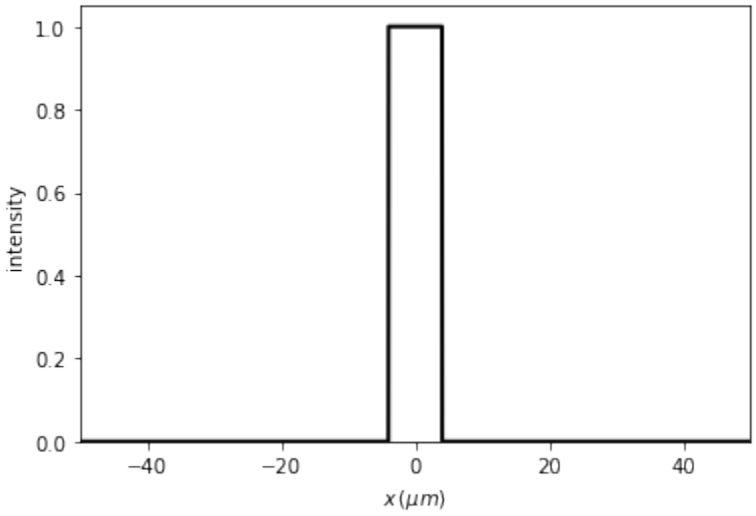
[23]:
```
plt.figure()
plt.plot(u_s0.x, np.sqrt(intensity0), 'k', lw=2, label='point source')
plt.plot(u_s[0].x, np.sqrt(intensity), 'r', lw=2, label='extended source')
plt.legend()
```



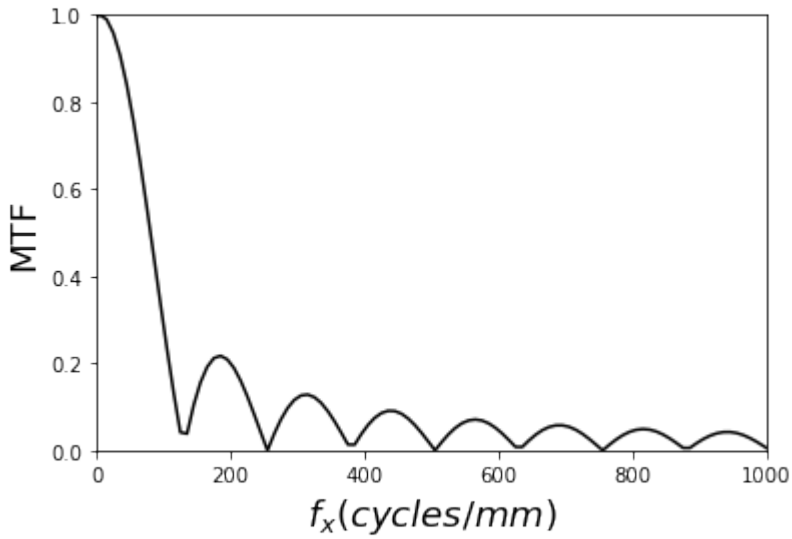
6.2.1.7 Modulation transfer function (MTF)

[24]:
```
num_data = 1024 * 128
x = np.linspace(-50 * um, 50 * um, num_data)
wavelength = 0.6328 * um

# intensidad de una onda plana
u0 = Scalar_mask_X(x, wavelength)
u0.slit(x0=0, size=8 * um)
u0.draw(kind='intensity')
```



[25]:
```
mtf1 = u0.MTF(has_draw=True)
plt.xlim(0, 1000)
plt.ylim(0, 1)
```



6.2.2 Example of light sources

6.2.2.1 Creating an instance

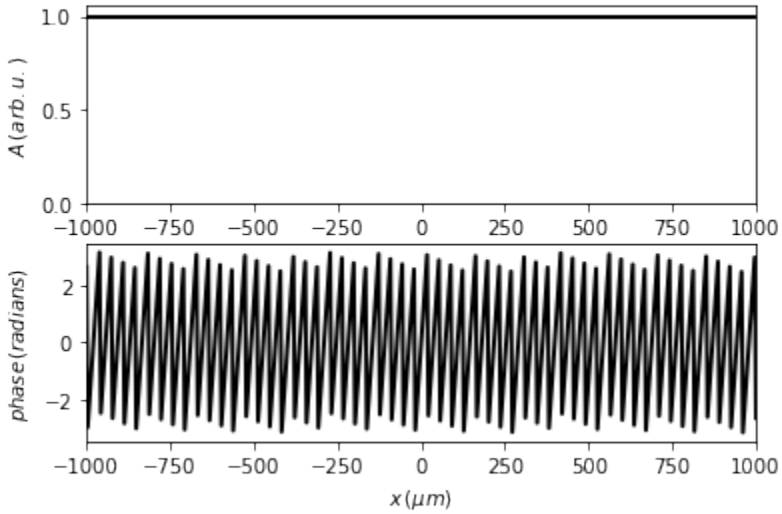
An instance must be created before starting to operate with light sources. The initialization accepts several arguments.

[1]:
```
from diffractio import np, plt
from diffractio import degrees, mm, um
from diffractio.scalar_sources_X import Scalar_source_X
```

6.2.2.2 Plane wave

[2]:
```
x0 = np.linspace(-1000 * um, 1000 * um, 512)
wavelength = 0.6328 * um

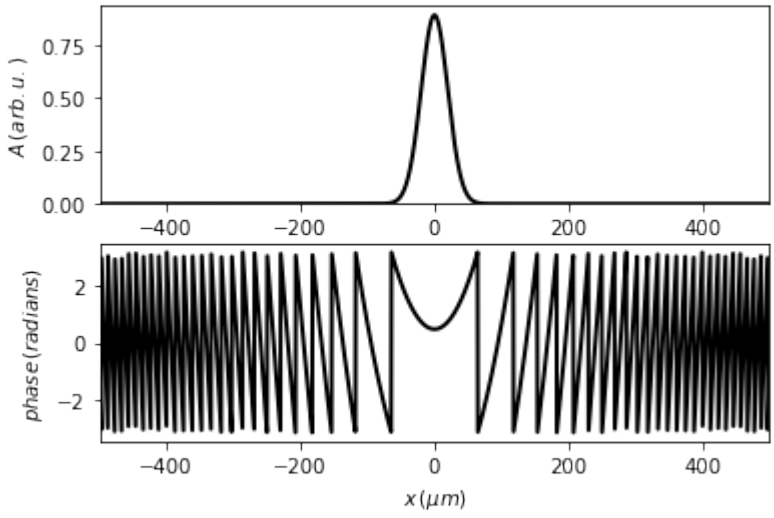
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(theta=1 * degrees, z0=0 * um)
u0.draw(kind='field')
```



6.2.2.3 Gauss beam

[3]:
```
x0 = np.linspace(-500 * um, 500 * um, 2048)
wavelength = .5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=-2000 * um, w0=25 * um, theta=0 * degrees)

u0.draw(kind='field')
```

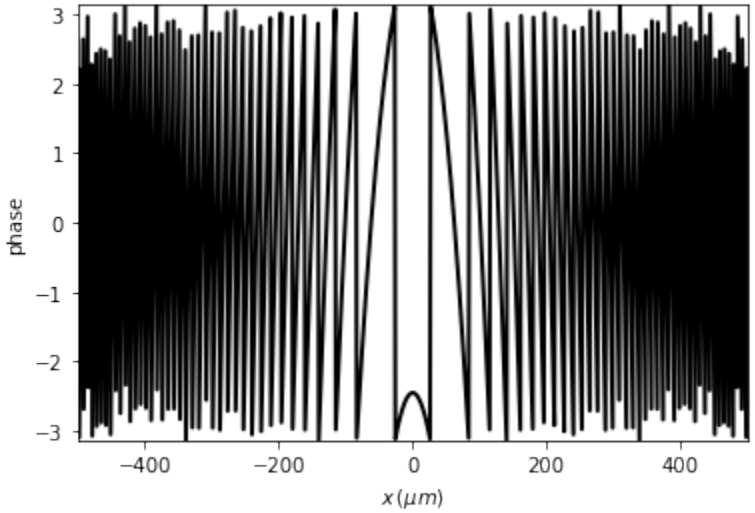


6.2.2.4 Spherical wave

Convergent

[4]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024)
wavelength = 0.6328 * um

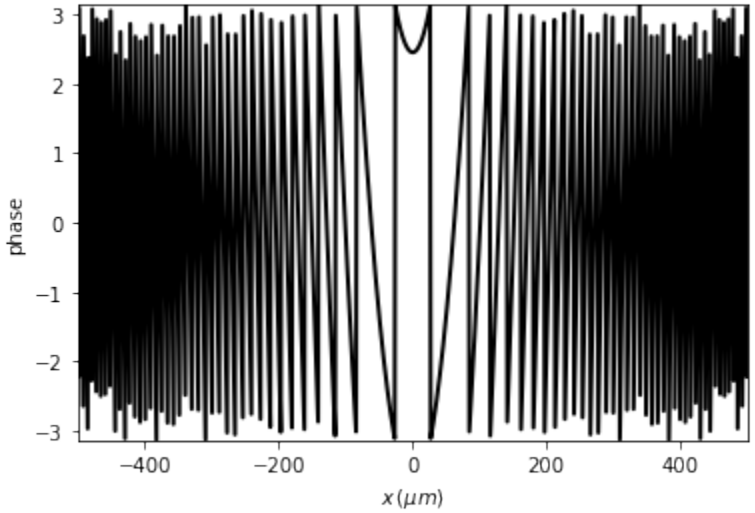
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.spherical_wave(A=1, x0=0 * um, z0=5 * mm)
u0.draw(kind='phase')
```



Divergent

[5]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024)
wavelength = 0.6328 * um

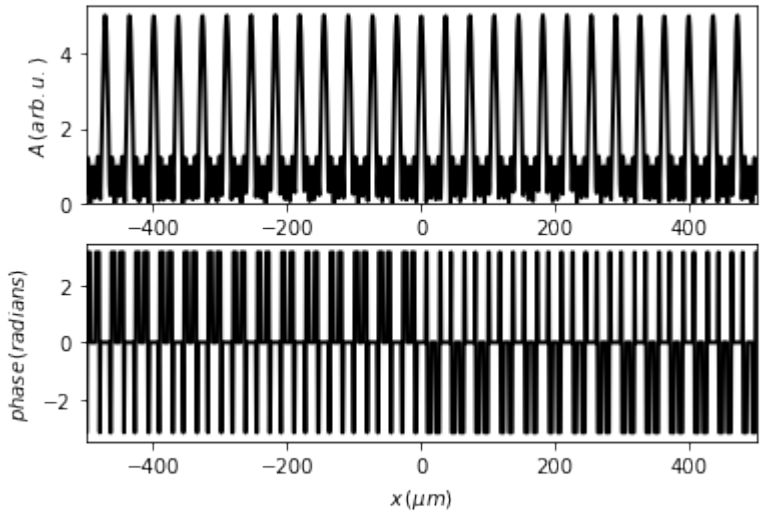
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.spherical_wave(A=1, x0=0 * um, z0=-5 * mm)
u0.draw(kind='phase')
```



6.2.2.5 Several plane waves with different inclination

[6]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024)
wavelength = 0.6328 * um

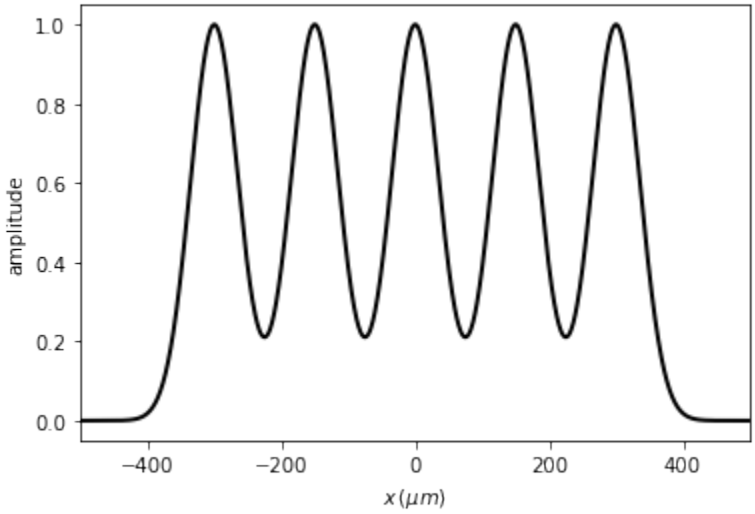
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_waves_several_inclined(A=1, num_beams=5, max_angle=5 * degrees)
u0.draw(kind='field')
```



6.2.2.6 Several gauss beams with different position

[7]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024)
wavelength = 0.6328 * um

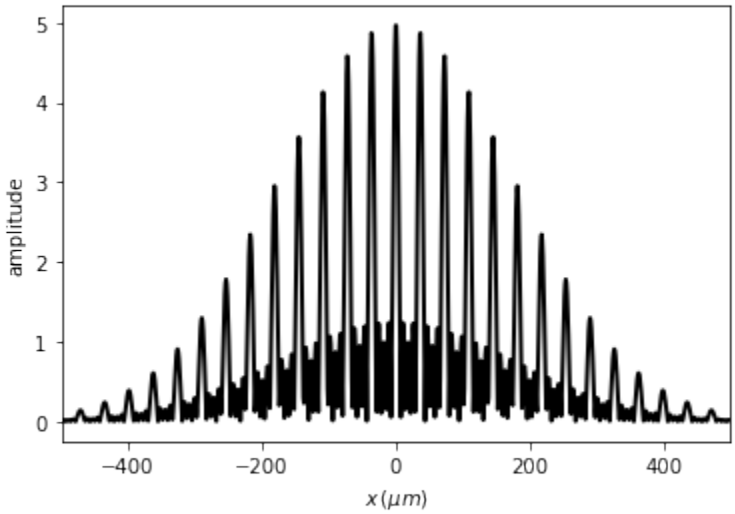
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beams_several_parallel(A=1,
                                num_beams=5,
                                w0=50 * um,
                                z0=0 * um,
                                x_central=0 * um,
                                x_range=750 * um,
                                theta=0 * degrees)
u0.draw(kind='amplitude')
```



6.2.2.7 Several Gauss beam with different inclination

[8]:
```
x0 = np.linspace(-500 * um, 500 * um, 1024)
wavelength = 0.6328 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beams_several_inclined(A=1,
                                num_beams=5,
                                w0=250 * um,
                                x0=0 * um,
                                z0=0 * um,
                                max_angle=5 * degrees)
u0.draw(kind='amplitude')
```



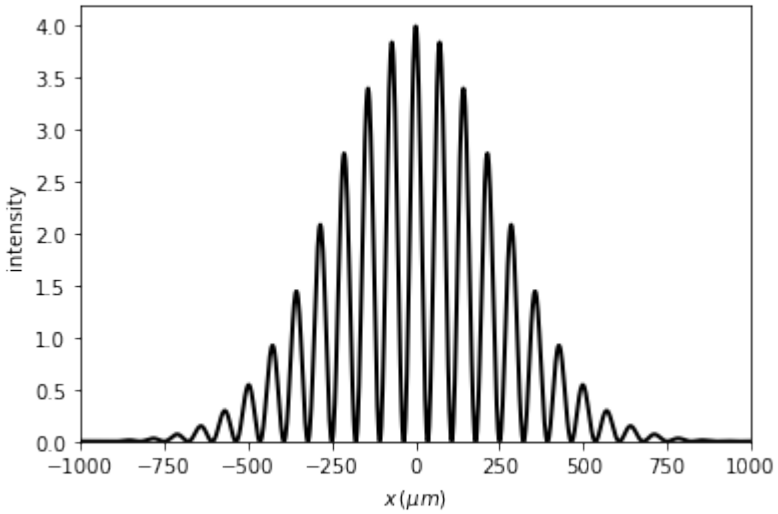
6.2.2.8 Sum of two gauss beams with different inclination: interference

[9]:
```
length = 2 * mm
x0 = np.linspace(-length / 2, length / 2, 1024)
wavelength0 = 0.6238 * um

u1 = Scalar_source_X(x=x0, wavelength=wavelength0)
u2 = Scalar_source_X(x=x0, wavelength=wavelength0)

u1.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=500 * um, theta=.25 * degrees)
u2.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=500 * um, theta=-.25 * degrees)

u0 = u1 + u2
u0.draw(kind='intensity')
```



6.2.3 Example of masks

6.2.3.1 Creating an instance

[1]:
```
from diffractio import np, plt
from diffractio import degrees, mm, um
from diffractio.scalar_masks_X import Scalar_mask_X

from numpy import loadtxt
```

6.2.3.2 Procedures to generate masks

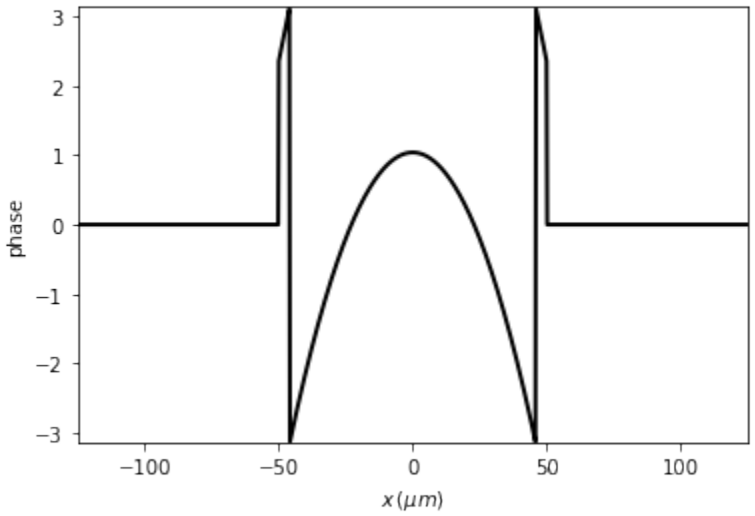
mask_from_function

[2]:
```
num_data = 1024
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

f1 = "R1-h1+np.sqrt(R1**2-(self.x-x0)**2)"
f2 = "R2-h2+np.sqrt(R2**2-(self.x-x0)**2)"
v_globals = {
    'R1': 5 * mm,
    'R2': 1 * mm,
    'x0': 0 * um,
    'y0': 0 * um,
    'h1': 1 * mm,
    'h2': -1 * mm
}
index = 1.5
print(v_globals)

t1 = Scalar_mask_X(x, wavelength)
t1.mask_from_function(x0=0 * um,
                      index=index,
                      f1=f1,
                      f2=f2,
                      v_globals=v_globals,
                      radius=100 * um)
t1.draw(kind='phase')
```

```
{'R1': 5000.0, 'R2': 1000.0, 'x0': 0.0, 'y0': 0.0, 'h1': 1000.0, 'h2': -1000.0}
```



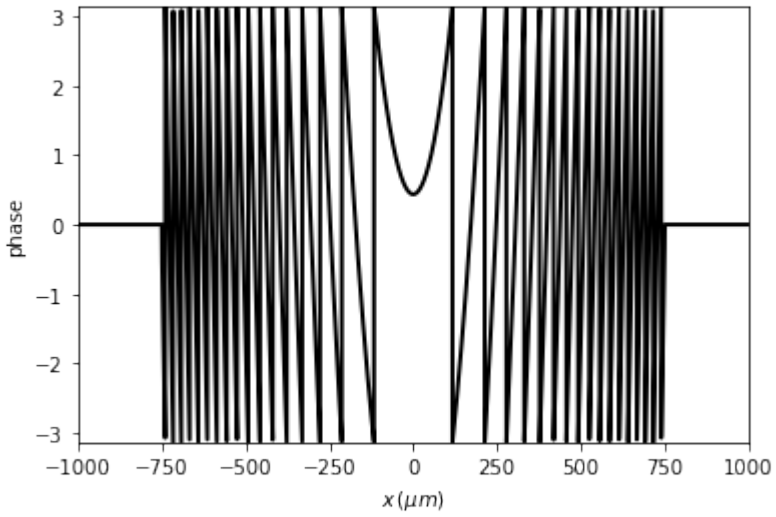
mask_from_array

[3]:
```
num_data = 1024 * 8
x = np.linspace(-1 * mm, 1 * mm, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_X(x, wavelength)

profile1 = loadtxt('profile1.txt')
profile2 = loadtxt('profile2.txt')
t1.mask_from_array(x0=0,
                   index=1.25,
                   array1=profile1 * mm,
                   array2=profile2 * mm,
                   interp_kind='quadratic',
                   radius=1.5 * mm)

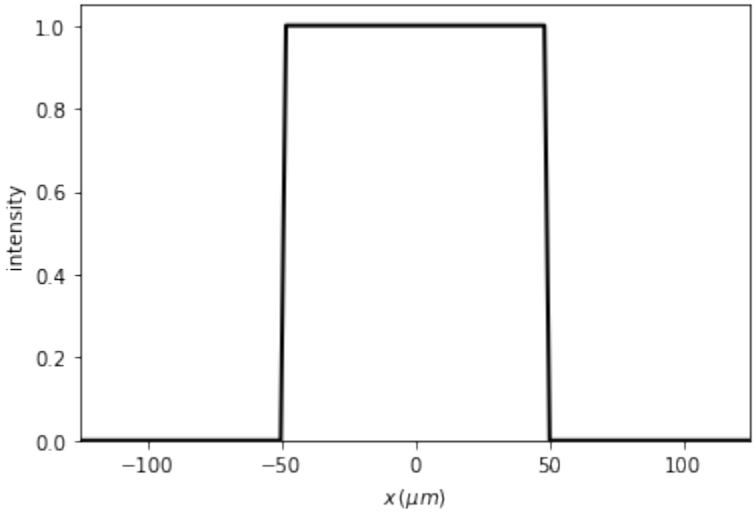
t1.draw(kind='phase')
```



6.2.3.3 slit

[4]:
```
num_data = 128
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

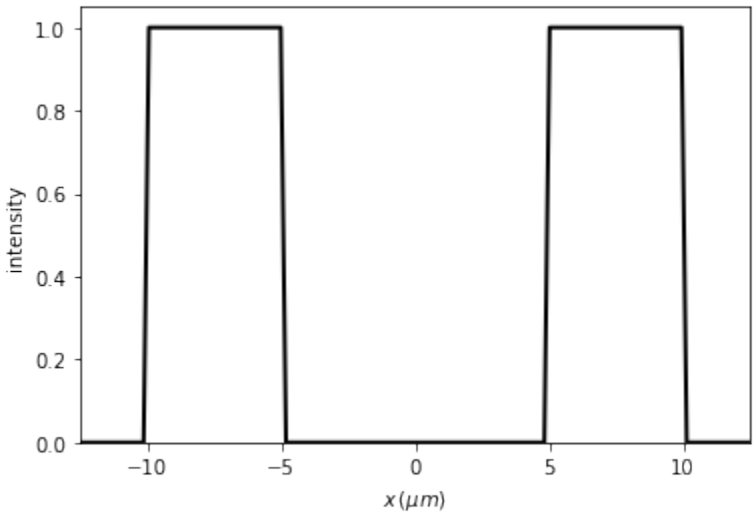
t1 = Scalar_mask_X(x, wavelength)
t1.slit(x0=0, size=100 * um)
t1.draw()
```



6.2.3.4 double_slit

[5]:
```
num_data = 128
length = 25 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

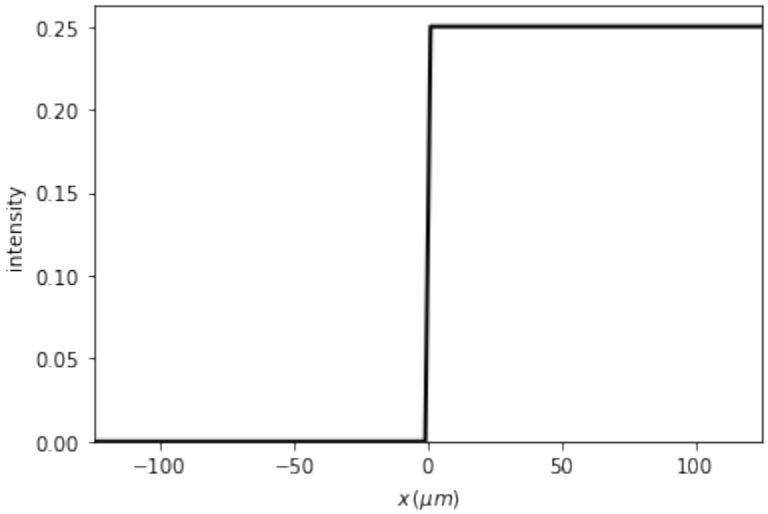
t1 = Scalar_mask_X(x, wavelength)
t1.double_slit(x0=0, size=5 * um, separation=15 * um)
t1.draw()
```



6.2.3.5 two levels

[6]:
```
num_data = 128
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

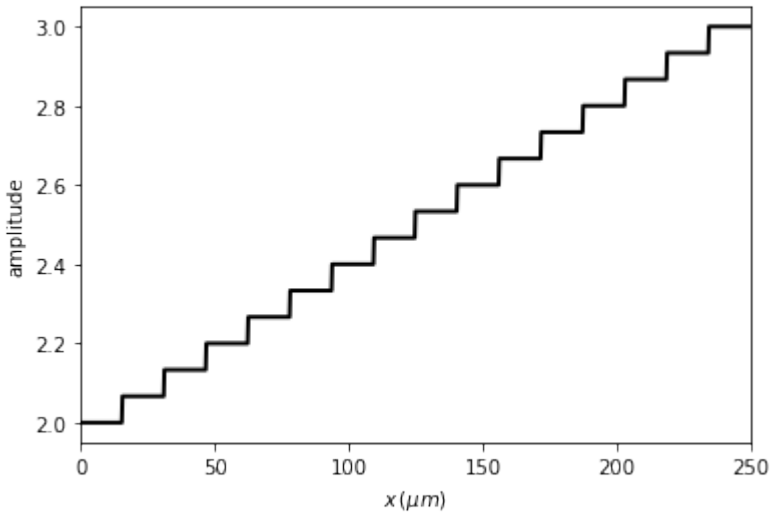
t1 = Scalar_mask_X(x, wavelength)
t1.two_levels(level1=0, level2=.5, x_edge=0)
t1.draw()
```



6.2.3.6 gray scale

[7]:
```
x = np.linspace(0, 250 * um, 1024)
wavelength = 0.6328 * um

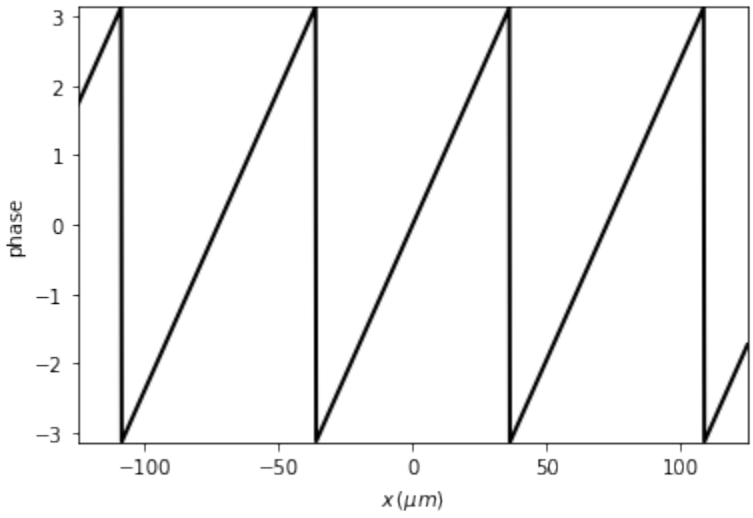
t1 = Scalar_mask_X(x, wavelength)
t1.gray_scale(num_levels=16, levelMin=2, levelMax=3)
t1.draw(kind='amplitude')
```



6.2.3.7 prism

[9]:
```
num_data = 1024
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

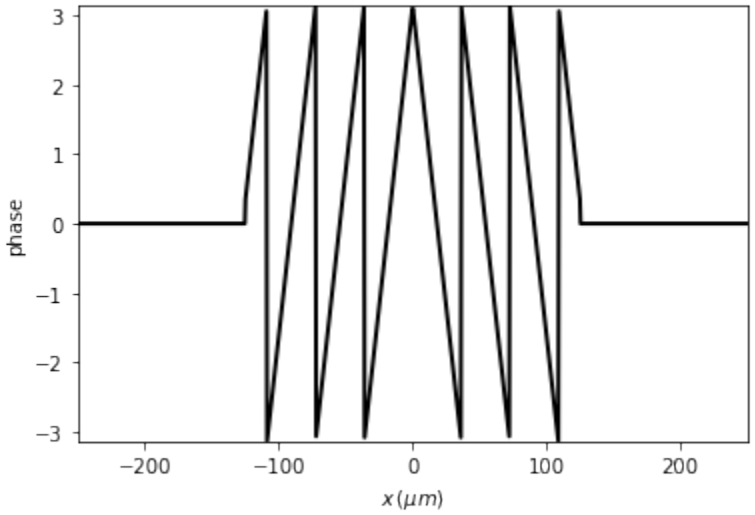
t1 = Scalar_mask_X(x, wavelength)
t1.prism(x0=0, n=1.5, angle=1 * degrees)
t1.draw(kind='phase')
```



6.2.3.8 Fresnel biprism

[10]:
```
num_data = 1024
length = 500 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_X(x, wavelength)
t1.biprism_fresnel(angle=1 * degrees, x0=0 * um, radius=125)
t1.draw(kind='phase')
```

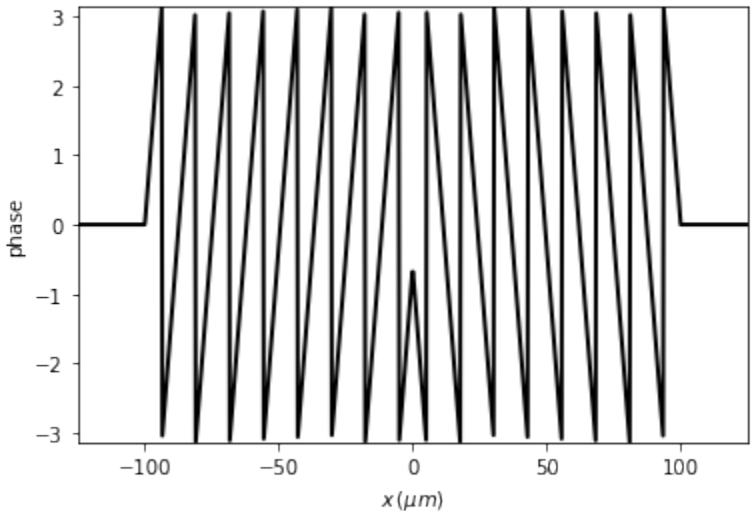


6.2.3.9 Fresnel biprism from material properties n and h

[11]:
```
num_data = 1024
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_X(x, wavelength)
t1.biprism_fresnel_nh(x0=0, width=100 * um, height=5 * um, n=1.5)

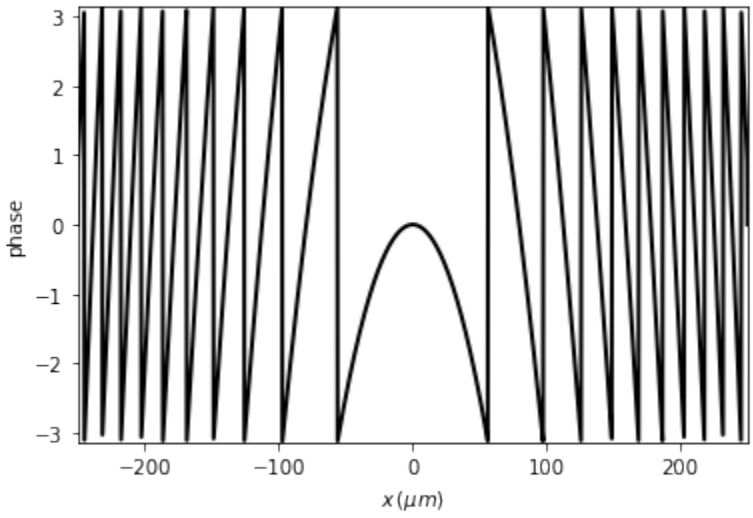
t1.draw(kind='phase')
```



6.2.3.10 standard lens

[12]:
```
num_data = 2048
length = 500 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

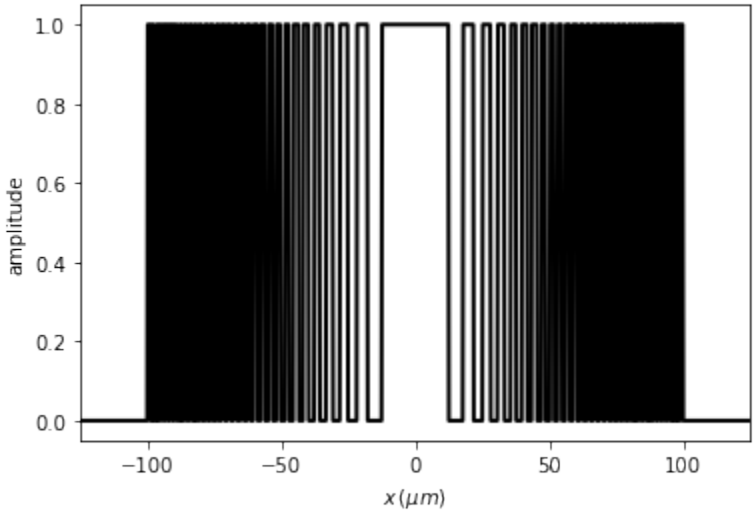
t1 = Scalar_mask_X(x, wavelength)
t1.lens(x0=0 * um, radius=250 * um, focal=5 * mm)
t1.draw(kind='phase')
```



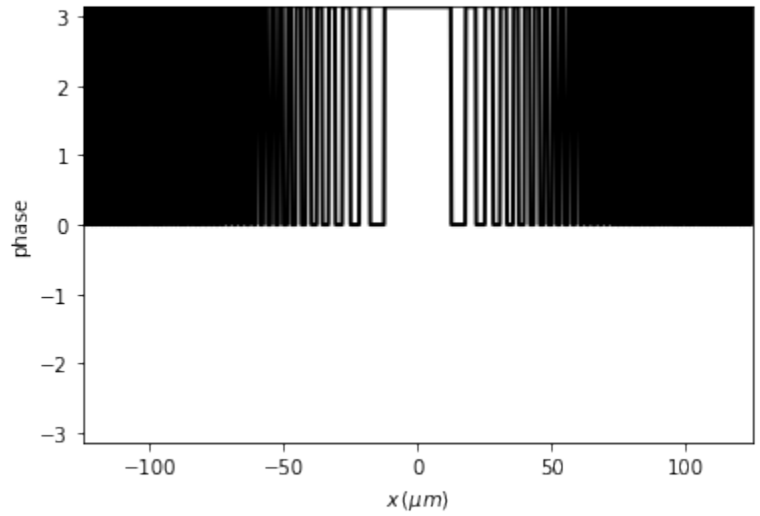
6.2.3.11 Fresnel lens: amplitude and phase

[13]:
```
num_data = 1024
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um
```

[14]:
```
t1 = Scalar_mask_X(x, wavelength)
t1.fresnel_lens(x0=0 * um,
                radius=100 * um,
                focal=.25 * mm,
                kind='amplitude',
                phase=np.pi)
t1.draw(kind='amplitude')
```



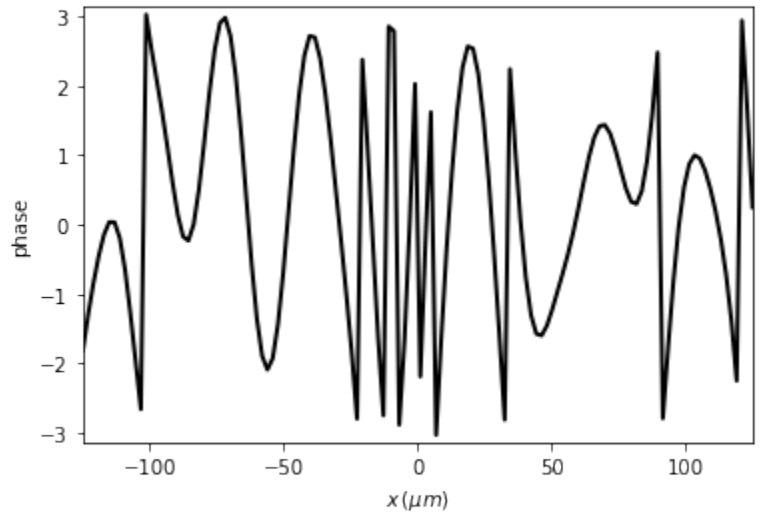
[15]:
```
t1.fresnel_lens(x0=0 * um,
                radius=100 * um,
                focal=.25 * mm,
                kind='phase',
                phase=np.pi)
t1.draw(kind='phase')
```



6.2.3.12 rough surface

[16]:
```
num_data = 128
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_X(x, wavelength)
t1.roughness(t=15 * um, s=.2 * um)
t1.draw(kind='phase')
```

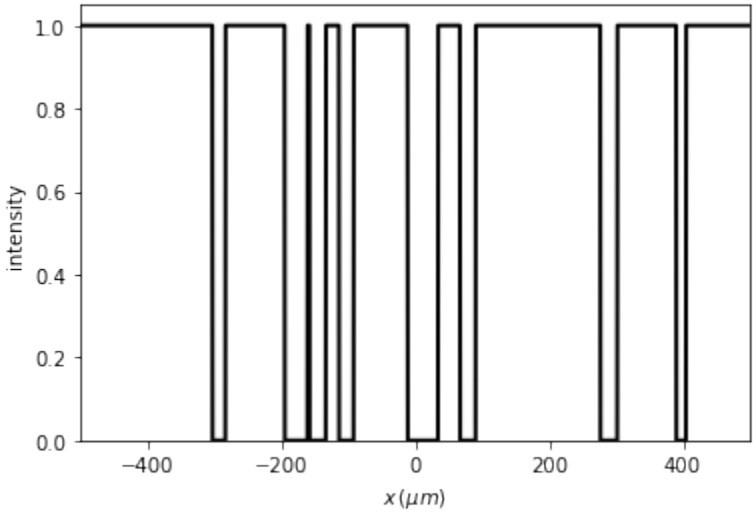


6.2.3.13 Dust particles with different sizas

TODO: does not work properly

[17]:
```
wavelength = 0.6328 * um
x = np.linspace(-500 * um, 500 * um, 2048)

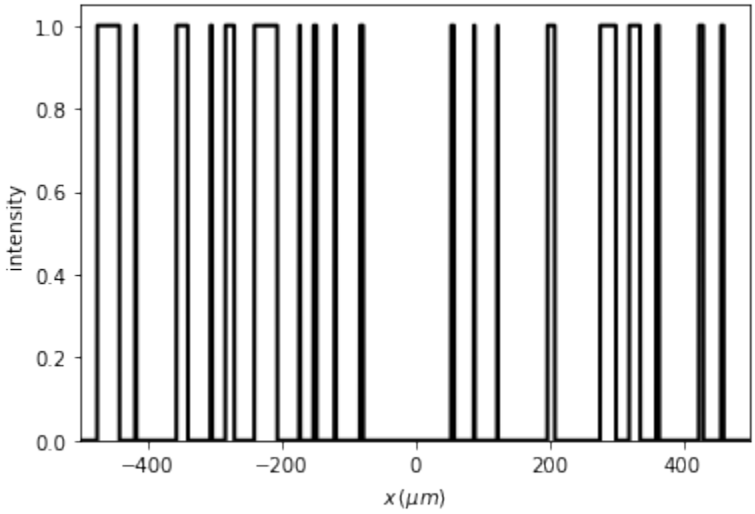
t1 = Scalar_mask_X(x, wavelength)
t1.dust_different_sizes(percentage=0.2, size=20 * um, std=5 * um)
t1.draw()
```



6.2.3.14 dust particles

[18]:
```
wavelength = 0.6328 * um
x = np.linspace(-500 * um, 500 * um, 2048 * 8)

t1 = Scalar_mask_X(x, wavelength)
t1.dust(percentage=0.9, size=20 * um)
t1.draw()
```

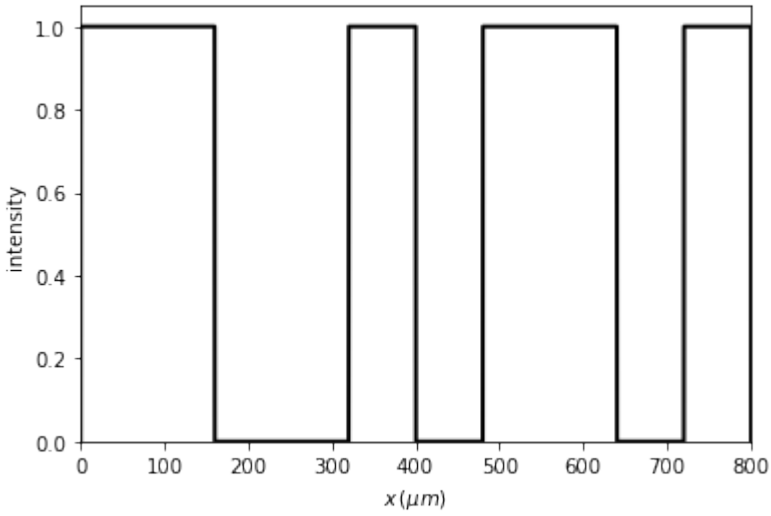


6.2.3.15 bit structure

[19]:
```
num_data = 2048 * 4

wavelength = 0.85 * um
code = [1, 1, 0, 0, 1, 0, 1, 1, 0, 1]
bit_width = 80 * um
x = np.linspace(0, bit_width * len(code), num_data)

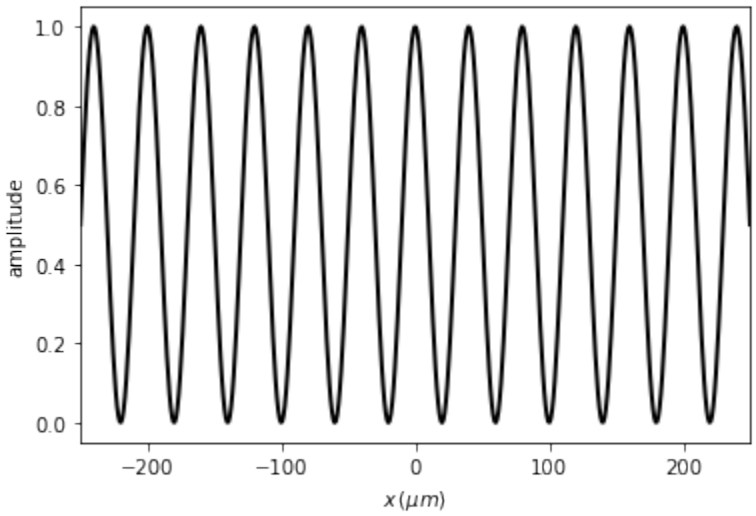
t1 = Scalar_mask_X(x, wavelength)
t1.binary_code(kind='normal', code=code, bit_width=bit_width, x0=0 * um)
t1.draw()
```



6.2.3.16 Gratings

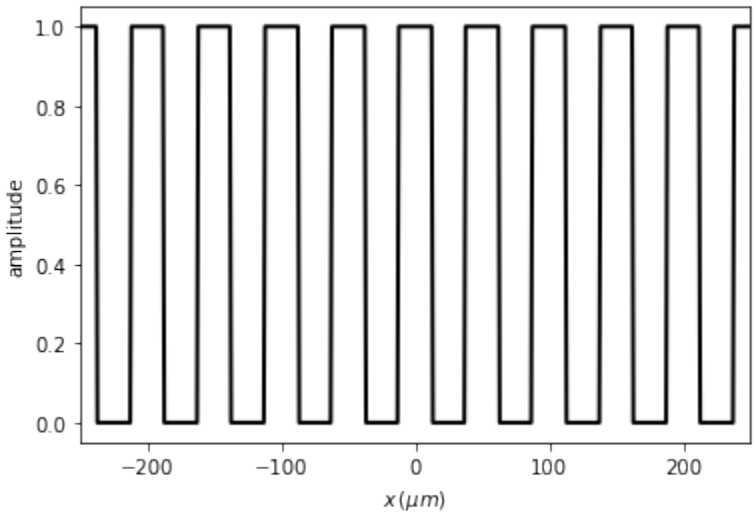
sine grating

[20]:
```
x = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
period = 40 * um
red = Scalar_mask_X(x, wavelength)
red.sine_grating(period=period, amp_min=0, amp_max=1, x0=0 * um)
red.draw(kind='amplitude')
```



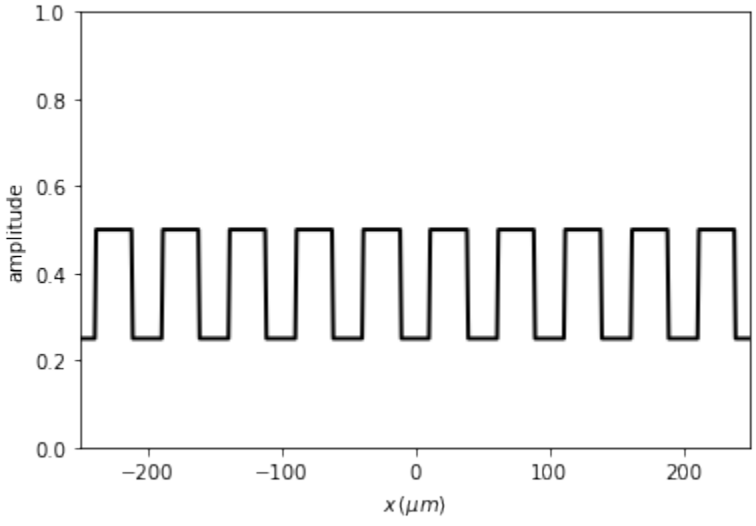
Ronchi grating

[21]:
```
x = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
red = Scalar_mask_X(x, wavelength)
red.ronchi_grating(period=50 * um, x0=0 * um, fill_factor=0.5)
red.draw(kind='amplitude')
```

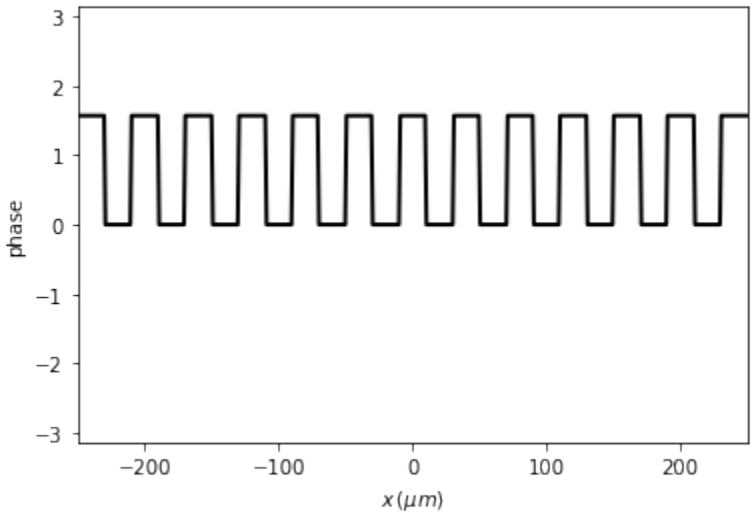


binary grating: amplitude and phase

[22]:
```
x = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
red = Scalar_mask_X(x, wavelength)
red.binary_grating(period=50 * um,
                   amin=0.25,
                   amax=0.5,
                   phase=np.pi,
                   x0=25 * um,
                   fill_factor=0.55)
red.draw(kind='amplitude')
plt.ylim(0, 1)
```

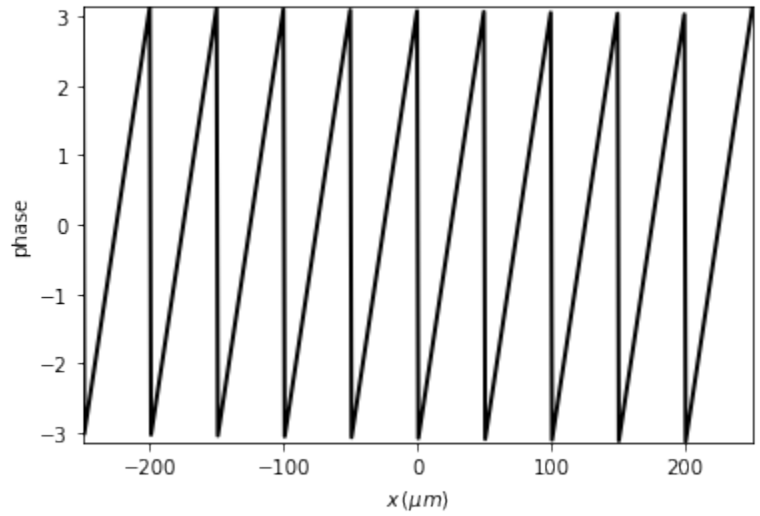


[23]:
```
x = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
period = 40 * um
red = Scalar_mask_X(x, wavelength)
red.binary_grating(period=period,
                   amin=1,
                   amax=1,
                   phase=np.pi / 2,
                   x0=0,
                   fill_factor=0.5)
red.draw(kind='phase')
```



blazed grating

[24]:
```
x = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
period = 50 * um
red = Scalar_mask_X(x, wavelength)
red.blazed_grating(x0=period/2, period=period, phase_max=2*np.pi)
red.draw(kind='phase')
```

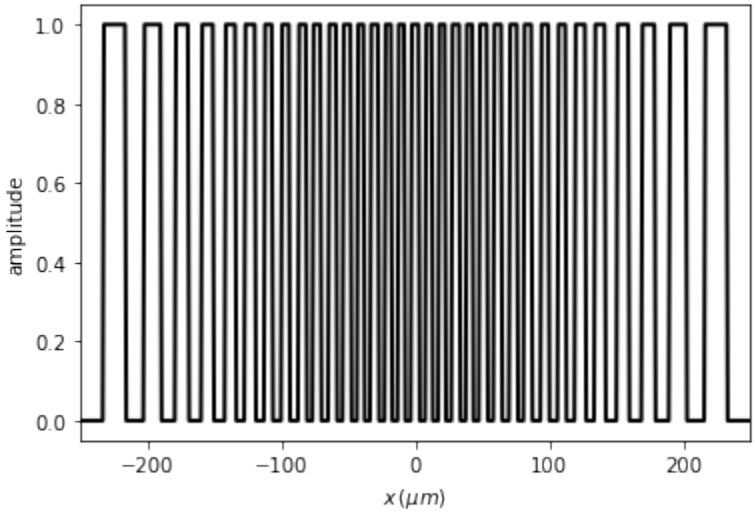


chirped grating: general

[25]:
```
x = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um

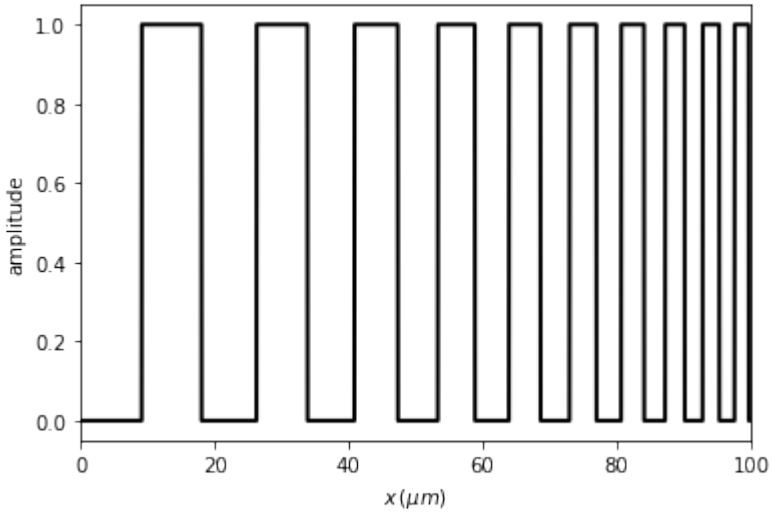
fx = '10+20*(self.x/length)**2'

red = Scalar_mask_X(x, wavelength)
red.chirped_grating(kind='amplitude_binary',
                    p_x=fx,
                    length=500 * um,
                    x0=0 * um,
                    amp_max=1,
                    amp_min=0,
                    delta_x=0,
                    phase_max=np.pi)
red.draw(kind='amplitude')
```



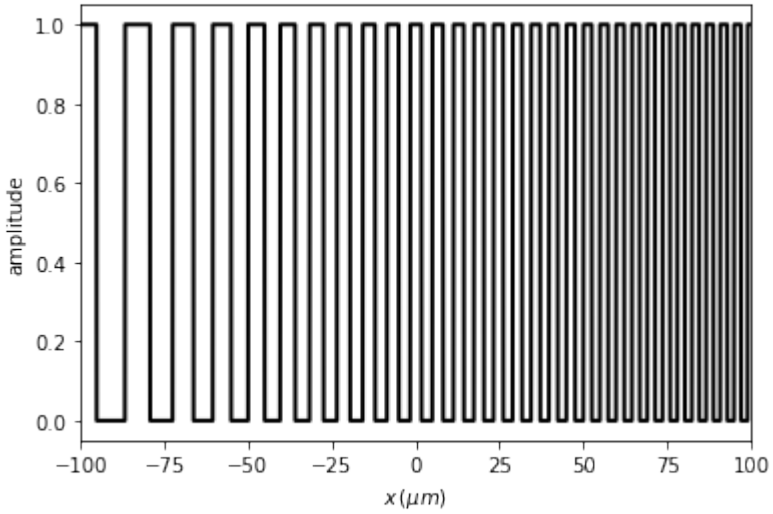
chirped grating: linear p

[26]:
```
x = np.linspace(0 * um, 100 * um, 4096 * 4)
wavelength = 0.6238 * um
red = Scalar_mask_X(x, wavelength)
conds = {
    'kind': 'amplitude_binary',
    'p0': 20 * um,
    'p1': 4 * um,
    'amp_min': 0,
    'amp_max': 1,
    'delta_x': 0,
    'phase_max': np.pi,
    'length': 0,  # equal to length of x
}
red.chirped_grating_p(**conds)
red.draw(kind='amplitude')
```



chirped grating: linear q

[27]:
```
x = np.linspace(-100 * um, 100 * um, 4096 * 4)
wavelength = 0.6238 * um
red = Scalar_mask_X(x, wavelength)
conds = {
    'kind': 'amplitude_binary',
    'p0': 20 * um,
    'p1': 4 * um,
    'amp_min': 0,
    'amp_max': 1,
    'delta_x': 0,
    'phase_max': np.pi,
    'length': 0,
}
red.chirped_grating_q(**conds)
red.draw(kind='amplitude')
```



6.3 Scalar_Z

6.3.1 Scalar field Z

[1]:
```
from diffractio import degrees, mm, um, nm
from diffractio import np, plt, sp

from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_sources_X import Scalar_source_X

from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_fields_Z import Scalar_field_Z
```

6.3.1.1 Definition of variables

[2]:
```
radius = 3 * mm

xin = np.linspace(-radius, radius, 4096)
xout = np.linspace(-150 * um, 150 * um, 128)

wavelength = 550 * nm

focal = 250 * mm
z = focal
```

[3]:
```
t0 = Scalar_mask_X(xin, wavelength)
t0.lens(x0=0, focal=focal, radius=radius)
```

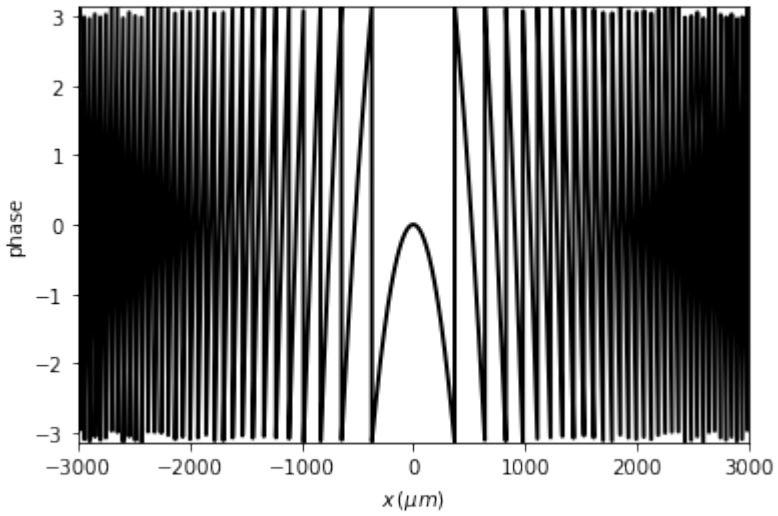
(continues on next page)

(continued from previous page)

```
u0 = Scalar_source_X(xin, wavelength)
u0.plane_wave(A=1)

u1 = t0 * u0
```

[4]:
```
u1.draw('phase')
```



[5]:
```
print(u1)
```

```
Scalar_field_X
 - x:  (4096,),   u:  (4096,)
 - xmin:       -3000.00 um,  xmax:      3000.00 um,  Dx:   1.47 um
 - Imin:       0.00,     Imax:      1.00
 - phase_min:  -179.70 deg, phase_max: 180.00 deg
 - wavelength: 0.55 um
 - date:       2023-10-07_17_30_14
```

6.3.1.2 CZT algorithm

[6]:
```
%%time
zs = np.linspace(focal - 50 * mm, focal + 50 * mm, 1024)
u_z = u1.CZT(zs,0,0)
```

```
CPU times: user 3.3 s, sys: 219 µs, total: 3.3 s
Wall time: 3.31 s
```

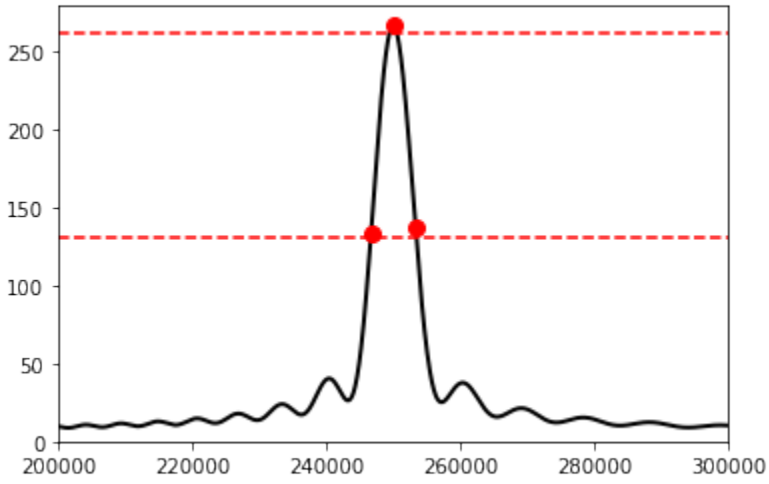
[7]:
```
print(u_z)
```

```
Scalar_field_Z
 - z:  (1024,),   u:  (1024, 1)
 - zmin:       200000.00 um,  zmax:      300000.00 um,  Dz:   97.75 um
 - Imin:       4.36,     Imax:      261.70
 - phase_min:  -179.73 deg, phase_max: 179.85 deg
 - wavelength: 0.55 um
 - date:       2023-10-07_17_30_18
```

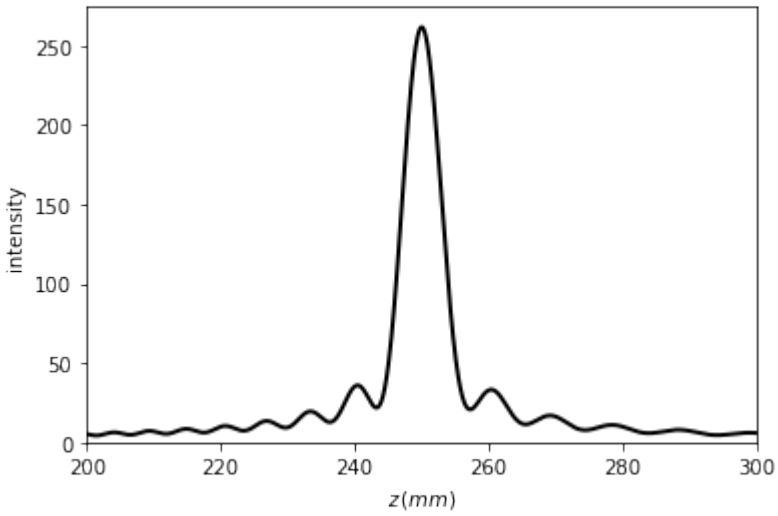
6.3.1.3 Scalar z

[8]:
```
dof = u_z.DOF(percentage=0.5, remove_background=True, has_draw=True)
print("DOF = {:2.2f} mm".format(dof / mm))
```

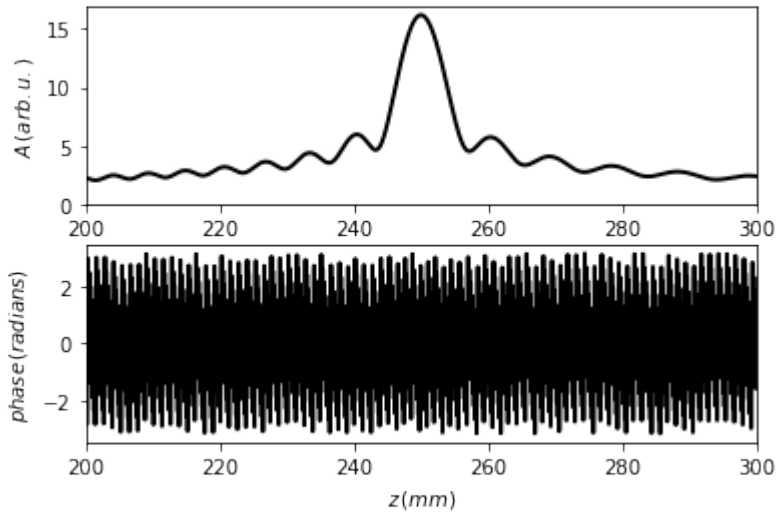
```
DOF = 6.64 mm
```



[9]:
```
u_z.draw(z_scale='mm')
```



[10]:
```
u_z.draw('field', z_scale='mm')
```



6.4 Scalar_XZ

6.4.1 Characteristics

Scalar_XZ is a set of two modules for:

- Generation of 2D (xz-axis) masks and diffractive optical elements.
- Drawing sources, masks and fields.
- Propagation of light, determination of parameters, and other functions.

Light sources are loaded into the module using Scalar_field_X, using the incident_field function.

These modules are named: scalar_fields_XZ.py and scalar_masks_XZ.py.

Each module present a main class:

- Scalar_field_XZ
- Scalar_masks_XZ

The main attributes for these classes are the following:

- self.x (numpy.array): linear array with x equidistant positions. The number of data is preferibly :math:`2^n` .
- self.z (numpy.array): linear array with z equidistant positions.
- self.wavelength (float): wavelength of the incident field.
- self.n (numpy.array): linear 2D array with size x * z storing the refraction index.
- self.u0 (numpy.array): linear array with x equidistant positions storing the incident complex field.
- self.u (numpy.array): linear 2D array with size x * z storing the complex field.
- self.n_background (numpy.float): refraction index of background.

We can also find these atributes:

- self.X (numpy.array): linear 2D array with size x * z storing X position.
- self.Z (numpy.array): linear 2D array with size x * z storing Z position.
- self.quality (float): quality of RS algorithm. Valid for values > 1.
- self.reduce_matrix (bool): Indicates if matrix are reduced for drawing purposes.
- self.info (str): description of data.
- self.fast (bool): If True, Rayleigh-Sommerfeld computations are performed using Fresnel approximation.
- self.type (str): Class of the field.
- self.date (str): date when performed.

The dimensional magnitudes are related to microns: `micron = 1.`

Creating an instance

[1]:
```
from diffractio import degrees, eps, mm, um, num_max_processors
from diffractio import np, plt, sp
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[2]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024)
z0 = np.linspace(-100 * um, 600 * um, 1024)
wavelength = 0.6238 * um

u1 = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
```

Creating a light source Light sources for XZ propagation are defined in the scalar_sources_x.py module.

Example: plane wave.

[3]:
```
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1.incident_field(u0)
```

The light beam can also be placed at any plane in the z scale, not only at the beginning:

[4]:
```
u1.incident_field(u0, z0=200 * um)
```

Masks

Masks are defined in the scalar_masks_xz.py module. Masks can be defined in several ways.

- In the first place, a linear plane mask can be added using Scalar_mask_X functions.
- Also, volumetric XZ masks can be defined using surfaces, arrays, etc.
- Specific XZ masks are defined.
- Most mask can be displaced in the x-z plane and rotated a certaint θ angle.
- The refraction index of the mask can be real numbers (transparent) of complex (absorbent).
- Refraction index defining the masks can be constant or defined by n(x,z) functions.

There are many methods that can be used to generate masks and diffractive optical elements:

Generation of general masks

- **extrude_mask**: Converts a Scalar_mask_X in volumetric xz.
- **mask_from_function**: Phase mask defined between two surfaces f1 and f1: h(x,z)=f2(x,z)-f1(x,z)
- **mask_from_array**: Mask defined between two surfaces given by arrays (x,z): h(x,z)=f2(x,z)-f1(x,z).
- **object_by_surfaces**: "Mask defined by n surfaces given in array Fs={f1, f2, ...}
- **add_surfaces**: A topography fx is added to one of the faces of object u (self.n).
- **discretize_refractive_index**: Takes a refraction index an discretize it according refraction indexes.

Specific XZ masks

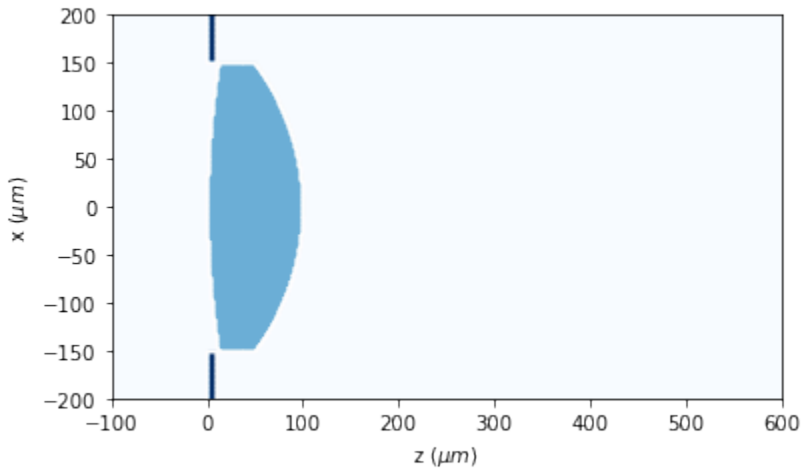
- **image**: Converts an image file in an xz-refraction index matrix.
- **semi_plane**: Inserts a semi-sphere in background (x>0).
- **layer**: Insert a layer with a certain depth.
- **rectangle**: Insert a rectangle in background.
- **slit**: Insert a slit in background.
- **sphere**: Insert a sphere in background.
- **semi_sphere**: Insert a semi_sphere in background.
- **lens_plane_convergent**: Insert a plane-convergent lens in background.
- **lens_plane_convergent**: Inserts a convergent lens in background.
- **lens_plane_divergent**: "Insert a plane-divergent lens in background.
- **lens_divergent**: Insert a divergent lens in background.
- **aspheric_surface_z** (unfinished now).
- **aspheric_lens**: Define an aspheric lens.
- **wedge**: Insert a wedge.
- **prism**: Similar to wedge but the use is different.
- **biprism**: Fresnel biprism.

- **ronchi_grating**: Insert a ronchi grating in background.
- **sine_grating**: Insert a sine grating in background
- **probe**: Probe with a sinusoidal shape.
- **rough_sheet**: Sheet with one rough surface.
- **wessenplatten**

For a more detailed description of each method, refer to the individual documentation of each one.

Example: convergent lens

[5]:
```
t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.lens_convergent(r0=(0, 0),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
                   thickness=100 * um,
                   refractive_index=2,
                   angle=0 * degrees,
                   mask=(10 * um, 3 + 0.05j))
t1.draw_refractive_index(scale='scaled')
```



Example: sphere with variable refraction index:

[6]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = 0.5 * um

t2 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.0)

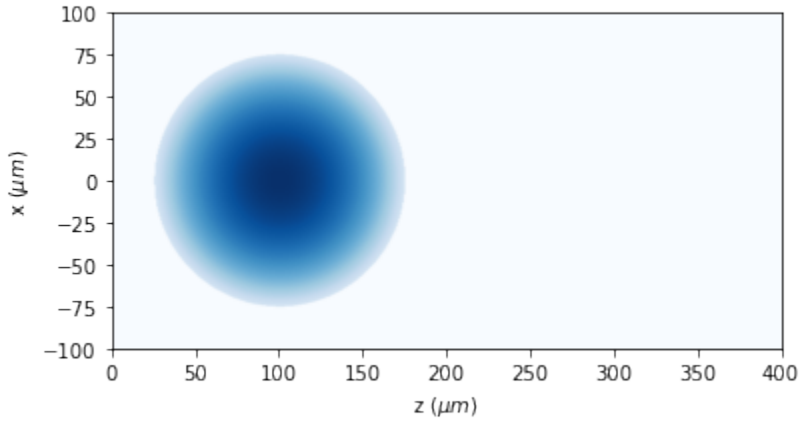
pn = dict(n_out=1.5, n_center=4, cx=0 * um, cz=100 * um, radius=75 * um)

center = (pn['cx'], pn['cz'])
radius = pn['radius']
ref_index = "{p[n_out]}+({p[n_center]}-{p[n_out]})*(1-((X-{p[cx]})**2+(Z-{p[cz]})**2)/
↪{p[radius]}**2)".format(
    p=pn)
```

(continues on next page)

(continued from previous page)

```
t2.sphere(r0=center,
          radius=(radius, radius),
          refractive_index=ref_index,
          angle=0)
t2.draw_refractive_index(draw_borders=False, scale='scaled')
```



6.4.2 Propagation and other parameters

Propagation and other actions and parameters of the optical fields are defined in the scalar_field_x.py module. There are several methods of determining the field at a given plane after the mask:

- **RS**: Rayleigh-Sommerfeld propagation at a certain distance
- **fft**: Fast Fourier propagation at the far field.
- **ifft**: Inverse Fast Fourier propagation at the far field.
- **BPM**: Beam propagation method at a near distance.

The field can be stored in the same instance, generate a new instance, or generate a numpy.array when fast computation is required.

6.4.2.1 Rayleigh-Sommerfeld propagation

For Rayleigh-Sommerfeld propagation, it is not possible to consider variations of refraction index, but only propagation in vaccum or an constant refraction index.

Nevertheless, the field at a distant plane from the object can be determined easily.

[7]:
```
wavelength = .5 * um
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(1 * mm, 3 * mm, 512)

t1 = Scalar_mask_X(x0, wavelength)
t1.lens(x0=0 * um, radius=100 * um, focal=2 * mm)

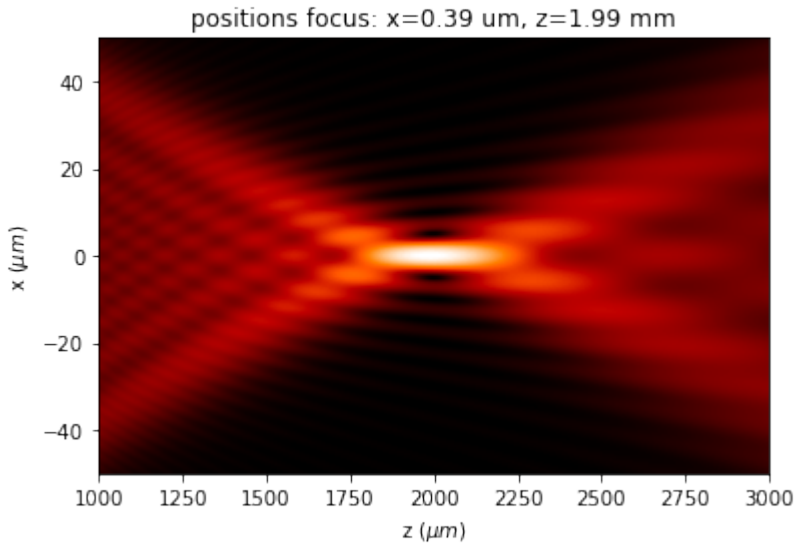
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

[8]:
```
u1 = u0 * t1

u2 = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
u2.incident_field(u1)
u2.RS()
```

[9]:
```
u2.draw(logarithm=True, draw_borders=False, scale='')
plt.ylim(-50, 50)

x_f, z_f = u2.search_focus(verbose=False)
text = "positions focus: x={:2.2f} um, z={:2.2f} mm".format(x_f, z_f / mm)
plt.title(text)
```



6.4.2.2 Wave Propagation Method (WPM)

For Wave Propagation Method, propagations should be performed incrementally, from a plane z to another z+Δz. The refraction index at each position is considered and, then, we can consider the shape of the optical elements.

[10]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 200 * um, 512)
wavelength = 5 * um

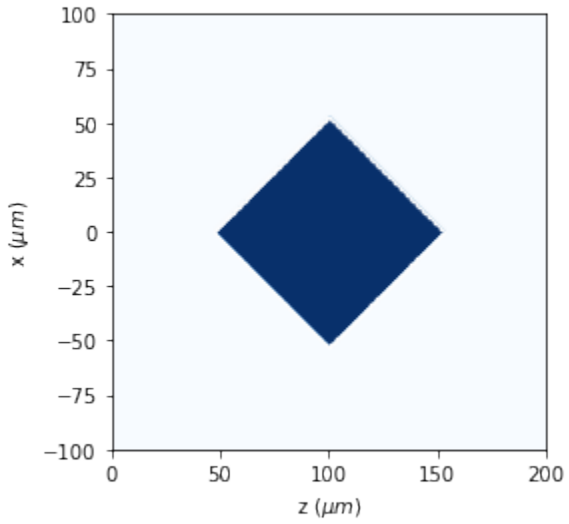
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=10 * um, theta=0. * degrees)
u0.plane_wave(A=1, theta=0 * degrees)
```

[11]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.rectangle(r0=(0 * um, 100 * um),
             size=(75 * um, 75 * um),
             angle=45 * degrees,
```

(continues on next page)

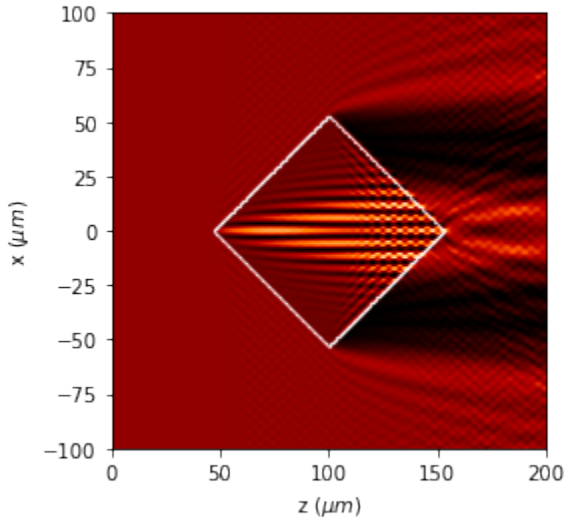
(continued from previous page)

```
            refractive_index=1.5)
u1.draw_refractive_index(scale='scaled')
```



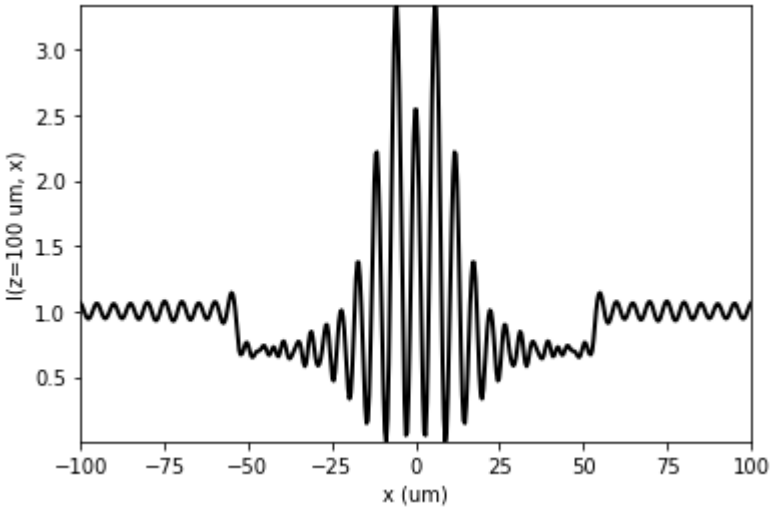
[12]:
```
edges = u1.surface_detection()
```

[13]:
```
u1.WPM(verbose=False, has_edges=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges,
        scale='scaled')
```

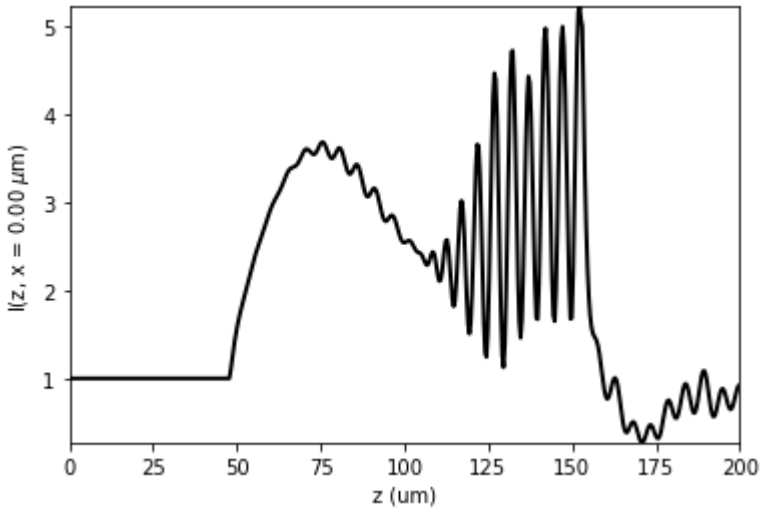


Also intensity, phase, amplitude profiles can be easily obtained using the class methods.

[14]:
```
u1.profile_transversal(z0=100 * um)
```



[15]:
```
u1.profile_longitudinal(x0=0 * um)
```



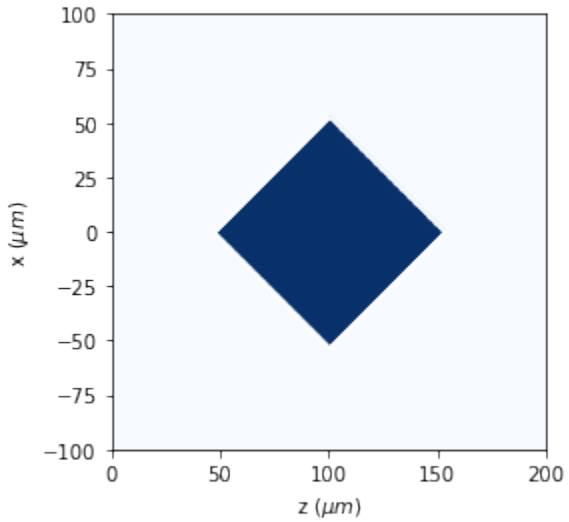
6.4.2.3 Beam Propagation Method (BPM)

For Beam Propagation Method, propagations should be performed incrementally, from a plane z to another z+Δz. The refraction index at each position is considered and, then, we can consider the shape of the optical elements.

[16]:
```
x0 = np.linspace(-100 * um, 100 * um, 1024)
z0 = np.linspace(0 * um, 200 * um, 1024)
wavelength = 5 * um

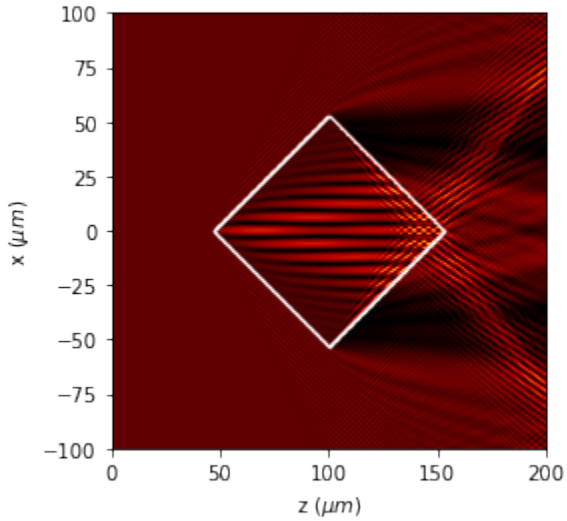
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=10 * um, theta=0. * degrees)
u0.plane_wave(A=1, theta=0 * degrees)
```

[17]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.rectangle(r0=(0 * um, 100 * um),
             size=(75 * um, 75 * um),
             angle=45 * degrees,
             refractive_index=1.5)
u1.draw_refractive_index(scale='scaled')
```



[18]:
```
edges = u1.surface_detection()
```

[19]:
```
u1.clear_field()
u1.BPM(verbose=False, has_edges=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges,
        scale='scaled')
```



6.4.3 Save and load masks and fields

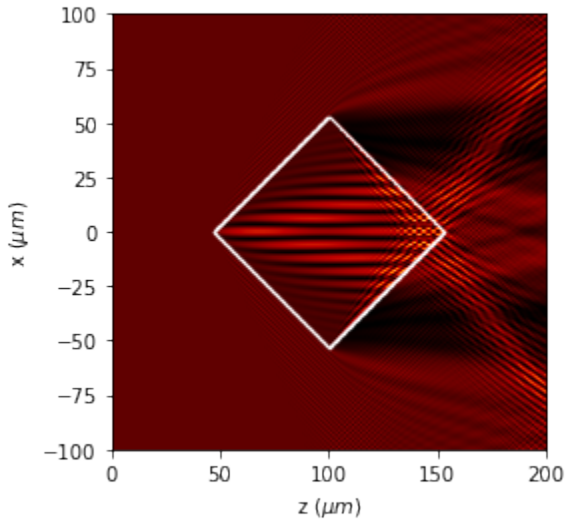
XZ module allows to save data to files and after recover those data.

[20]:
```
u1.save_data(filename='save_load.npz')
```

[21]:
```
del u1
```

[22]:
```
u2 = Scalar_field_XZ(None, None, None)
u2.load_data(filename='save_load.npz', verbose=True)
u2.draw(logarithm=True, draw_borders=True, scale='scaled')
```

```
dict_keys(['x', 'z', 'wavelength', 'n_background', 'fast', 'quality', 'borders', 'CONF_
↪DRAWING', 'X', 'Z', 'u0', 'u', 'n', 'info', 'reduce_matrix', 'type', 'date',
↪'description'])
```

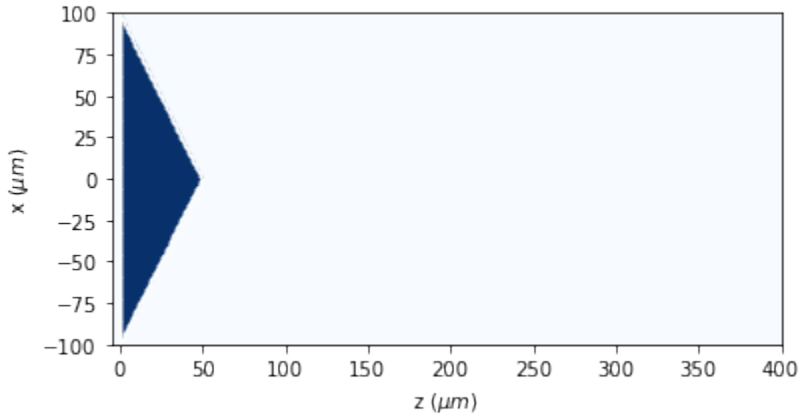


6.4.4 Surface detection of masks

As XZ masks present a certain volume, we can detect the edges of the optical elements in order to a better representation.

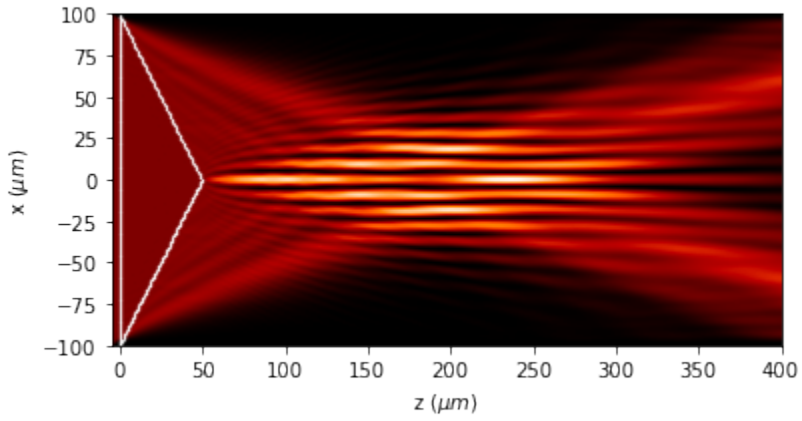
[23]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(-5 * um, 400 * um, 512)
wavelength = 5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(theta=0. * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.biprism(r0=(0, 0),
           length=200 * um,
           height=50 * um,
           refractive_index=1.5,
           angle=0)

u1.draw_refractive_index(draw_borders=True, scale='scaled')
```



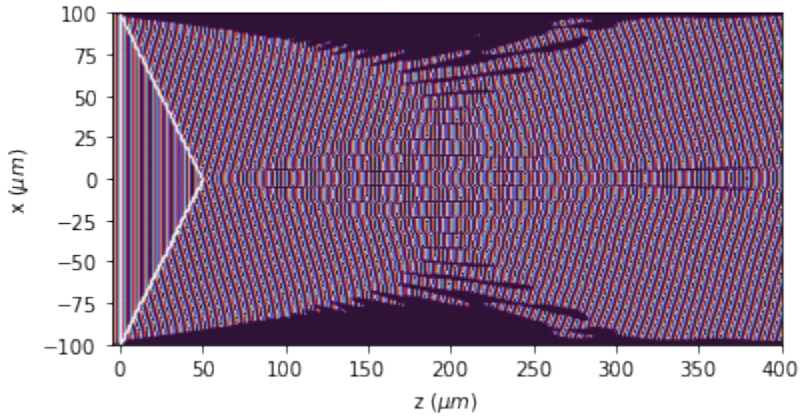
The edges can be included in other figures, in order to improve the representation of fields. For example in the intensity distribution of the optical field.

[24]:
```
u1.WPM()
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
```



Also, a better analysis of the phase can be performed using the edge detection tool.

[25]:
```
u1.draw(kind='phase', draw_borders=True, scale='scaled')
```



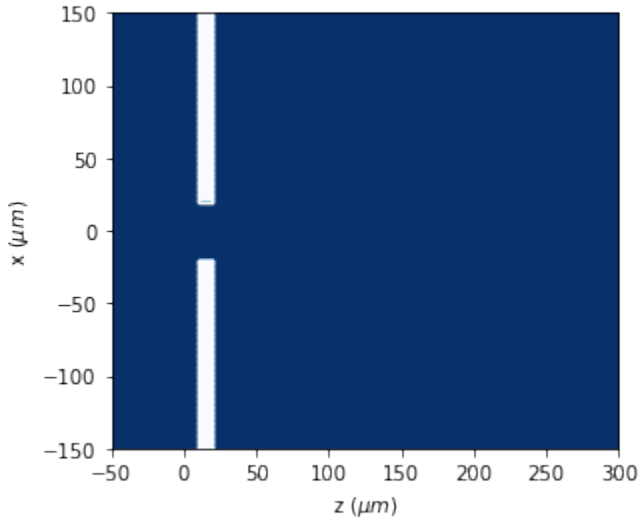
6.4.5 Reduction of the field

Sometimes, as when light is focused by a lens, we need to analyze a small part of the processed area. We can do this using cut_resample function.

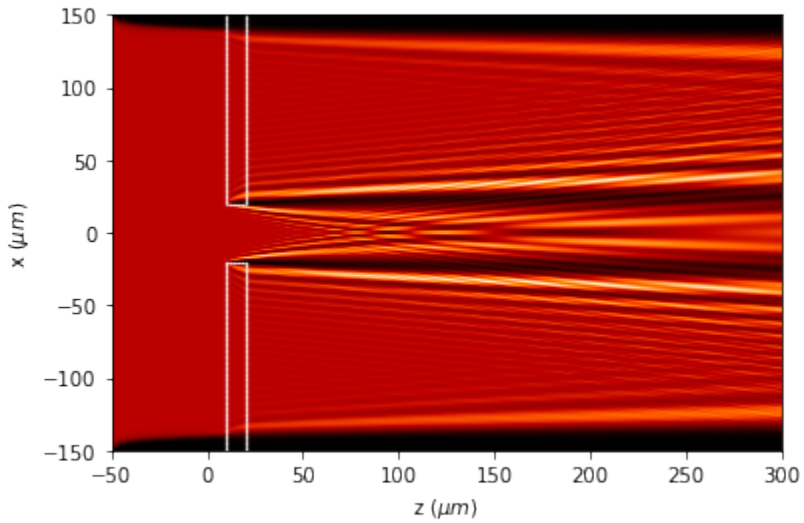
[26]:
```
x0 = np.linspace(-150 * um, 150 * um, 512)
z0 = np.linspace(-50 * um * um, 300 * um, 512)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)

u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=4)
u1.incident_field(u0)
u1.slit(r0=(0 * um, 10 * um),
        aperture=40 * um,
        depth=10 * um,
        refractive_index=1,
        refractive_index_center='',
        angle=0 * degrees)
u1.draw_refractive_index(draw_borders=True, scale='scaled')
```

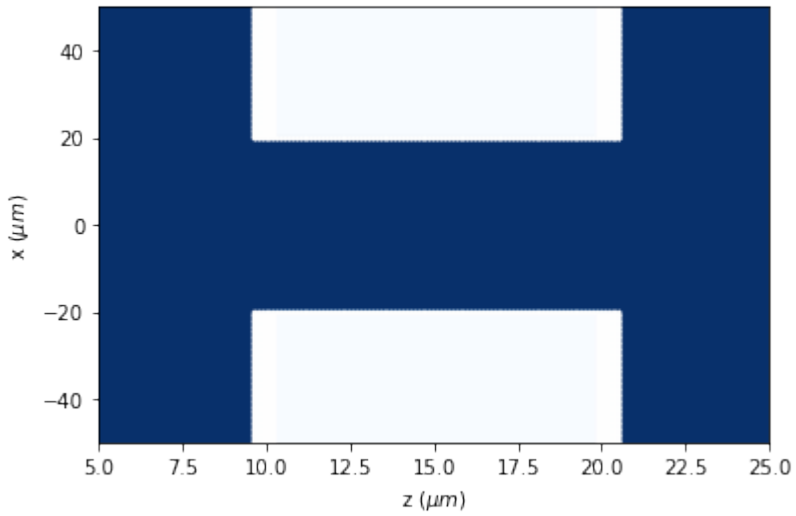


[27]:
```
u1.WPM(verbose=False)
u1.draw(kind='intensity', draw_borders=True)
```

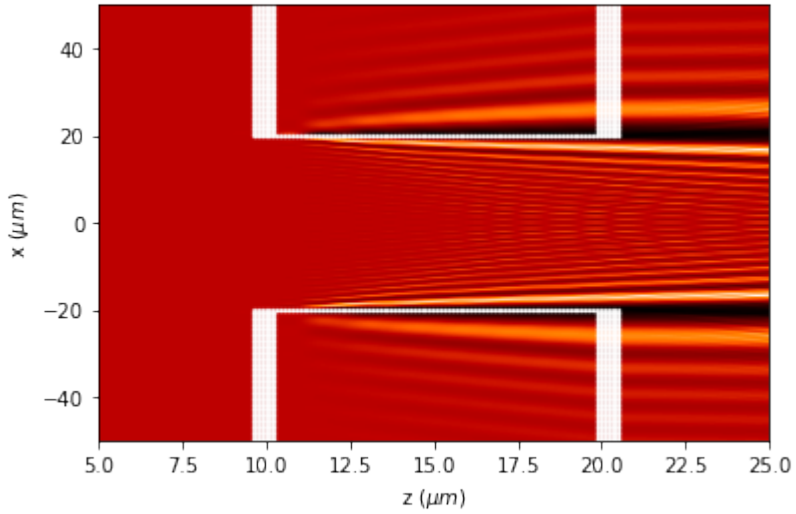


[28]:
```
u1.cut_resample(x_limits=(-50, 50),
                z_limits=(5, 25),
                num_points=(512, 512),
                new_field=False)

u1.draw_refractive_index(draw_borders=True)
```



[29]:
```
u1.draw(kind='intensity', draw_borders=True)
```



6.4.6 Incident field: location

When defining a light source, it is usually placed at the first plane of the XZ area.

[30]:
```
x0 = np.linspace(-60 * um, 60 * um, 512)
z0 = np.linspace(0 * um, 120 * um, 512)
wavelength = 5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=10 * um, theta=0. * degrees)

u1 = Scalar_source_X(x=x0, wavelength=wavelength)
u1.gauss_beam(A=1, x0=40 * um, z0=0 * um, w0=10 * um, theta=-30. * degrees)
```

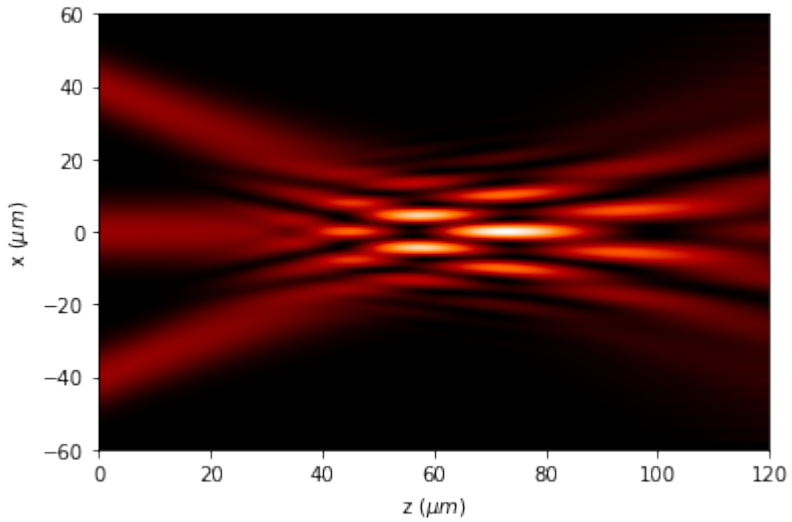
(continues on next page)

(continued from previous page)

```
u2 = Scalar_source_X(x=x0, wavelength=wavelength)
u2.gauss_beam(A=1, x0=-40 * um, z0=0 * um, w0=10 * um, theta=30. * degrees)

u3 = u0 + u1 + u2

u4 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u4.incident_field(u3)
u4.WPM()
u4.draw(kind='intensity', logarithm=True)
```



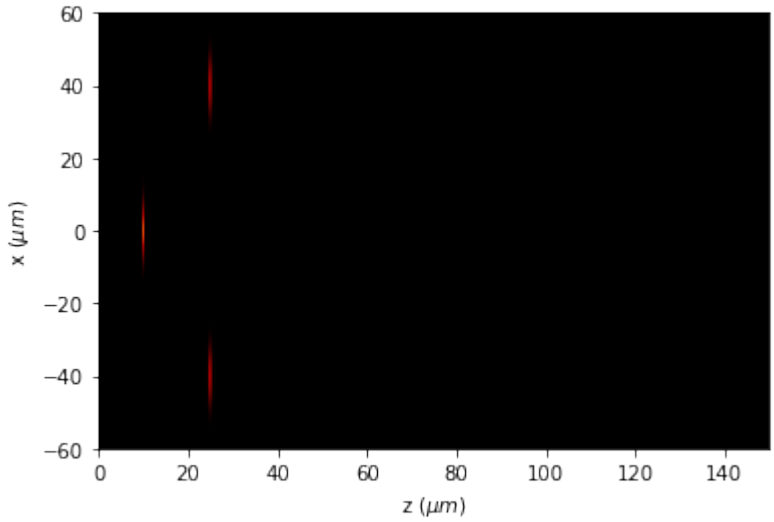
Nevertheless, we can locate the source at any plane:

[31]:
```
x0 = np.linspace(-60 * um, 60 * um, 512)
z0 = np.linspace(0 * um, 150 * um, 512)
wavelength = 5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=10 * um, theta=0. * degrees)

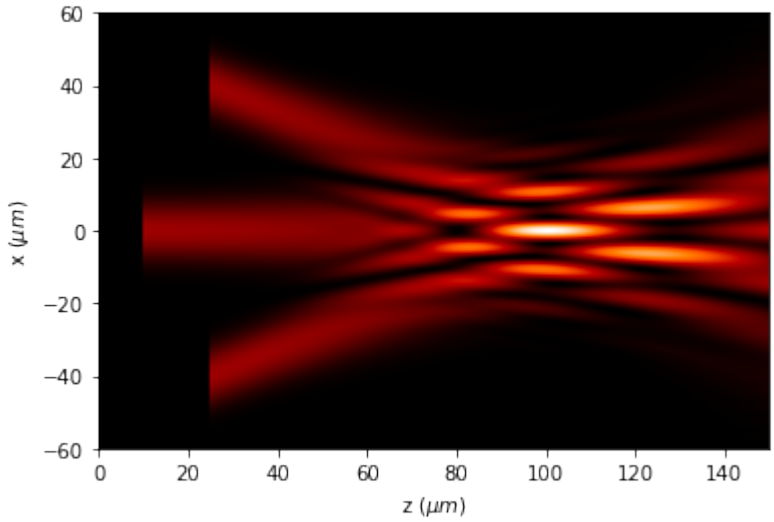
t0 = Scalar_source_X(x=x0, wavelength=wavelength)
t0.gauss_beam(A=1, x0=40 * um, z0=0 * um, w0=10 * um, theta=-25. * degrees)

t1 = Scalar_source_X(x=x0, wavelength=wavelength)
t1.gauss_beam(A=1, x0=-40 * um, z0=0 * um, w0=10 * um, theta=25. * degrees)

u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0, z0=10 * um)
u1.incident_field(t0, z0=25 * um)
u1.incident_field(t1, z0=25 * um)
u1.draw(kind='intensity', logarithm=True)
```



[32]:
```
u1.WPM()
u1.draw(kind='intensity', logarithm=True)
```



6.4.7 Recovering the final field

After processing the simulation, many a times the final result is the last plane of the simulation, which can be easily recovered using final_field() method.

[33]:
```
x0 = np.linspace(-125 * um, 125 * um, 2048)
z0 = np.linspace(0 * um, 500 * um, 2048)
wavelength = 5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)

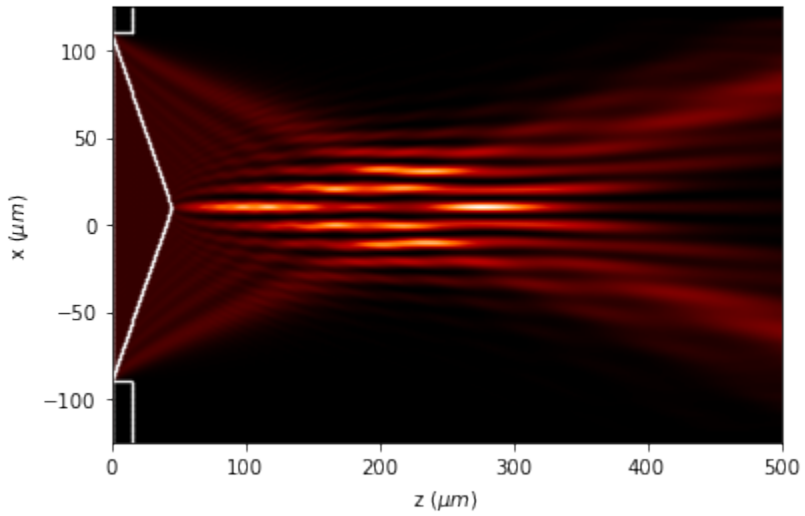
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
```

(continues on next page)

(continued from previous page)

```
u1.incident_field(u0)
u1.slit(r0=(10 * um, 0 * um),
        aperture=200 * um,
        depth=15 * um,
        refractive_index=1 + .5j,
        refractive_index_center='',
        angle=0 * degrees)
u1.biprism(r0=(10 * um, 0 * um),
           length=200 * um,
           height=45 * um,
           refractive_index=1.5,
           angle=0 * degrees)
u1.WPM()

u1.draw(kind='intensity', draw_borders=True)
```



6.4.8 Beam Propagation Method: Filtering

When the pixel size is great enough, sometimes the shape of the objects is not correctly sampled. Then, artifacts can appear in the propagation. To solve this, a filtering of the refraction index can be performed. Obviously, this filtering should be the minimum so that the simulation is accurate.

Due to the simulation process, fitering can be performed in BPM propagation, but not for WPM.

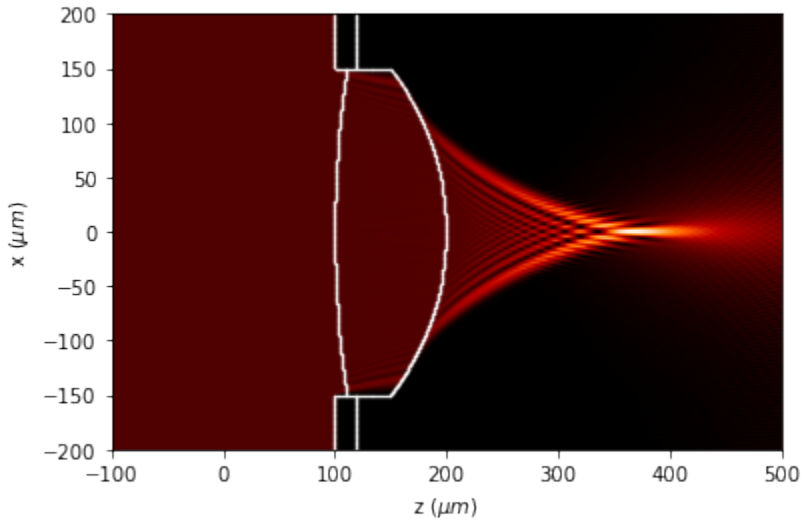
6.4.8.1 Simulation properly sampled

[34]:
```
x0 = np.linspace(-200 * um, 200 * um, 2048)
z0 = np.linspace(-100 * um, 500 * um, 1024 * 8)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

u1.lens_convergent(r0=(0, 100),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
                   thickness=100 * um,
                   refractive_index=2,
                   angle=0 * degrees,
                   mask=(20 * um, 3 + 0.5j))

edges = u1.surface_detection()
u1.borders = edges
u1.WPM(has_edges=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges)
```



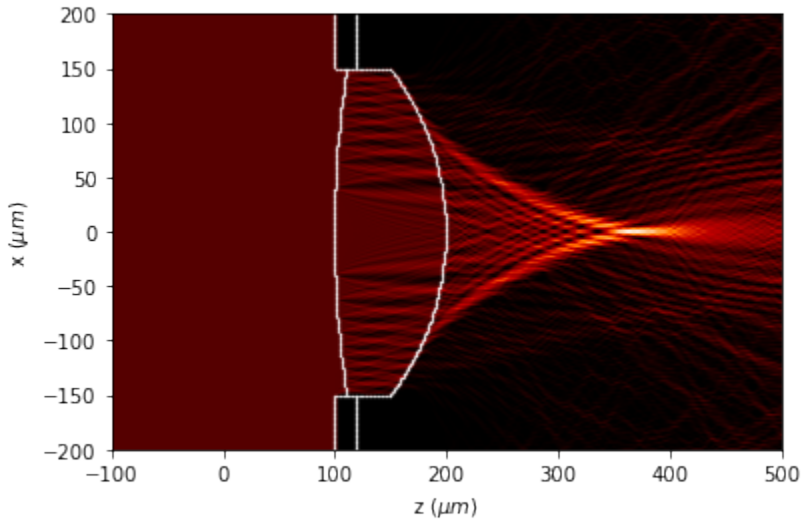
6.4.8.2 Simulation with bad sampling

[35]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(-100 * um, 500 * um, 512)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

u1.lens_convergent(r0=(0, 100),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
                   thickness=100 * um,
                   refractive_index=2,
                   angle=0 * degrees,
                   mask=(20 * um, 3 + 0.5j))

edges = u1.surface_detection()
u1.borders = edges
u1.WPM(has_edges=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges)
```



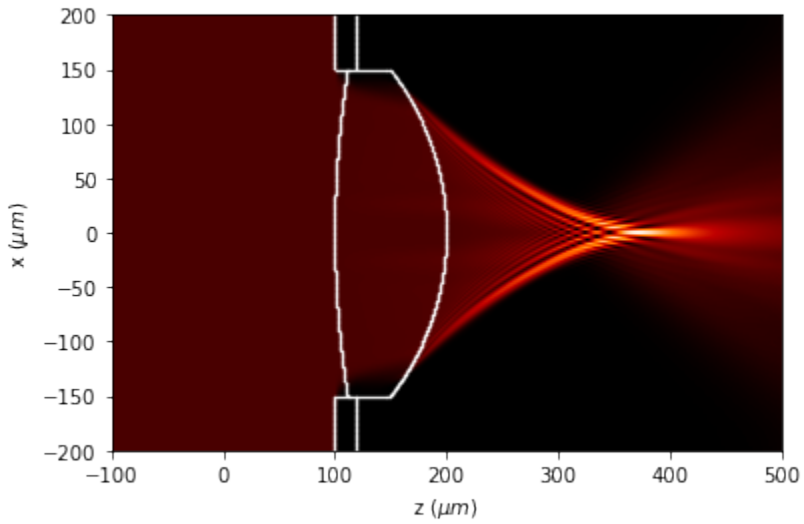
6.4.8.3 Simulation with filtering

[36]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024)
z0 = np.linspace(-100 * um, 500 * um, 1024)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

u1.lens_convergent(r0=(0, 100),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
                   thickness=100 * um,
                   refractive_index=2,
                   angle=0 * degrees,
                   mask=(20 * um, 3 + 0.5j))

edges = u1.surface_detection()
u1.borders = edges
u1.smooth_refractive_index(type_filter=2,
                           pixels_filtering=8,
                           max_diff_filter=0.05,
                           draw_check=False)
u1.BPM(has_edges=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges)
```



6.4.9 Rotating masks and fields

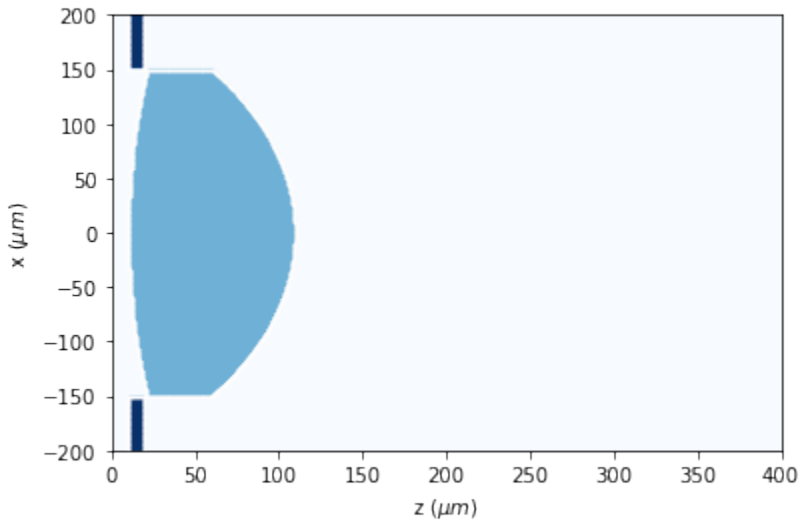
When developing a simulation, most functions present the posibility of rotating light sources and masks.

[37]:
```
x0 = np.linspace(-200 * um, 200 * um, 2048)
z0 = np.linspace(-0 * um, 400 * um, 2048)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

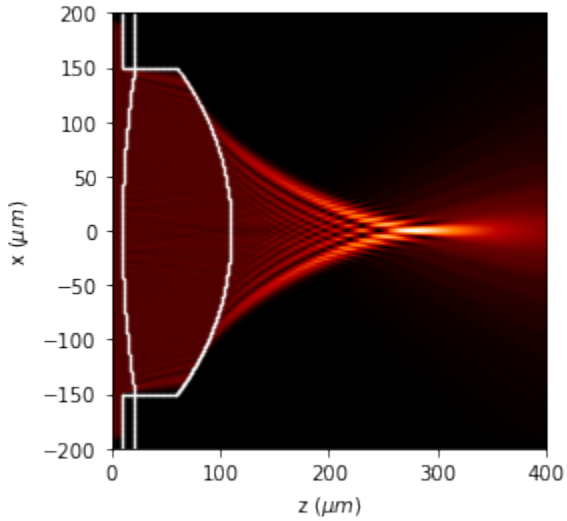
u1.lens_convergent(r0=(0, 10),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
                   thickness=100 * um,
                   refractive_index=2,
                   angle=0 * degrees,
                   mask=(10 * um, 3 + 0.5j))

edges = u1.surface_detection()
u1.draw_refractive_index(draw_borders=True)
```



[38]:
```
u1.WPM(verbose=False)
```

[39]:
```
u1.draw(logarithm=True,
        normalize='maximum',
        scale='scaled',
        draw_borders=True,
        edge_matrix=edges)
```



Rotate_field rotates all the field, including the light fields.

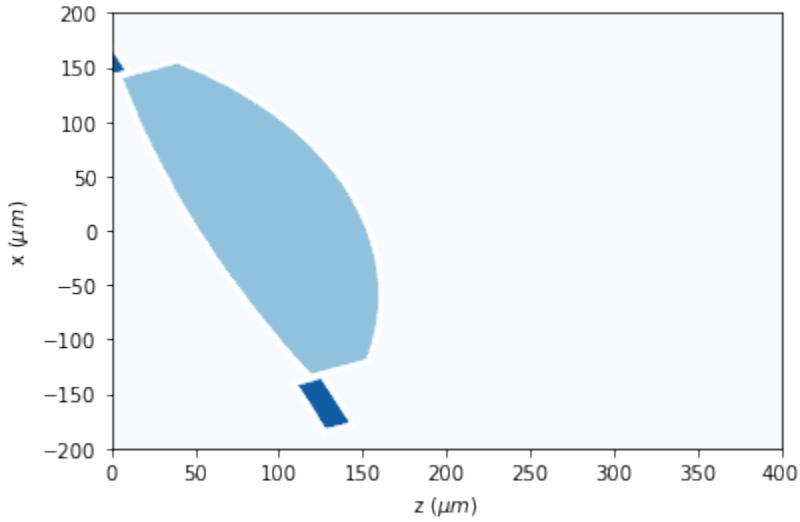
[40]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024)
z0 = np.linspace(-0 * um, 400 * um, 1024)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

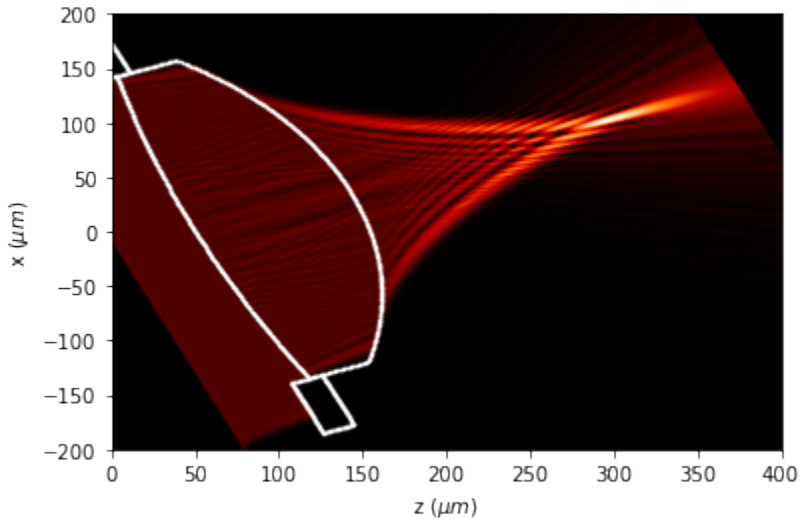
u1.lens_convergent(r0=(0, 50),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
                   thickness=100 * um,
                   refractive_index=2,
                   angle=0 * degrees,
                   mask=(20 * um, 3 + 0.5j))
```

[41]:
```
u1.WPM(verbose=False)

u1.rotate_field(angle=22.5 * degrees, center_rotation=(0, 50), kind='all')
edges = u1.surface_detection()
u1.draw_refractive_index(draw_borders=True)
```



[42]:
```
u1.draw(logarithm=True, draw_borders=True, edge_matrix=edges)
```



Nevertheless, we can clear the fields, and compute it again with the masks rotated.

[43]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024)
z0 = np.linspace(-10 * um, 400 * um, 1024)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)

u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

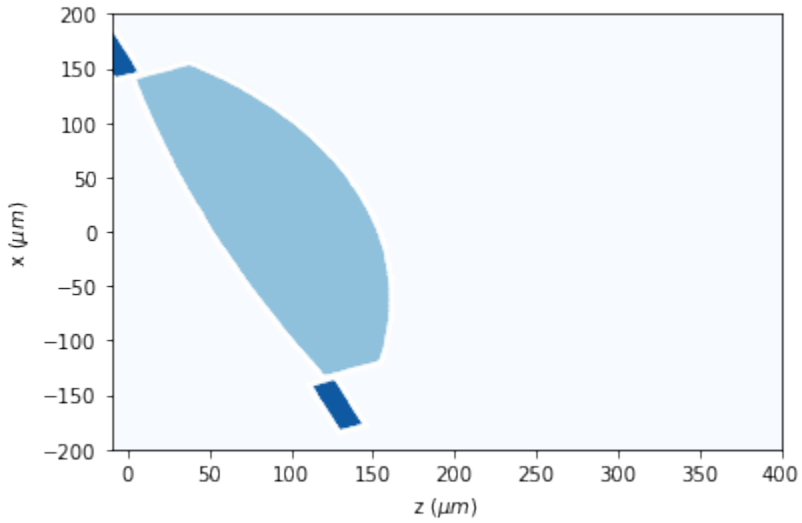
u1.lens_convergent(r0=(0, 50),
                   aperture=300 * um,
                   radius=(1000 * um, -250 * um),
```

(continues on next page)

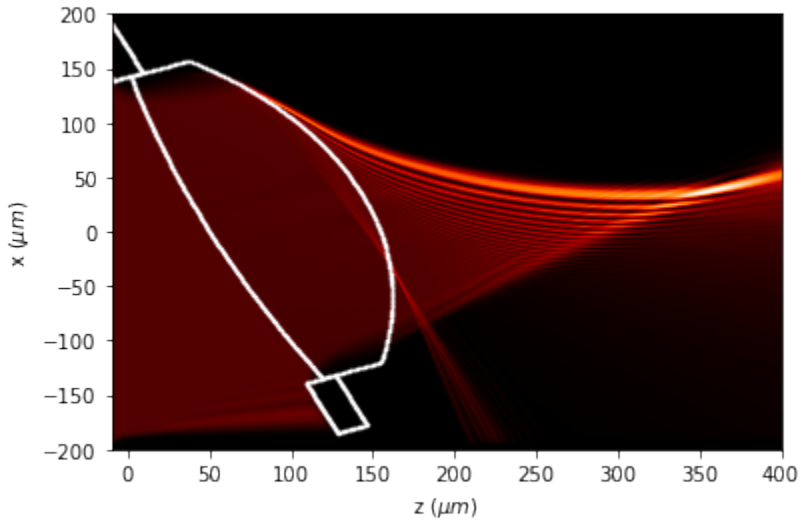
(continued from previous page)

```
                    thickness=100 * um,
                    refractive_index=2,
                    angle=0 * degrees,
                    mask=(20 * um, 3 + 0.5j))

u1.rotate_field(angle=22.5 * degrees, center_rotation=(0, 50), kind='all')
edges = u1.surface_detection()
u1.draw_refractive_index(draw_borders=True)
```



[44]:
```
u1.smooth_refractive_index(type_filter=2,
                           pixels_filtering=8,
                           max_diff_filter=0.05,
                           draw_check=False)
u1.BPM(verbose=False, has_edges=True)
u1.draw(logarithm=True, draw_borders=True, edge_matrix=edges)
```



6.4.10 Find focus tool

When focusing a light field, with a lens for example, sometimes we need to determine the location of maximum intensity. This can be easily determined using search_focus() method.

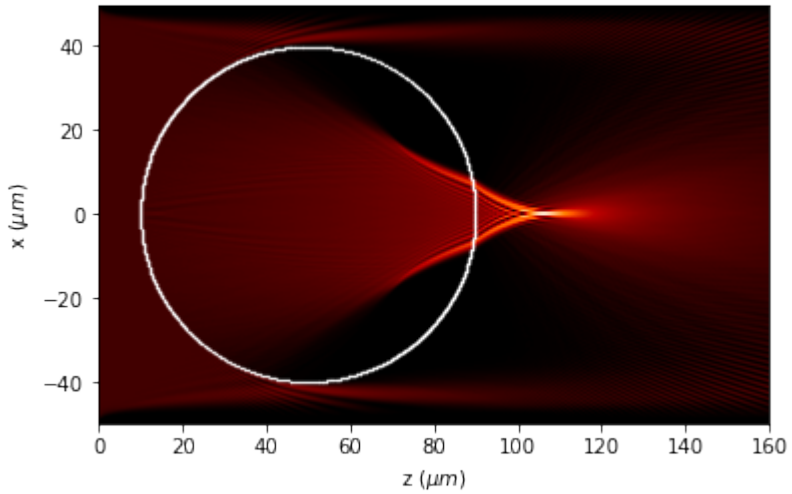
[45]:
```
radius_sphere = 40 * um
x0 = np.linspace(-1.25 * radius_sphere * um, 1.25 * radius_sphere * um, 2048)
z0 = np.linspace(0 * um, 4 * radius_sphere, 2048)
wavelength = .85 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

u1.sphere(r0=(0, 1.25 * radius_sphere),
          radius=(radius_sphere, radius_sphere),
          refractive_index=1.5)

edges = u1.surface_detection()

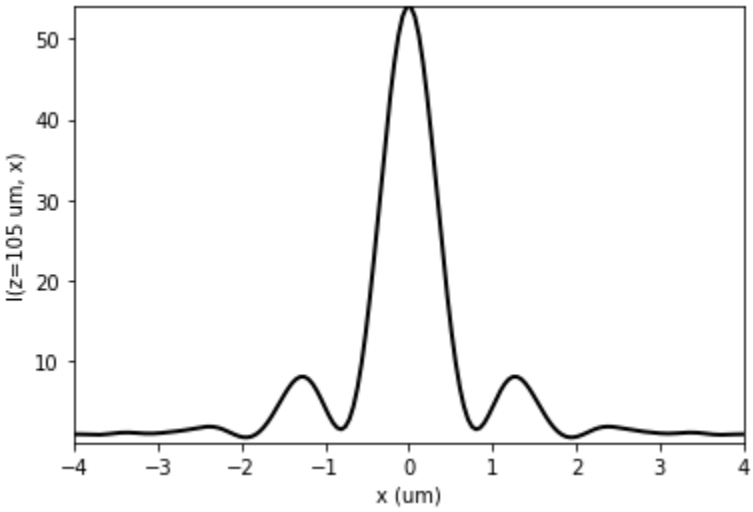
u1.WPM(verbose=False)

u1.draw(logarithm=True, normalize=False, draw_borders=True, scale='scaled')
```

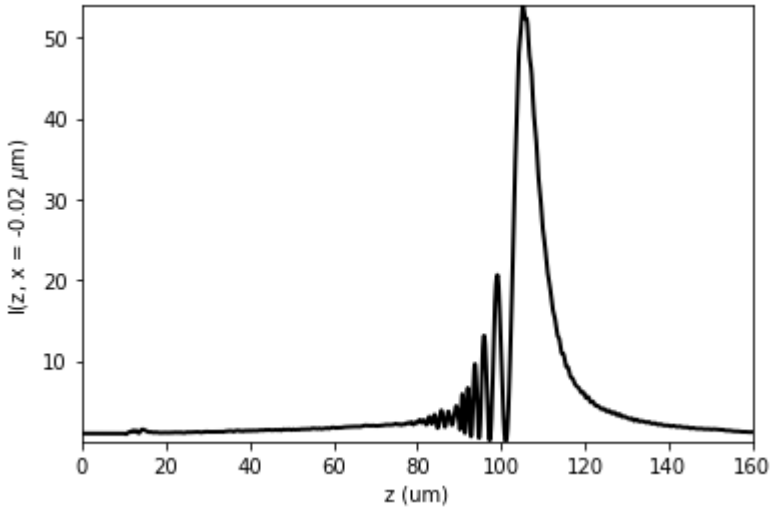


[46]:
```
x_max, z_max = u1.search_focus()
u1.profile_transversal(z0=z_max)
plt.xlim(-4, 4)
```

```
x = -0.024 um, z = 105.286 um
```



[47]:
```
u1.profile_longitudinal(x0=x_max)
```



6.4.11 Examples of multiprocessing

6.4.11.1 Rayleigh-Sommerfeld: polychromatic light

[48]:
```
def _func_polychromatic_RS_(wavelength):
    x0 = np.linspace(-100 * um, 100 * um, 512)

    f1 = Scalar_source_X(x0, wavelength)
    f1.plane_wave(A=1)
    t1 = Scalar_mask_X(x0, wavelength)
    t1.ronchi_grating(x0=0 * um, period=10 * um, fill_factor=0.5)

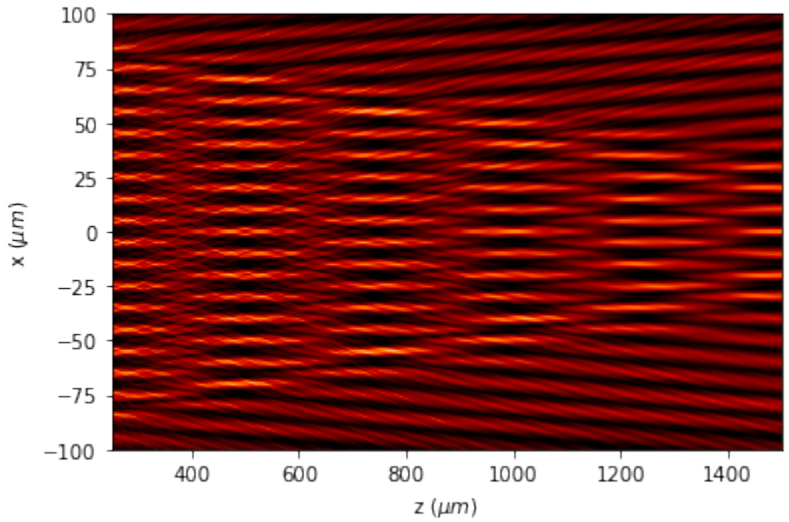
    return f1 * t1
```

[49]:
```
wavelengths = np.linspace(0.4, 0.8, 21)
w_central = 0.6
Dw = 0.07 * um
spectrum = np.exp(-(wavelengths - w_central)**2 / (2 * Dw**2))
initial_field = _func_polychromatic_RS_(wavelengths[0])
x0 = initial_field.x

z0 = np.linspace(250 * um, 1500 * um, 512)
```

[50]:
```
%%time
u1 = Scalar_mask_XZ(x0, z0, wavelengths[0], n_background=1)
initial_field = _func_polychromatic_RS_(wavelengths[0])
u1.incident_field(initial_field)
u1.RS()
u1.draw(logarithm=True, normalize='intensity');
```

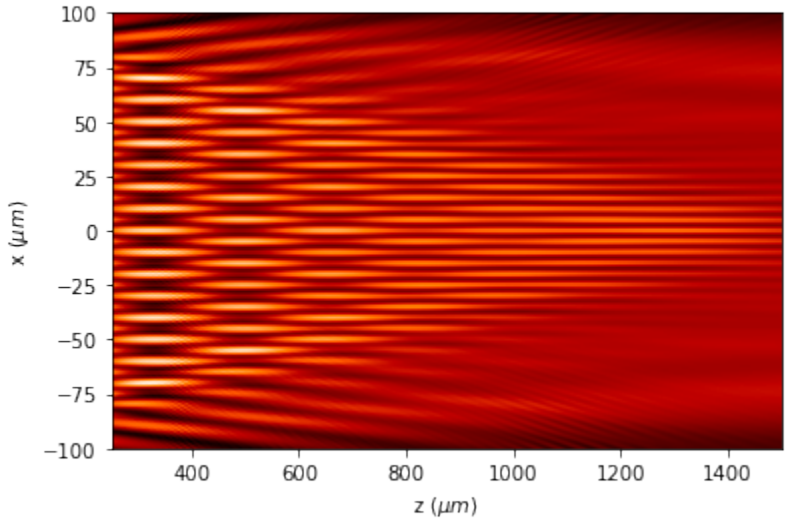
```
CPU times: user 602 ms, sys: 1 s, total: 1.6 s
Wall time: 2.17 s
```



[51]:
```
%%time
u1 = Scalar_mask_XZ(x0, z0, wavelengths[0], n_background=1)
u_poly = u1.RS_polychromatic(
    _func_polychromatic_RS_,
    wavelengths,
    spectrum=spectrum,
    verbose=False,
    num_processors=num_max_processors)

u_poly.draw(logarithm=True, normalize='intensity', draw_borders=True);
```

```
CPU times: user 8.78 s, sys: 15.7 s, total: 24.5 s
Wall time: 33.3 s
```



6.4.11.2 Rayleigh-Sommerfeld: polychromatic light without multiprocessing

When using polychromatic light in xz module, we can process each wavelength, and then sum the intensities distribution for each wavelength. We can see two examples, valid for RS and BPM techniques:

[52]:
```
def _func_polychromatic_BPM_(wavelength):
    x0 = np.linspace(-200 * um, 200 * um, 2048)
    z = np.linspace(0 * um, 800 * um, 2048)

    f1 = Scalar_source_X(x0, wavelength)
    f1.gauss_beam(A=1, x0=0, z0=0 * um, w0=600 * um, theta=0 * degrees)
    f1.plane_wave(A=1)
    t1 = Scalar_mask_X(x0, wavelength)
    t1.ronchi_grating(x0=0 * um, period=10 * um, fill_factor=0.5)

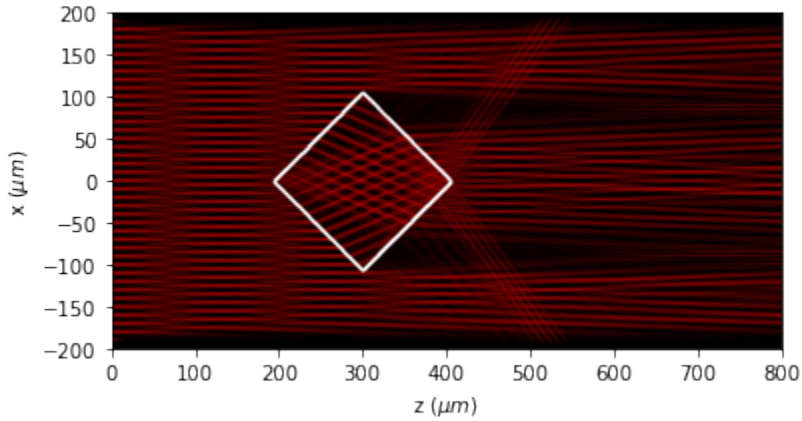
    u1 = Scalar_mask_XZ(x0, z, wavelength, n_background=1)
    u1.incident_field(f1 * t1)

    u1.rectangle(r0=(0, 300 * um),
                 size=(150 * um, 150 * um),
                 angle=45 * degrees,
                 refractive_index=1.5)

    return u1
```

[53]:
```
%%time
u1 = _func_polychromatic_BPM_(.75 * um)
u1.BPM(has_edges=True)
u1.draw(logarithm=True, normalize='intensity', draw_borders=True, scale='scaled');
```

```
CPU times: user 1.35 s, sys: 112 ms, total: 1.46 s
Wall time: 1.41 s
```



6.4.11.3 BPM: polychromatic light

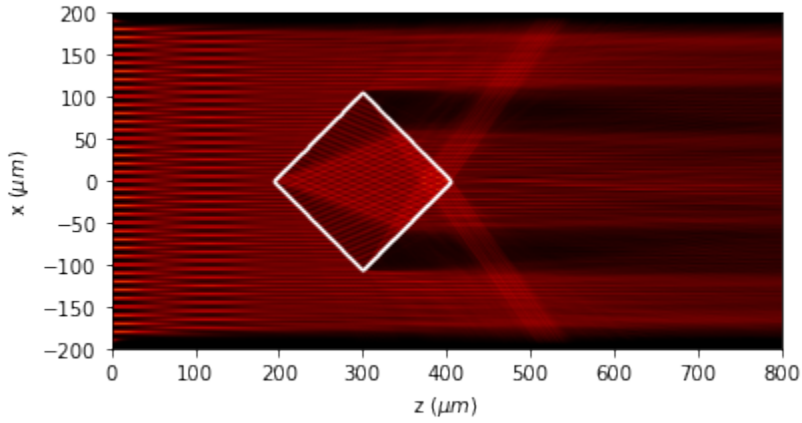
[54]:
```
wavelengths = np.linspace(.4, 1.2, 21)
spectrum = np.ones_like(wavelengths) / len(wavelengths)

initial_field = _func_polychromatic_BPM_(wavelengths[0])
z0 = initial_field.z
x0 = initial_field.x
```

[55]:
```
%%time

u1 = Scalar_mask_XZ(x0, z0, wavelengths[0], n_background=1)
u_poly = u1.BPM_polychromatic(
    _func_polychromatic_BPM_,
    wavelengths,
    spectrum,
    verbose=True,
    num_processors=num_max_processors)
u_poly.draw(
    logarithm=True, normalize='intensity', draw_borders=True, scale='scaled');
```

```
CPU times: user 20.7 s, sys: 11.7 s, total: 32.4 s
Wall time: 31.8 s
```



6.4.12 Example of masks

6.4.12.1 Creating an instance

[1]:
```
from diffractio import np, plt, sp
from diffractio import degrees, mm, um
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X

from numpy import loadtxt
```

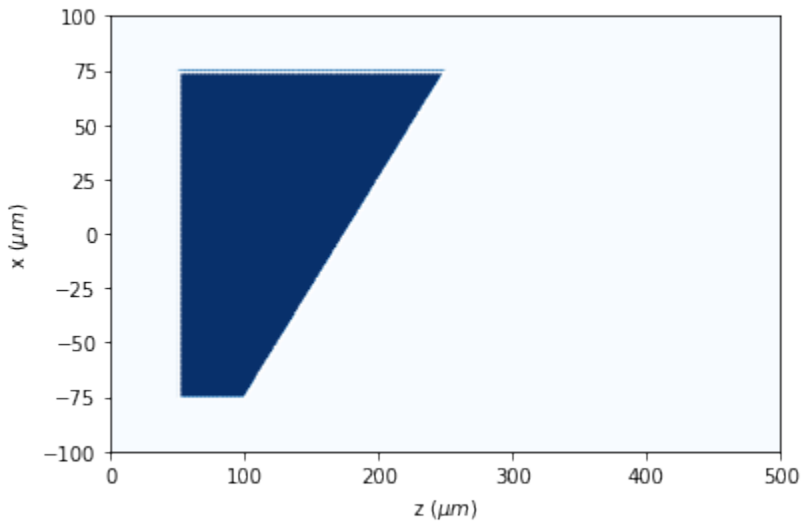
6.4.12.2 mask from two functions

[2]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 500 * um, 512)
wavelength = 0.6238 * um

f1 = '50 * um'
f2 = "175*um+np.tan(45*degrees)*(self.X-0*um)"
z_sides = (-75 * um, 75 * um)
v_globals = {'um': 1, 'np': np}

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.mask_from_function(r0=(0, 0),
                      refractive_index=1.5,
                      f1=f1,
                      f2=f2,
                      z_sides=z_sides,
                      angle=0 * degrees,
                      v_globals=v_globals)

t1.draw_refractive_index()
```



6.4.12.3 mask from surfaces defined in file

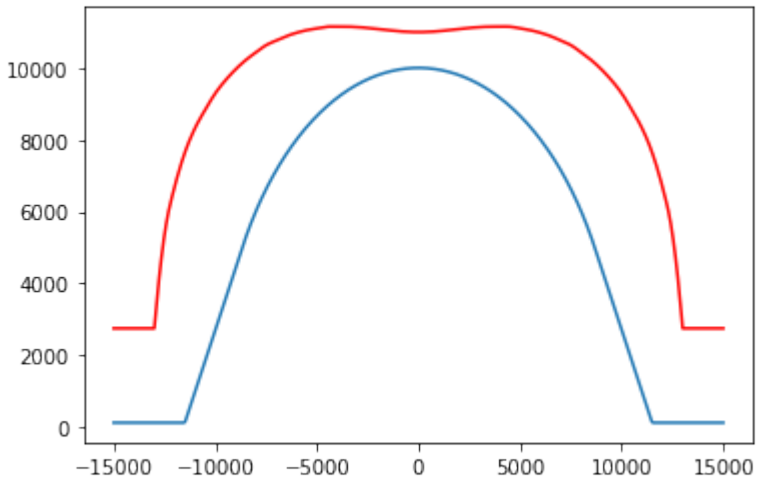
[3]:
```
x = np.linspace(-15 * mm, 15 * mm, 512)
z = np.linspace(-15 * mm, 15 * mm, 512)
wavelength = 0.6328 * um

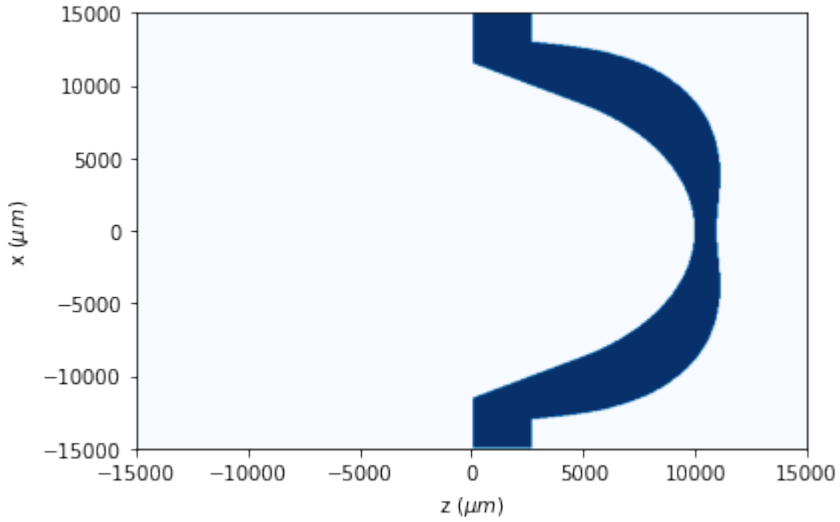
t1 = Scalar_mask_XZ(x, z, wavelength)

profile1 = loadtxt('profile1.txt')
profile2 = loadtxt('profile2.txt')

profile1[:, 1] = np.abs(profile1[:, 1])
profile2[:, 1] = np.abs(profile2[:, 1])
t1.mask_from_array(
    r0=(0 * um, 0 * um),
    refractive_index=2,
    array1=profile2 * 1000,  # propasar a micras
    array2=profile1 * 1000,  # pasar a micras
    x_sides=(-15 * mm, 15 * mm),
    angle=0 * degrees,
    v_globals={},
    interp_kind='quadratic',
    has_draw=True)

t1.draw_refractive_index(draw_borders=False)
```





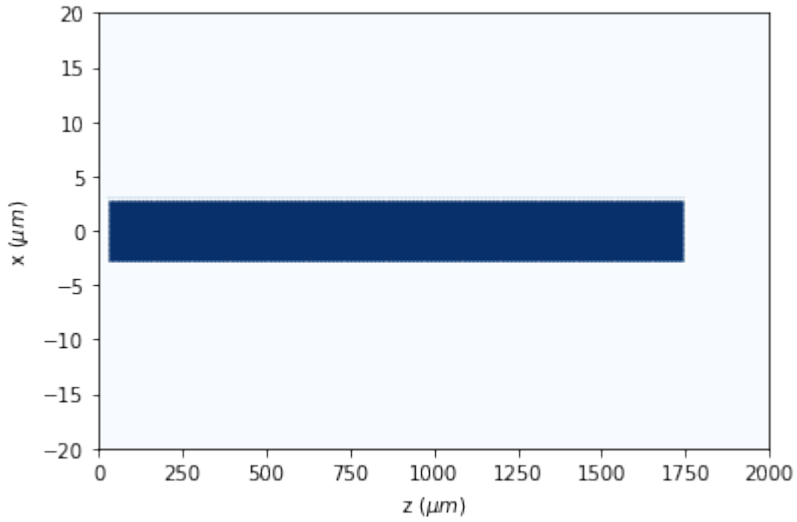
6.4.12.4 mask from several surfaces

[4]:
```
x0 = np.linspace(-20 * um, 20 * um, 512)
z0 = np.linspace(0 * um, 2000 * um, 512)
wavelength = 2 * um

r0 = (0, 0)
refractive_index = 4
Fs = ['Xrot<3*um', 'Xrot>-3*um', 'Zrot>25*um', 'Zrot<1750*um']
t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)

t1.object_by_surfaces(r0,
                      refractive_index,
                      Fs,
                      angle=0 * degrees,
                      v_globals={})

t1.draw_refractive_index(draw_borders=True)
```



6.4.12.5 Refraction index from functions

[5]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)

wavelength = 0.5 * um

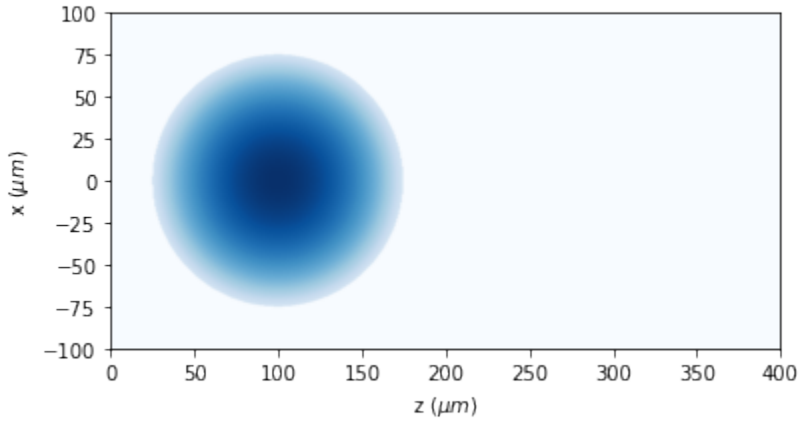
t0 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.0)

pn = dict(n_out=1.5, n_center=4, cx=0 * um, cz=100 * um, radius=75 * um)

center = (pn['cx'], pn['cz'])
radius = pn['radius']
# ref_index = '2*(((X-0)**2+(Z-300)**2)/75**2-0)'
ref_index = "{p[n_out]}+({p[n_center]}-{p[n_out]})*(1-((X-{p[cx]})**2+(Z-{p[cz]})**2)/
↪{p[radius]}**2)".format(
    p=pn)

t0.sphere(r0=center,
          radius=(radius, radius),
          refractive_index=ref_index,
          angle=0)

t0.draw_refractive_index(draw_borders=False, scale='scaled')
```



6.4.12.6 xz mask from a x mask

[6]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = .55 * um

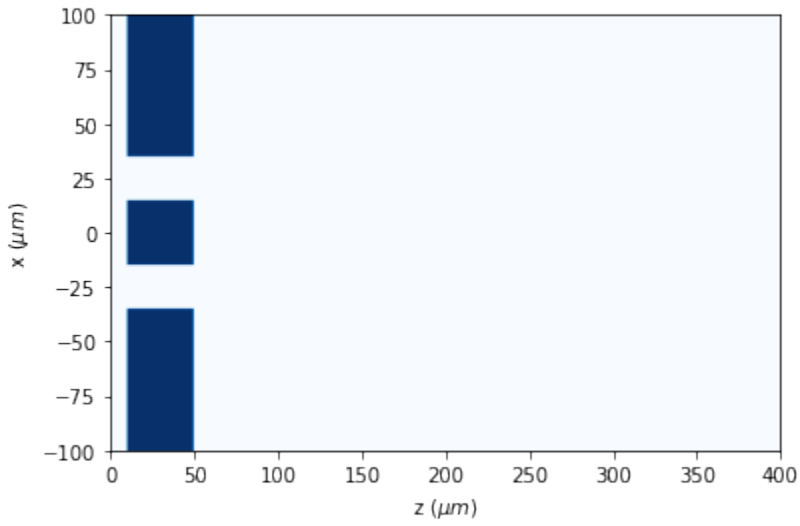
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.double_slit(x0=0, size=20 * um, separation=50 * um)

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
z0 = 10 * um
z1 = 50 * um

v_globals = dict(z0=z0, z1=z1)

t1.extrude_mask(t=t0, z0=z0, z1=z1, refractive_index=1.5, v_globals=v_globals)

t1.draw_refractive_index(draw_borders=False)
```



6.4.12.7 xz mask from a xy mask

There are many masks defined in the XY frame. We can use them using the 'refractive_index_from_scalar_mask_XY' method.

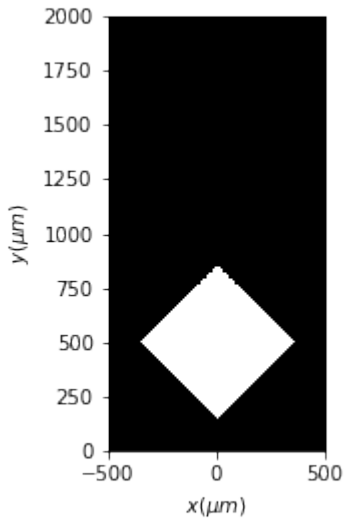
[7]:
```
from diffractio.scalar_masks_XY import Scalar_mask_XY

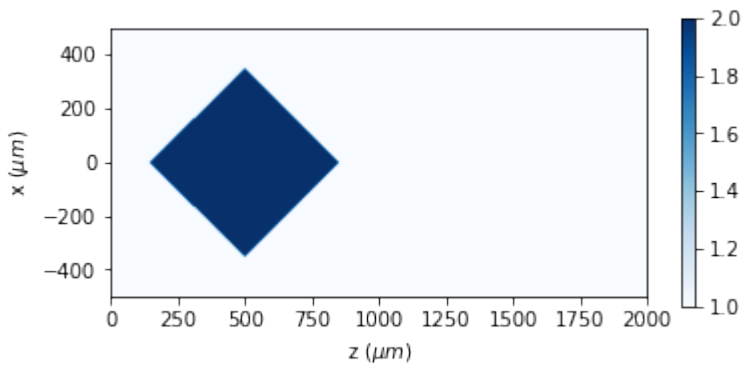
x = np.linspace(-500*um, 500*um, 256)
y = z = np.linspace(0*um, 2*mm, 512)

wavelength = 0.6328*um

mask_xy = Scalar_mask_XY(x,y, wavelength)
mask_xy.square(r0=(0, 500*um), size=500*um, angle=45*degrees)
mask_xy.draw()

mask_xz = Scalar_mask_XZ(x,z, wavelength=wavelength, n_background=1)
mask_xz.refractive_index_from_scalar_mask_XY(mask_xy,2+0.02j)
mask_xz.draw_refractive_index(draw_borders=False, scale='scaled', colorbar_kind='vertical
↪')
```





6.4.12.8 Variable refraction index

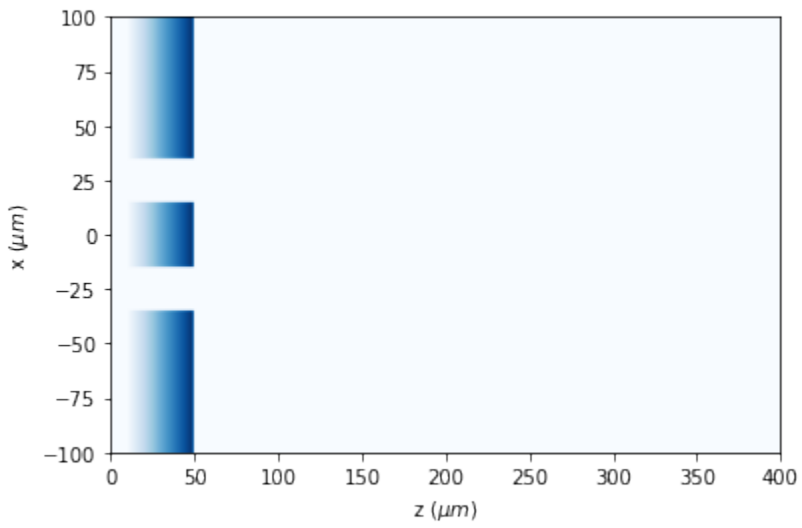
z direction

[8]:
```
# In this example the variation is in z

x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = .55 * um

t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.double_slit(x0=0, size=20 * um, separation=50 * um)

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
z0 = 10 * um
z1 = 50 * um
v_globals = dict(z0=z0, z1=z1)
t1.extrude_mask(t=t0,
                z0=z0,
                z1=z1,
                refractive_index='1+0.25*(z-z0)/(z1-z0)',
                v_globals=v_globals)
t1.draw_refractive_index(draw_borders=False, )
```

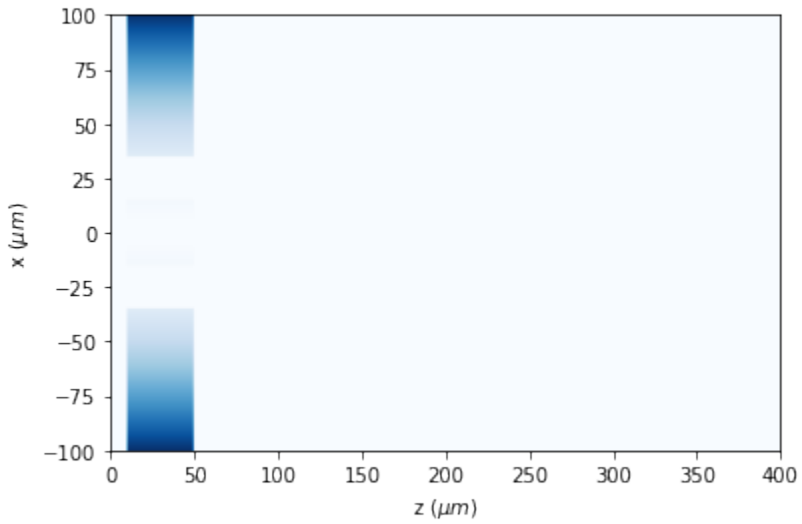


x direction

[9]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = .5 * um

t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.double_slit(x0=0, size=20 * um, separation=50 * um)

z_min = 10 * um
z_max = 50 * um
v_globals = dict(np=np)
t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
t1.extrude_mask(t=t0,
                z0=z_min,
                z1=z_max,
                refractive_index='1+0.25*np.abs(x/200)**2',
                v_globals=v_globals)
t1.draw_refractive_index(draw_borders=False)
```



discretize_refractive_index

[10]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
print("Dx={}".format(x0[1] - x0[0]))
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = 50 * um

t0 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.0)

pn = dict(n_out=1.5, n_center=4, cx=0 * um, cz=100 * um, radius=75 * um)

center = (pn['cx'], pn['cz'])
```

(continues on next page)

(continued from previous page)

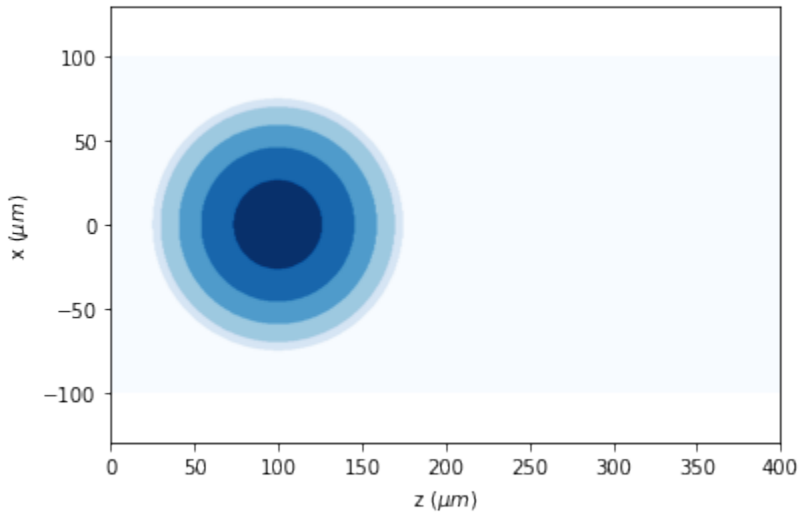
```
radius = pn['radius']
# ref_index = '2*(((X-0)**2+(Z-300)**2)/75**2-0)'
ref_index = "{p[n_out]}+({p[n_center]}-{p[n_out]})*(1-((X-{p[cx]})**2+(Z-{p[cz]})**2)/
↪{p[radius]}**2)".format(
    p=pn)

t0.sphere(r0=center,
          radius=(radius, radius),
          refractive_index=ref_index,
          angle=0)

t0.discretize_refractive_index(num_layers=5)

t0.draw_refractive_index(draw_borders=False, scale='equal')
```

```
Dx=0.391389432485326
```



6.4.12.9 add_masks

Complex structures can be performed by adding different masks.

[11]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = 2 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)

t1.sphere(r0=(0, 100 * um),
          radius=(80 * um, 80 * um),
          refractive_index=2.5,
          angle=0)

t1.sphere(r0=(0, 100 * um),
```

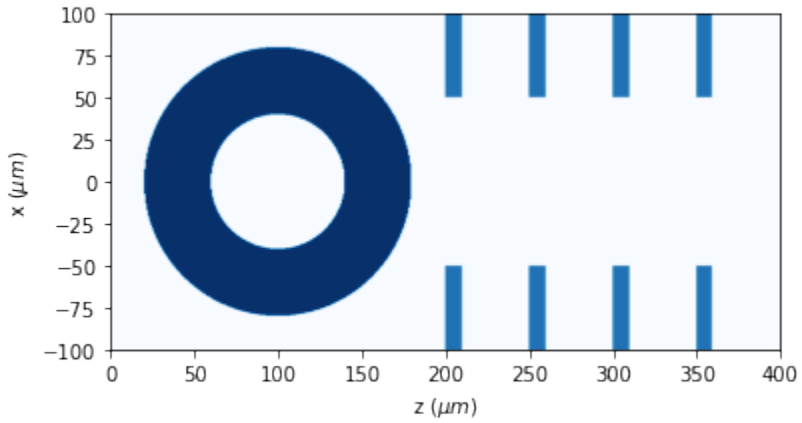
(continues on next page)

(continued from previous page)

```
          radius=(40 * um, 40 * um),
          refractive_index=1,
          angle=0)

for pos_slit in [200, 250, 300, 350]:
    t1.slit(r0=(0 * um, pos_slit * um),
            aperture=100 * um,
            depth=10 * um,
            refractive_index=1.5 - 1.5j,
            refractive_index_center='',
            angle=0 * degrees)

t1.draw_refractive_index(draw_borders=False, scale='scaled')
```

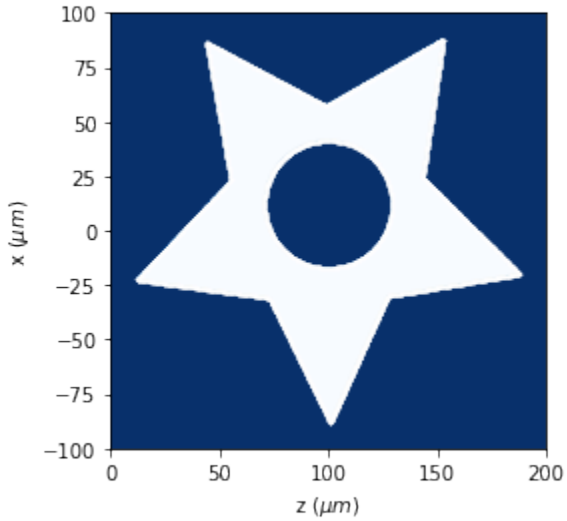


6.4.12.10 mask from an image

[12]:
```
x0 = np.linspace(-100 * um, 100 * um, 1024)
z0 = np.linspace(0 * um, 200 * um, 1024)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

filename = "star_hole.png"
t1.image(filename=filename, n_max=2, n_min=1, angle=0 * degrees, invert=False)
t1.draw_refractive_index(scale='scaled')
```

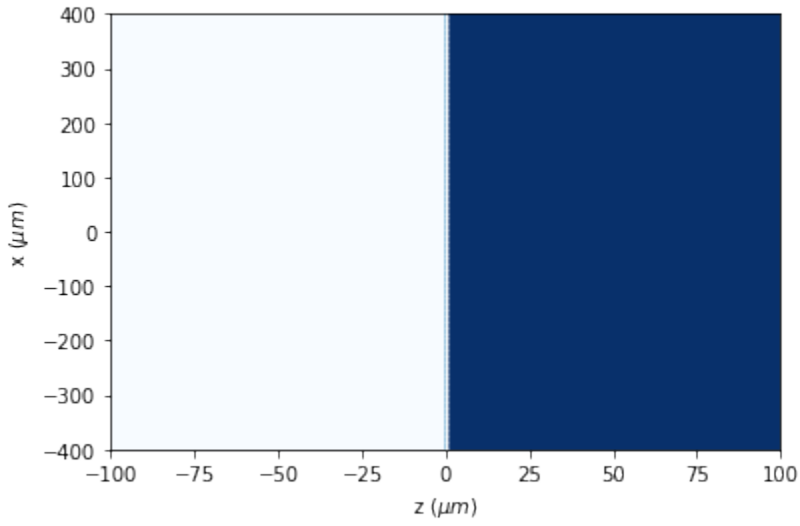


6.4.13 Definite masks

6.4.13.1 semi plane

[13]:
```
x0 = np.linspace(-400 * um, 400 * um, 512)
z0 = np.linspace(-100 * um, 100 * um, 512)
wavelength = .5 * um

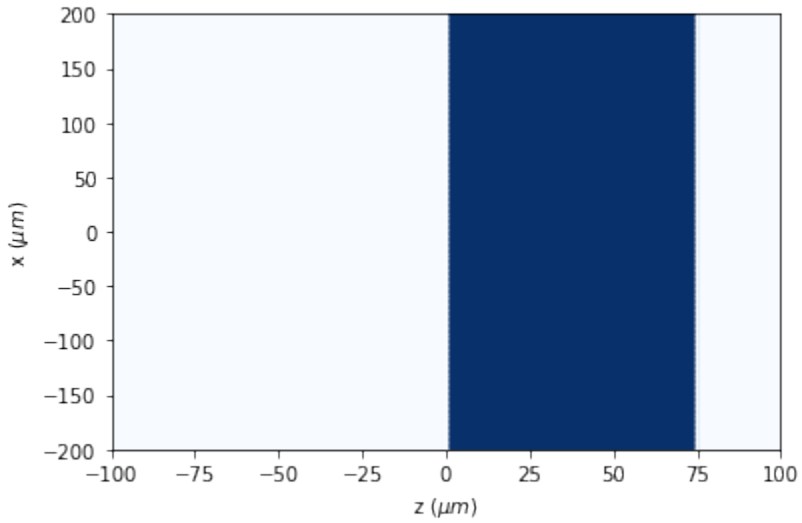
t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.semi_plane(r0=(0, 0),
              refractive_index=2,
              angle=0 * degrees,
              rotation_point=None)
t1.draw_refractive_index()
```



6.4.13.2 layer

[14]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(-100 * um, 100 * um, 512)
wavelength = .5 * um

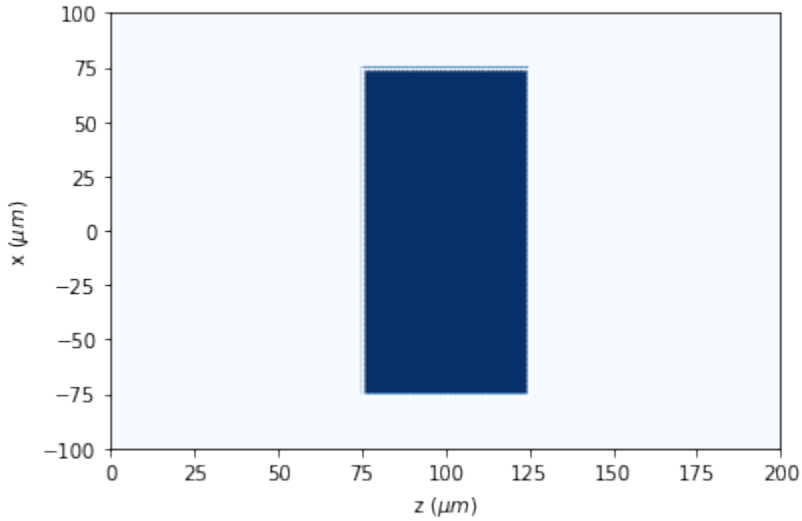
t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.layer(r0=(50, 0),
         depth=75 * um,
         refractive_index=2,
         angle=0 * degrees,
         rotation_point=None)
t1.draw_refractive_index()
```



6.4.13.3 rectangle

[15]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 200 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.rectangle(r0=(0 * um, 100 * um),
             size=(150 * um, 50 * um),
             angle=0 * degrees,
             refractive_index=1.5)
t1.draw_refractive_index()
```



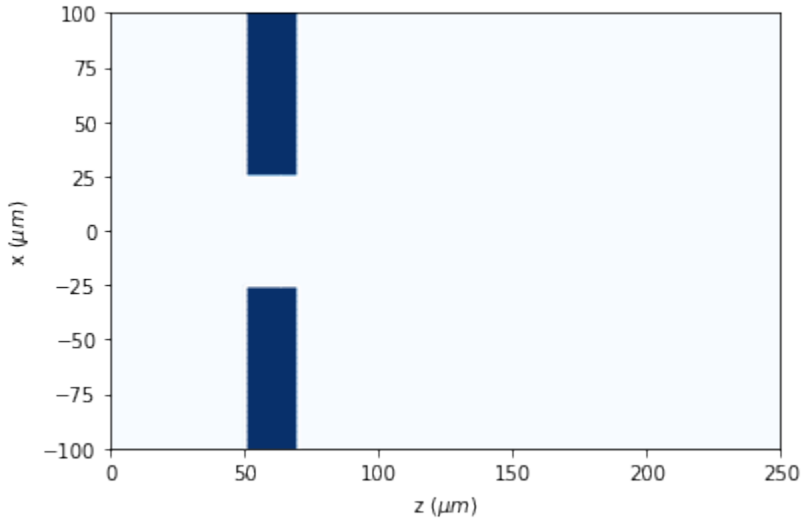
6.4.13.4 slit

[16]:
```
x0 = np.linspace(-100 * um, 100 * um, 512 * 4)
z0 = np.linspace(0 * um, 250 * um, 512 * 4)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

t1.slit(r0=(0 * um, 50 * um),
        aperture=50 * um,
        depth=20 * um,
        refractive_index=1.5 + 1j,
        refractive_index_center='',
        angle=0 * degrees)

t1.draw_refractive_index()
```



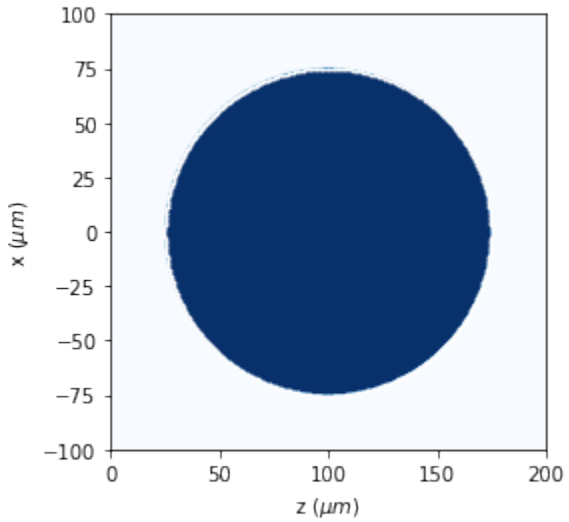
6.4.13.5 sphere or ellipsoid

[17]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 200 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

t1.sphere(r0=(0, 100 * um),
          radius=(75 * um, 75 * um),
          refractive_index=1.5,
          angle=0 * degrees)

t1.draw_refractive_index(scale='scaled')
```

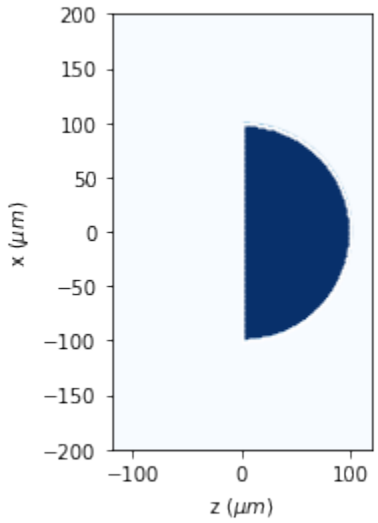


6.4.13.6 semi sphere

[18]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(-120 * um, 120 * um, 512)
wavelength = .5 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.semi_sphere(r0=(0, 0),
               radius=(100, 100),
               refractive_index=2,
               angle=0 * degrees)

t1.draw_refractive_index(draw_borders=True, min_incr=0.01, scale='scaled')
```

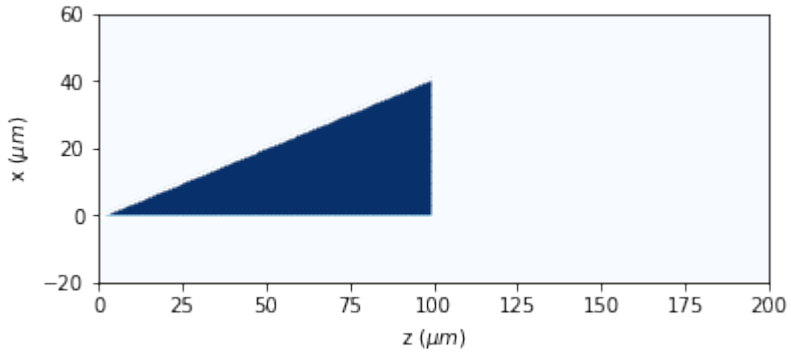


6.4.13.7 wedge

[19]:
```
x0 = np.linspace(-20 * um, 60 * um, 512)
z0 = np.linspace(0 * um, 200 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

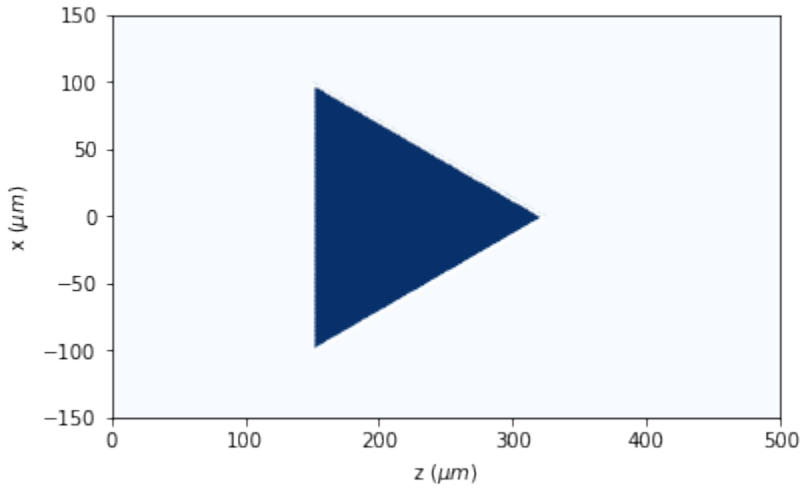
t1.wedge(r0=(0, 0),
         length=100 * um,
         refractive_index=1.5,
         angle_wedge=22.5 * degrees,
         angle=0 * degrees,
         rotation_point=None)
t1.draw_refractive_index(scale='scaled')
```



6.4.13.8 prism

[20]:
```
x0 = np.linspace(-150 * um, 150 * um, 512)
z0 = np.linspace(0 * um, 500 * um, 4096)
wavelength = 2 * um

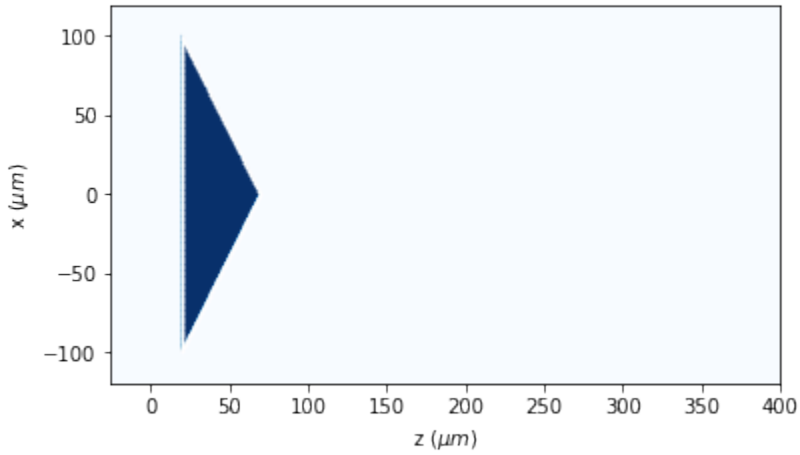
t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.prism(r0=(100 * um, 150 * um),
         length=200 * um,
         refractive_index=2,
         angle_prism=60 * degrees,
         angle=90 * degrees)
t1.draw_refractive_index(scale='scaled')
```



6.4.13.9 biprism

[21]:
```
x0 = np.linspace(-120 * um, 120 * um, 512)
z0 = np.linspace(-25 * um, 400 * um, 4096)
wavelength = .5 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.biprism(r0=(0, 20 * um),
           length=200 * um,
           height=50 * um,
           refractive_index=1.5,
           angle=0)
t1.draw_refractive_index(draw_borders=True, scale='scaled')
```



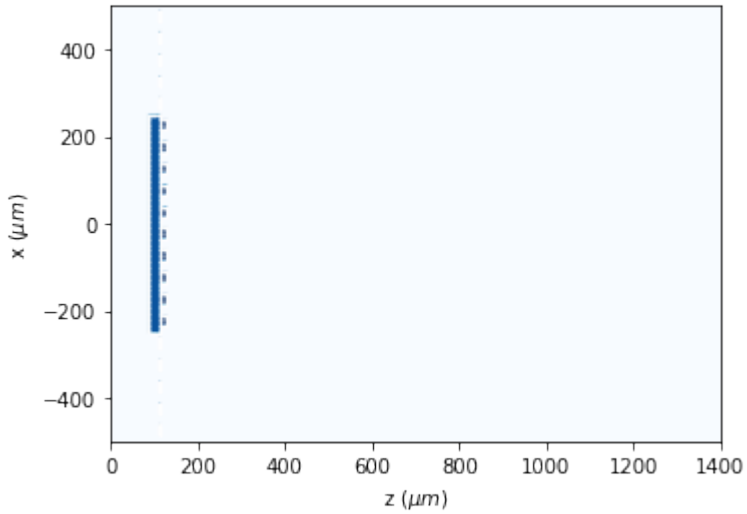
6.4.13.10 Ronchi grating

[22]:
```
x0 = np.linspace(-500 * um, 500 * um, 512)
z0 = np.linspace(0 * um, 1400 * um, 512)
wavelength = 0.5 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

t1.ronchi_grating(period=50 * um,
                  fill_factor=.5,
                  length=500 * um,
                  height=20 * um,
                  r0=(0 * um, 100 * um),
                  Dx=2 * um,
                  refractive_index=1.5 + 0.5j,
                  heigth_substrate=25 * um,
                  refractive_index_substrate=1.5,
                  angle=0 * degrees)

t1.draw_refractive_index(scale='scaled')
```



6.4.13.11 grating with substrate

[23]:
```
x0 = np.linspace(-500 * um, 500 * um, 512)
z0 = np.linspace(0 * um, 400 * um, 512)
wavelength = .55 * um

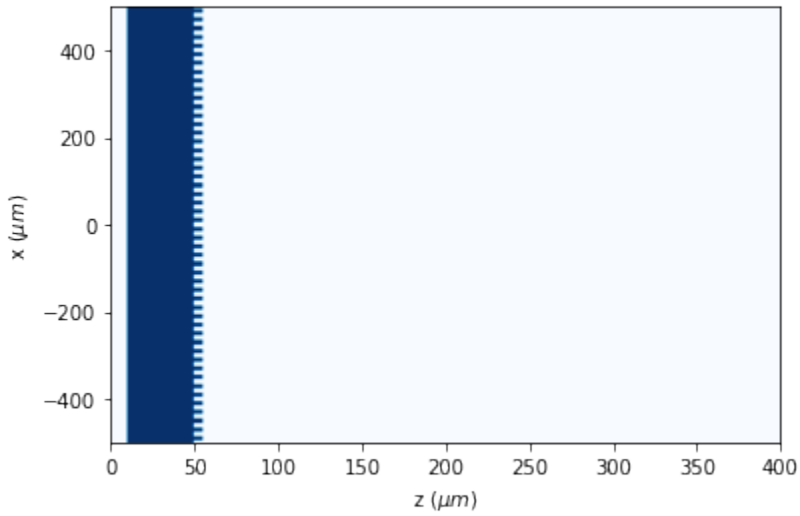
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.slit(x0=0, size=0 * um)

t1 = Scalar_mask_X(x=x0, wavelength=wavelength)
t1.ronchi_grating(period=20 * um, x0=0 * um, fill_factor=0.5)

t2 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)

t2.extrude_mask(t=t0, z0=10 * um, z1=50 * um, refractive_index=1.5)
t2.extrude_mask(t=t1, z0=50 * um, z1=55.5 * um, refractive_index=1.5)

t2.draw_refractive_index(draw_borders=False)
```



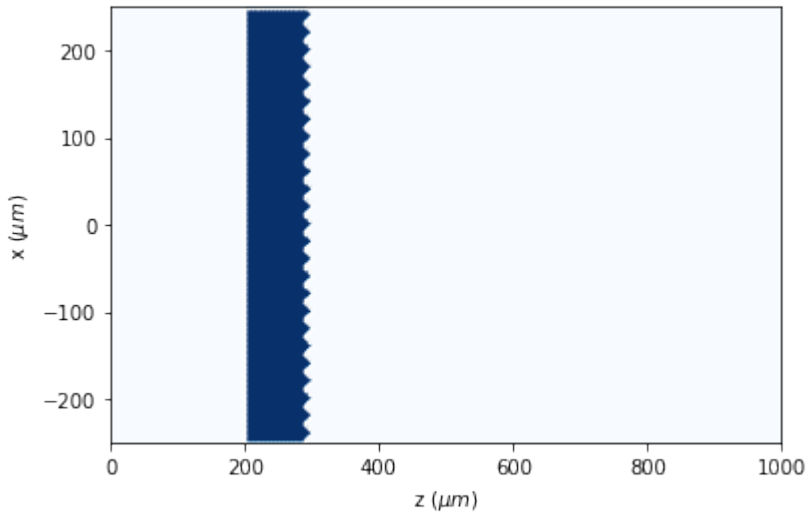
6.4.13.12 sine grating

[24]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
z0 = np.linspace(0 * um, 1000 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

t1.sine_grating(period=20 * um,
                heigth_sine=10 * um,
                heigth_substrate=100 * um,
                r0=(0 * um, 200 * um),
                length=500 * um,
                Dx=2 * um,
                refractive_index=1.5,
                angle=0 * degrees)

t1.draw_refractive_index()
```

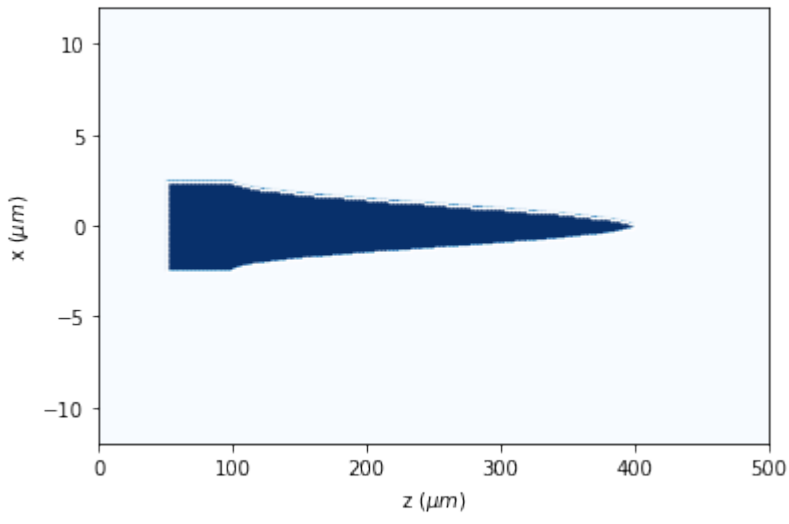


6.4.13.13 probe

[25]:
```
x0 = np.linspace(-12 * um, 12 * um, 512)
z0 = np.linspace(0 * um, 500 * um, 512)
wavelength = .6 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.probe(r0=(0, 50 * um),
         base=5 * um,
         length=300 * um,
         refractive_index=1.5,
         angle=0 * degrees)

t1.draw_refractive_index()
```



6.4.13.14 convergent plane lens

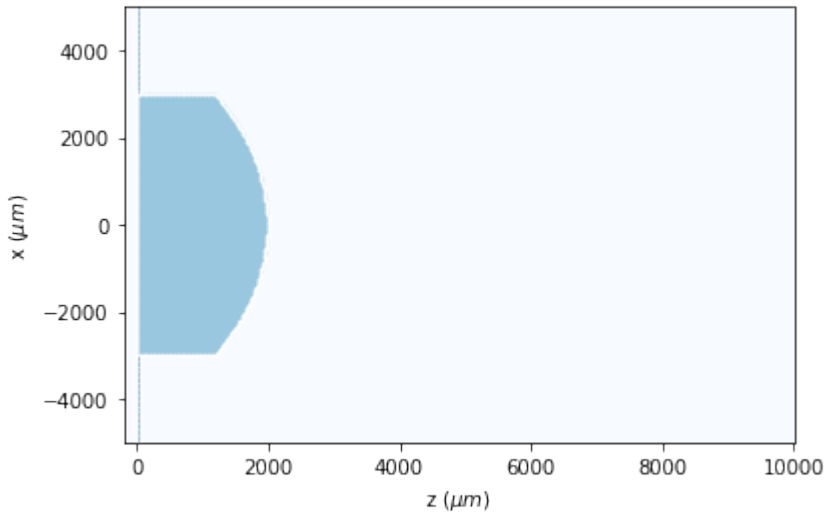
[26]:
```
x0 = np.linspace(-5 * mm, 5 * mm, 512)
z0 = np.linspace(-200 * um, 10 * mm, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

focal, _ = t1.lens_plane_convergent(r0=(0, 0),
                                    aperture=6 * mm,
                                    radius=6 * mm,
                                    thickness=2 * mm,
                                    refractive_index=1.5,
                                    angle=0 * degrees,
                                    mask=(50 * um, 1 + 2.05j))
print("focus distance f={} um".format(focal))

t1.draw_refractive_index()
```

```
focus distance f=12000.0 um
```



6.4.13.15 convergent lens

[27]:
```
x0 = np.linspace(-2000 * um, 2000 * um, 512)
z0 = np.linspace(-100 * um, 4 * mm, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

focal, _ = t1.lens_convergent(r0=(0, 0),
                              aperture=3 * mm,
                              radius=(8 * mm, -6 * mm),
                              thickness=500 * um,
                              refractive_index=2,
```

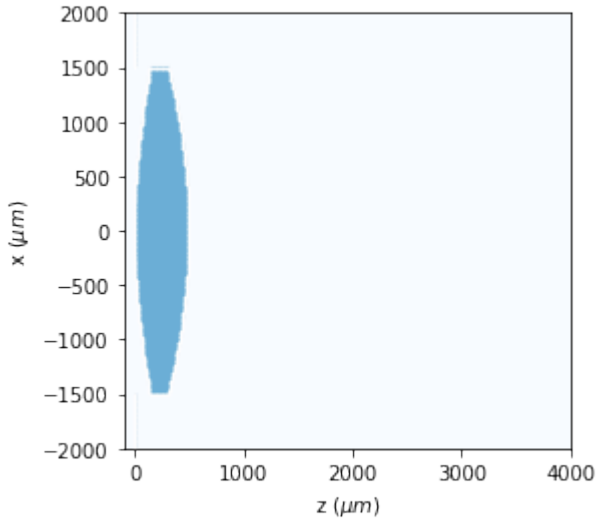
(continues on next page)

(continued from previous page)

```
                                  angle=0 * degrees,
                                  mask=(10 * um, 3 + 0.05j))
print("focus distance f={:2.2f} um".format(focal))

t1.draw_refractive_index(scale='scaled')
```

```
focus distance f=3368.42 um
```



6.4.13.16 divergent plane lens

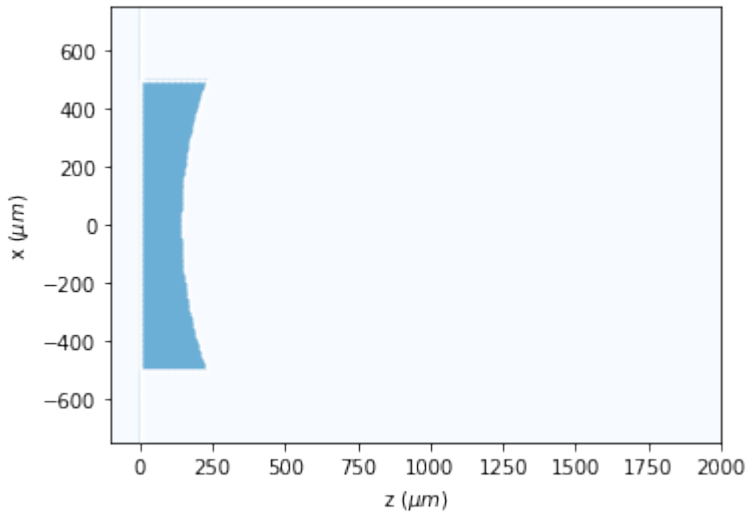
[28]:
```
x0 = np.linspace(-750 * um, 750 * um, 512)
z0 = np.linspace(-100 * um, 2000 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

focal, _ = t1.lens_plane_divergent(r0=(0, 0),
                                   aperture=1000 * um,
                                   radius=1500 * um,
                                   thickness=150 * um,
                                   refractive_index=2,
                                   angle=0 * degrees,
                                   mask=(10 * um, 3 + 0.05j))
print("focus distance f={} um".format(focal))

t1.draw_refractive_index(scale='scaled')
```

```
focus distance f=1500.0 um
```



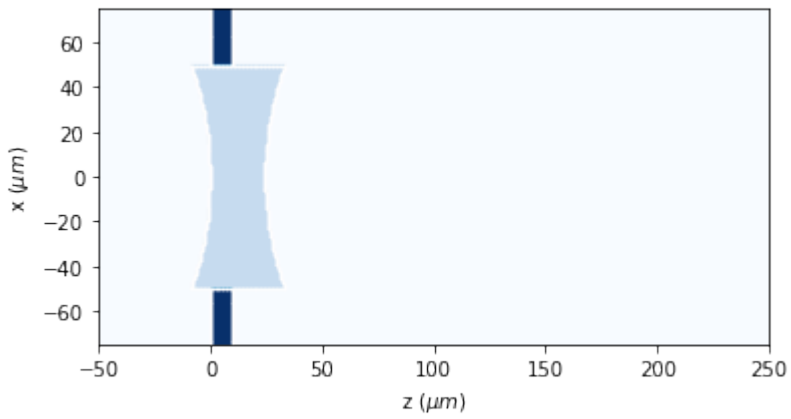
6.4.13.17 divergent lens

[29]:
```
x0 = np.linspace(-75 * um, 75 * um, 512)
z0 = np.linspace(-50 * um, 250 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)

focal, _ = t1.lens_divergent(r0=(0, 0),
                             aperture=100 * um,
                             radius=(-150 * um, 150 * um),
                             thickness=25 * um,
                             refractive_index=1.5,
                             angle=0 * degrees,
                             mask=(10 * um, 3 + 0.05j))
print("focus distance f={} um".format(focal))
t1.draw_refractive_index(scale='scaled')
```

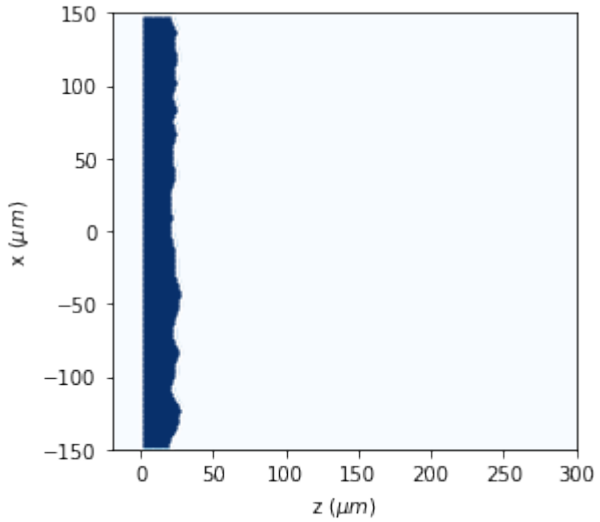
```
focus distance f=-154.28571428571428 um
```



6.4.13.18 rough surface

[30]:
```
x0 = np.linspace(-150 * um, 150 * um, 512)
z0 = np.linspace(-20 * um, 300 * um, 512)
wavelength = 0.6238 * um

t1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
t1.rough_sheet(r0=(0 * um, 0 * um),
               size=(300 * um, 25 * um),
               t=10 * um,
               s=10 * um,
               refractive_index=1.5,
               angle=0,
               rotation_point=None)
t1.draw_refractive_index(scale='scaled')
```



6.4.14 Drawing XZ: focal area of an apheric lens

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
```

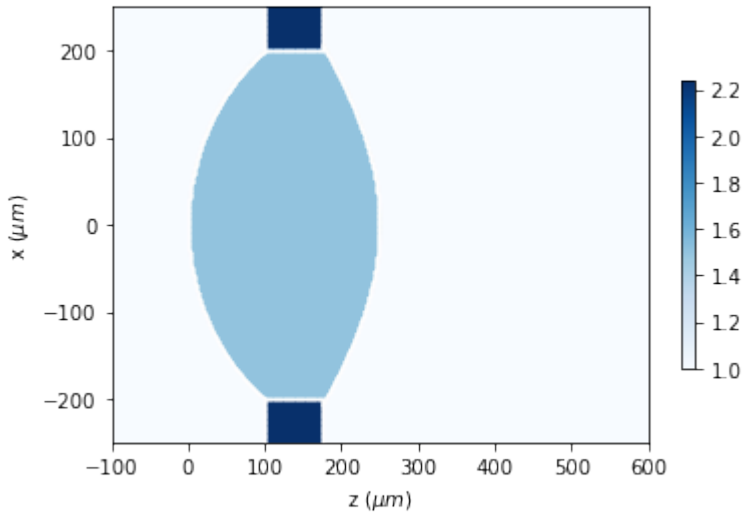
[2]:
```
x0 = np.linspace(-250 * um, 250 * um, 1024 * 4)
z0 = np.linspace(-0.1 * mm, .6 * mm, 1024 * 4)
wavelength = 10 * um
```

[3]:
```
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

[4]:
```
lens = Scalar_mask_XZ(x0, z0, wavelength, n_background=1, info='')
ipasa, conds = lens.aspheric_lens(r0=(0 * mm, 0 * mm),
                                  angle=(0 * degrees, (0 * mm, 0 * mm)),
                                  refractive_index=1.5,
                                  cx=(1 / (.25 * mm), -1 / (.25 * mm)),
                                  Qx=(0, 0),
                                  a2=(0, 1e-13),
                                  a3=(0, 0),
                                  a4=(0, 0),
                                  depth=.25 * mm,
                                  size=0.8 * mm)

lens.slit(r0=(0, 100 * um),
          aperture=400 * um,
          depth=75 * um,
          refractive_index=1 + 2j)
```

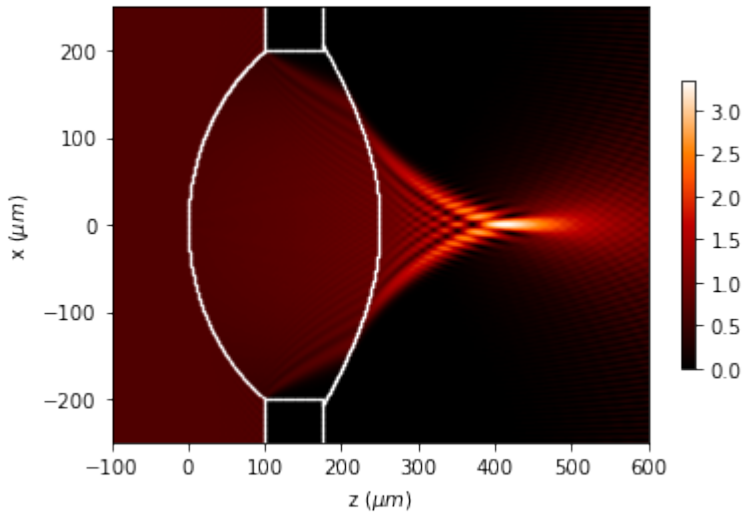
[5]:
```
lens.draw_refractive_index(draw_borders=True,
                           min_incr=0.01,
                           colorbar_kind='vertical')
```



[6]:
```
lens.incident_field(u0)
```

[7]:
```
lens.clear_field()
lens.WPM(verbose=False, has_edges=False)
```

[8]:
```
lens.draw(kind='intensity',
          logarithm=True,
          draw_borders=True,
          colorbar_kind='vertical')
```



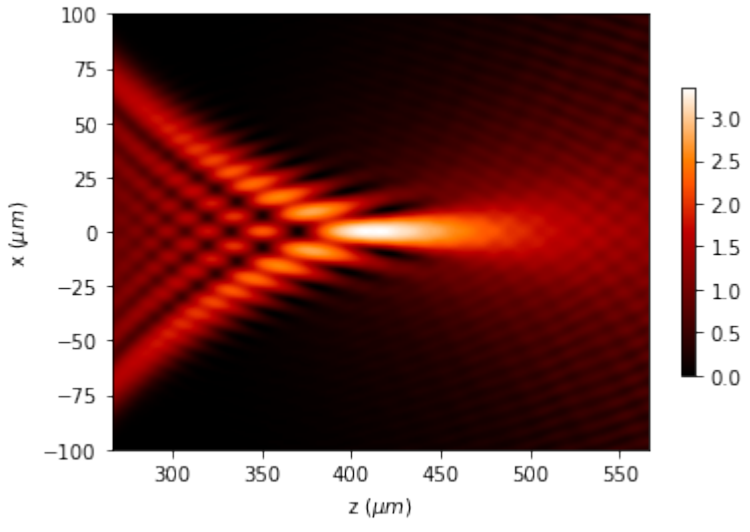
[9]:
```
Intensity_WPM = lens.intensity()
```

[10]:
```
x_f, z_f = lens.search_focus()
```

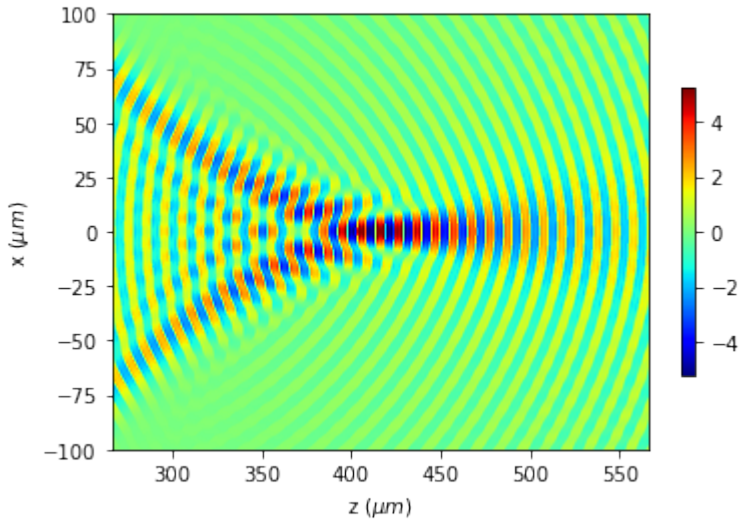
```
x = -0.061 um, z = 415.897 um
```

[11]:
```
ylim_max = 100 * um
zlim_max = 150 * um
```

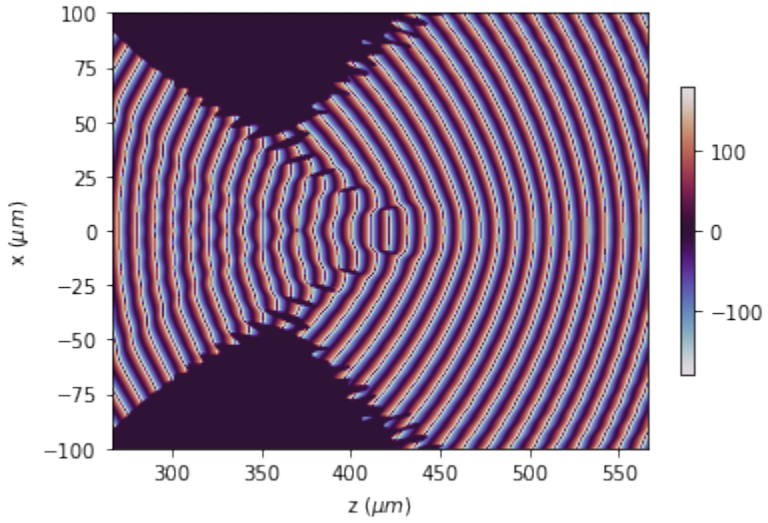
[12]:
```
lens.draw(kind='intensity', logarithm=True, colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f - zlim_max, z_f + zlim_max)
```



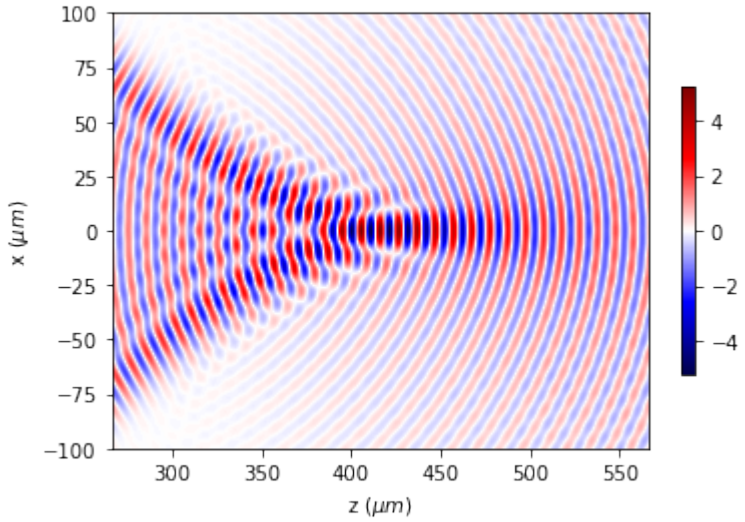
[13]:
```
lens.draw(kind='amplitude', colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f - zlim_max, z_f + zlim_max)
```



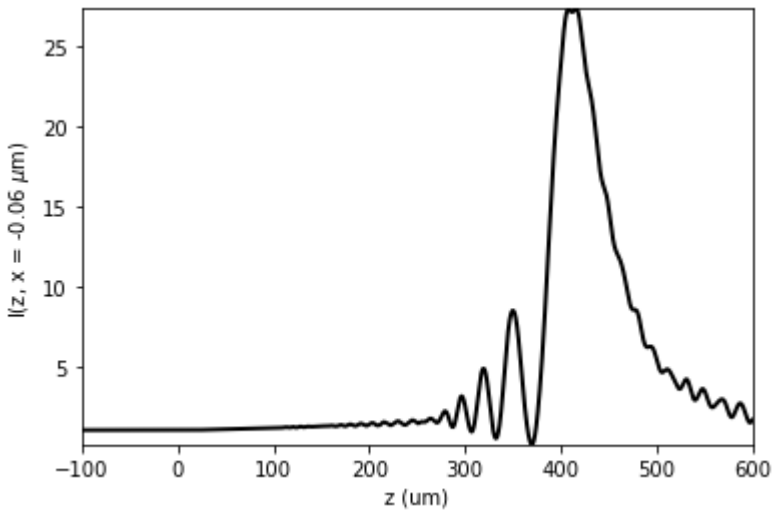
[14]:
```
lens.draw(kind='phase',
          draw_borders=True,
          colorbar_kind='vertical',
          percentage_intensity=0.005)
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f - zlim_max, z_f + zlim_max)
```



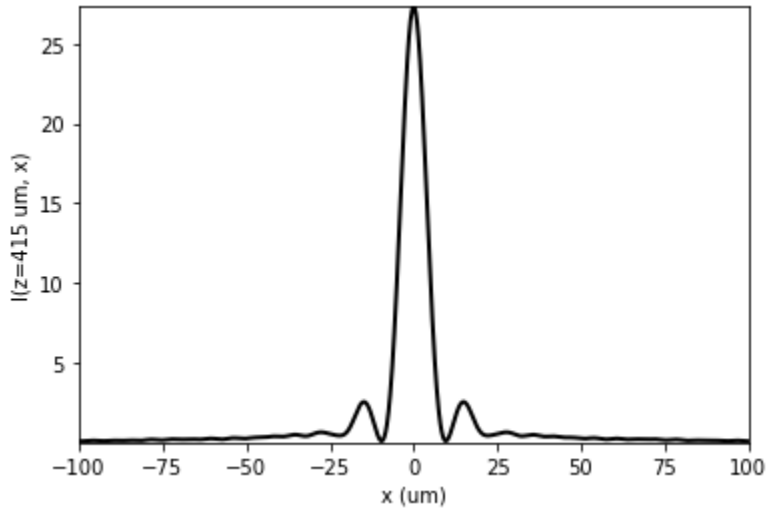
[15]:
```
lens.draw(kind='real', colorbar_kind='vertical')
plt.ylim(-ylim_max, ylim_max)
plt.xlim(z_f - zlim_max, z_f + zlim_max)
```



[16]:
```
lens.profile_longitudinal(kind='intensity', logarithm=False, x0=x_f)
```



[17]:
```
lens.profile_transversal(kind='intensity', logarithm=False, z0=z_f)
plt.xlim(-100, 100)
```



[18]:
```
filename = 'video_xz'

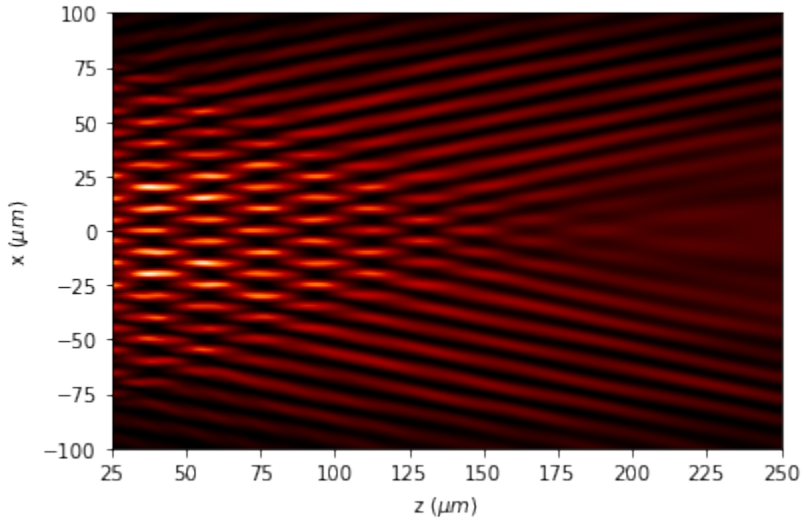
length = 200 * um
wavelength = 5 * um
period = 25 * um
z_talbot = 2 * period**2 / wavelength
x0 = np.linspace(-length / 2, length / 2, 256)
z0 = np.linspace(25 * um, 1 * z_talbot, 64)

u0 = Scalar_source_X(x0, wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=-100 * um, w0=100 * um, theta=0 * degrees)

t1 = Scalar_mask_X(x0, wavelength)
t1.ronchi_grating(x0=0 * um, period=10 * um, fill_factor=0.5)

u1 = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(t1 * u0)
u1.RS()

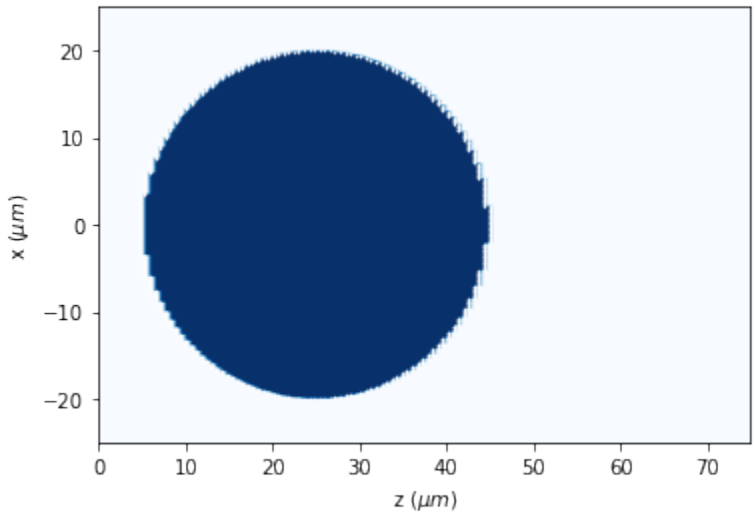
u1.draw(kind='intensity',
        logarithm=False,
        normalize='maximum',
        draw_borders=True,
        filename='')
```



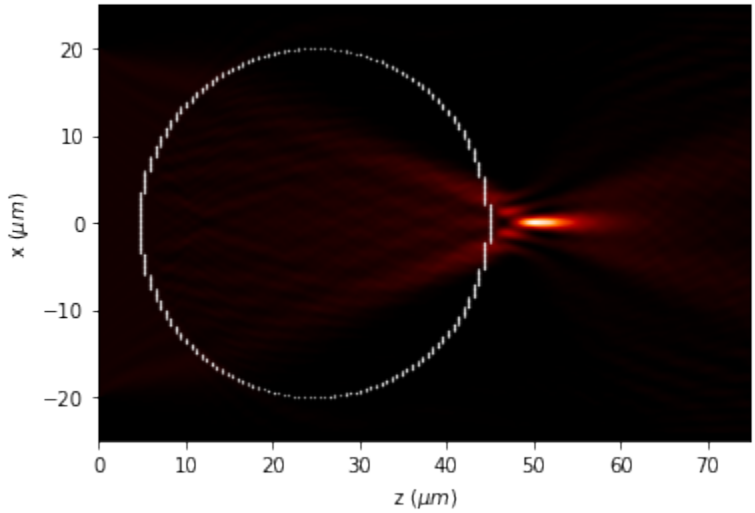
[19]:
```
seconds = 1.
u1.video(kind='intensity',
         logarithm=True,
         normalize=False,
         time_video=10 * seconds,
         frames_reduction=5,
         filename='creating_video.mp4',
         dpi=100)
```

6.4.14.1 Visualizing fields using interactive tools

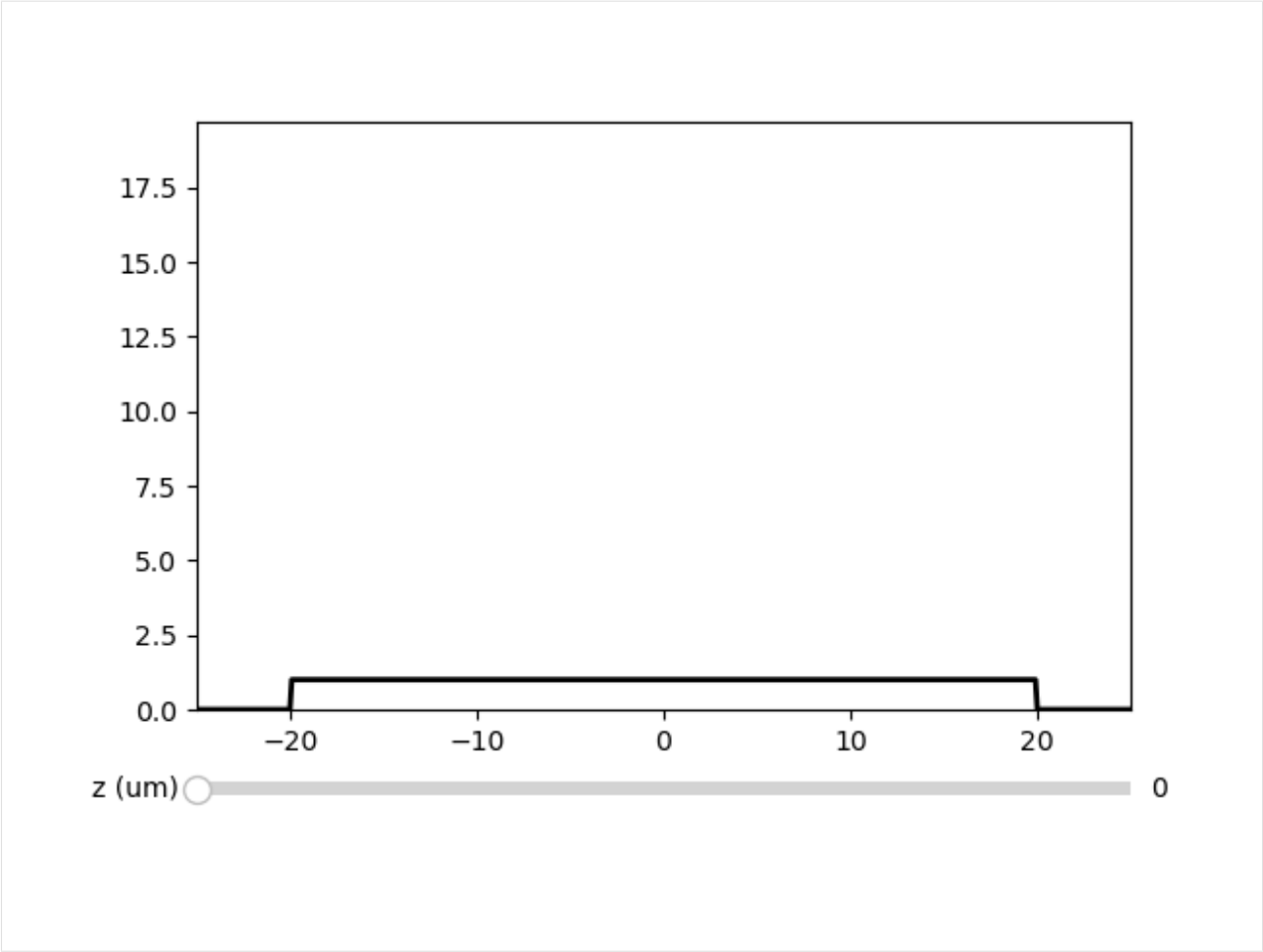
[20]:
```
x0 = np.linspace(-25 * um, 25 * um, 512)
z0 = np.linspace(0 * um, 75 * um, 128)
wavelength = 2 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.mask_field(size_edge=5 * um)
u1.sphere(r0=(0 * um, 25 * um),
          radius=(20 * um, 20 * um),
          refractive_index=1.5)
u1.draw_refractive_index(scale='scaled')
```

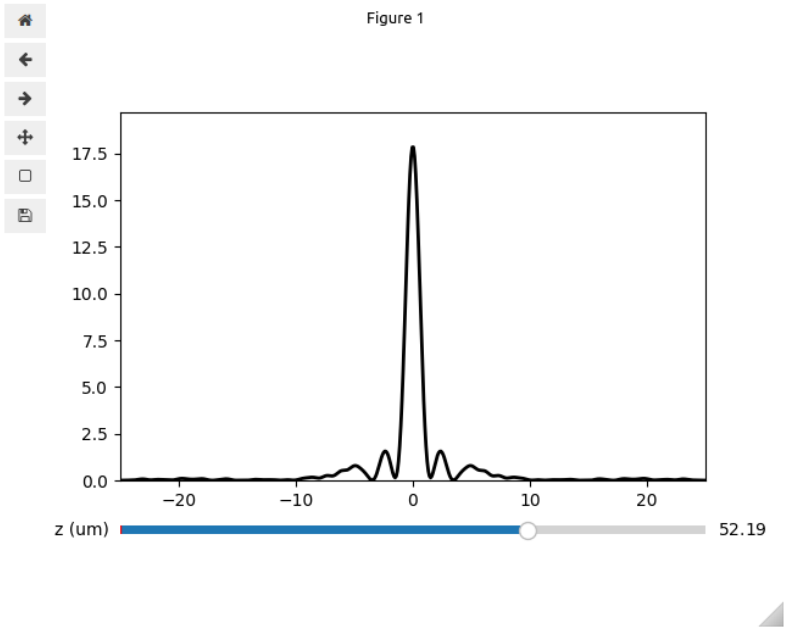


[21]:
```
u1.clear_field()
u1.WPM(verbose=False)
u1.draw(scale='scaled', draw_borders=True)
```



[22]:
```
%matplotlib widget
u1.draw_profiles_interactive(kind='intensity');
```





[23]:
```
%matplotlib inline
```

6.5 Scalar_XY

6.5.1 Characteristics

Scalar_XY is a set of three modules for:

- Generation of 2D (xy-axis) light source.
- Generation of 2D (xy-axis) masks and diffractive optical elements.
- Propagation of light, determination of parameters, and other functions.
- Drawing sources, masks and fields.

These modules are named: scalar_fields_XY.py, scalar_sources_XY.py, and scalar_masks_XY.py.

Each module present a main class:

- Scalar_field_XY
- Scalar_masks_XY
- Scalar_sources_XY

The main attributes for these classes are the following:

- self.x (numpy.array): linear array with equidistant positions. The number of data is preferibly :math:`2^n` .
- self.y (numpy.array): linear array with equidistant positions. The number of data is preferibly :math:`2^n` .
- self.wavelength (float): wavelength of the incident field.
- self.u (numpy.array): complex field with size 2D x.y

We can also find these atributes: * self.X (numpy.array): equal size to x * y. complex field * self.Y (numpy.array): equal size to x * y. complex field * self.quality (float): quality of RS algorithm. Valid for values > 1. * self.info (str): description of data. * self.type (str): Class of the field. * self.date (str): date when performed.

The dimensional magnitudes are related to microns: `micron = 1.`

Creating an instance

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

Creating a light beam. An instance must be created before starting to operate with light sources. The initialization accepts several arguments.

[2]:
```
wavelength = 0.5 * um
x0 = np.linspace(-0.25 * mm, 0.25 * mm, 512)
y0 = np.linspace(-0.25 * mm, 0.25 * mm, 512)

u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength, info='u0')
```

Light sources are defined in the scalar_sources_xy.py module. When the field is initialized, the amplitude of the field is zero. There are many methods that can be used to generate a light source:

- **plane_wave**: Generates a plane wave with a given direction and amplitude
- **gauss_beam**: Generates a gauss beam with a given amplitude, direction, beam-waist and position of beam-waist.

- **spherical_wave**: Generates a spherical wave.
- **vortex_beam**: Generates a vortex beam.
- **laguerre_beam**: Generates a Laguerre beam.
- **hermite_gauss_beam**: Generates a Hermite-Gauss beam.
- **zernike_beam**: Generates aberrated beams using Zernike functions.
- **bessel_beam**: Generates a bessel beam.
- **plane_waves_several_inclined**: Generate several plane waves with different incident angles.
- **plane_waves_dict**: Generate several plane waves with parameters defined in a dictionary.
- **gauss_beams_several_parallel**: Generates a number of parallel gauss beam at different positions.
- **gauss_beams_several_inclined**: Generates a number of gauss beams all placed at the same position but with different incidence angles.

For a more detailed description of each method, refer to the individual documentation of each one.

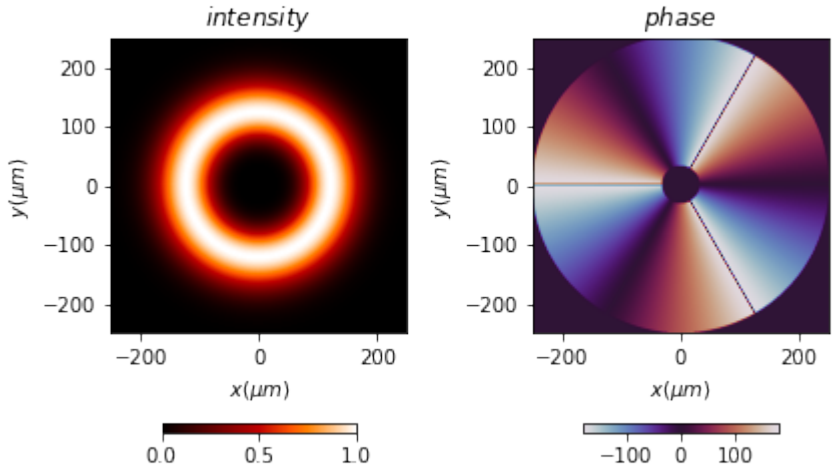
Example: vortex_beam

[3]:
```
u0.vortex_beam(A=1, r0=(0 * um, 0 * um), w0=100 * um, m=3)
print(u0)
```

```
Scalar_source_XY
 - x:  (512,),   y:  (512,),   u:  (512, 512)
 - xmin:       -250.00 um,  xmax:      250.00 um,  Dx:   0.98 um
 - ymin:       -250.00 um,  ymax:      250.00 um,  Dy:   0.98 um
 - Imin:       0.00,     Imax:      1.00
 - phase_min:  -180.00 deg, phase_max: 180.00 deg
 - wavelength: 0.50 um
 - date:       2023-12-22_17_18_30
 - info:       u0
```

Several ways to show the field are defined: ‘amplitude’, ‘intensity’, ‘field’, ‘phase’, …

[4]:
```
u0.draw(kind='field')
```

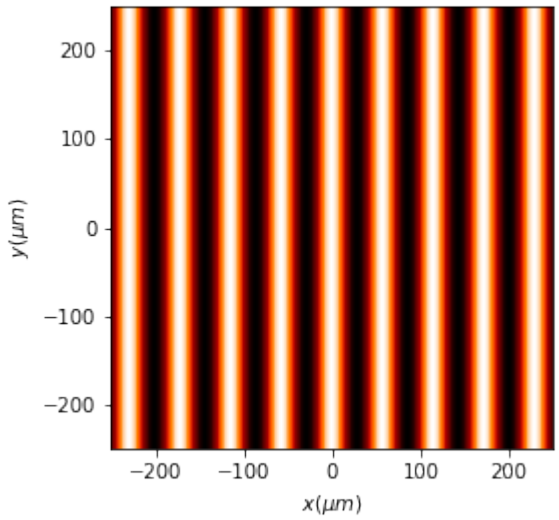


6.5.1.1 Basic operations: add to light sources.

[5]:
```
u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength, info='u1')
u2 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength, info='u2')

u1.plane_wave(A=1, theta=-.25 * degrees)
u2.plane_wave(A=1, theta=+.25 * degrees)

u_sum = u1 + u2
u_sum.draw(kind='intensity')
```



Since we have two plane waves propagating at different angles, an interference effect is produced, shown in the intensity variation of the optical field.

Masks

Masks are defined in the scalar_masks_xy.py module. There are many methods that can be used to generate masks and diffractive optical elements:

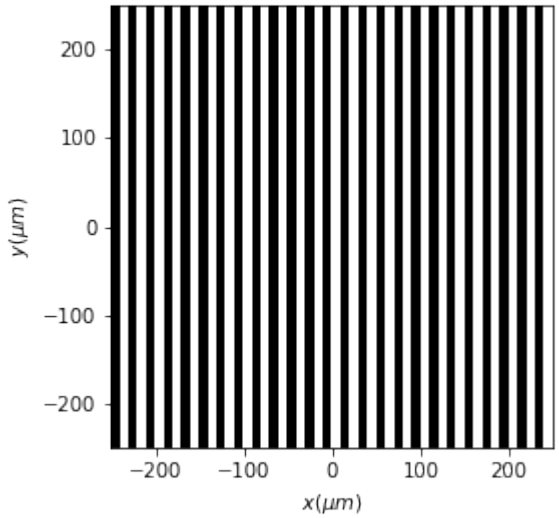
- set_amplitude, set_phase
- binarize, two_levels, gray_scale
- a_dataMatrix
- area
- save_mask
- inverse_amplitude, inverse_phase
- widen
- image
- slit, double_slit, square, circle, super_gauss, square_circle, ring, cross
- mask_from_function
- prism, lens, fresnel_lens

- sine_grating, sine_edge_grating ronchi_grating, binary_grating, blazed_grating, forked_grating, grating2D, grating_2D_ajedrez
- axicon, biprism_fresnel,
- radial_grating, angular_grating, hyperbolic_grating, archimedes_spiral, laguerre_gauss_spiral
- hammer
- roughness, circle_rough, ring_rough, fresnel_lens_rough,

For a more detailed description of each method, refer to the individual documentation of each one.

Example: Ronchi grating.

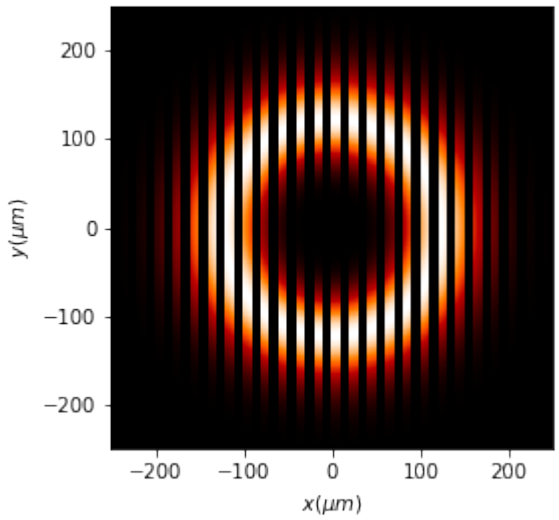
[6]:
```
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.ronchi_grating(x0=0 * um,
                  period=20 * um,
                  fill_factor=0.5,
                  angle=0 * degrees)
t0.draw(kind='intensity')
```



6.5.1.2 Basic operations: multiplication of masks and fields

When the light u_0 passes through a mask t_0, the field just after the mask, according to the Thin Element Approximation (TEA), is $u_1 = t_0 * u_0$. This can be represented in the following way

[7]:
```
u1 = u0 * t0
u1.draw(kind='intensity')
```



6.5.2 Propagation

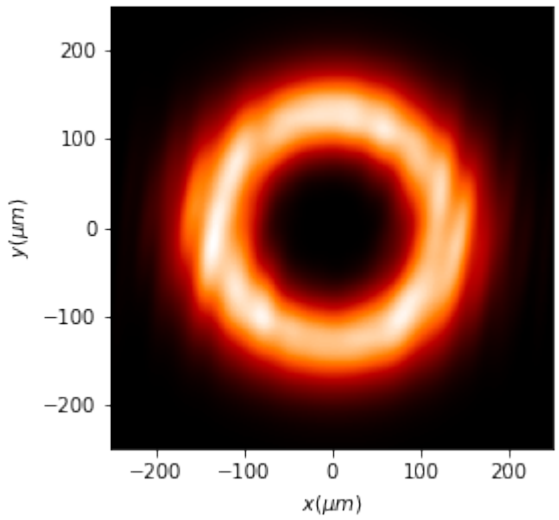
Propagation and other actions and parameters of the optical fields are defined in the scalar_field_x.py module. There are several methods of determining the field at a given plane after the mask:

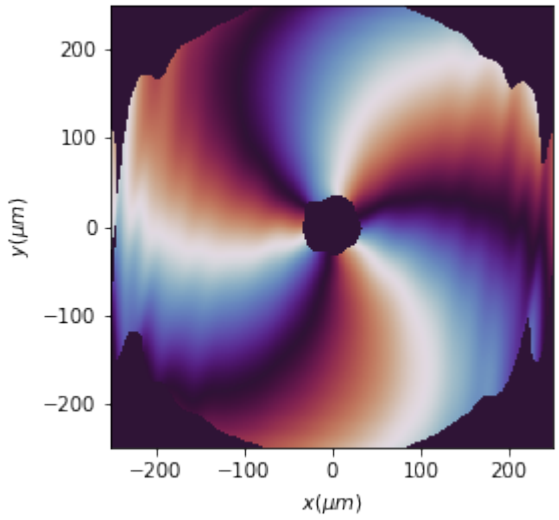
- **RS**: Rayleigh-Sommerfeld propagation at a certain distance
- **fft**: Fast Fourier propagation at the far field.
- **ifft**: Inverse Fast Fourier propagation at the far field.

The field can be stored in the same isntance, generate a new instance, or generate a numpy.array when fast computation is required.

6.5.2.1 Near field: Rayleigh-Sommerfeld approach

[8]:
```
u2 = u1.RS(z=20 * mm, new_field=True)
u2.draw(kind='intensity')
u2.draw(kind='phase')
```



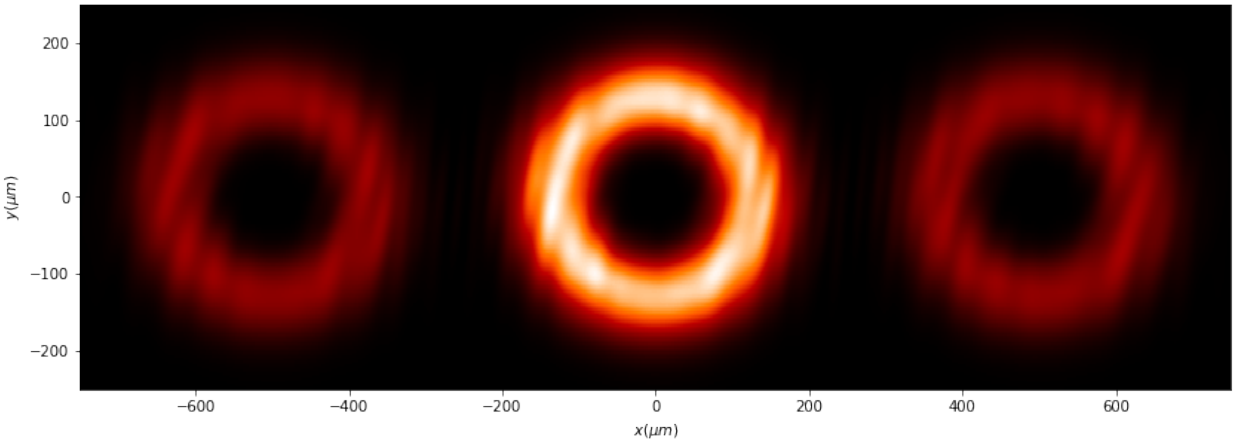


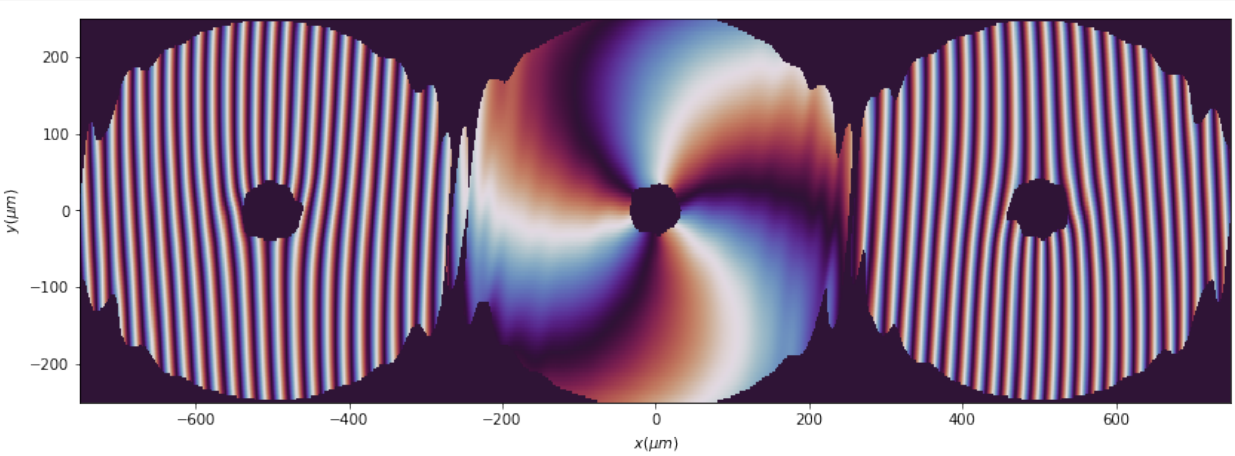
6.5.2.2 Amplification of the field

The Rayleigh-Sommefeld implementation allows an amplification of the field since, sometimes, light diverges:

[9]:
```
u2 = u1.RS(z=20 * mm, new_field=True, amplification=(3, 1))
u2.draw(kind='intensity', logarithm=False)
fig = plt.gcf()
fig.set_size_inches(13, 6)

u2.draw(kind='phase')
fig = plt.gcf()
fig.set_size_inches(13, 6)
```





6.5.2.3 Reduction of the field

There are some cases where we need to analyze the field in a small area, such as focus of a lens. Then, we can cut and resample the field.

[10]:
```
wavelength = 0.5 * um
x0 = np.linspace(-0.5 * mm, 0.5 * mm, 1024)
y0 = np.linspace(-0.5 * mm, 0.5 * mm, 1024)

focal = 25 * mm

u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength, info='u0')
u0.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.lens(r0=(0, 0), radius=(500 * um, 500 * um), focal=(focal, focal))
```

The field at focal distance results:

[11]:
```
%%time
```

(continues on next page)

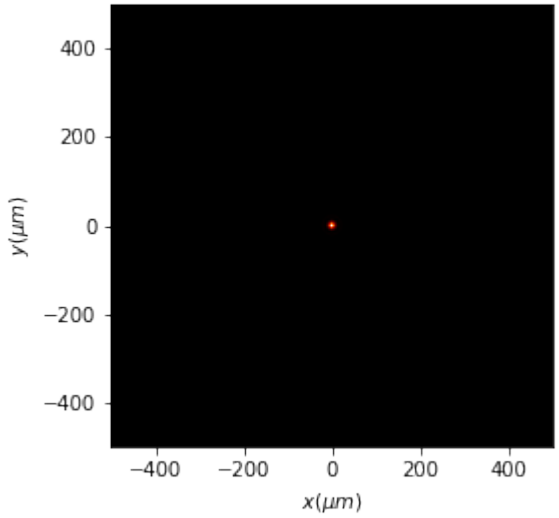
(continued from previous page)

```
u1=t0*u0

u2=u1.RS(z=focal, verbose=False)
u2.normalize();
```

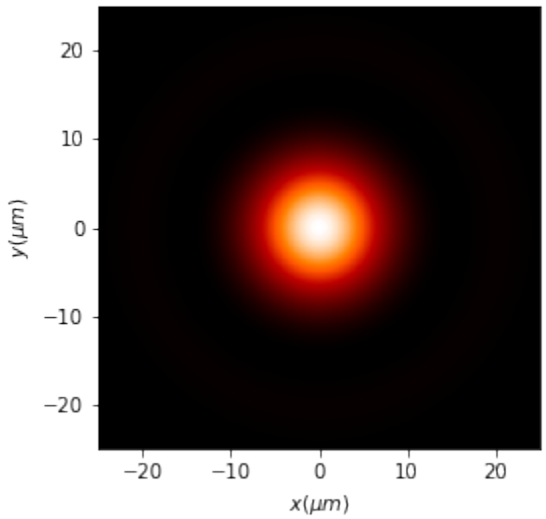
```
CPU times: user 1.94 s, sys: 151 ms, total: 2.09 s
Wall time: 2.09 s
```

[12]:
```
u2.draw(kind='intensity')
```



As, the focus cannot be seen properly, we can reduce the field to the desired size.

[13]:
```
u2.cut_resample(x_limits=(-25 * um, 25 * um),
                y_limits=(-25 * um, 25 * um),
                num_points=(512, 512),
                new_field=False)
u2.draw(kind='intensity', logarithm=False)
```



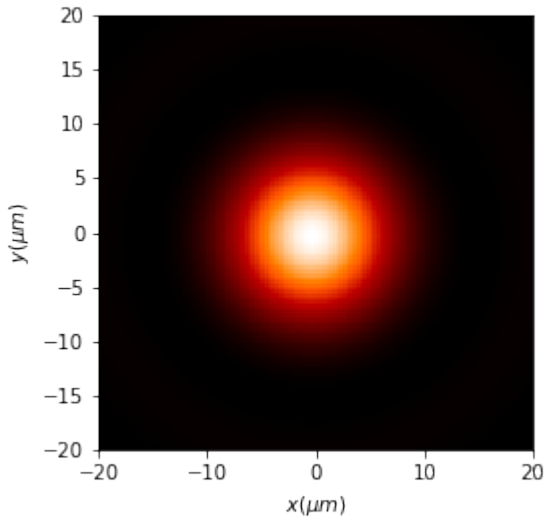
A better possibility is to use CZT algorithm to determine the field at reduced output coordinates.

[14]:
```
%%time
xout=np.linspace(-20,20,128)
yout=np.linspace(-20,20,128)


u2 = u1.CZT(z=focal, xout=xout, yout=yout)
u2.normalize()
```

```
CPU times: user 323 ms, sys: 41.9 ms, total: 365 ms
Wall time: 364 ms
```

[15]:
```
u2.draw(kind='intensity')
```



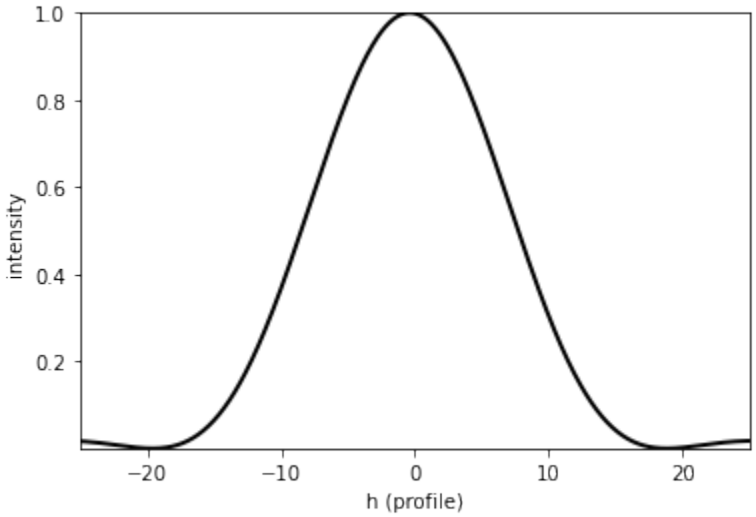
6.5.2.4 Focus position and profiles

For focus and other application, the software can detect the position of maximum intensity and determine the profiles in order to a better analysis.

[16]:
```
x_max, y_max = u2.search_focus(verbose=True)

u2.draw_profile(point1=[-25 * um, x_max],
                point2=[25 * um, x_max],
                npixels=512,
                kind='intensity',
                order=2)
```

```
x = -0.283 um, y = -0.283 um
```



6.5.3 Example of light sources

6.5.3.1 Creating an instance

An instance must be created before starting to operate with light sources. The initialization accepts several arguments.

Creating an instance

[1]:
```
from diffractio import degrees, mm, nm, um
from diffractio import np, plt, sp
from diffractio.scalar_sources_XY import Scalar_source_XY
```

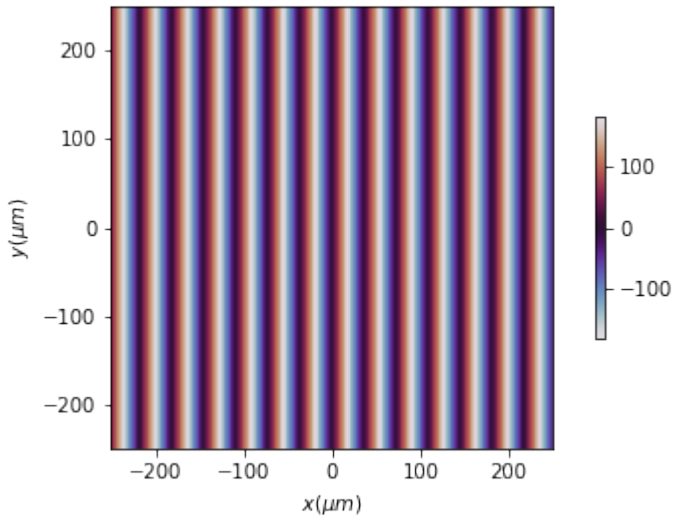
6.5.3.2 plane wave

[2]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u.plane_wave(theta=1 * degrees)

u.draw(kind='phase', has_colorbar='vertical')
```



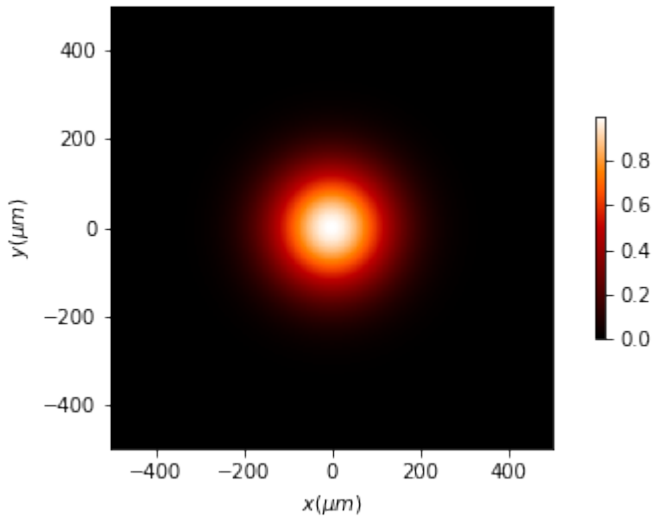
6.5.3.3 Gauss beam

[3]:
```
x0 = np.linspace(-500 * um, 500 * um, 512)
y0 = np.linspace(-500 * um, 500 * um, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

u.gauss_beam(A=1,
             r0=(0 * um, 0 * um),
             z0=0,
             w0=(200 * um, 200 * um),
             phi=0 * degrees,
             theta=0 * degrees)

u.draw(kind='intensity', has_colorbar='vertical')
```



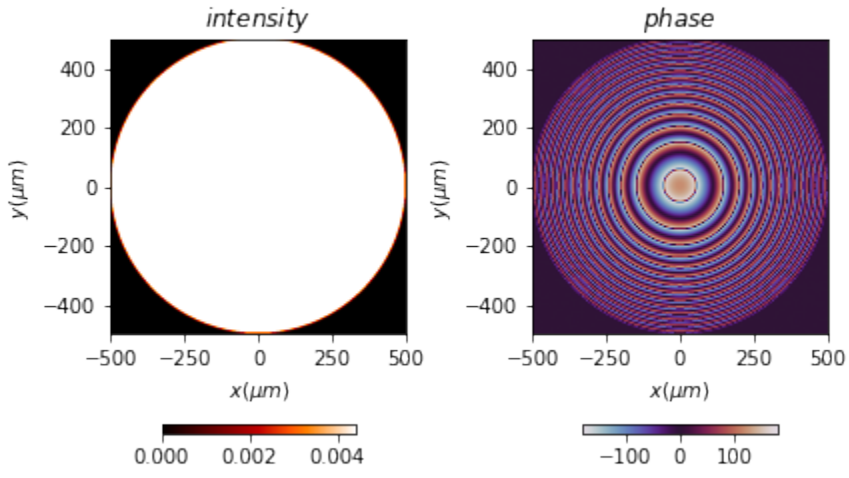
6.5.3.4 spherical wave

[4]:
```
x0 = np.linspace(-500 * um, 500 * um, 512)
y0 = np.linspace(-500 * um, 500 * um, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

u.spherical_wave(A=1,
                 r0=(0 * um, 0 * um),
                 z0=-15 * mm,
                 radius=500 * um)
u.pupil()

u.draw(kind='field', normalize='maximum')
```

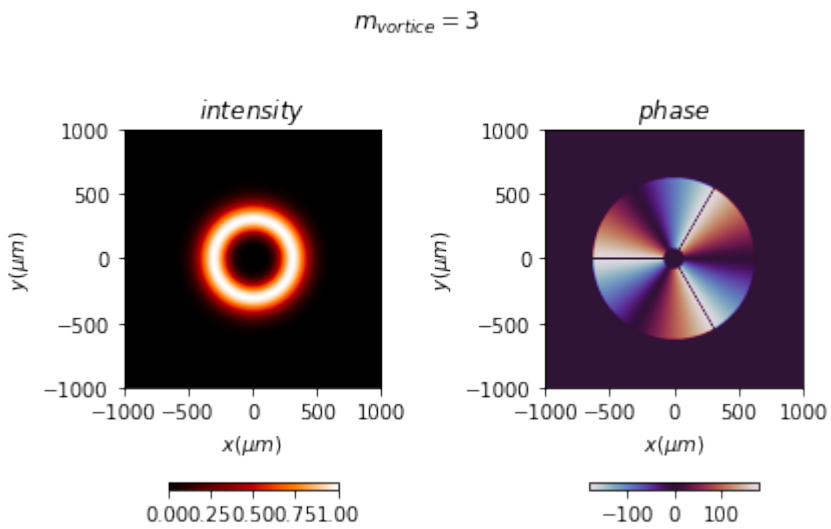


6.5.3.5 vortex beam

[5]:
```
x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

m = 3
u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u.vortex_beam(A=1, r0=(0, 0), w0=250 * um, m=m)

title = '$m_{vortice}= %d$' % (m)
u.draw(kind='field', title=title)
```



6.5.3.6 Vortex beams

[6]:
```
M = 5

x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

ID1 = plt.figure(figsize=(12, 6))
plt.suptitle("$Vortices$", fontsize=20)

for m in range(M):
    u.vortex_beam(A=1, r0=(0 * um, 0 * um), w0=400 * um, m=m)
    intensity = np.abs(u.u)**2
    phase = np.angle(u.u) / degrees
    phase[intensity < 0.005] = 0

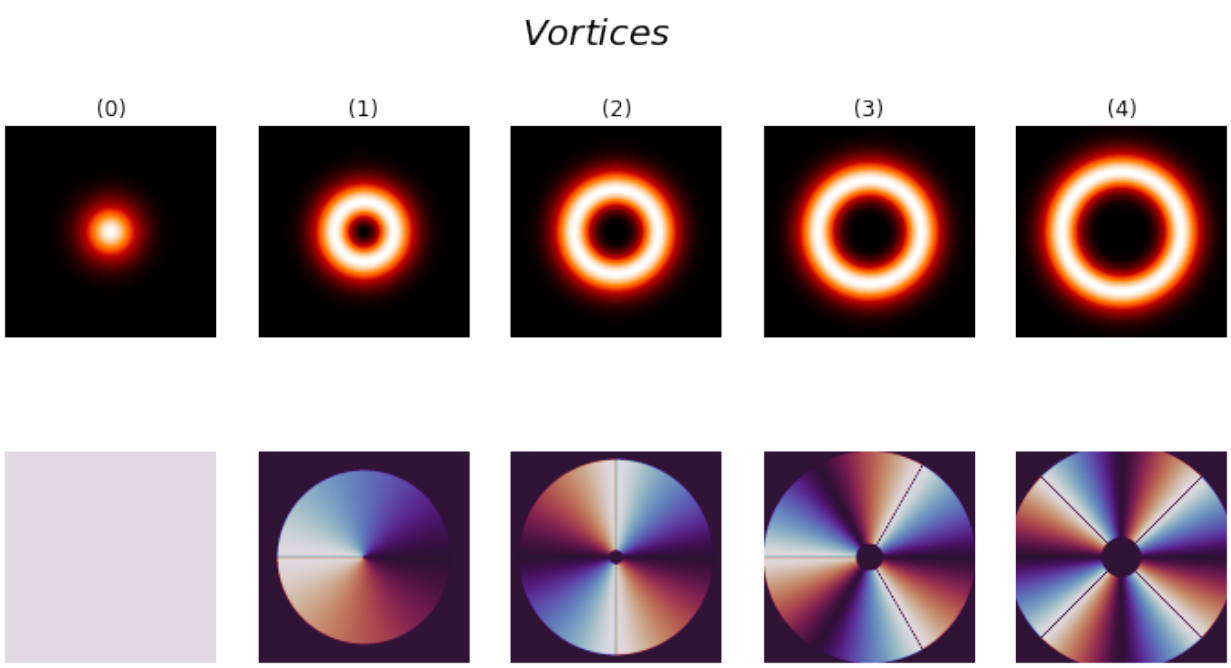
    # drawings
```

(continues on next page)

(continued from previous page)

```
    title = "(%d)" % (m)
    plt.subplot(2, M, m + 1)
    plt.axis('off')
    plt.title(title)
    h1 = plt.imshow(intensity)
    h1.set_cmap("gist_heat")

    plt.subplot(2, M, m + M + 1)
    plt.axis('off')
    h2 = plt.imshow(phase)
    h2.set_cmap("twilight")
```



6.5.3.7 Laguerre beam

[7]:
```
N = 2
K = 2

x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

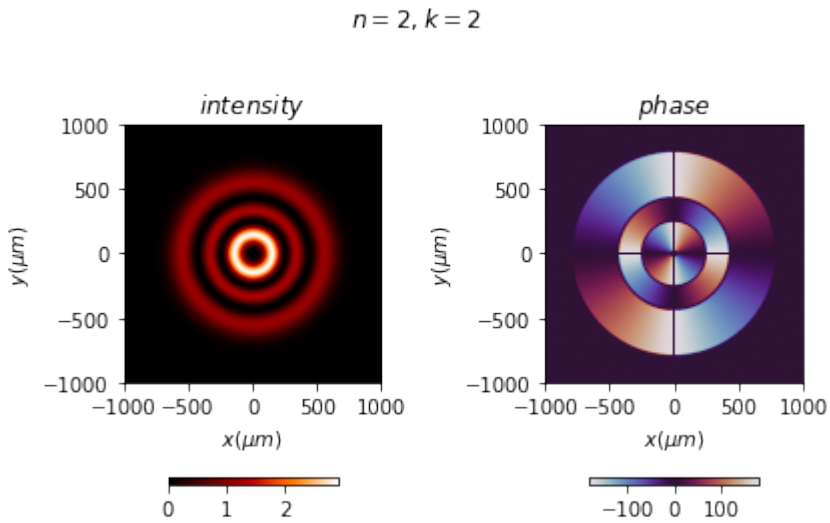
u.laguerre_beam(A=1,
                n=N,
                l=K,
                r0=(0 * um, 0 * um),
                w0=250 * um,
```

(continues on next page)

(continued from previous page)

```
                z0=0,
                z=0.01 * um)

title = r'$n=%d, k=%d$' % (N, K)
u.draw(kind='field', title=title)
```



6.5.3.8 Laguerre beams

[8]:
```
N = 3
K = 3

x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

ID1 = plt.figure(figsize=(8, 8))
plt.suptitle("amplitude", fontsize=20)
ID2 = plt.figure(figsize=(8, 8))
plt.suptitle("phase", fontsize=20)

for n in range(N + 1):
    for k in range(K + 1):

        u.laguerre_beam(A=1,
                        n=n,
                        l=k,
                        r0=(0 * um, 0 * um),
                        w0=250 * um,
                        z0=0,
                        z=0.01 * um)
```

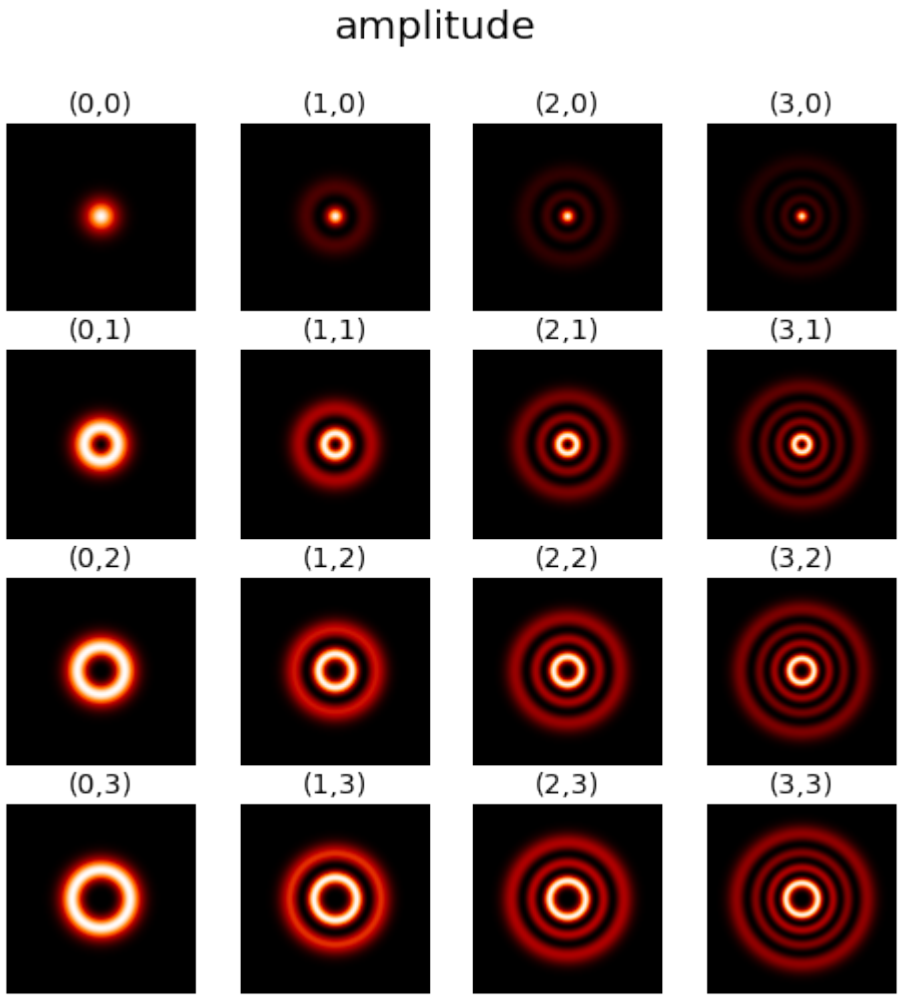
(continues on next page)

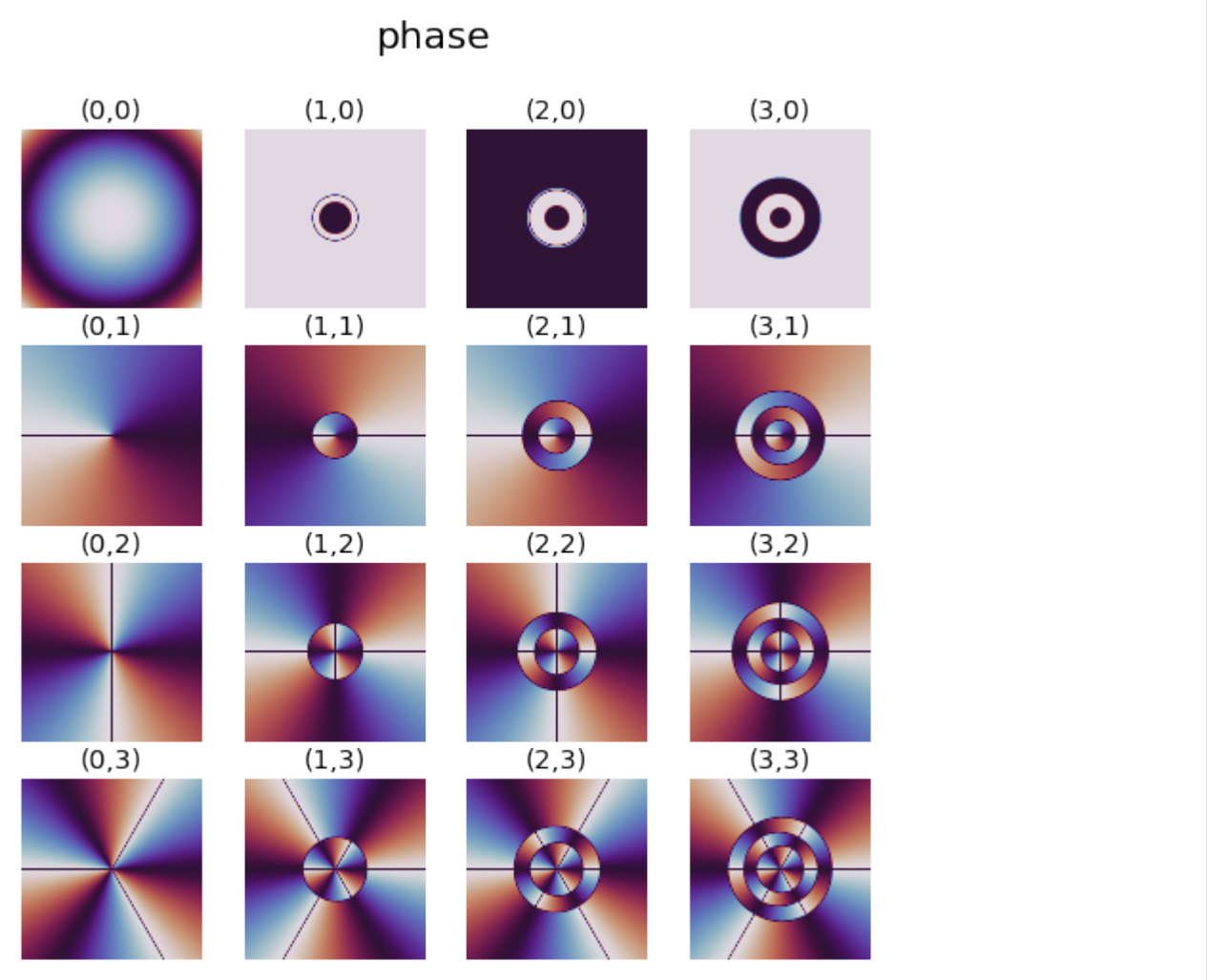
(continued from previous page)

```
        title = "(%d,%d)" % (n, k)
        plt.figure(ID1.number)
        plt.subplot(N + 1, K + 1, (N + 1) * (k) + n + 1)
        plt.axis('off')
        plt.title(title, fontsize=14)
        h1 = plt.imshow(np.abs(u.u)**2)
        h1.set_cmap("gist_heat")

        plt.figure(ID2.number)
        plt.subplot(N + 1, K + 1, (N + 1) * (k) + n + 1)
        plt.axis('off')
        plt.title(title, fontsize=14)
        h2 = plt.imshow(np.angle(u.u))
        h2.set_cmap("twilight")

plt.figure(ID1.number)
plt.figure(ID2.number)
```





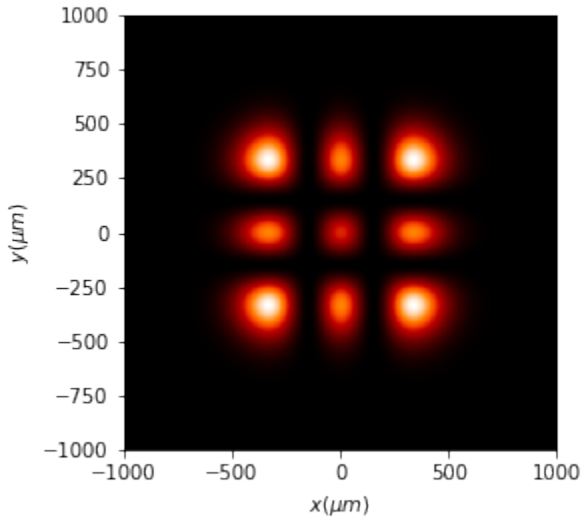
6.5.3.9 Hermite_Gauss

[9]:
```
x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

u.hermite_gauss_beam(A=1,
                     r0=(0, 0),
                     w0=[.3 * mm, .3 * mm],
                     z0=(0, 0 * mm),
                     z=0,
                     n=2,
                     m=2)

u.draw(kind='intensity')
```



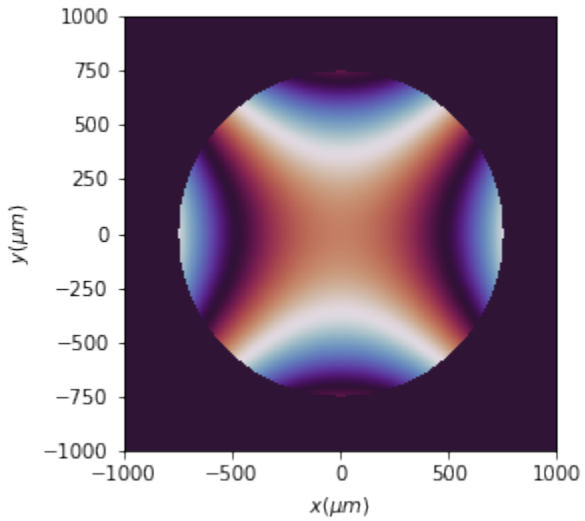
6.5.3.10 Zernike beam

[10]:
```
N = np.array([0, 2])
M = np.array([0, 2])
c_nm = np.array([2, 2])

x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

u.zernike_beam(A=1, r0=(0, 0), radius=750 * um, n=N, m=M, c_nm=c_nm)
u.draw(kind='phase')
```



6.5.3.11 Zernike beams

[11]:
```
N = 4

x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

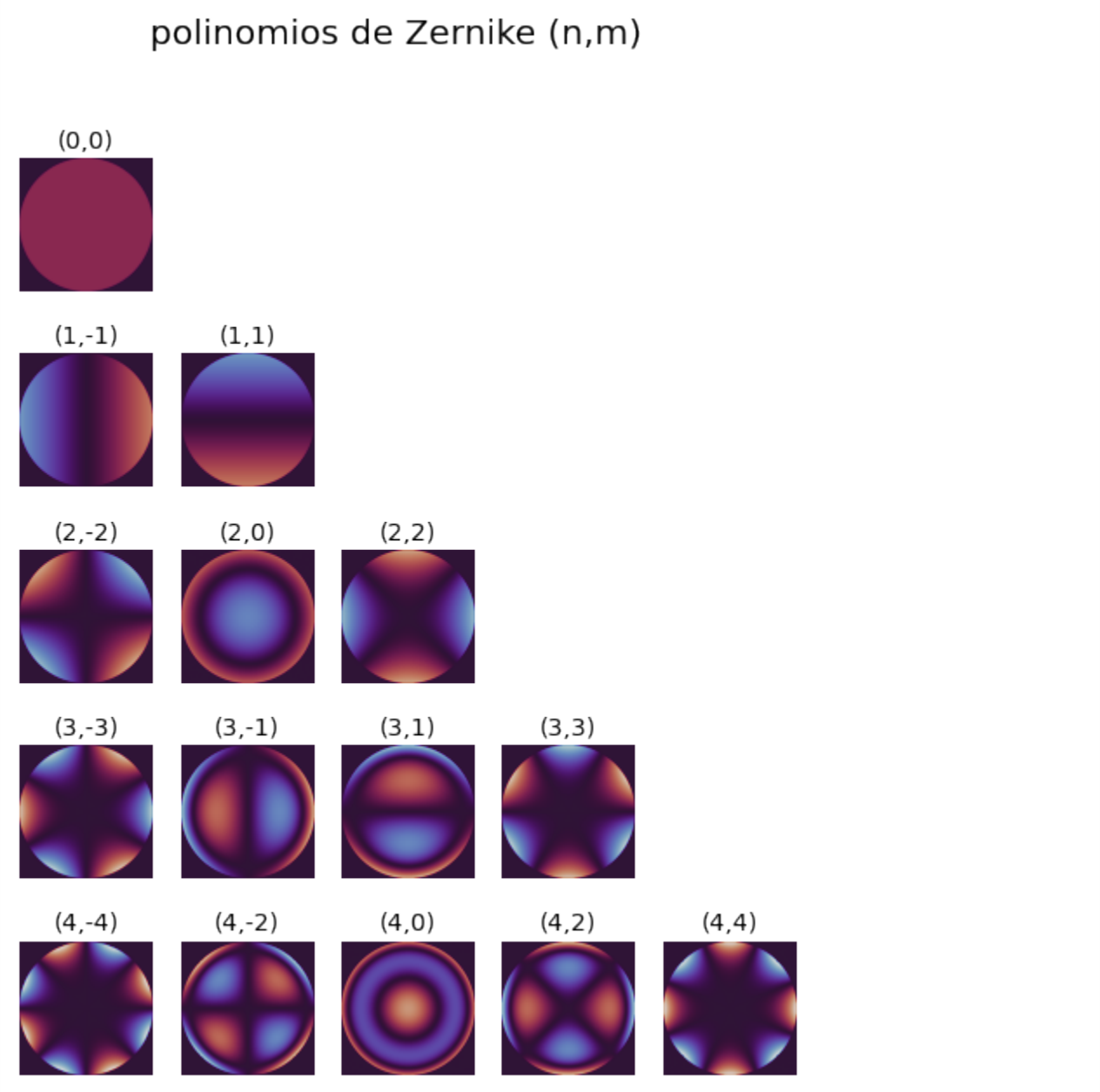
plt.figure(figsize=(8, 10))
plt.suptitle(u"polinomios de Zernike (n,m)", fontsize=20)

u = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

for n in range(0, N + 1):
    pos = 0
    for m in range(-n, n + 1, 2):
        pos = pos + 1

        u.zernike_beam(A=1,
                       r0=(0, 0),
                       radius=1 * mm,
                       n=np.array([n]),
                       m=np.array([m]),
                       c_nm=np.array([1]))
        u.pupil()

        title = "(%d,%d)" % (n, m)
        plt.subplot(N + 1, N + 1, n * (N + 1) + pos)
        plt.axis('off')
        plt.title(title, fontsize=14)
        h2 = plt.imshow(np.angle(u.u))
        plt.clim(vmin=-np.pi, vmax=np.pi)
        h2.set_cmap("twilight")
```



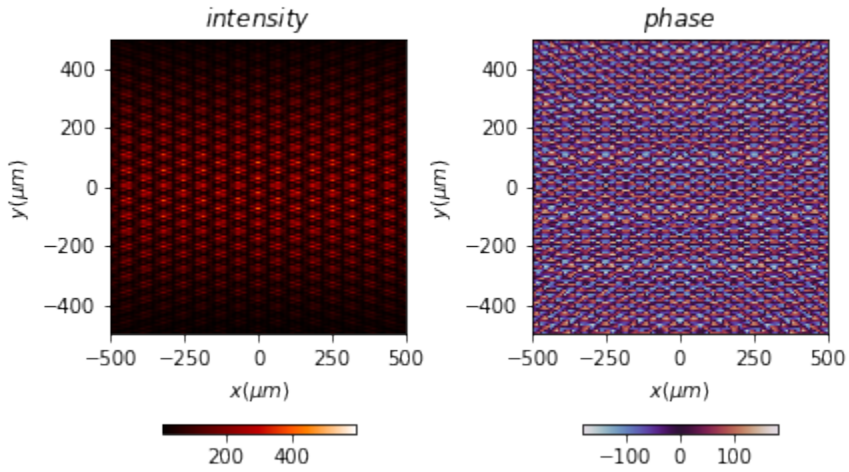
6.5.3.12 Several plane waves inclined

[12]:
```
x0 = np.linspace(-500 * um, 500 * um, 256)
y0 = np.linspace(-500 * um, 500 * um, 256)
wavelength = 0.6328 * um

u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.plane_waves_several_inclined(A=1,
                                num_beams=(5, 5),
                                max_angle=(5 * degrees, 45 * degrees))
u0.draw(kind='field')
```

(continues on next page)

(continued from previous page)



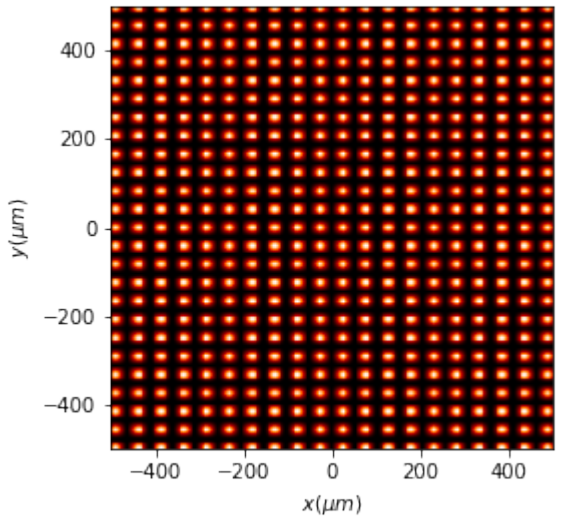
6.5.3.13 plane waves with parameters from dictionary

[13]:
```
x0 = np.linspace(-500 * um, 500 * um, 256)
y0 = np.linspace(-500 * um, 500 * um, 256)
wavelength = 0.6328 * um

params = []
p1 = dict(A=1, theta=5 * degrees, phi=5 * degrees, z0=0)
p2 = dict(A=1, theta=-5 * degrees, phi=5 * degrees, z0=0)
p3 = dict(A=1, theta=5 * degrees, phi=-5 * degrees, z0=0)
p4 = dict(A=1, theta=-5 * degrees, phi=-5 * degrees, z0=0)

params.append(p1)
params.append(p2)
params.append(p3)
params.append(p4)

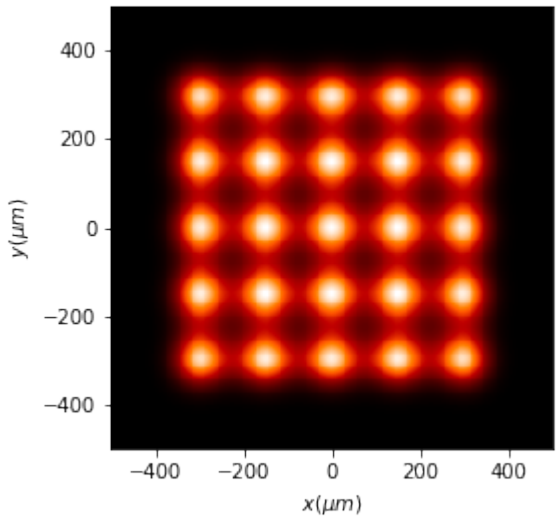
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.plane_waves_dict(params)
u0.draw()
```



6.5.3.14 several parallel Gauss beams

[14]:
```
x0 = np.linspace(-500 * um, 500 * um, 256)
y0 = np.linspace(-500 * um, 500 * um, 256)
wavelength = 0.6328 * um

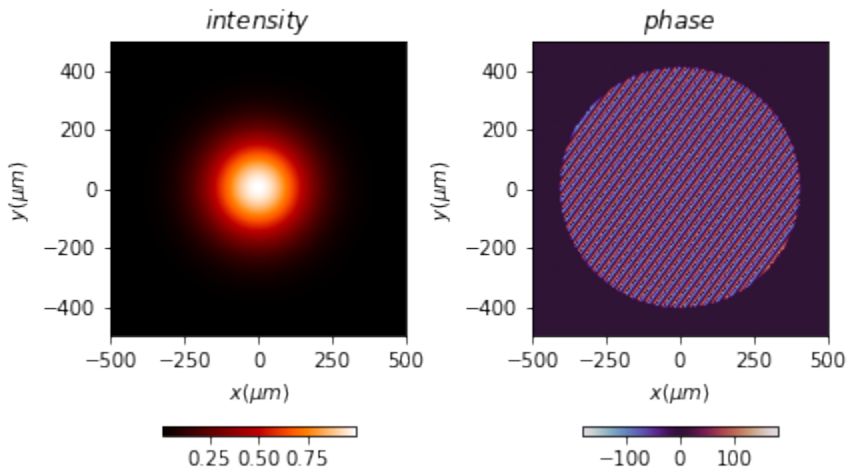
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.gauss_beams_several_parallel(A=1,
                                num_beams=(5, 5),
                                w0=75 * um,
                                z0=0 * um,
                                r0=(0 * um, 0 * um),
                                r_range=(750 * um, 750 * um),
                                theta=0 * degrees,
                                phi=0 * degrees)
u0.draw(kind='intensity')
```



6.5.3.15 several inclined Gauss beams

[15]:
```
x0 = np.linspace(-500 * um, 500 * um, 256)
y0 = np.linspace(-500 * um, 500 * um, 256)
wavelength = 0.6328 * um

u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.gauss_beams_several_inclined(A=1,
                                num_beams=(5, 5),
                                w0=250 * um,
                                r0=(0 * um, 0 * um),
                                z0=0 * um,
                                max_angle=(10 * degrees, 180 * degrees))
u0.draw(kind='field')
```



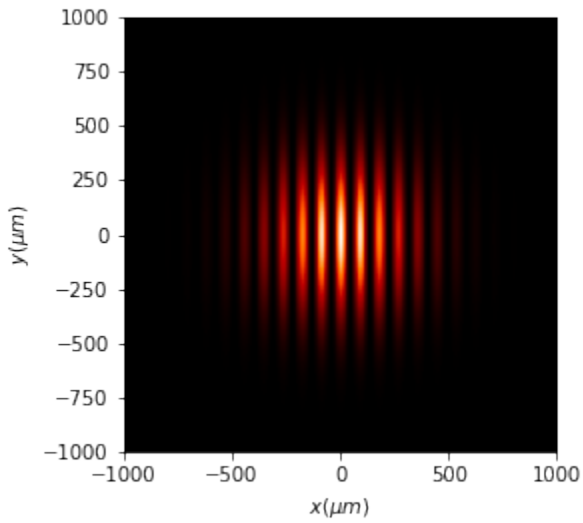
6.5.3.16 Interference

[16]:
```
x0 = np.linspace(-1 * mm, 1 * mm, 512)
y0 = np.linspace(-1 * mm, 1 * mm, 512)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u2 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)

u1.gauss_beam(A=1,
              r0=(0 * um, 0 * um),
              w0=(500 * um, 500 * um),
              z0=0 * um,
              theta=.2 * degrees)
u2.gauss_beam(A=1,
              r0=(0 * um, 0 * um),
              w0=(500 * um, 500 * um),
              z0=0 * um,
              theta=-.2 * degrees)

u3 = u1 + u2
u3.draw(kind='intensity')
```



[ ]:
```
```

6.5.4 Examples of masks

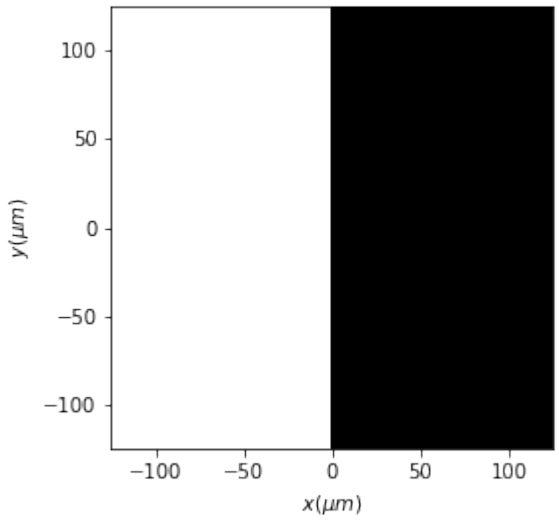
[1]:
```
from diffractio import sp, plt, np
from diffractio import nm, mm, degrees, um
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.utils_drawing import draw_several_fields
```

6.5.4.1 Standard masks

Two_levels

[2]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.two_levels(level1=1, level2=0, x_edge=0)
t1.draw(kind='intensity')
```

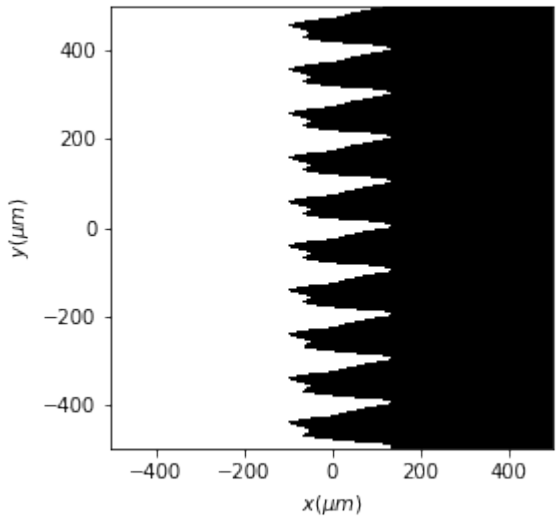


Edge Series

[3]:
```
num_pixels = 512
length = 1000 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.06238 * um

a_coef = np.array([[0, 1, 2], [0, 100, 25]])
b_coef = np.array([[0, 3], [0, 25]])

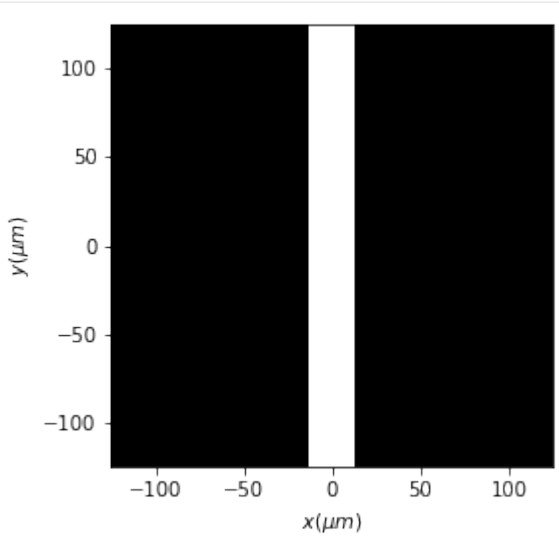
t2 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t2.edge_series(r0=(0, 0),
               period=100,
               a_coef=a_coef,
               b_coef=b_coef,
               angle=0 * degrees,
               invert=False)
t2.draw()
```



slit

[4]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

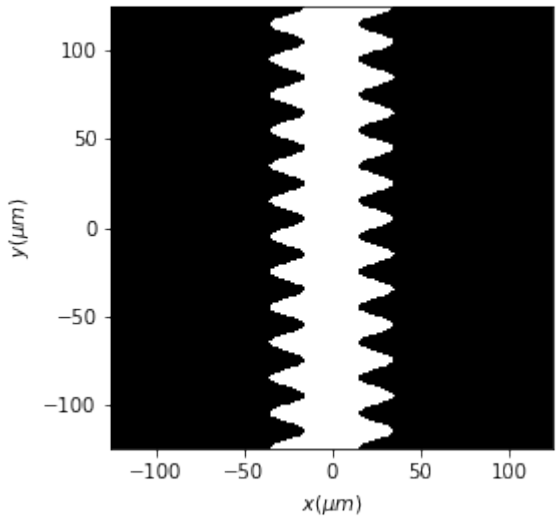
t1 = Scalar_mask_XY(x, y, wavelength)
t1.slit(x0=0, size=25 * um, angle=0 * degrees)
t1.draw(kind='intensity')
```



sinusoidal slit

[5]:
```
x = np.linspace(-125 * um, 125 * um, 512)
y = np.linspace(-125 * um, 125 * um, 512)
wavelength = 1 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.sinusoidal_slit(x0=0 * um,
                   size=50 * um,
                   amplitude=10 * um,
                   phase=0 * degrees,
                   angle=0 * degrees,
                   period=20 * um)
t1.draw(kind='intensity')
```



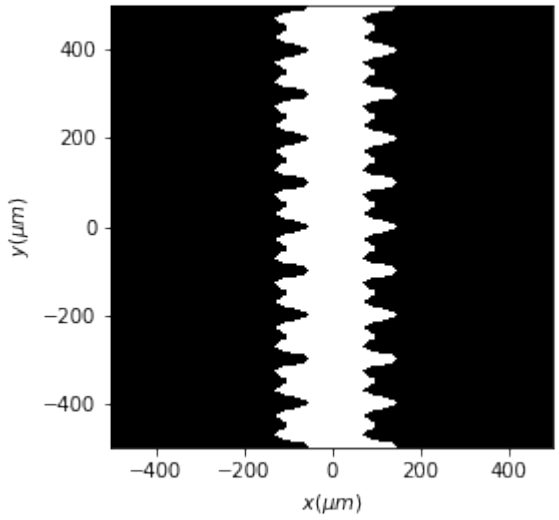
slit_series

[6]:
```
num_data = 256
length = 1000 * um
x0 = np.linspace(-length / 2, length / 2, num_data)
y0 = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

a_coef1 = np.array([[0, 1, 2], [0, 25, 25]])
a_coef2 = np.array([[0, 1, 2], [0, 25, 25]])

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.slit_series(x0=0,
               width=200,
               period1=100,
               period2=100,
               Dy=(0, 0),
               a_coef1=a_coef1,
               a_coef2=a_coef2,
               angle=0 * degrees)

t1.draw()
```



double_slit

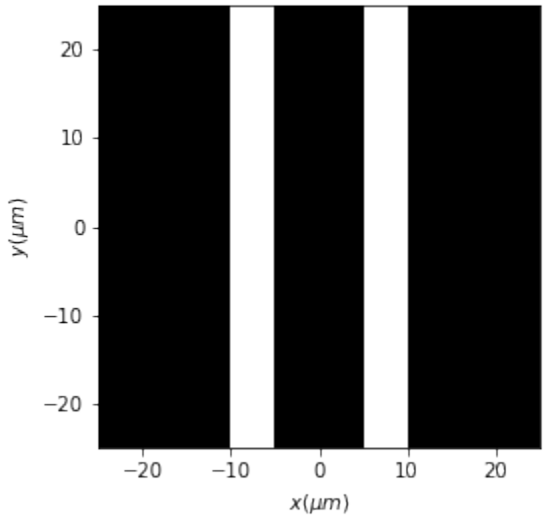
[7]:
```
num_data = 512
length = 50 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
```

(continues on next page)

(continued from previous page)

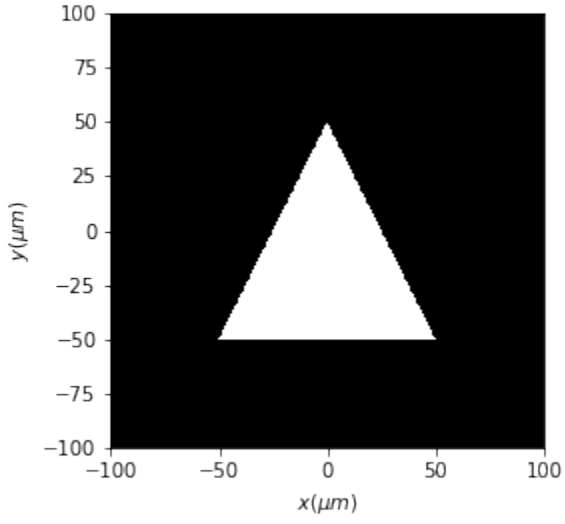
```
t1.double_slit(x0=0, size=5 * um, separation=15 * um, angle=0 * degrees)
t1.draw(kind='intensity')
```



triangle

[8]:
```
x = np.linspace(-100 * um, 100 * um, 512)
y = np.linspace(-100 * um, 100 * um, 512)
wavelength = 1 * um
t1 = Scalar_mask_XY(x, y, wavelength)

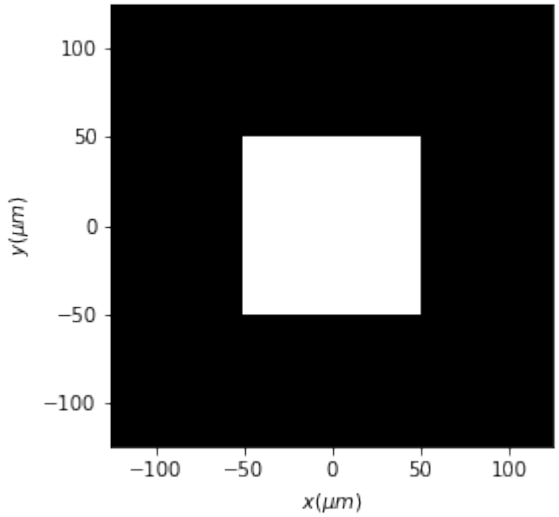
t1.triangle(r0=(0, 50 * um), slope=2, height=100 * um, angle=0 * degrees)
t1.draw(kind='intensity')
```



square

[9]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

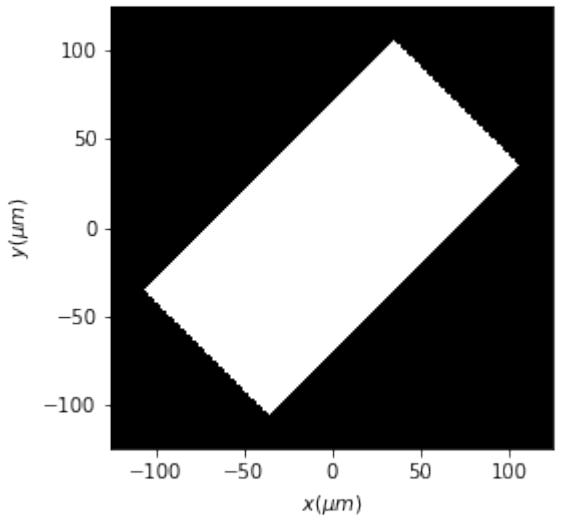
t1 = Scalar_mask_XY(x, y, wavelength)
t1.square(r0=(0 * um, 0 * um), size=100*um, angle=0 * degrees)
t1.draw(kind='intensity')
```



With this function we can also generate rectangles

[10]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.square(r0=(0 * um, 0 * um), size=(200 * um, 100 * um), angle=45 * degrees)
t1.draw(kind='intensity')
```



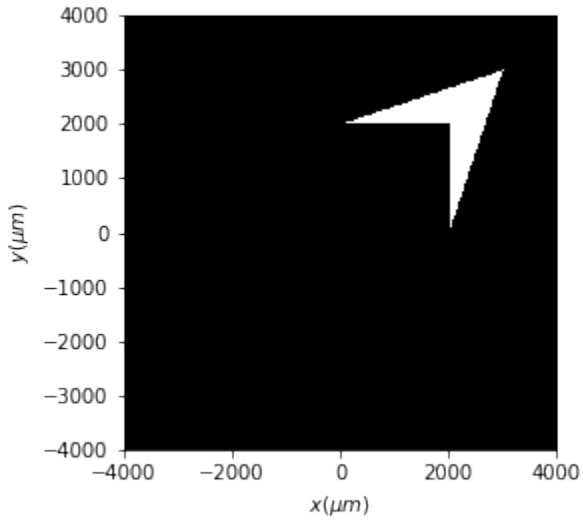
polygon

[11]:
```
x0 = np.linspace(-4 * mm, 4 * mm, 512)
y0 = np.linspace(-4 * mm, 4 * mm, 512)
wavelength = 0.6328 * um

vertices = np.array([(2 * mm, 2 * mm), (2 * mm, 0 * mm), (3 * mm, 3 * mm),
                     (0 * mm, 2 * mm)])

t = Scalar_mask_XY(x0, y0, wavelength)

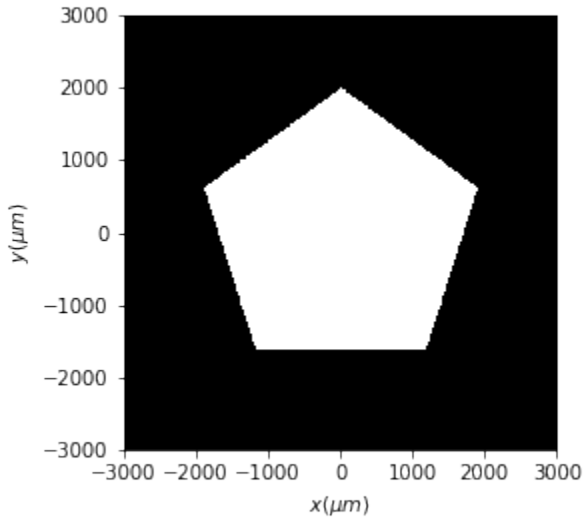
t.polygon(vertices)
t.draw()
```



regular polygon

[12]:
```
x0 = np.linspace(-3 * mm, 3 * mm, 512)
y0 = np.linspace(-3 * mm, 3 * mm, 512)
wavelength = 0.6328 * um

t = Scalar_mask_XY(x0, y0, wavelength)
vertices = t.regular_polygon(num_vertices = 5, radius = 2 * mm, angle = 0 * degrees)
t.draw()
```

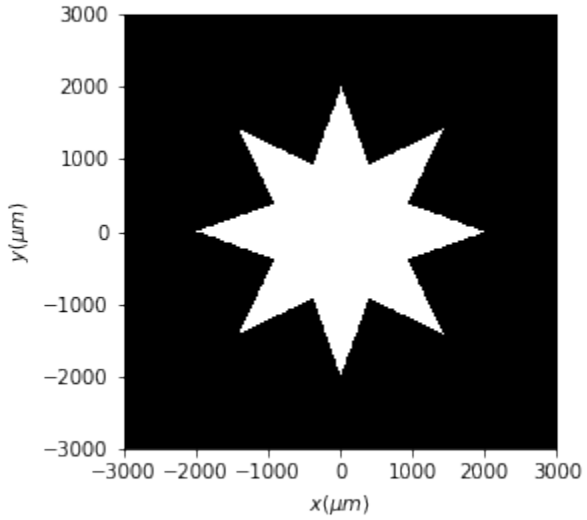


star

[13]:
```
x0 = np.linspace(-3 * mm, 3 * mm, 512)
y0 = np.linspace(-3 * mm, 3 * mm, 512)
wavelength = 0.6328 * um



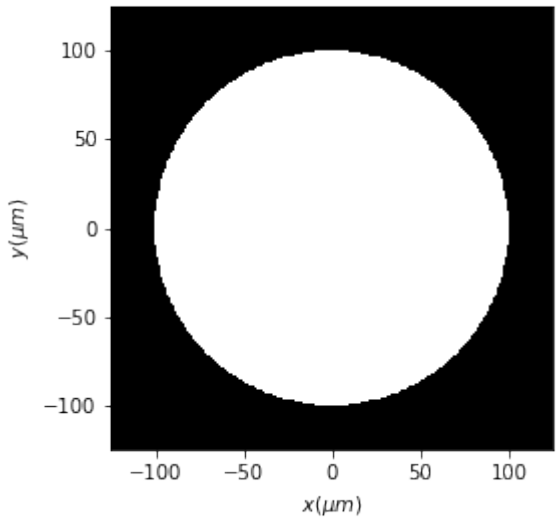
t = Scalar_mask_XY(x0, y0, wavelength)
vertices = t.star(num_peaks = 8, radii = (2 * mm, 1*mm), angle = 0 * degrees)
t.draw()
```



circle

[14]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

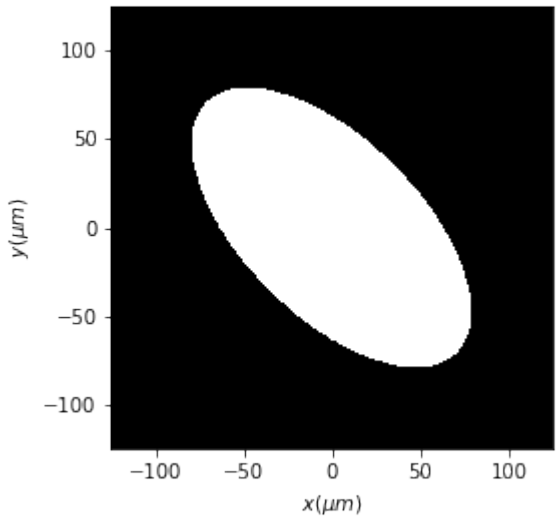
t1 = Scalar_mask_XY(x, y, wavelength)
t1.circle(r0=(0 * um, 0 * um), radius=100*um)
t1.draw(kind='intensity')
```



With this function we can also generate ellipses

[15]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.circle(r0=(0 * um, 0 * um), radius=(50 * um, 100 * um), angle=45*degrees)
t1.draw(kind='intensity')
```

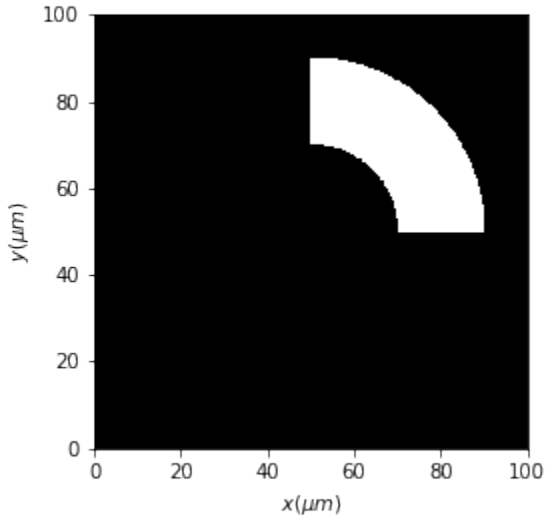


and also angular sectors.

[16]:
```
r0 = (50*um, 50*um)
radii = (20*um, 40 *um)
angles = (0 * degrees, 90 * degrees)
size = 100*um
xin = np.linspace(0, size, 1024*2)
yin = np.linspace(0, size, 1024*2)
wavelength = 0.6328*um


t = Scalar_mask_XY(x=xin, y=yin, wavelength=wavelength)

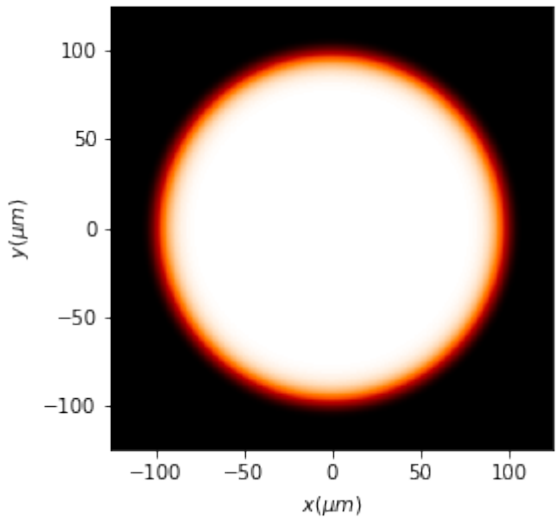
t.circular_sector(r0=r0, radii=radii, angles=angles)
t.draw()
```



super_gauss

[17]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

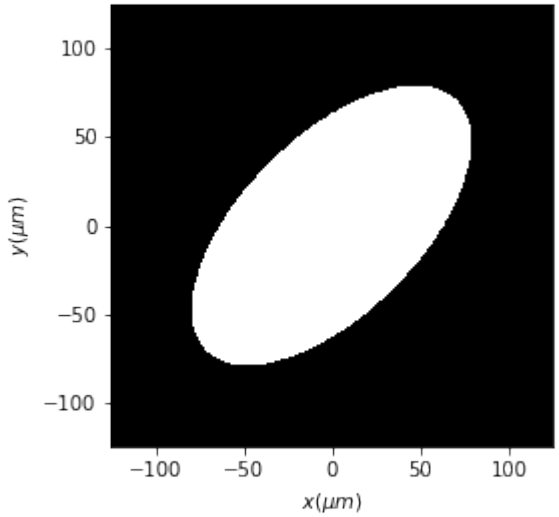
t1 = Scalar_mask_XY(x, y, wavelength)
t1.super_gauss(r0=(0 * um, 0 * um), radius=100 * um, power=22)
t1.draw(kind='intensity')
```



square_circle

[18]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

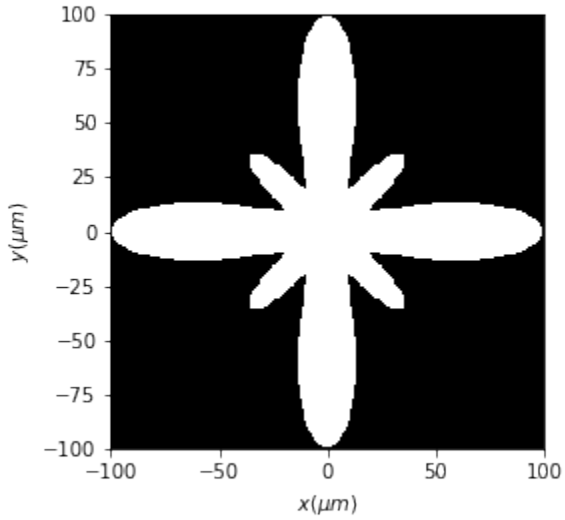
t1 = Scalar_mask_XY(x, y, wavelength)
t1.circle(r0=(0 * um, 0 * um), radius=(100 * um, 50 * um), angle=45 * degrees)
t1.draw(kind='intensity')
```



angular_aperture

[19]:
```
num_pixels = 256
length = 200 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

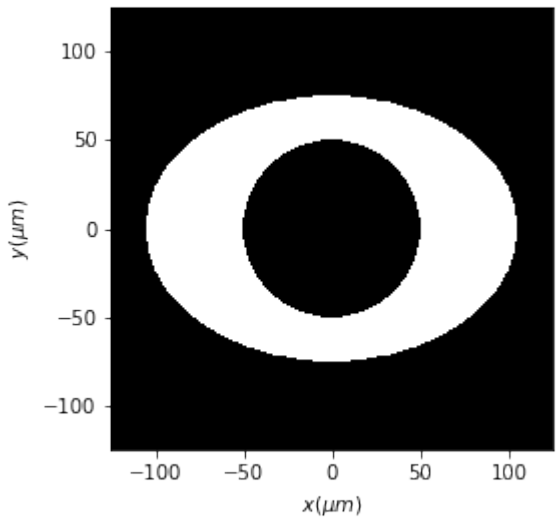
t2 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
a_coef = np.array([[0, 4, 8], [50, 25, 25]])
t2.angular_aperture(a_coef=a_coef, angle=0 * degrees)
t2.draw()
```



ring

[20]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.ring(r0=(0 * um, 0 * um),
        radius1=(50 * um, 50 * um),
        radius2=(75 * um, 105 * um),
        angle=90 * degrees)
t1.draw(kind='intensity')
```



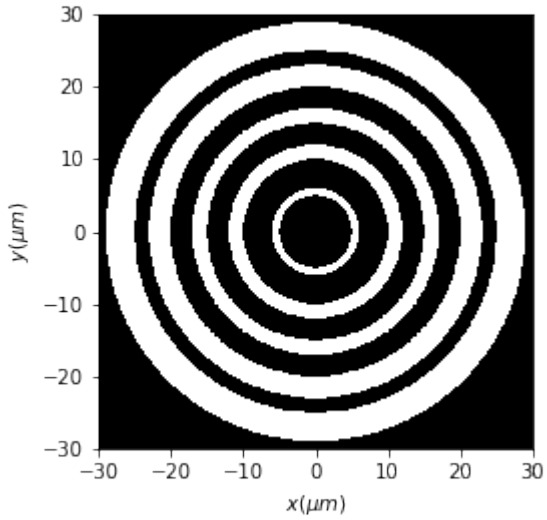
rings

[21]:
```
num_data = 512
length = 60 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

inner_radius = np.array([5., 10, 15, 20, 25])

outer_radius = np.array([6., 12, 17, 23, 29])

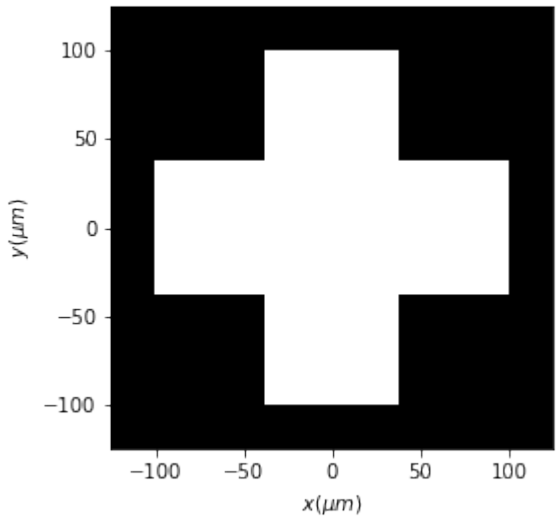
t1 = Scalar_mask_XY(x, y, wavelength)
t1.rings(r0=(0, 0), inner_radius=inner_radius, outer_radius=outer_radius)
t1.pupil()
t1.draw(kind='intensity')
```



cross

[22]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.cross(r0=(0 * um, 0 * um), size=(200 * um, 75 * um), angle=0 * degrees)
t1.draw(kind='intensity')
```

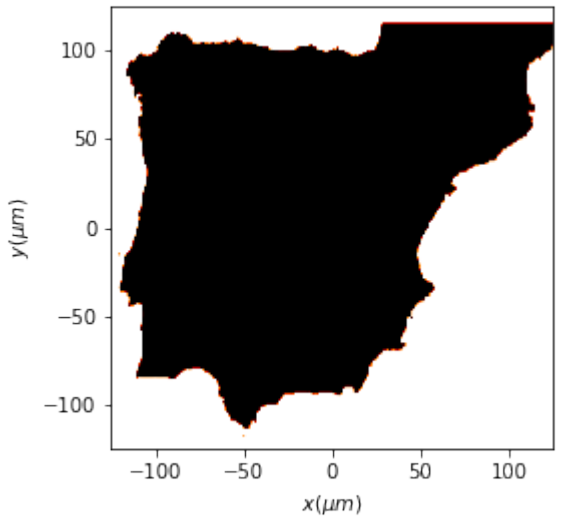


image

[23]:
```
num_data = 1204
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.image(filename="spain.png",
         normalize=True,
         canal=0,
         lengthImage=True,
         angle=0 * degrees)

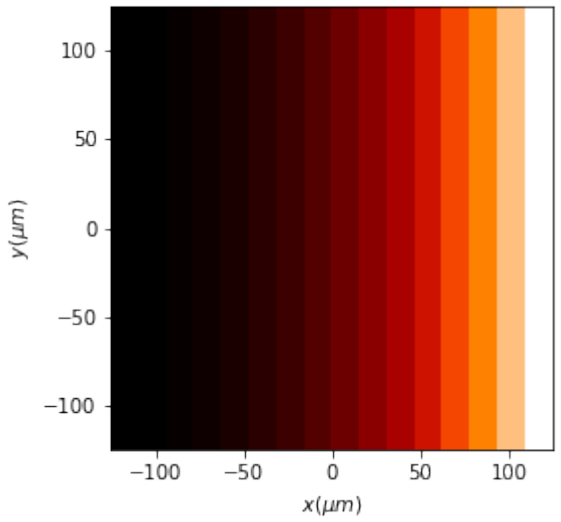
t1.draw(kind='intensity')
```



gray_scale

[24]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.gray_scale(num_levels=16, levelMin=0, levelMax=1)
t1.draw(kind='intensity')
```

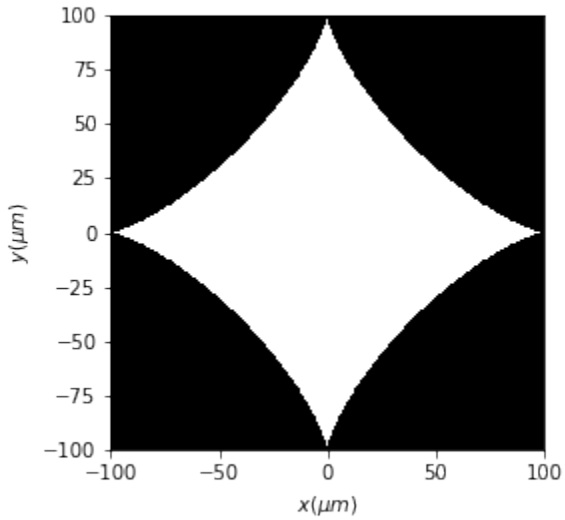


hiperellipse

[25]:
```
x = np.linspace(-100 * um, 100 * um, 512)
y = np.linspace(-100 * um, 100 * um, 512)
wavelength = 1 * um

t1 = Scalar_mask_XY(x, y, wavelength)

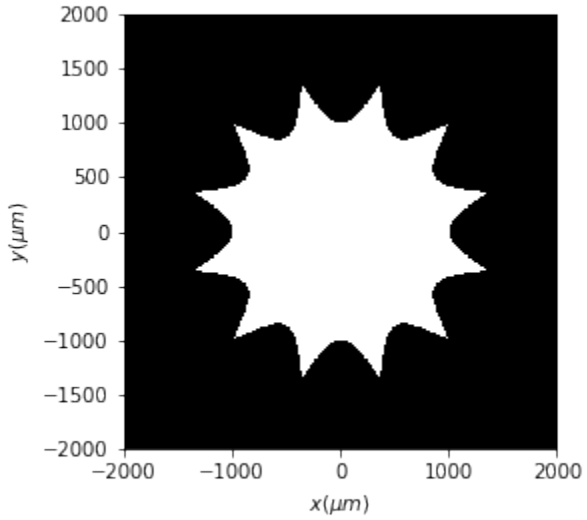
t1.super_ellipse(r0=(0, 0),
                 radius=(100 * um, 100 * um),
                 angle=0 * degrees,
                 n=[0.75, 0.75])
t1.draw(kind='intensity')
```



superformula

[26]:
```
x = np.linspace(-2 * mm, 2 * mm, 512)
y = np.linspace(-2 * mm, 2 * mm, 512)
wavelength = 1 * um

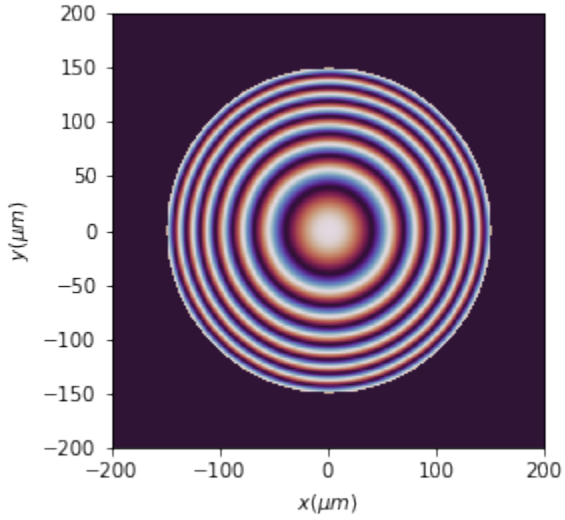
t = Scalar_mask_XY(x, y, wavelength)
t.superformula(r0=(0, 0), radius=(1 * mm, 1 * mm), n=(160, 160, 160), m=12)
t.draw()
```



lens

[27]:
```
num_data = 512
length = 400 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

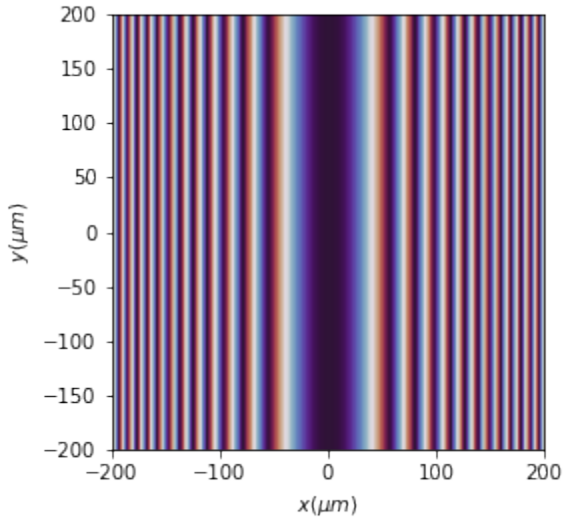
t0 = Scalar_mask_XY(x, y, wavelength)
t0.lens(r0=(0 * um, 0 * um),
        radius=(150 * um, 150 * um),
        focal=(2.5 * mm, 2.5 * mm),
        angle=0 * degrees)
t0.draw(kind='phase')
```



Cylindrical lens

[28]:
```
num_data = 512
length = 400 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

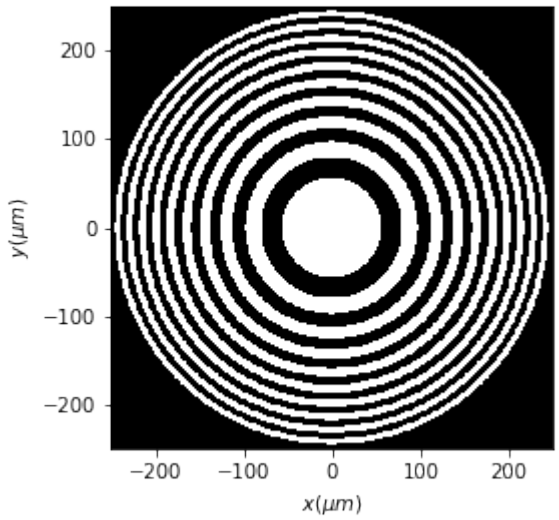
t0 = Scalar_mask_XY(x, y, wavelength)
t0.lens_cylindrical(x0=0 * um, focal=2.5 * mm, angle=0 * degrees)
t0.draw(kind='phase')
```



Fresnel lens

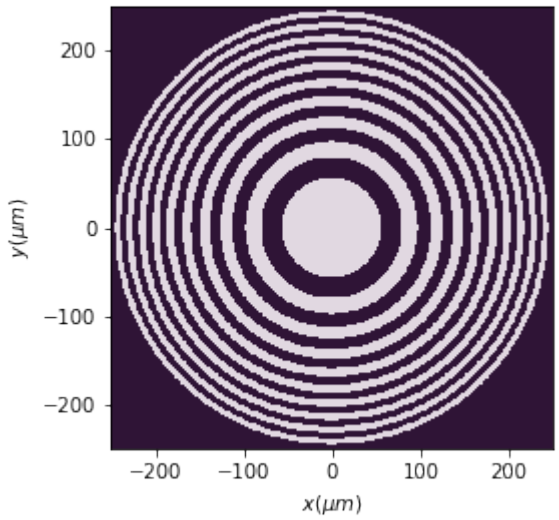
[29]:
```
# Amplitude lens
num_data = 512
length = 500 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.fresnel_lens(r0=(0 * um, 0 * um),
                radius=(250 * um, 250 * um),
                focal=(5 * mm, 5 * mm),
                angle=45 * degrees,
                kind='amplitude',
                phase=np.pi)
t1.draw(kind='intensity')
```



[30]:
```
# Phase lens
num_data = 512
length = 500 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

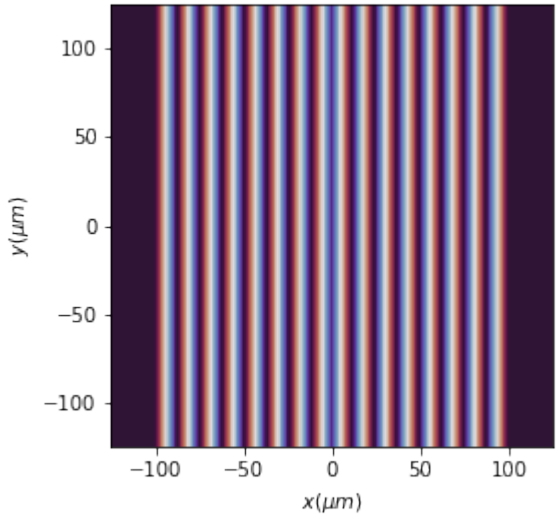
t1.fresnel_lens(r0=(0 * um, 0 * um),
                radius=(250 * um, 250 * um),
                focal=(5 * mm, 5 * mm),
                angle=0 * degrees,
                kind='phase',
                phase=np.pi)
t1.draw(kind='phase')
```



biprism_fresnel

[31]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

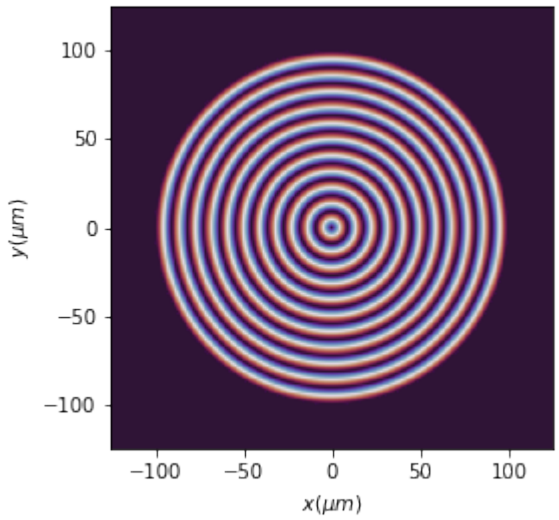
t1 = Scalar_mask_XY(x, y, wavelength)
t1.biprism_fresnel(r0=(0 * um, 0 * um), width=100 * um, height=5 * um, n=1.5)
t1.draw(kind='phase')
```



axicon

[32]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

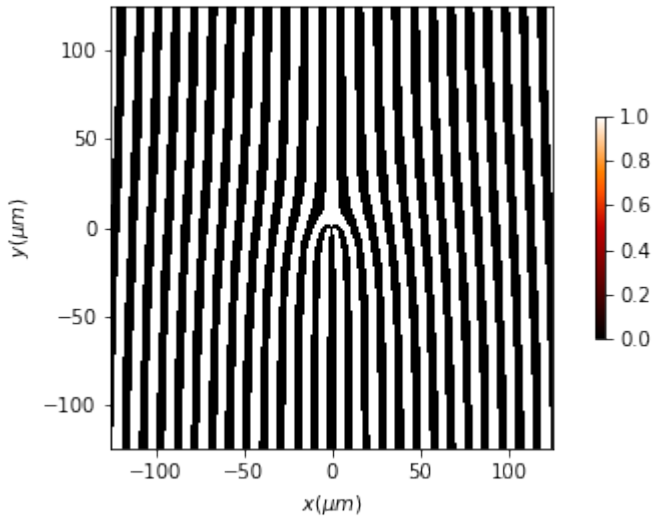
t1 = Scalar_mask_XY(x, y, wavelength)
t1.axicon(r0=(0 * um, 0 * um),
          radius=100 * um,
          angle=2 * degrees,
          refractive_index=1.5,
          off_axis_angle=0,
          reflective=True)
t1.draw(kind='phase')
```



grating_forked

[33]:
```
num_data = 1024
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

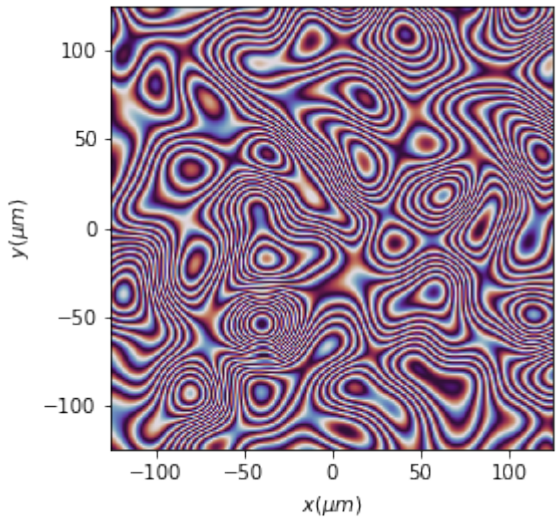
t = Scalar_mask_XY(x, y, wavelength)
t.forked_grating(r0=(0 * um, 0 * um),
                 period=10 * um,
                 l=3,
                 alpha=2,
                 kind='amplitude',
                 angle=0 * degrees)
t.draw(has_colorbar='vertical')
```



roughness

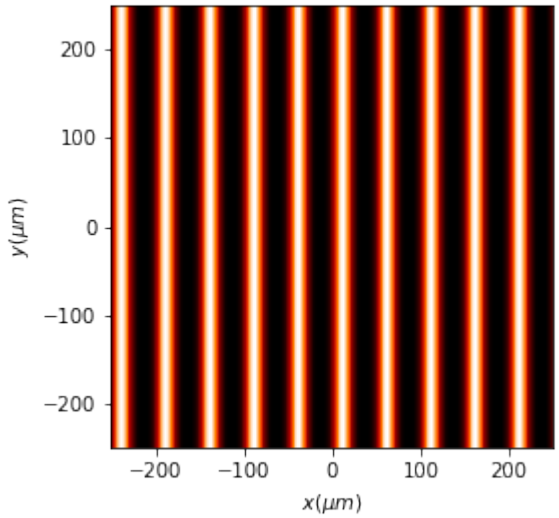
[34]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.roughness(t=(25 * um, 25 * um), s=1 * um)
t1.draw(kind='phase')
```



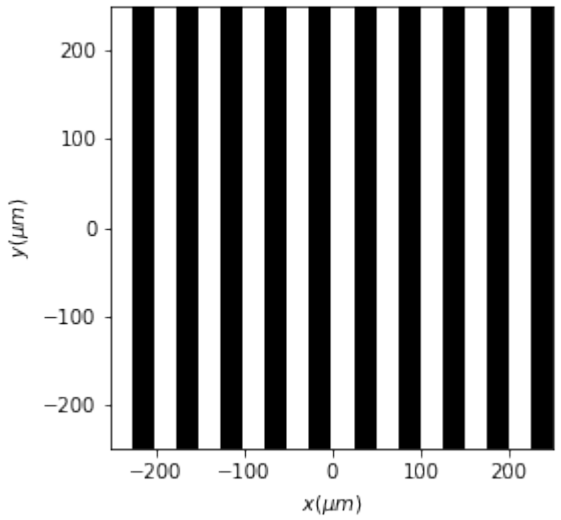
Sine grating

[35]:
```
x = np.linspace(-250 * um, 250 * um, 512)
y = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
period = 50 * um
red = Scalar_mask_XY(x, y, wavelength)
red.sine_grating(period=period,
                 amp_min=0,
                 amp_max=1,
                 x0=0 * um,
                 angle=0 * degrees)
red.draw(kind='intensity')
```



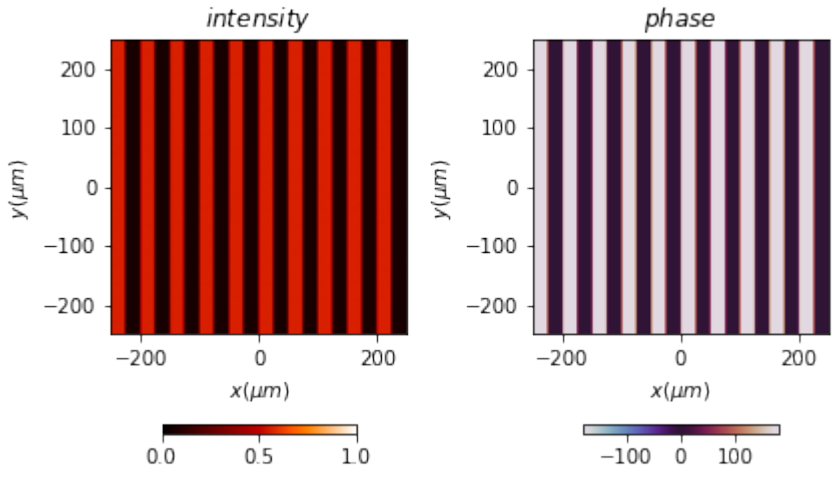
Ronchi grating

[36]:
```
x = np.linspace(-250 * um, 250 * um, 512)
y = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
red = Scalar_mask_XY(x, y, wavelength)
red.ronchi_grating(period=50 * um,
                   x0=0 * um,
                   angle=0 * degrees,
                   fill_factor=0.5)
red.draw(kind='intensity')
```



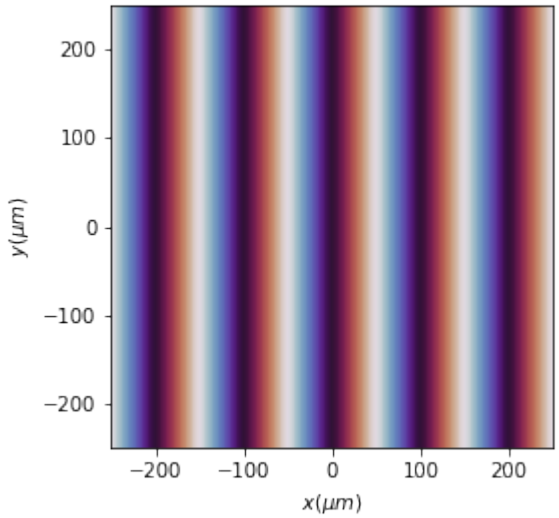
Binary grating (amplitude + phase)

[37]:
```
x = np.linspace(-250 * um, 250 * um, 512)
y = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
red = Scalar_mask_XY(x, y, wavelength)
red.binary_grating(period=50 * um,
                   amin=.25,
                   amax=.75,
                   phase=np.pi,
                   x0=0,
                   fill_factor=0.5,
                   angle=0 * degrees)
red.draw(kind='field')
```



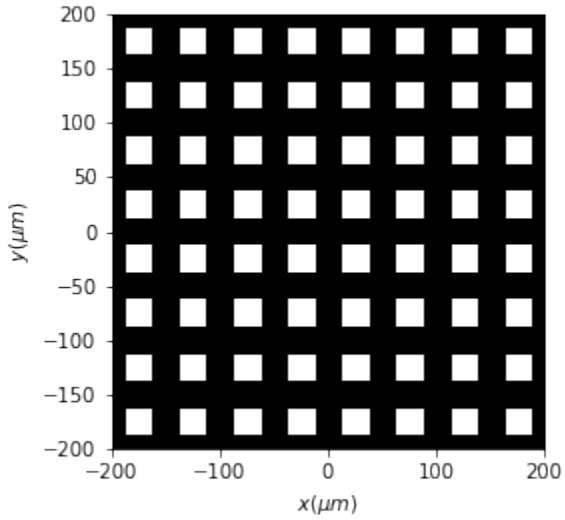
Blazed grating

[38]:
```
x = np.linspace(-250 * um, 250 * um, 512)
y = np.linspace(-250 * um, 250 * um, 512)
wavelength = 0.6238 * um
period = 100 * um
red = Scalar_mask_XY(x, y, wavelength)
red.blazed_grating(period=period,
                   phase_max=2*np.pi,
                   x0=0,
                   angle=0 * degrees)
red.draw(kind='phase')
```



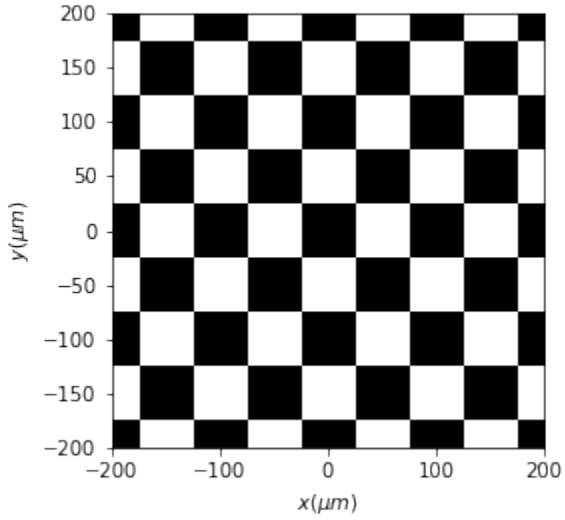
2D grating

[39]:
```
x = np.linspace(-200 * um, 200 * um, 512)
y = np.linspace(-200 * um, 200 * um, 512)
wavelength = 0.6238 * um
period = 50 * um
red = Scalar_mask_XY(x, y, wavelength)
red.grating_2D(r0=(0,0),
               period=period,
               amin=0,
               amax=1.,
               phase=0 * np.pi / 2,
               fill_factor=0.5,
               angle=0 * degrees)
red.draw(kind='intensity')
```



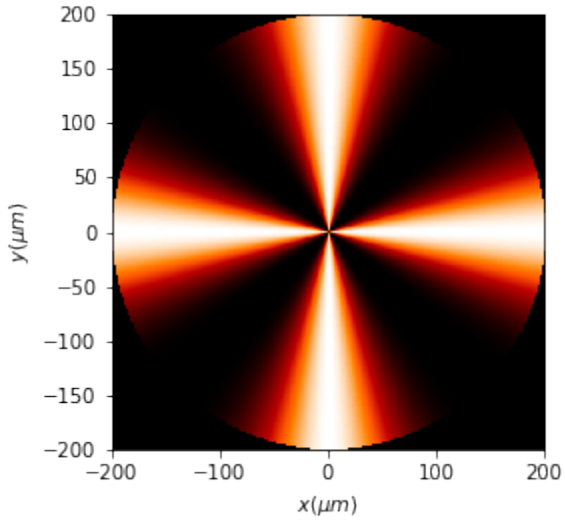
Chess grating

[40]:
```
x = np.linspace(-200 * um, 200 * um, 512)
y = np.linspace(-200 * um, 200 * um, 512)
wavelength = 0.6238 * um
period = 100 * um
red = Scalar_mask_XY(x, y, wavelength)
red.grating_2D_chess(r0=(period / 8, period / 4),
                     period=period,
                     amin=0,
                     amax=1.,
                     phase=0 * np.pi / 2,
                     fill_factor=0.5,
                     angle=(0 * degrees))
red.draw(kind='intensity')
```

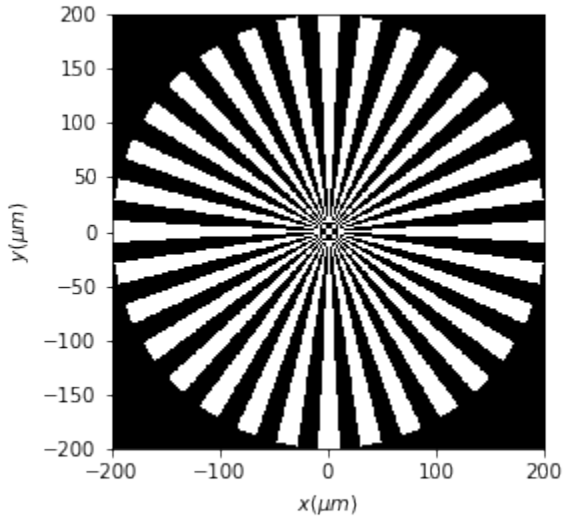


Angular grating

[41]:
```
x = np.linspace(-200 * um, 200 * um, 512)
y = np.linspace(-200 * um, 200 * um, 512)
wavelength = 0.6238 * um
period = 50 * um
red = Scalar_mask_XY(x, y, wavelength)
red.angular_grating(r0=(0,0), num_petals=4,  radius= 200*um, phase=0*degrees, is_
↪binary=False)
red.draw(kind='intensity')
```



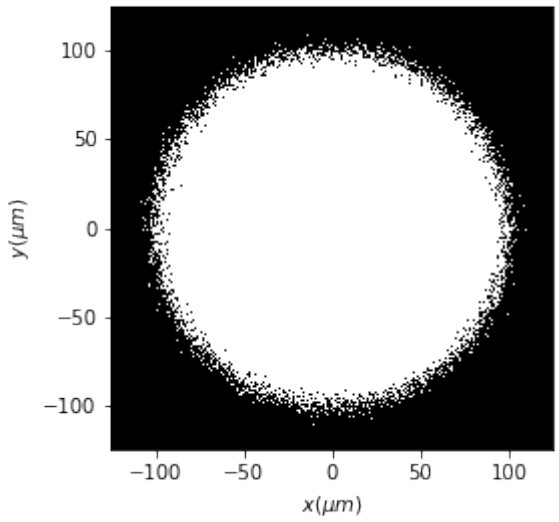
[42]:
```
x = np.linspace(-200 * um, 200 * um, 512)
y = np.linspace(-200 * um, 200 * um, 512)
wavelength = 0.6238 * um
period = 50 * um
red = Scalar_mask_XY(x, y, wavelength)
red.angular_grating(r0=(0, 0), num_petals=32, radius=200 * um, phase=0, is_binary=True)
red.draw(kind='intensity')
```



random circle edge

[43]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6238 * um

t = Scalar_mask_XY(x, y, wavelength)
t.circle_rough(r0=(0, 0), radius=100 * um, angle=0 * degrees, sigma=4 * um)
t.draw(kind='intensity')
```

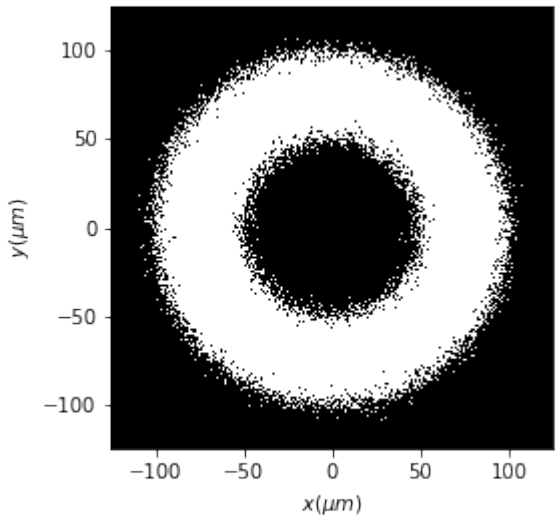


Random ring edge

[44]:
```
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)

wavelength = 0.6238 * um

t = Scalar_mask_XY(x, y, wavelength)
t.ring_rough(r0=(0, 0),
             radius1=50 * um,
             radius2=100 * um,
             angle=0 * degrees,
             sigma=4 * um)
t.draw(kind='intensity')
```



6.5.5 Other procedures to generate XY masks

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.utils_drawing import draw_several_fields
```

6.5.5.1 Adding masks

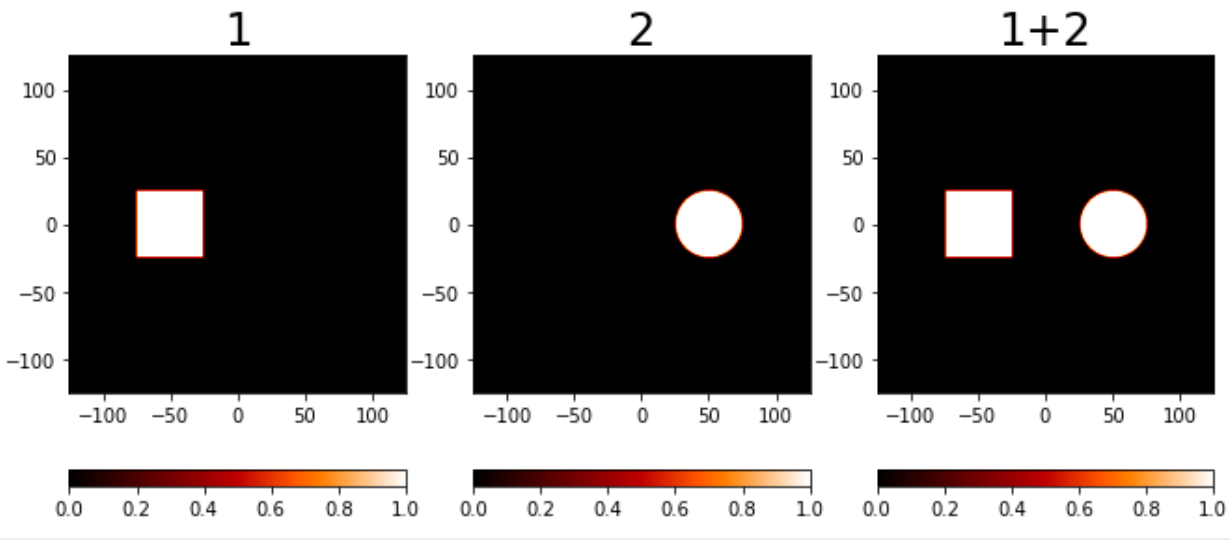
[2]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.square(r0=(-50 * um, 0 * um), size=(50 * um, 50 * um), angle=0 * degrees)

t2 = Scalar_mask_XY(x, y, wavelength)
t2.circle(r0=(50 * um, 0 * um), radius=(25 * um, 25 * um), angle=0 * degrees)

t3 = t2 + t1

draw_several_fields([t1, t2, t3], titles=['1', '2', '1+2'])
```



[3]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.square(r0=(0 * um, 0 * um), size=(100 * um, 100 * um), angle=0 * degrees)

t2 = Scalar_mask_XY(x, y, wavelength)
t2.circle(r0=(80 * um, 0 * um), radius=(40 * um, 40 * um), angle=0 * degrees)

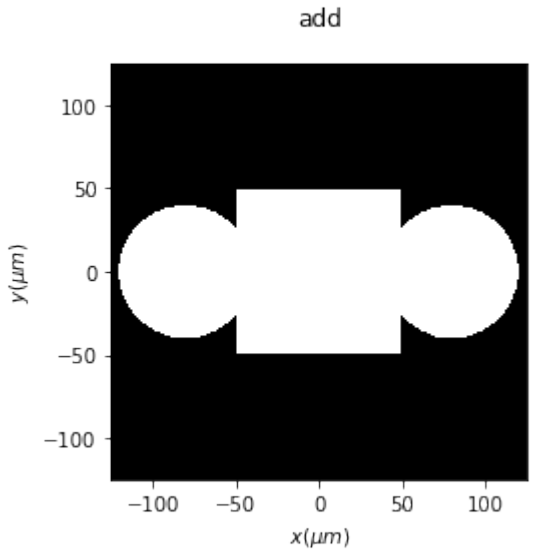
t3 = Scalar_mask_XY(x, y, wavelength)
t3.circle(r0=(-80 * um, 0 * um), radius=(40 * um, 40 * um), angle=0 * degrees)

t4 = t2.add(t1, 'maximum1')
```

(continues on next page)

(continued from previous page)

```
t5 = t4.add(t3, 'maximum1')
t5.draw(title='add')
```



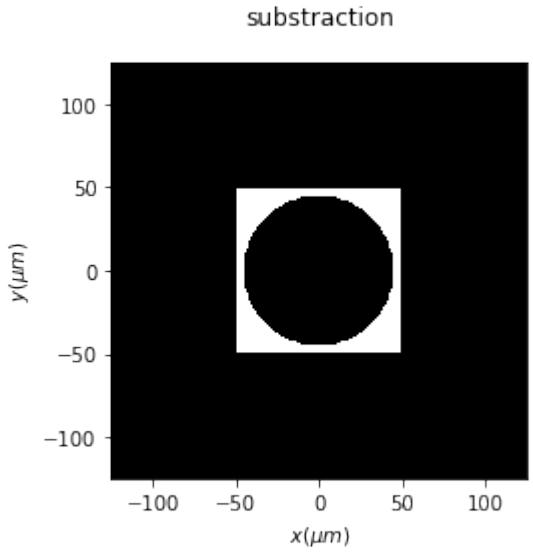
6.5.5.2 Substracting masks

[4]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.square(r0=(0 * um, 0 * um), size=(100 * um, 100 * um), angle=0 * degrees)

t2 = Scalar_mask_XY(x, y, wavelength)
t2.circle(r0=(0 * um, 0 * um), radius=(45 * um, 45 * um), angle=0 * degrees)

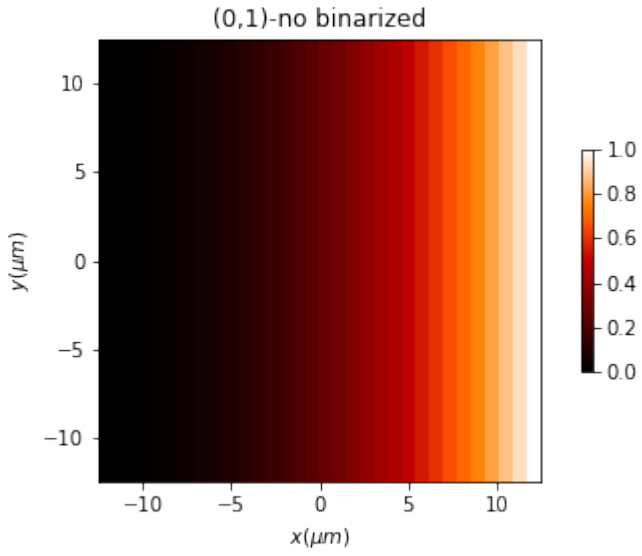
t3 = t2 - t1
t3.draw(title='substraction')
```



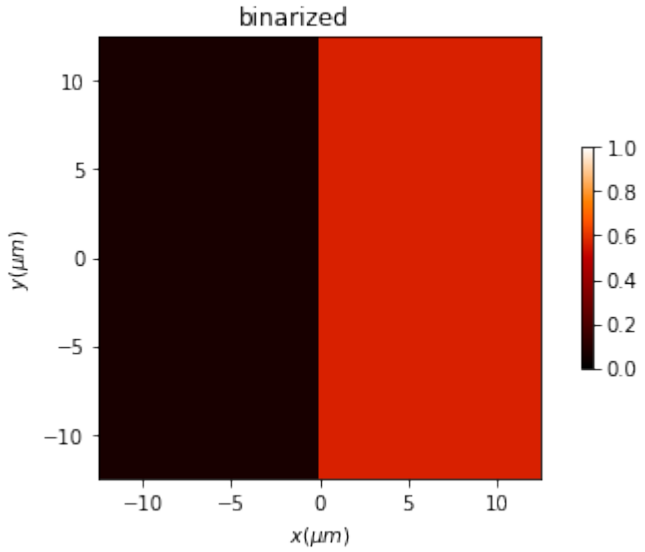
6.5.5.3 Binarization of a previous mask

[5]:
```
num_data = 512
length = 25 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.gray_scale(num_levels=32, levelMin=0, levelMax=1)
t1.draw(kind='intensity', has_colorbar='vertical')
plt.title('(0,1)-no binarized')
```



[6]:
```
t1.binarize(kind='amplitude',
            bin_level=None,
            level0=0.25,
            level1=.75,
            new_field=False,
            matrix=False)
t1.draw(kind='intensity', has_colorbar='vertical')
plt.suptitle('binarized')
```



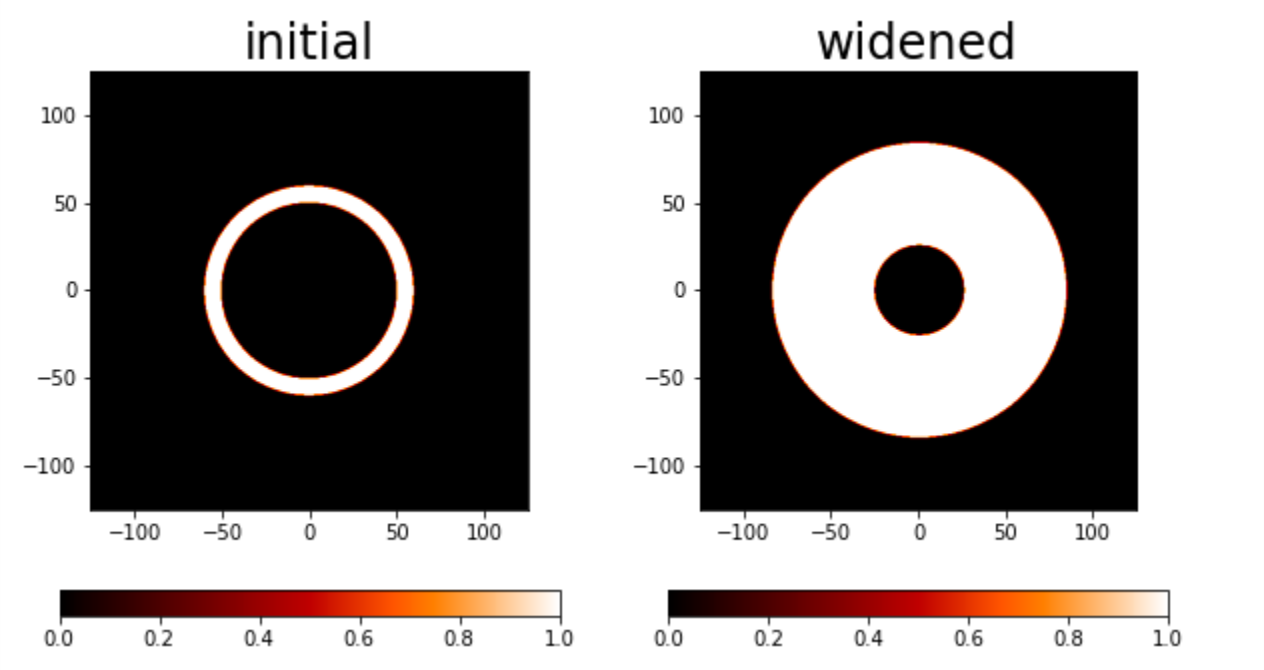
6.5.5.4 Widen the shape of a previous mask

[7]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.ring(r0=(0 * um, 0 * um),
        radius1=(50 * um, 50 * um),
        radius2=(60 * um, 60 * um),
        angle=45 * degrees)

t2 = t1.widen(radius=25 * um, new_field=True, binarize=True)

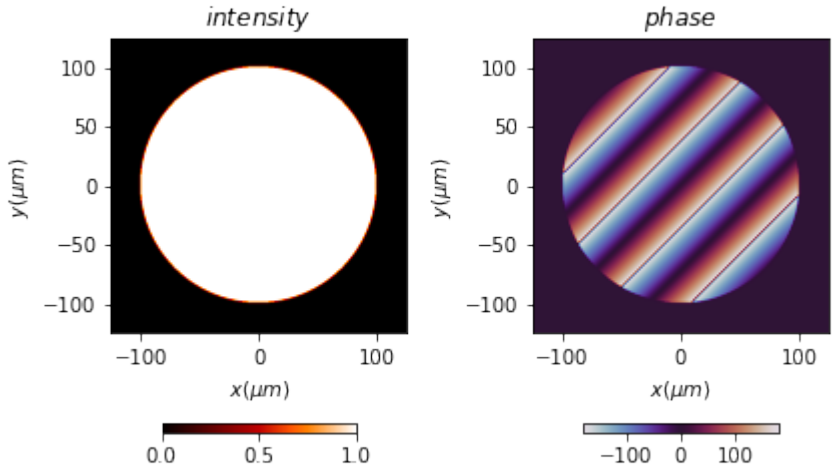
draw_several_fields((t1, t2), titles=('initial', 'widened'))
```



6.5.5.5 Mask from surface functions

[8]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

t1 = Scalar_mask_XY(x, y, wavelength)
t1.mask_from_function(r0=(0 * um, 0 * um),
                      index=1.5,
                      f1='1*degrees*self.Y',
                      f2='1*degrees*self.X',
                      v_globals=None,
                      radius=(100 * um, 100 * um))
t1.draw(kind='field')
```

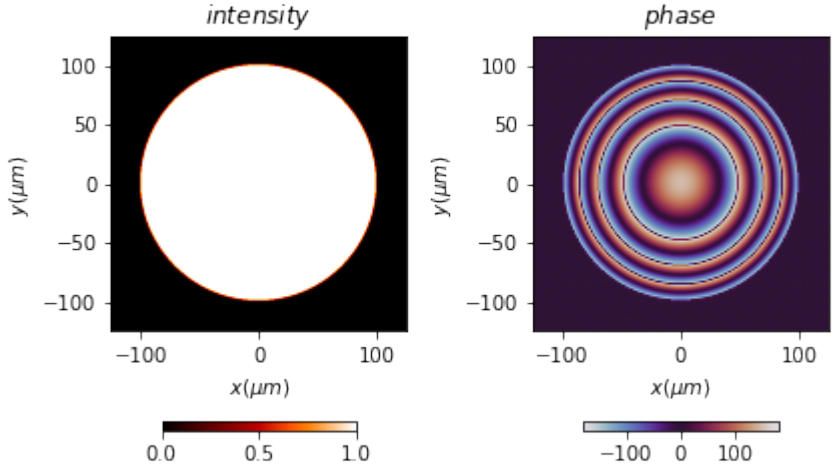


[9]:
```
num_data = 512
length = 250 * um
x = np.linspace(-length / 2, length / 2, num_data)
y = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328 * um

f1 = "np.zeros_like(self.X,dtype=float)"
f2 = "R2-h2+np.sqrt(R2**2-(self.X-x0)**2-(self.Y-y0)**2)"
v_globals = {
    'R1': 5 * mm,
    'R2': 1 * mm,
    'x0': 0 * um,
    'y0': 0 * um,
    'h1': 1 * mm,
    'h2': -1 * mm,
    'np': np,
}
index = 1.5

t1 = Scalar_mask_XY(x, y, wavelength)
t1.mask_from_function(r0=(0 * um, 0 * um),
                      index=index,
                      f1=f1,
                      f2=f2,
                      v_globals=v_globals,
                      radius=(100 * um, 100 * um))
t1.draw(kind='field')

focal = v_globals['R2'] / (index - 1)
```



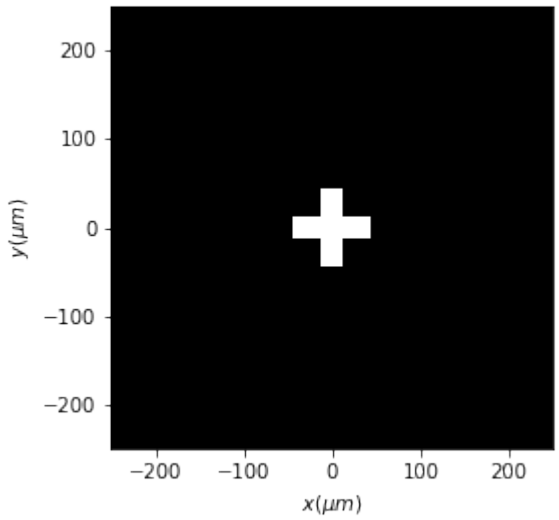
6.5.5.6 Insert_array_masks

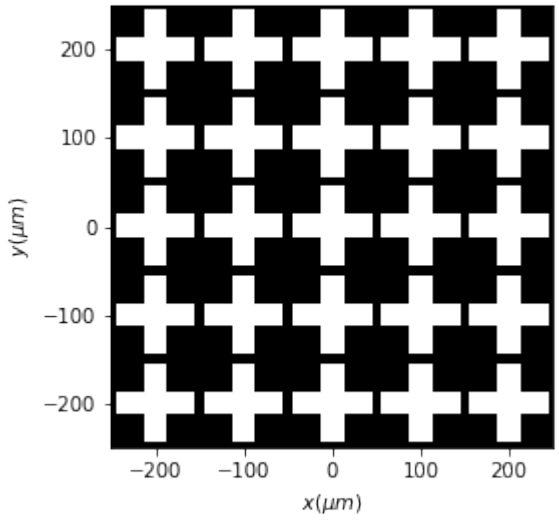
The mask is repited in x-y axis separated a certain distance

[10]:
```
x = np.linspace(-250 * um, 250 * um, 512)
y = np.linspace(-250 * um, 250 * um, 512)
wavelength = 1 * um

s = Scalar_mask_XY(x, y, wavelength)
s.cross(r0=(0 * um, 0 * um), size=(90 * um, 25 * um), angle=0 * degrees)
s.draw()

t1 = Scalar_mask_XY(x, y, wavelength)
num_points = t1.insert_array_masks(t1=s,
                                   space=[100 * um, 100 * um],
                                   margin=50 * um,
                                   angle=90 * degrees)
t1.draw(kind='intensity')
```





[11]:
```
print(s)
```

```
Scalar_mask_XY
 - x:  (512,),   y:  (512,),   u:  (512, 512)
 - xmin:       -250.00 um,  xmax:      250.00 um,  Dx:   0.98 um
 - ymin:       -250.00 um,  ymax:      250.00 um,  Dy:   0.98 um
 - Imin:       0.00,     Imax:      1.00
 - phase_min:  0.00 deg, phase_max: 0.00 deg
 - wavelength: 1.00 um
 - date:       2023-12-22_17_17_30
```

[12]:
```
print(t1)
```

```
Scalar_mask_XY
 - x:  (512,),   y:  (512,),   u:  (512, 512)
 - xmin:       -250.00 um,  xmax:      250.00 um,  Dx:   0.98 um
 - ymin:       -250.00 um,  ymax:      250.00 um,  Dy:   0.98 um
```

(continues on next page)

(continued from previous page)

```
- Imin:       0.00,     Imax:      1.00
- phase_min:  0.00 deg, phase_max: 180.00 deg
- wavelength: 1.00 um
- date:       2023-12-22_17_17_31
```

6.5.5.7 repeat_structure

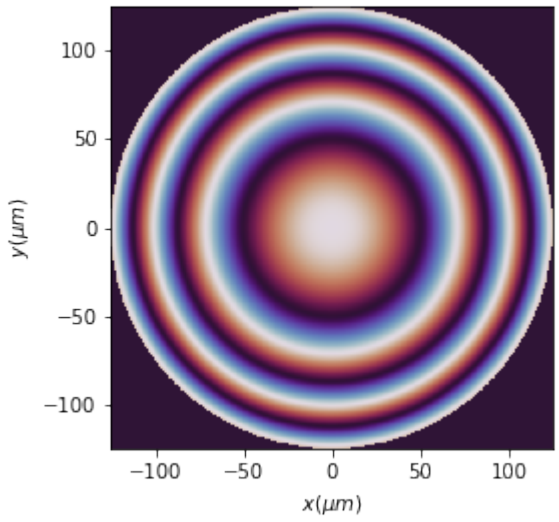
The mask si repeated (n,m) times. The size of the new mask is (n,m) times greater.

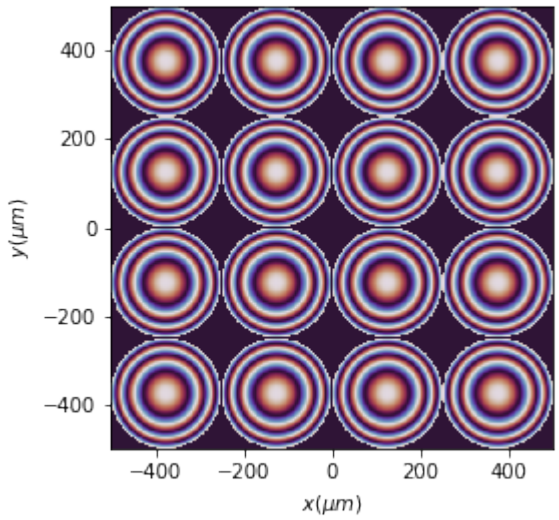
[13]:
```
size = 250 * um
x0 = np.linspace(-size / 2, size / 2, 512)
y0 = np.linspace(-size / 2, size / 2, 512)

wavelength = 0.6328 * um

focal = 4 * mm
lenslet = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
lenslet.lens(r0=(0, 0), radius=size / 2, focal=focal)
lenslet.draw('phase')

lens_array = lenslet.repeat_structure(num_repetitions=(4, 4),
                                      position='center',
                                      new_field=True)
lens_array.draw('phase')
```





[14]:
```
print(lenslet)
```

```
Scalar_mask_XY
 - x:  (512,),   y:  (512,),   u:  (512, 512)
 - xmin:       -125.00 um,  xmax:      125.00 um,  Dx:   0.49 um
 - ymin:       -125.00 um,  ymax:      125.00 um,  Dy:   0.49 um
 - Imin:       0.00,     Imax:      1.00
 - phase_min:  -179.93 deg, phase_max: 180.00 deg
 - wavelength: 0.63 um
 - date:       2023-12-22_17_17_32
```

[15]:
```
print(lens_array)
```

```
Scalar_mask_XY
 - x:  (2048,),   y:  (2048,),   u:  (2048, 2048)
 - xmin:       -500.00 um,  xmax:      500.00 um,  Dx:   0.49 um
 - ymin:       -500.00 um,  ymax:      500.00 um,  Dy:   0.49 um
 - Imin:       0.00,     Imax:      1.00
 - phase_min:  -179.93 deg, phase_max: 180.00 deg
 - wavelength: 0.63 um
 - date:       2023-12-22_17_17_32
```

6.6 Scalar_XYZ

6.6.1 Characteristics

Scalar_XYZ is a set of two modules for:

- Generation of 3D masks and diffractive optical elements.
- Propagation of light, determination of parameters, and other functions.
- Drawing sources, masks and fields

For light generation, scalar_sources_XY.py is used.

Warning: This module is not very mature yet.

These modules are named: scalar_fields_XYZ.py,and scalar_masks_XYZ.py.

Each module present a main class:

- Scalar_field_XYZ
- Scalar_masks_XYZ

The main attributes for these classes are the following:

- self.x (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n .
- self.y (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n .
- self.z (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n .
- self.wavelength (float): wavelength of the incident field.
- self.u (numpy.array): equal size than x * y * z. complex field.

We can also find these atributes:

- self.X (numpy.array): linear 2D array with size x * y * z storing X position.
- self.Y (numpy.array): linear 2D array with size x * y * z storing Y position.
- self.Z (numpy.array): linear 2D array with size x * y * z storing Z position.
- self.quality (float): quality of RS algorithm. Valid for values > 1.
- self.info (str): description of data.
- self.fast (bool): If True, Rayleigh-Sommerfeld computations are performed using Fresnel approximation.
- self.type (str): Class of the field.
- self.date (str): date when performed.

The dimensional magnitudes are related to microns: `micron = 1.`

[2]:
```
from diffractio import degrees, eps, mm, no_date, np, plt, um
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
from diffractio.scalar_sources_XY import Scalar_source_XY
```

6.6.1.1 save_load

[3]:
```
length = 100 * um
numdata = 16  # 256
x0 = np.linspace(-length / 2, length / 2, numdata)
y0 = np.linspace(-length / 2, length / 2, numdata)
z0 = np.linspace(-length / 2, length / 2, numdata)
wavelength = 0.5 * um
filename = 'save_load_xyz.npz'

t1 = Scalar_field_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
```

(continues on next page)

(continued from previous page)

```
t1.u = np.ones_like(t1.u)
t1.save_data(filename=filename, add_name='')
```

[4]:
```
t2 = Scalar_field_XYZ(x=None, y=None, z=None, wavelength=None)
t2.load_data(filename=filename, verbose=False)
```

6.6.1.2 clear_field

[5]:
```
length = 100 * um
numdata = 32  # 256
x0 = np.linspace(-length / 2, length / 2, numdata)
y0 = np.linspace(-length / 2, length / 2, numdata)
z0 = np.linspace(-length / 2, length / 2, numdata)
wavelength = 0.5 * um

u0 = Scalar_field_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
u0.u = np.ones_like(u0.u)
print(u0.u.max())

u0.clear_field()
print(u0.u.max())
```

```
(1+0j)
0j
```

6.6.1.3 show_index_refraccion

[5]:
```
length = 150 * um

x0 = np.linspace(-length / 2, length / 2, 128)
y0 = np.linspace(-length / 2, length / 2, 128)
z0 = np.linspace(-length / 2, length / 2, 128)
wavelength = 0.55 * um

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.circle(r0=(0 * um, 0 * um), radius=(20 * um, 20 * um), angle=0 * degrees)

uxyz = Scalar_mask_XYZ(x=x0,
                       y=y0,
                       z=z0,
                       wavelength=wavelength,
                       n_background=1.,
                       info='')
uxyz.incident_field(t1)
uxyz.cylinder(r0=(0 * um, 0 * um, 0),
              radius=(20 * um, 20 * um),
              length=75 * um,
              refractive_index=1.5,
```

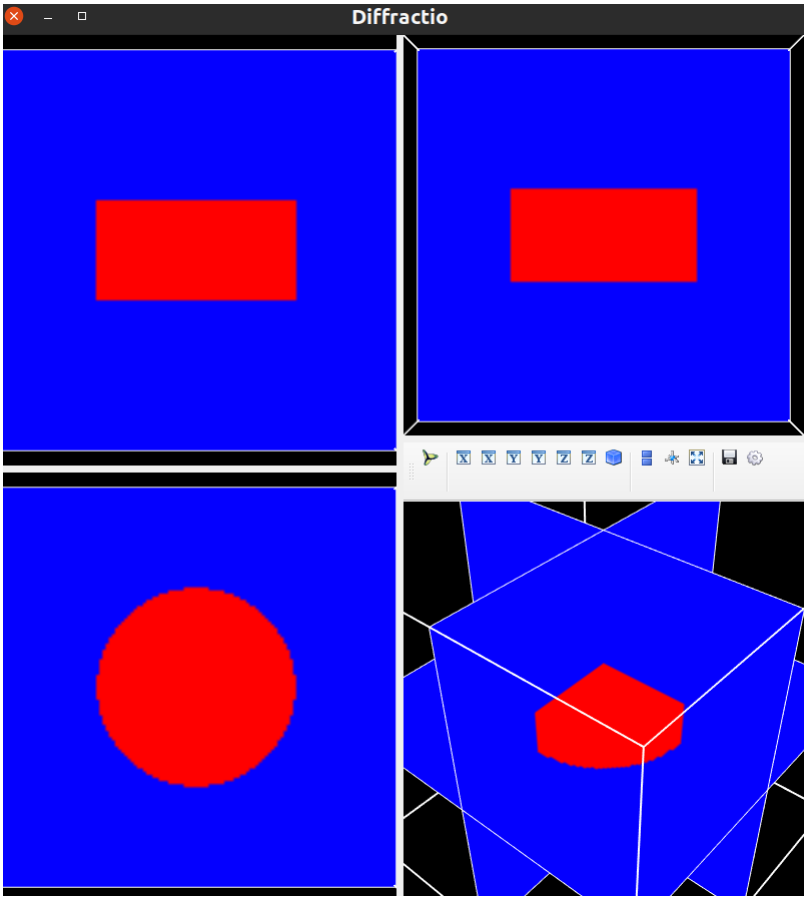
(continues on next page)

(continued from previous page)

```
                axis=(0, 0, 0),
                angle=0 * degrees)
```

[ ]:
```
uxyz.draw_volume()
```

[ ]:
```
uxyz.draw_refractive_index()
```



6.6.1.4 RS propagation of a circular aperture

[7]:
```
length = 100 * um
numdata = 128
x0 = np.linspace(-length / 2, length / 2, numdata)
y0 = np.linspace(-length / 2, length / 2, numdata)
z0 = np.linspace(200 * um, 2 * mm, 128)
wavelength = 0.5 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.gauss_beam(A=1, r0=(0, 0), z0=0, w0=(150 * um, 150 * um))

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
```

(continues on next page)

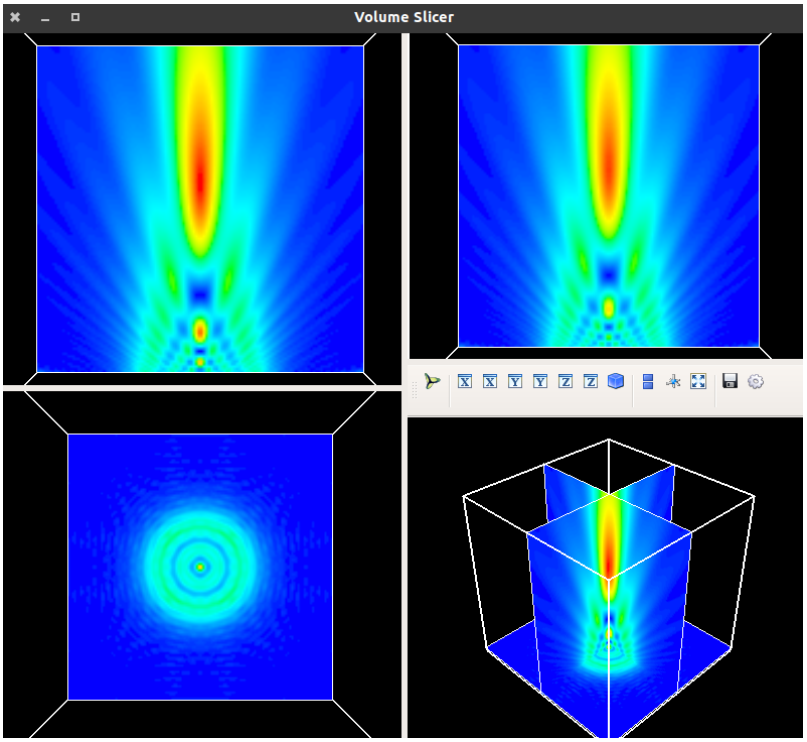
(continued from previous page)

```
t1.circle(r0=(0, 0), radius=(25 * um, 25 * um))
t3 = u1 * t1
```

[8]:
```
uxyz = Scalar_field_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
uxyz.incident_field(t3)
uxyz.RS()
```

```
time in RS= 8.317139387130737. num proc= 6
```

[ ]:
```
uxyz.draw_XYZ(logarithm=False, normalize='maximum')
```



6.6.1.5 Rayleigh-Sommerfeld propagation of a gauss beam passing through a Grating

[9]:
```
length = 500 * um
x0 = np.linspace(-length / 2, length / 2, 128)
y0 = np.linspace(-length / 2, length / 2, 128)
z0 = np.linspace(2 * mm, 7 * mm, 128)
wavelength = 0.6328 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.gauss_beam(A=1, r0=(0, 0), z0=0, w0=(150 * um, 150 * um))

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.ronchi_grating(x0=0 * um, period=20 * um, angle=0 * degrees)
t2 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t2.lens(r0=(0 * um, 0 * um),
```

(continues on next page)

(continued from previous page)

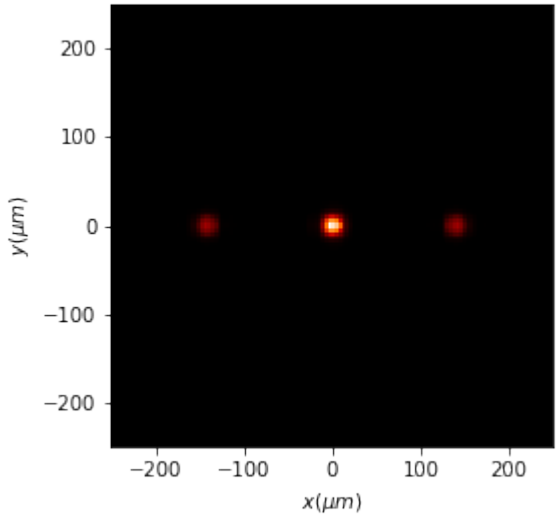
```
        radius=(200 * um, 200 * um),
        focal=(5 * mm, 5 * mm),
        angle=0 * degrees)

t3 = u1 * t1 * t2
```

[10]:
```
uxyz = Scalar_field_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
uxyz.incident_field(t3)
uxyz.RS()
uxyz.normalize()
```

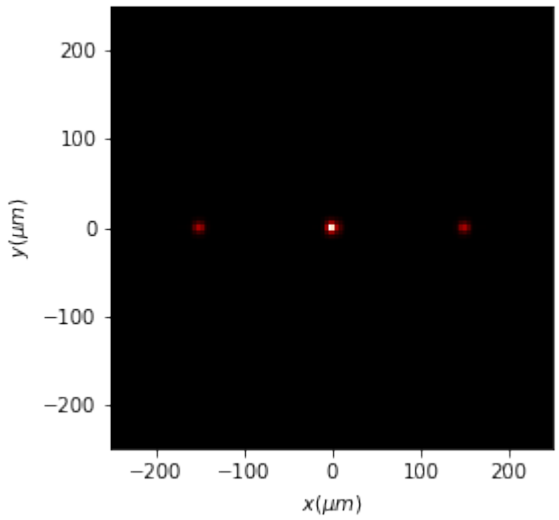
```
time in RS= 9.478154182434082. num proc= 6
```

[11]:
```
uxyz.draw_XY(z0=4.5 * mm)
```



A particular plane, can be passed to scalar_field_XY, and then, it can be drawn.

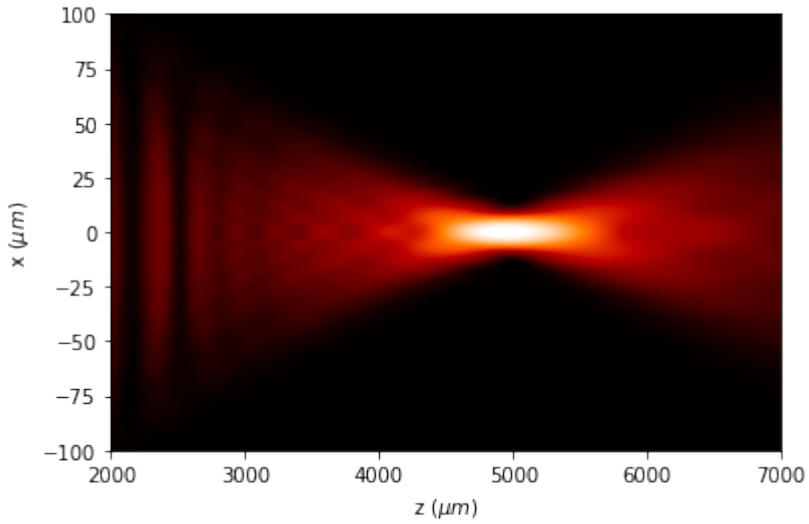
[12]:
```
u_xy = uxyz.to_Scalar_field_XY(z0=4.75 * mm, is_class=True, matrix=False)
u_xy.draw(kind='intensity')
```



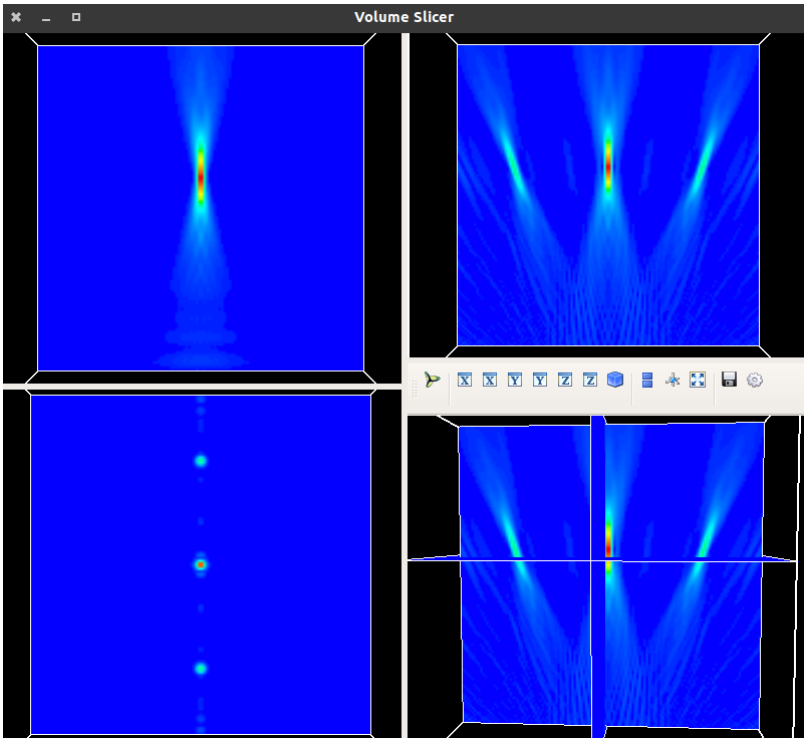
Also, the intensity at the XZ plane can be obtained.

[13]:
```
uxyz.draw_XZ(y0=0 * mm, logarithm=1e2)
plt.ylim(-100, 100)
```

```
<Figure size 432x288 with 0 Axes>
```



[ ]:
```
uxyz.draw_XYZ(logarithm=False, normalize='maximum')
```



6.6.1.6 Video from a XYZ distribution

[6]:
```
length = 100 * um
numdata = 128  # 256
x0 = np.linspace(-length / 2, length / 2, numdata)
y0 = np.linspace(-length / 2, length / 2, numdata)
z0 = np.linspace(200 * um, 2 * mm, 32)  #128
wavelength = 0.5 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.gauss_beam(A=1, r0=(0, 0), z0=0, w0=(150 * um, 150 * um))

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.circle(r0=(0, 0), radius=(25 * um, 25 * um))
t3 = u1 * t1
```

[7]:
```
uxyz = Scalar_field_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)
uxyz.incident_field(t3)
uxyz.RS()
uxyz.normalize()
```

```
time in RS= 1.6388328075408936. num proc= 6
```

[8]:
```
filename = "video_RS"
```

[9]:
```
uxyz.video(filename=filename + '_int.mp4', kind='intensity', frame=False)
```

[11]:
```
%%HTML
<div align="middle">
      <video width="50%" controls>
            <source src="video_RS_int.mp4" type="video/mp4">
      </video>
</div>
```

```
<IPython.core.display.HTML object>
```

6.6.1.7 benchmark_RS_multiprocessing

[ ]:
```
x0 = np.linspace(-25 * um, 25 * um, 128)
y0 = np.linspace(-25 * um, 25 * um, 128)
z0 = np.linspace(100 * um, 500 * um, 128)
wavelength = 0.55 * um

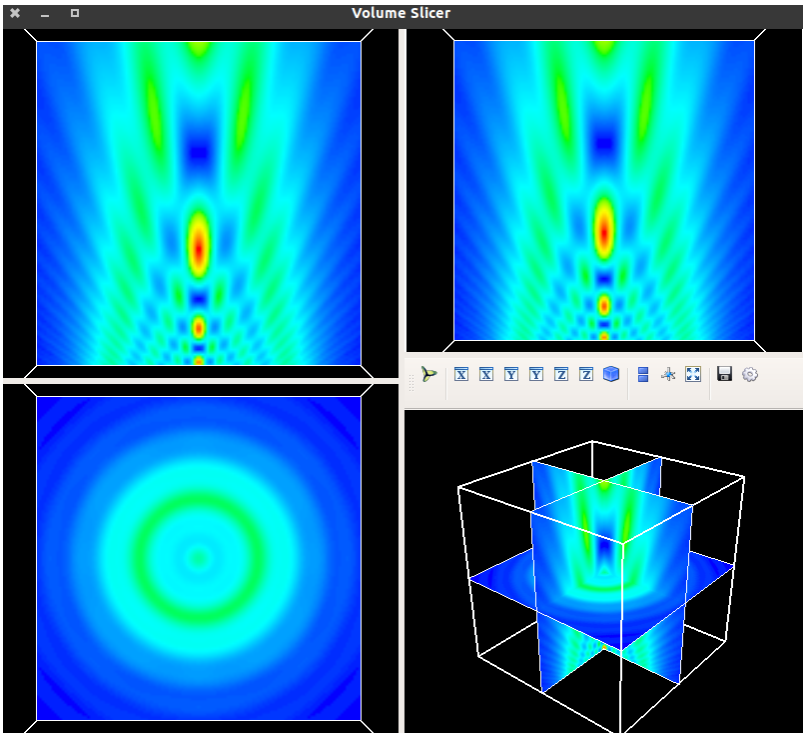
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.circle(r0=(0 * um, 0 * um), radius=(20 * um, 20 * um), angle=0 * degrees)

uxyz = Scalar_mask_XYZ(x=x0,
                       y=y0,
                       z=z0,
                       wavelength=wavelength,
                       n_background=1.,
                       info='')
uxyz.incident_field(t1)
uxyz.sphere(r0=(0 * um, 0 * um, 0 * um),
            radius=(10 * um, 30 * um, 50 * um),
            refractive_index=2,
            angles=(0 * degrees, 0 * degrees, 45 * degrees))
uxyz.u0 = t1
```

[ ]:
```
time = uxyz.RS(num_processors=1, verbose=False)
print("time in RS_multiprocessing {}: {} seconds".format(1, time))
```

[ ]:
```
time = uxyz.RS(num_processors=4, verbose=False)
print("time in RS_multiprocessing {}: {} seconds".format(4, time))
```

[ ]:
```
uxyz.draw_XYZ()
```



[ ]:

6.6.2 Wave Propagation Method and BPM in 3D

WPM method is very fast. It is based on S. Schmidt et al., "Wave-optical modeling beyond the thin-element-approximation," Opt. Express, vol. 24, no. 26, p. 30188, 2016.

[1]:
```
from diffractio import np, plt, sp, um, mm, degrees
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ

from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
```

6.6.2.1 Propagation at vacuum

[2]:
```
x = np.linspace(-50 * um, 50 * um, 256)
y = np.linspace(-50 * um, 50 * um, 256)
z = np.linspace(0, 500 * um, 256)
wavelength = .6 * um
```

[3]:
```
t0 = Scalar_mask_XY(x, y, wavelength)
t0.circle(r0=(0, 0), radius=45 * um, angle=0)
```

[4]:
```
u = Scalar_mask_XYZ(x, y, z, wavelength)
u.incident_field(u0=t0)
```

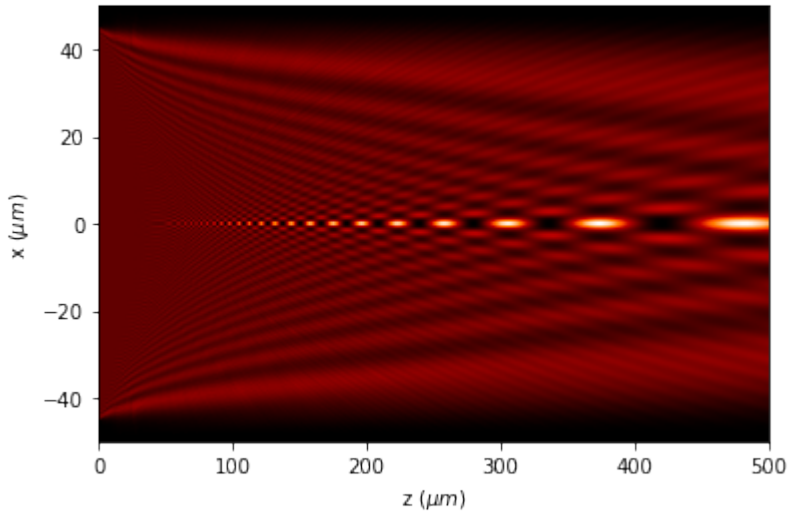
6.6.2.2 WPM

[5]:
```
%%time
u.clear_field()
u.WPM(verbose=True, has_edges=True)
```

```
Time = 6.44 s, time/loop = 25.17 ms
CPU times: user 5.88 s, sys: 730 ms, total: 6.61 s
Wall time: 6.58 s
```

[6]:
```
u.draw_XZ(y0=0, logarithm=False)
```

```
<Figure size 432x288 with 0 Axes>
```

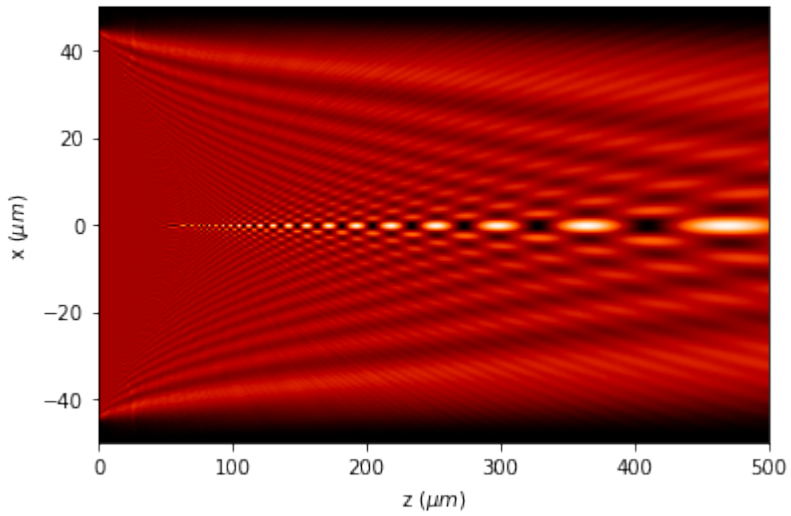


6.6.2.3 BPM

[9]:
```
%%time
u.clear_field()
u.BPM(verbose=True, has_edges=True)
u.draw_XZ(y0=0, logarithm=True);
```

```
CPU times: user 2.61 s, sys: 120 ms, total: 2.73 s
Wall time: 2.72 s
```

```
<Figure size 432x288 with 0 Axes>
```



At vacuum both techniques work fine.

6.6.2.4 Diffraction by an sphere

WPM and BPM also allow propagation through a XYZ refraction index structure.

[11]:
```
x = np.linspace(-15 * um, 15 * um, 256)
y = np.linspace(-15 * um, 15 * um, 256)
z = np.linspace(0, 30 * um, 256)
wavelength = 0.6328 * um
```

[12]:
```
t0 = Scalar_mask_XY(x, y, wavelength)
t0.circle(r0=(0, 0), radius=12.5 * um, angle=0)

u0 = Scalar_source_XY(x, y, wavelength)
u0.plane_wave(A=1)
```

[13]:
```
u = Scalar_mask_XYZ(x, y, z, wavelength)
u.sphere(r0=(0, 0, 12.5), radius=10 * um, refractive_index=2, angles=(0, 0, 0))
```

[14]:
```
u.incident_field(u0=u0 * t0)
```

[15]:
```
# u.draw_refractive_index()
```

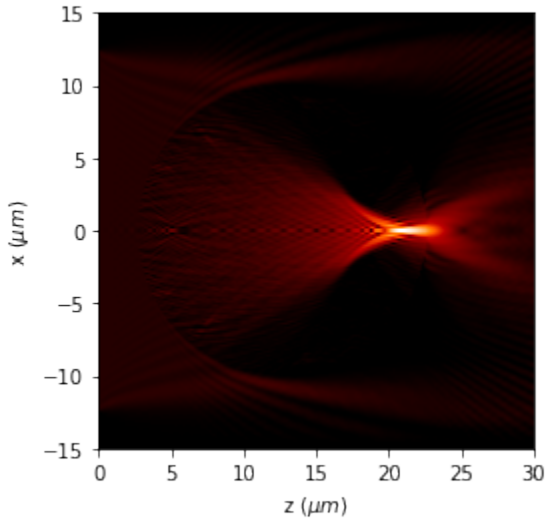
WPM

[16]:
```
%%time
u.clear_field()
u.WPM(verbose=True,has_edges=True)
```

```
Time = 7.79 s, time/loop = 30.42 ms
CPU times: user 7.86 s, sys: 117 ms, total: 7.97 s
Wall time: 7.93 s
```

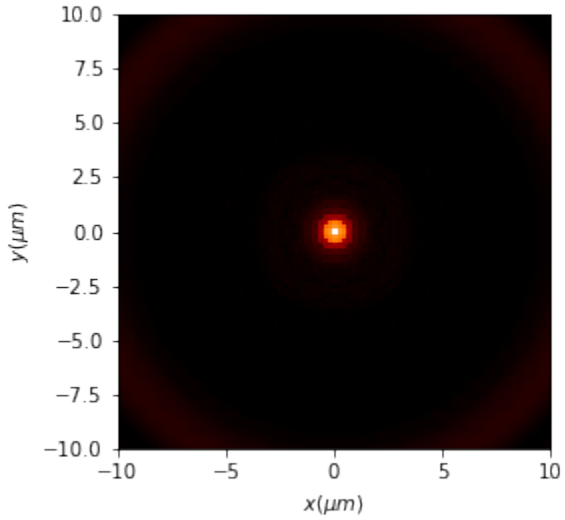
[17]:
```
u.draw_XZ(y0=0, logarithm=True, scale='scaled')
```

```
<Figure size 432x288 with 0 Axes>
```



[18]:
```
u2 = u.cut_resample([-10, 10], [-10, 10],
                    num_points=(128, 128, 128),
                    new_field=True)
```

[19]:
```
u2.draw_XY(z0=20.5, logarithm=True)
```



[20]:
```
# u.draw_XYZ(logarithm=True)
```

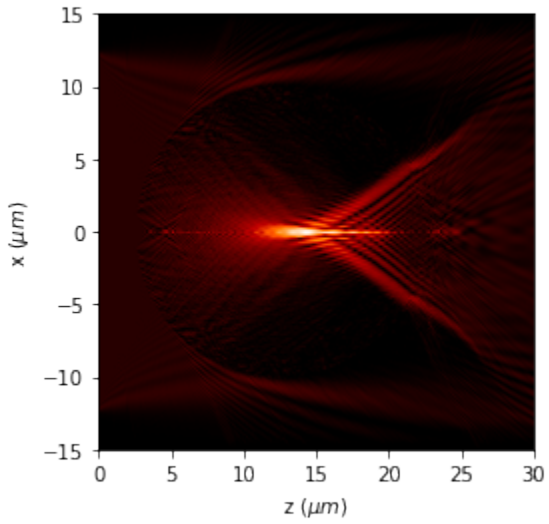
BPM

[21]:
```
%%time
u.clear_field()
u.BPM(verbose=True,has_edges=True)
```

```
CPU times: user 2.8 s, sys: 140 ms, total: 2.94 s
Wall time: 2.98 s
```

[22]:
```
u.draw_XZ(y0=0, scale='scaled', logarithm=True)
```

```
<Figure size 432x288 with 0 Axes>
```



In this case, the results are quite different. As the difference of refraction indexes is high, WPM produces better results.

6.6.3 WPM 3D with storing intermediate data

3D drawings are very nice, but sometimes we have not enough memory to store data. In this case we can use the following functions at Scalar_fields_XY scheme: WPM and BPM.

It is quite easy to execute them at vacuum, as only a XY initial field is required. However it is also possible to use a 3d refraction index structure. In order to not storing it, a function with the XYZ definition is required. However, the data are only extracted for each z position, and then no problems with storage are produced.

In my compute I have got the following results: - 128 x 128 x 128 = 600 ms - 256 x 256 x 256 = 5 seconds - 512 x 512 x 512 = 44 seconds (I could not resolve it in XYZ). - 1024 x 1024 x 1024 = 7 minutes (I could not resolve it in XYZ).

6.6.3.1 Modules

[1]:
```
from diffractio import np, plt, sp, um, mm, degrees
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ

from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ

from diffractio.utils_math import get_k, ndgrid_deprecated, nearest
from diffractio.scalar_fields_XY import WPM_schmidt_kernel

import time
import sys
```

[2]:
```
num_z = 256

x = np.linspace(-50 * um, 50 * um, 256)
y = np.linspace(-50 * um, 50 * um, 256)
z = np.linspace(0, 70 * um, num_z)
z_obs = z[-1]
wavelength = 1 * um
```

[3]:
```
t0 = Scalar_mask_XY(x, y, wavelength)
t0.circle(r0=(0, 0), radius=45 * um, angle=0)

u0 = Scalar_source_XY(x, y, wavelength)
u0.gauss_beam(r0=(0, 0), w0=45 * um, z0=0)

u_iter = u0 * t0
```

[4]:
```
def fn(x, y, z, wavelegth):
    u = Scalar_mask_XYZ(x, y, z, wavelength)
    u.sphere(r0=(0, 0, 25),
             radius=20 * um,
             refractive_index=2,
             angles=(0, 0, 0))
    index = u.n
```

(continues on next page)

(continued from previous page)

```
    del u
    return np.squeeze(index)
```

[5]:
```
# Example of structure at a given k-position in z distance

refractive_index = fn(x, y, np.array([
    44,
]), wavelength)
```

[6]:
```
zs = np.linspace(z[0], z[-1], num_z)
```

[7]:
```
%%time
u_iter.WPM(fn, zs =zs, has_edges=True, matrix=False, verbose=True)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
File <timed eval>:1, in <module>

File ~/bitbucket/diffractio/diffractio/scalar_fields_XY.py:1201, in Scalar_field_XY.
↪WPM(self, fn, zs, num_sampling, ROI, r_pos, z_pos, get_u_max, has_edges, pow_edge,␣
↪matrix, verbose)
   1198 # STORING global view matrices
   1199 if num_sampling is not None:
-> 1201     len_x, len_y, len_z = num_sampling
   1202     xout_gv = np.linspace(x[0], x[-1], len_x)
   1203     yout_gv = np.linspace(y[0], y[-1], len_y)

ValueError: not enough values to unpack (expected 3, got 2)
```

[ ]:
```
```

6.6.3.2 Comparison with 3D

We perform the same experiment, but storing data. Finally, a comparison between both procedures is performed.

[ ]:
```
u_3d = Scalar_mask_XYZ(x, y, z, wavelength)
u_3d.n = fn(x, y, z, wavelength)
```

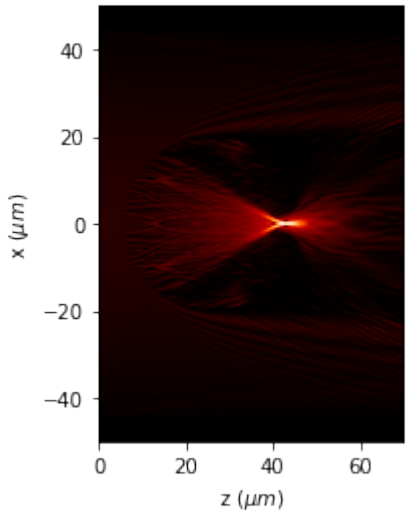
[ ]:
```
u_3d.incident_field(u0=u0 * t0)
```

[ ]:
```
%%time
u_3d.clear_field()
u_3d.WPM(verbose=True, has_edges=True)
```

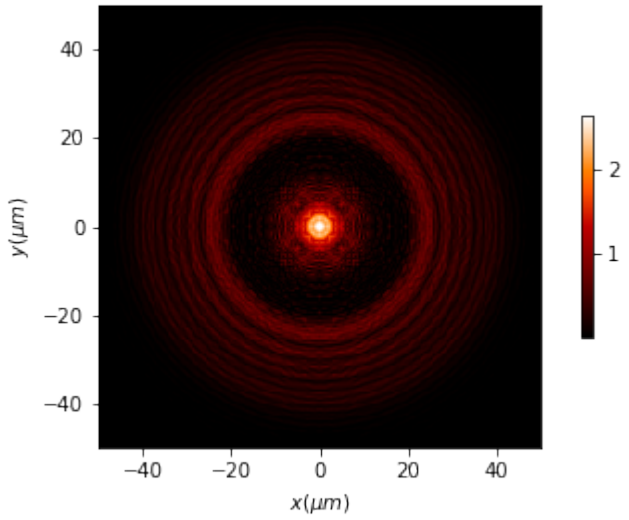
```
Time = 7.17 s, time/loop = 28.0 ms
CPU times: user 7.24 s, sys: 135 ms, total: 7.38 s
Wall time: 7.35 s
```

[ ]:
```
u_3d.draw_XZ(y0=0, logarithm=True)
plt.axis('scaled')
```

```
<Figure size 432x288 with 0 Axes>
```



[ ]:
```
u_3d.draw_XY(z0=50, logarithm=True, normalize='', has_colorbar='vertical')
```



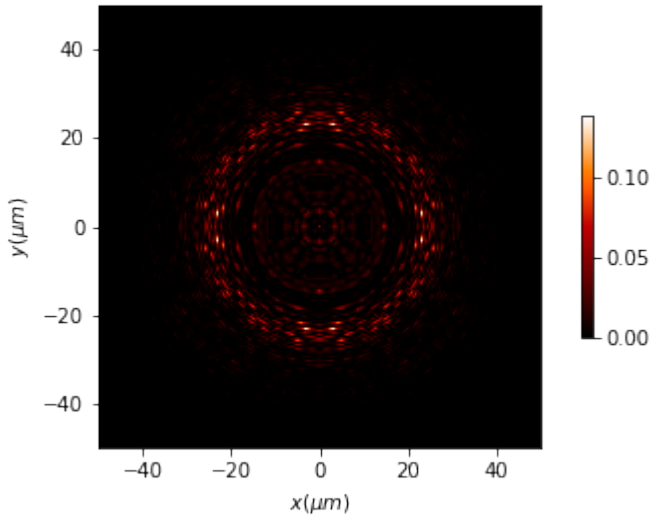
6.6.3.3 Differences between both algorithms

[ ]:
```
I_3d_final = np.abs(u_3d.u[:, :, -1])**2
I_iter = np.abs(u_iter.u)**2

u_diffs = Scalar_mask_XY(x, y, wavelength)
u_diffs.u = (I_iter - I_3d_final) / I_iter.max()

u_diffs.draw(has_colorbar='vertical', logarithm=False)
print("Differences = {:2.2f} %".format((100 * np.abs(
    (I_iter - I_3d_final) / I_iter.max()).mean())))
```

```
Differences = 3.06 %
```



[1]:
```
from diffractio import degrees, np, um
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
```

6.6.4 Sphere

[3]:
```
length = 100 * um
numdataX = 128
numdataZ = 128


x0 = np.linspace(-length / 2, length / 2, numdataX)
y0 = np.linspace(-length / 2, length / 2, numdataX)
z0 = np.linspace(-length / 2, length / 2, numdataX)
wavelength = 0.55 * um

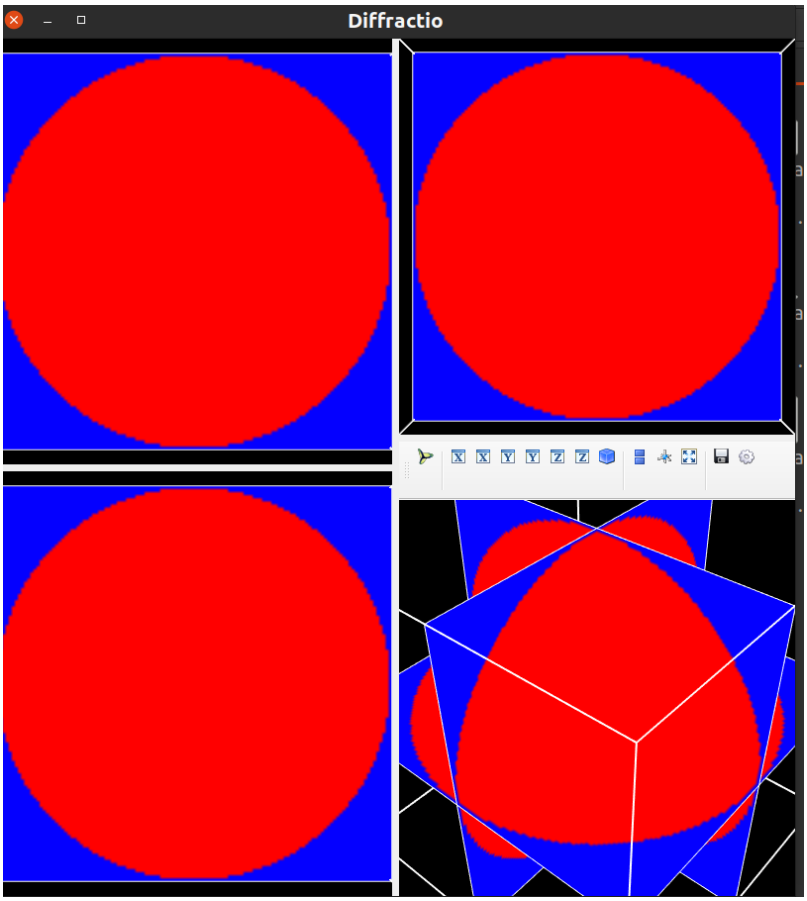
uxyz = Scalar_mask_XYZ(
    x=x0, y=y0, z=z0, wavelength=wavelength, n_background=1., info='')
uxyz.sphere(
```

(continues on next page)

(continued from previous page)

```
    r0=(0 * um, 0 * um, 0 * um),
    radius=(50 * um, 50 * um, 50 * um),
    refractive_index=2,
    angles=(0 * degrees, 0 * degrees, 45 * degrees))

uxyz.draw_refractive_index()
```



6.6.5 Square

[3]:
```
length = 100 * um
numdataX = 128
numdataZ = 128


x0 = np.linspace(-length / 2, length / 2, numdataX)
y0 = np.linspace(-length / 2, length / 2, numdataX)
z0 = np.linspace(-length / 2, length / 2, numdataX)
wavelength = 0.55 * um

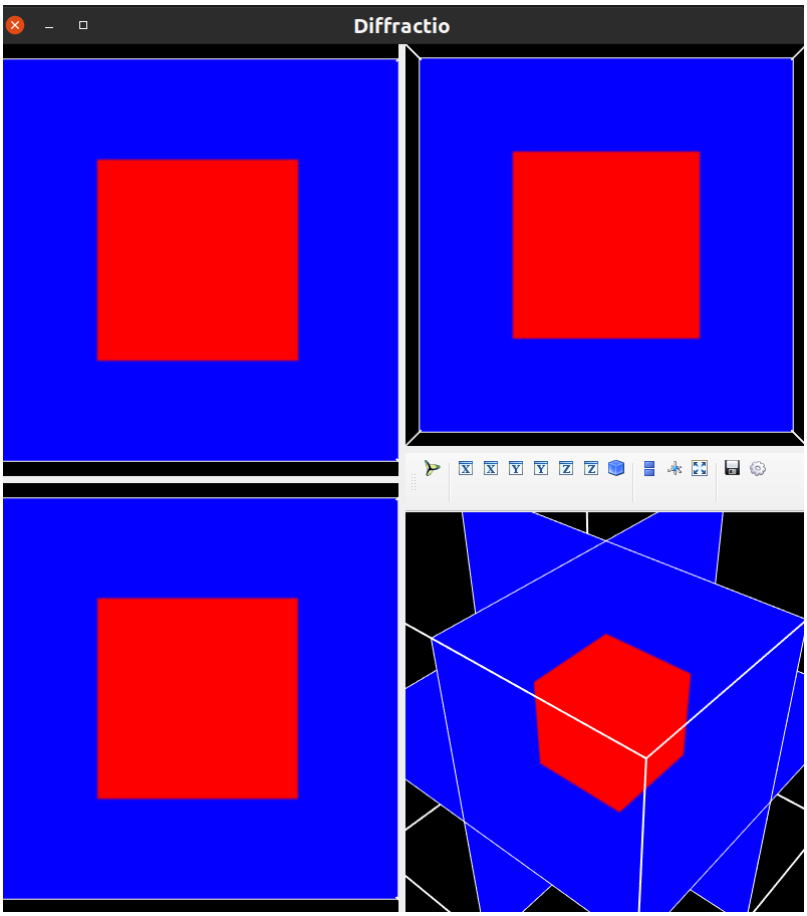

uxyz = Scalar_mask_XYZ(
    x=x0, y=y0, z=z0, wavelength=wavelength, n_background=1., info='')
```

(continues on next page)

(continued from previous page)

```
uxyz.square(
    r0=(0 * um, 0 * um, 0 * um),
    length=(50 * um, 50 * um, 50 * um),
    refractive_index=2,
    angles=None,
    rotation_point=None)
uxyz.draw_refractive_index()
```

```
close the window to continue
```



6.6.6 Cylinder

[4]:
```
length = 100 * um
numdataX = 128
numdataZ = 128

longitud = 50 * um

x0 = np.linspace(-length / 2, length / 2, numdataX)
y0 = np.linspace(-length / 2, length / 2, numdataX)
z0 = np.linspace(-longitud, longitud, numdataZ)
```

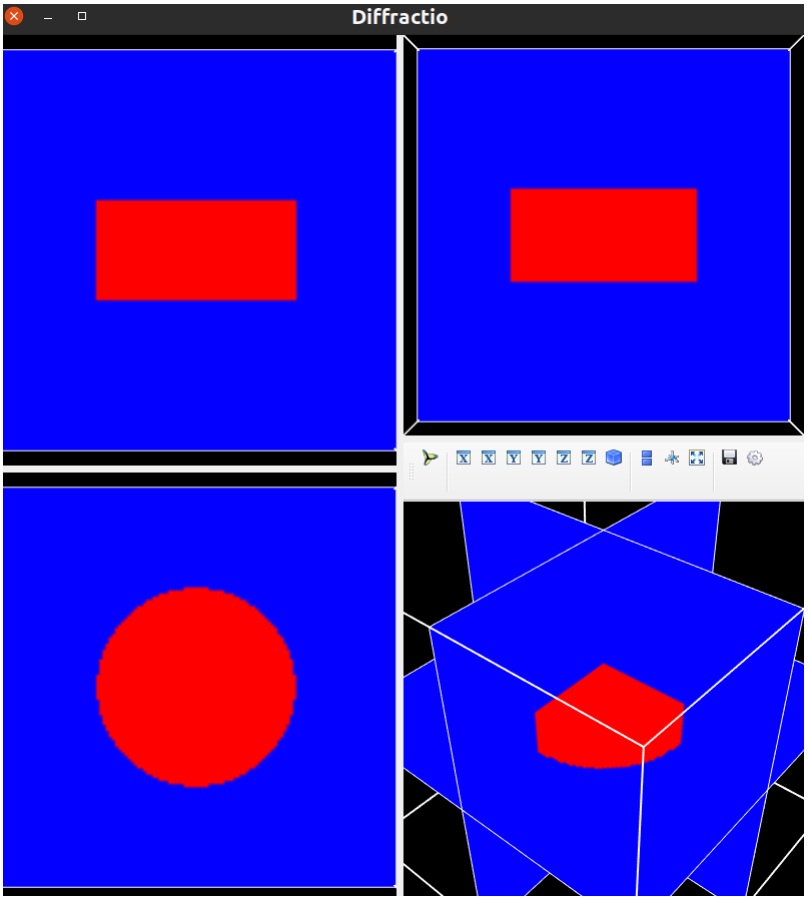
(continues on next page)

(continued from previous page)

```
wavelength = 0.55 * um


uxyz = Scalar_mask_XYZ(
    x=x0, y=y0, z=z0, wavelength=wavelength, n_background=1., info='')
uxyz.cylinder(
                  r0=(0 * um, 0 * um, 0 * um),
                  radius=(25 * um, 25 * um),
                  length=25 * um,
                  refractive_index=2,
                  eje=(1, 1, 1),
                  angle=45 * degrees)
uxyz.draw_refractive_index()
```

```
close the window to continue
```



6.7 Vector_XY

6.7.1 Characteristics

Vector_XY is a set of three modules for:

- Generation of 2D (xy-axis) paraxial vector light source.
- Generation of 2D (xy-axis) paraxial vector masks and diffractive optical elements.
- Propagation of light, determination of parameters, and other functions.
- Drawing sources, masks and fields.

These modules are named: vector_fields_XY.py, vector_sources_XY.py, and vector_masks_XY.py.

Each module present a main class:

- Vector_field_XY
- Vector_mask_XY
- Vector_source_XY

The main attributes for these classes are the following:

- self.x (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n.
- self.y (numpy.array): linear array with equidistant positions. The number of data is preferibly 2^n.
- self.wavelength (float): wavelength of the incident field.
- (numpy.array): complex field with size 2D x.y
  - self.Ex
  - self.Ey
  - self.Ez

We can also find these atributes:

- self.X (numpy.array): equal size to x * y. complex field.
- self.Y (numpy.array): equal size to x * y. complex field.
- self.quality (float): quality of RS propagation. Valid for values > 1.
- self.info (str): description of data.
- self.type (str): Class of the field.
- self.date (str): date when performed.

The propagation techiniques implemented are:

- RS - Rayleigh-Sommerfeld propagation for Ex and Ey fields
- VRS - Complete vector propagation (Ex, Ey, Ez) in Rayleigh-Sommerfeld approach.
- VFFT - Vector propagation to the focal distance of a lens.

The dimensional magnitudes are related to microns: `micron = 1.`

6.7.1.1 Generation of light sources

An instance must be created before starting to operate with light sources. The initialization accepts several arguments.

6.7.1.2 Creating an instance

[1]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees

from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XY import Scalar_field_XY

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_fields_XY import Vector_field_XY
```

6.7.1.3 Generating a vectorial light source

Light sources are defined in the scalar_sources_xy.py module. When the field is initialized, the amplitude of the field is zero. Vector sources are generated using a scalar source and providing a vector polarization. There are many methods to convert a scalar source to a vector source:

- **constant_polarization**: Generates a plane wave with a given direction and amplitude.
- **radial_wave**: Generates a plane wave with radial polarization.
- **azimuthal_wave**: Generates a plane wave with transversa polarization.
- **radial_inverse_wave**:
- **local_polarized_vector_wave**: Local radial polarized vector wave.
- **local_polarized_vector_wave_radial**: Local radial polarized vector wave.
- **local_polarized_vector_wave_hybrid**: Local hibrid polarized vector wave.
- **spiral_polarized_beam**: Spiral polarized wave.

For a more detailed description of each method, refer to the individual documentation of each one.

Example: gauss azimuthal_wave

[2]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)

wavelength = 0.6328 * um
```

Scalar source

[3]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0), w0=100 * um, z0=0 * um, A=1)
```

Vector source

[4]:
```
EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.radial_wave(u=u0, r0=(0, 0), radius=(200, 200))
```

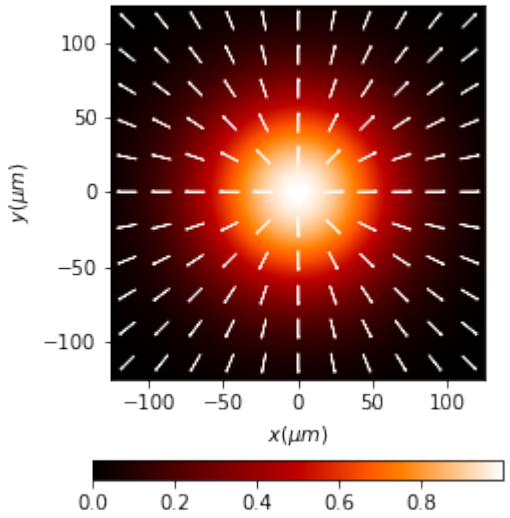
If we need, we can convert the E_x and E_y fields in Scalar_field_XY.

[5]:
```
Ex, Ey, _ = EM1.get(is_matrix=False)
print(Ex)
```

```
Scalar_field_XY
 - x:  (512,),   y:  (512,),   u:  (512, 512)
 - xmin:       -125.00 um,  xmax:      125.00 um,  Dx:   0.49 um
 - ymin:       -125.00 um,  ymax:      125.00 um,  Dy:   0.49 um
 - Imin:       0.00,     Imax:      0.99
 - phase_min:  -180.00 deg, phase_max: 0.00 deg
 - wavelength: 0.63 um
 - date:       2022-10-15_23_24_06
```

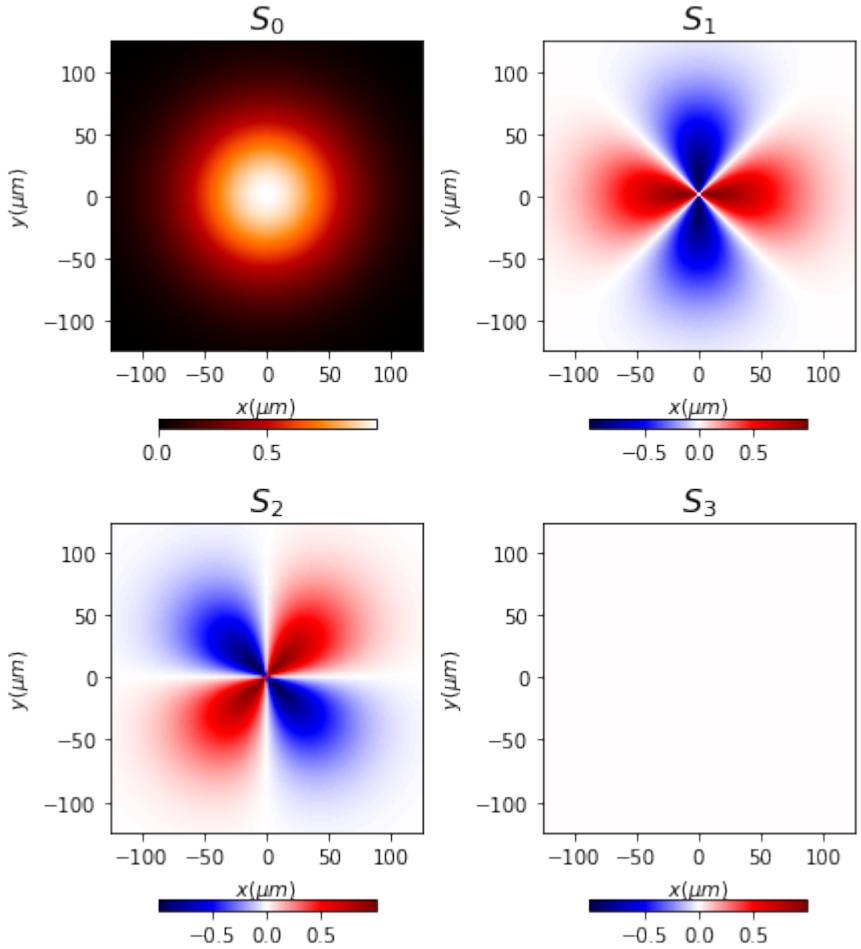
We can obtain a variety of representation, simply using the draw method.

[6]:
```
EM1.draw(kind='ellipses')
```



Polarimetric representation can be performed in a simple way.

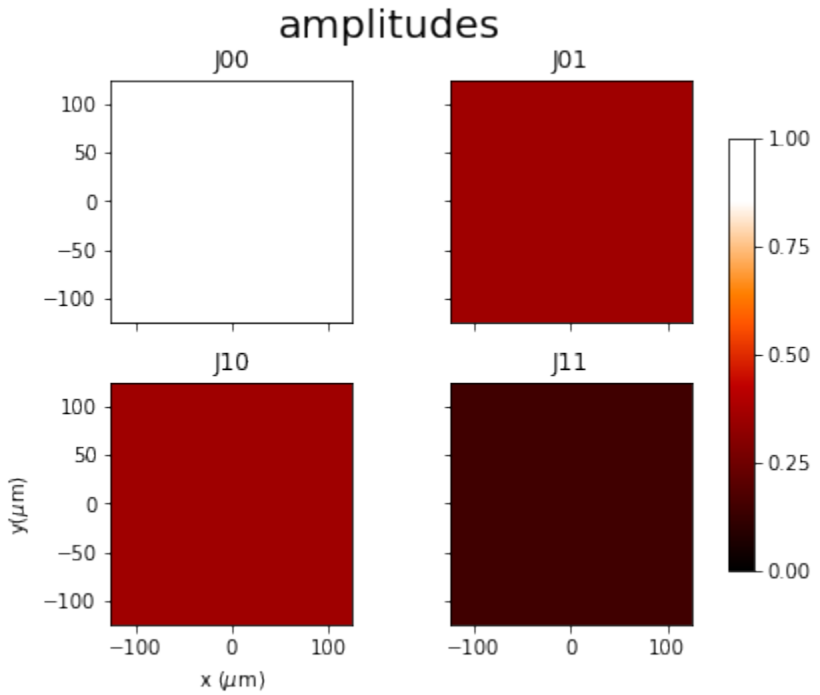
[7]:
```
EM1.draw('stokes')
```



6.7.2 A vector light source passes through a linear polarizer

[8]:
```
M1 = Vector_mask_XY(x=x0, y=y0, wavelength=wavelength)
M1.polarizer_linear(azimuth=22.5 * degrees)
M1.draw()
```

```
<Figure size 432x288 with 0 Axes>
```

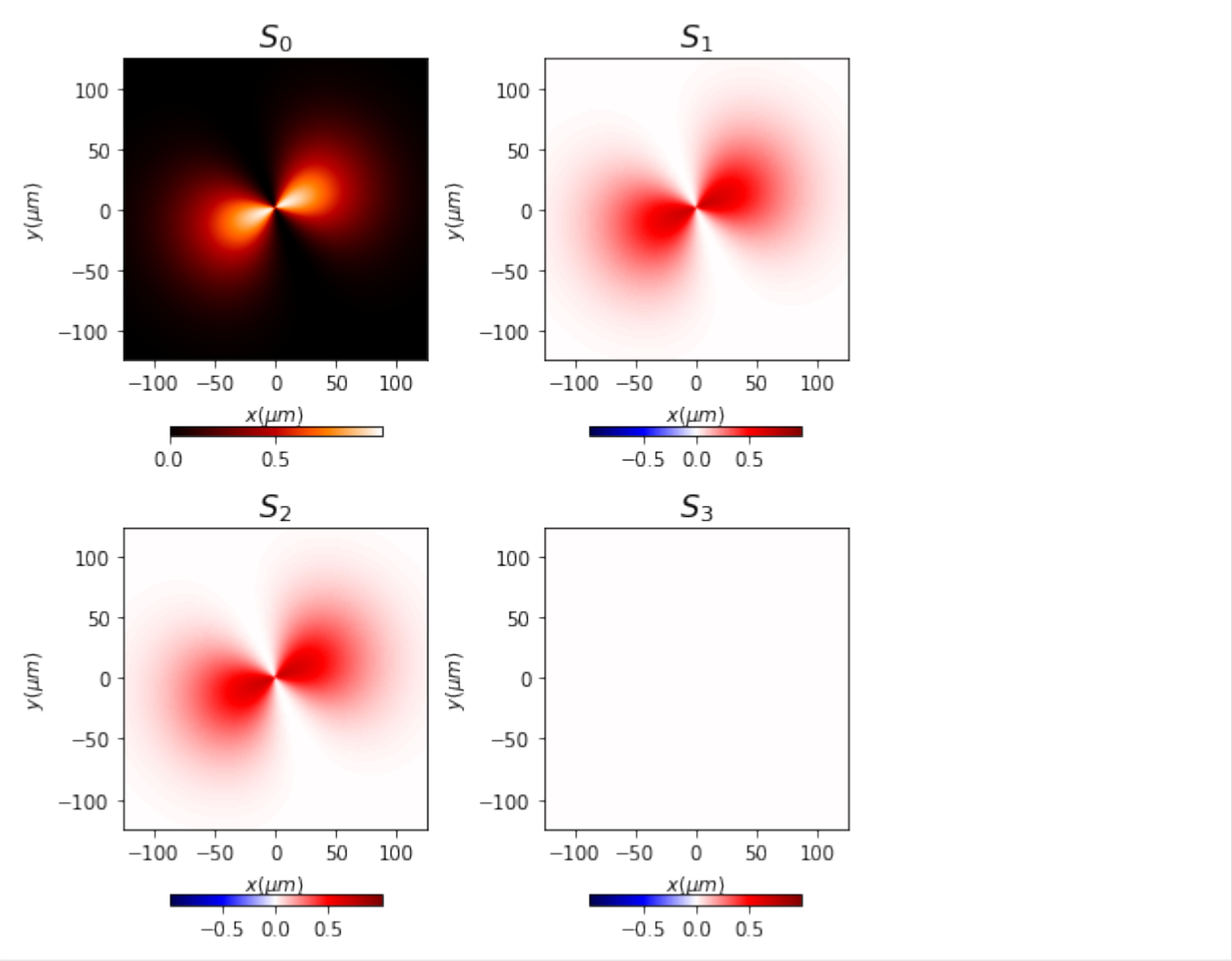


6.7.2.1 Multiplying fields by mask

We can have complex vector mask using the functions based on Scalar_XY masks. Also, vector masks are provided which act as simple polarizers.

[9]:
```
EM2 = EM1 * M1
```

[10]:
```
EM2
EM2.draw('stokes')
```



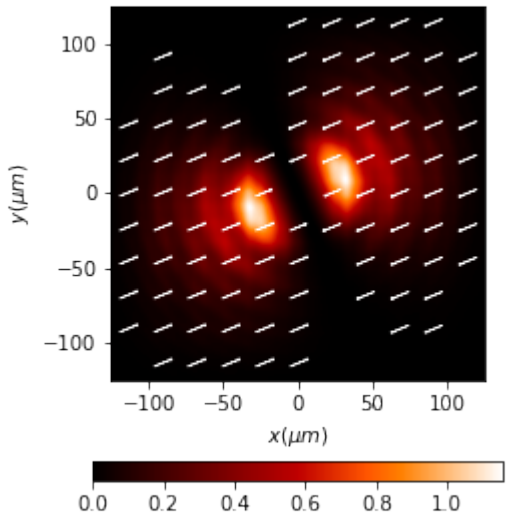
As the polarizer is linear at 45º, only S_2 parameter is obtained.

6.7.2.2 Propagation

Vector fields can be propagated using Rayleigh-Sommerfeld approach.

[11]:
```
EM3 = EM2.VRS(z=2 * mm, new_field=True)
```

[12]:
```
EM3.draw('ellipses')
```



Light is propagated, but as the polarization is the same for all the positions, the result is the same as an scalar propagation.

6.7.3 Propagation of an spiral polarized beam

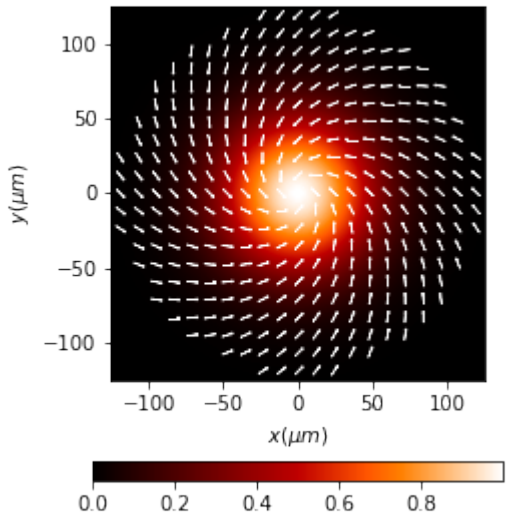
[13]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)

wavelength = 0.6328 * um
```

[14]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(75 * um, 75 * um),
              z0=0 * um,
              A=1,
              theta=0. * degrees,
              phi=0 * degrees)
```

[15]:
```
EM0 = Vector_source_XY(x0, y0, wavelength)
EM0.spiral_polarized_beam(u0, r0=(0, 0), alpha=45 * degrees)
```

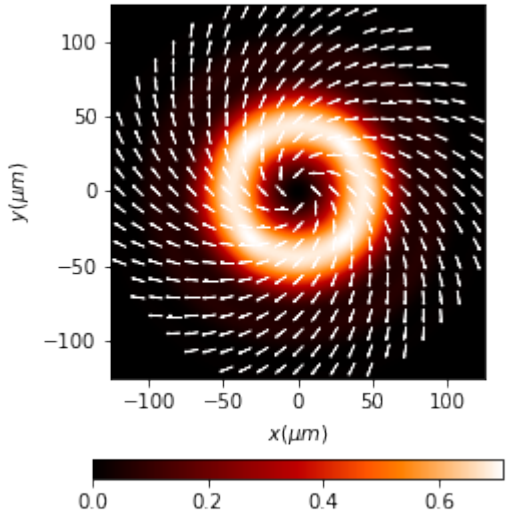
[16]:
```
EM0.draw('ellipses', num_ellipses=(21, 21), amplification=0.5)
```



[17]:
```
EM1 = EM0.VRS(z=5 * mm, verbose=True)
```

```
Good result: factor 24.69
```

[18]:
```
EM1.draw('ellipses', num_ellipses=(21, 21), amplification=0.75)
```



6.7.4 Vector Rayleigh-Sommerfeld Approximation

We have implemented the Vector Rayleigh-Sommerfeld approach to propagation of vectorial fields as used in H. Ye, C.-W. Qiu, K. Huang, J. Teng, B. Luk'yanchuk, y S. P. Yeo, «Creation of a longitudinally polarized subwavelength hotspot with an ultra-thin planar lens: vectorial Rayleigh–Sommerfeld method», Laser Phys. Lett., vol. 10, n.º 6, p. 065004, jun. 2013. DOI: 10.1088/1612-2011/10/6/065004 (http://stacks.iop.org/1612-202X/10/i=6/a=065004?key=crossref.890761f053b56d7a9eeb8fc6da4d9b4e). In the propagation, there is a transference of field from the Ex, Ey polarizations to Ez polarization.

[19]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

from diffractio.vector_fields_XY import Vector_field_XY
```

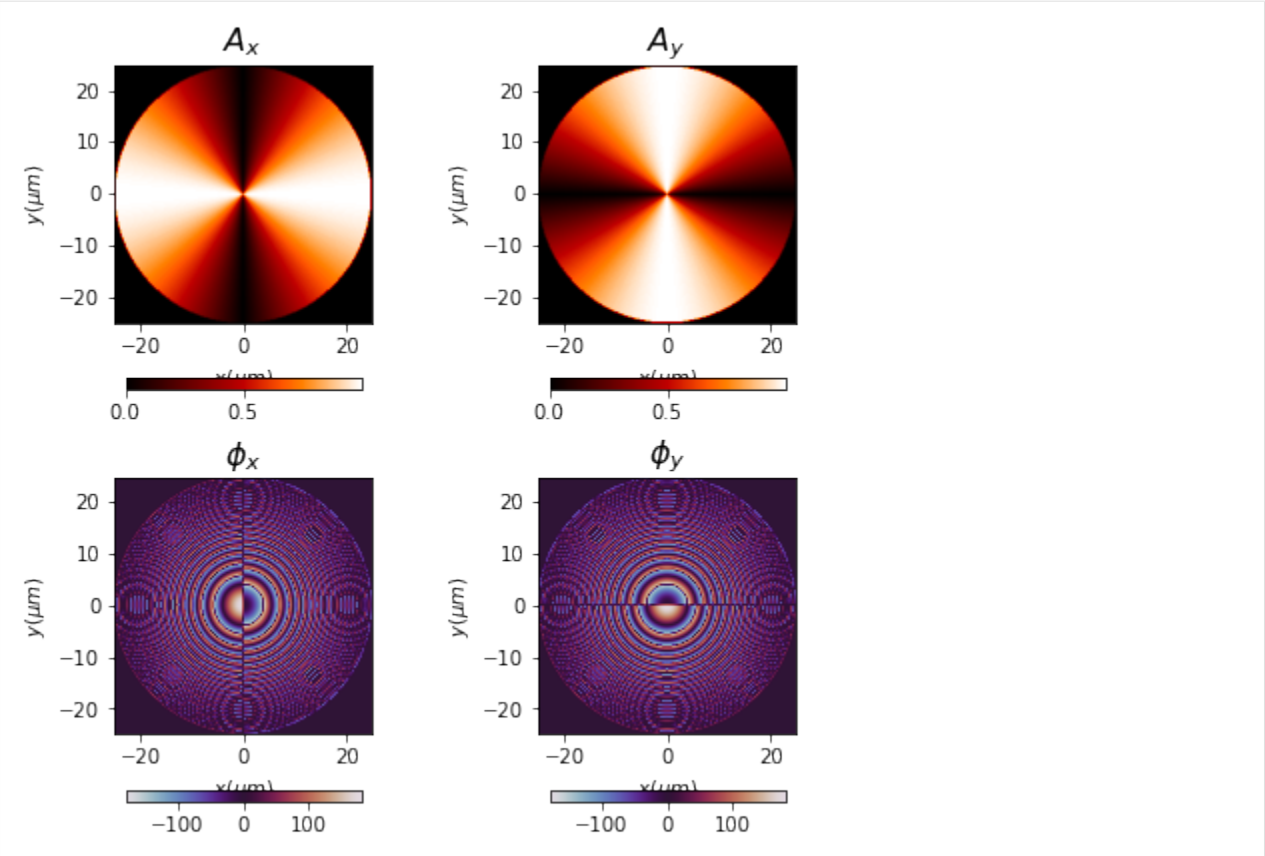
[20]:
```
size = 25 * um
x0 = np.linspace(-size, size, 256)
y0 = np.linspace(-size, size, 256)
wavelength = .640 * um
```

[21]:
```
focal = 25 * um

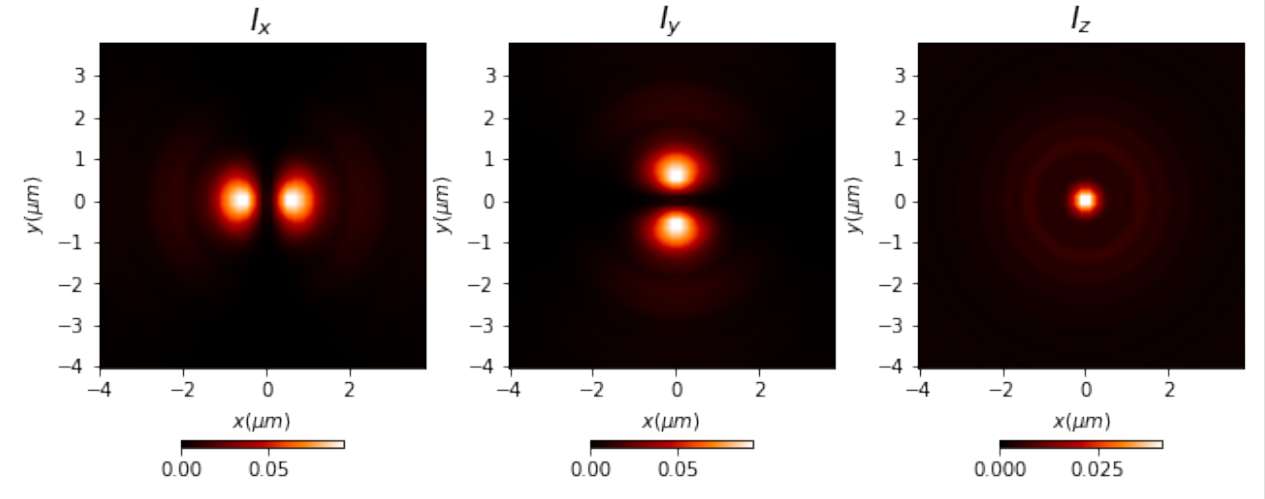
t = Scalar_mask_XY(x0, y0, wavelength)
t.lens(r0=(0, 0), focal=focal, radius=size)
```

[22]:
```
E0 = Vector_source_XY(x0, y0, wavelength)
E0.radial_wave(u=t)
```

[24]:
```
E0.draw('fields')
```



[25]:
```
E1 = E0.VRS(z=focal, n=1, new_field=True, verbose=False, amplification=(1, 1))
E1.cut_resample(x_limits=(-4, 4), y_limits=(-4, 4))
E1.draw(kind='intensities', logarithm=1e-3)
```



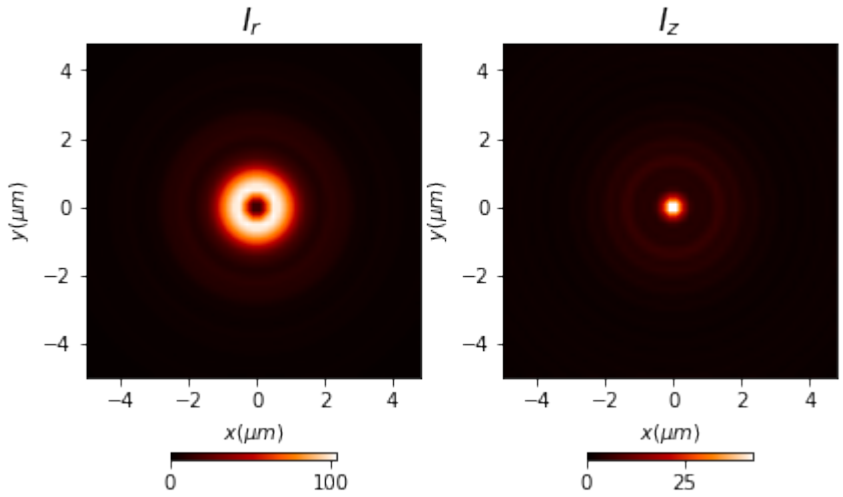
Additionally, r-z fields can be drawn for vector fields:

[26]:
```
E1 = E0.VRS(z=focal, n=1, new_field=True, verbose=False, amplification=(1, 1))
E1.cut_resample((-5, 5), (-5, 5))
```

(continues on next page)

(continued from previous page)

```
E1.draw(kind='intensities_rz', logarithm=False)
```



6.7.5 Vector Fast Fourier Transform

We have implemented an algorithm to obtain the vector field distribution at the focal plane of a lens. The algorithm is based on Richards–Wolf theory, according to K. Jahn and N. Bokor, "Intensity control of the focal spot by vectorial beam shaping," Opt. Commun., vol. 283, no. 24, pp. Nevertheless, 2D Fast Fourier Transform (FFT) is used to calculate the double integral, instead of 2D Chirp-z Transform (CZT). This procedure will be implemented in the future.

[27]:
```
from diffractio.scalar_sources_XY import Scalar_source_XY

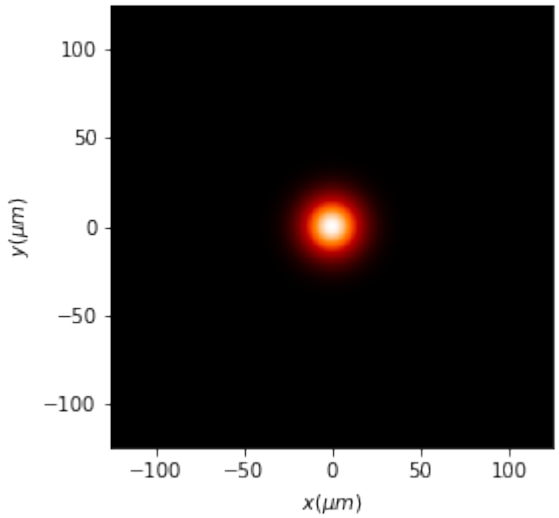
from diffractio.vector_fields_XY import Vector_field_XY
from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
```

[28]:
```
size = 250 * um
x0 = np.linspace(-size / 2, size / 2, 1024)
y0 = np.linspace(-size / 2, size / 2, 1024)

wavelength = 0.6328 * um
```

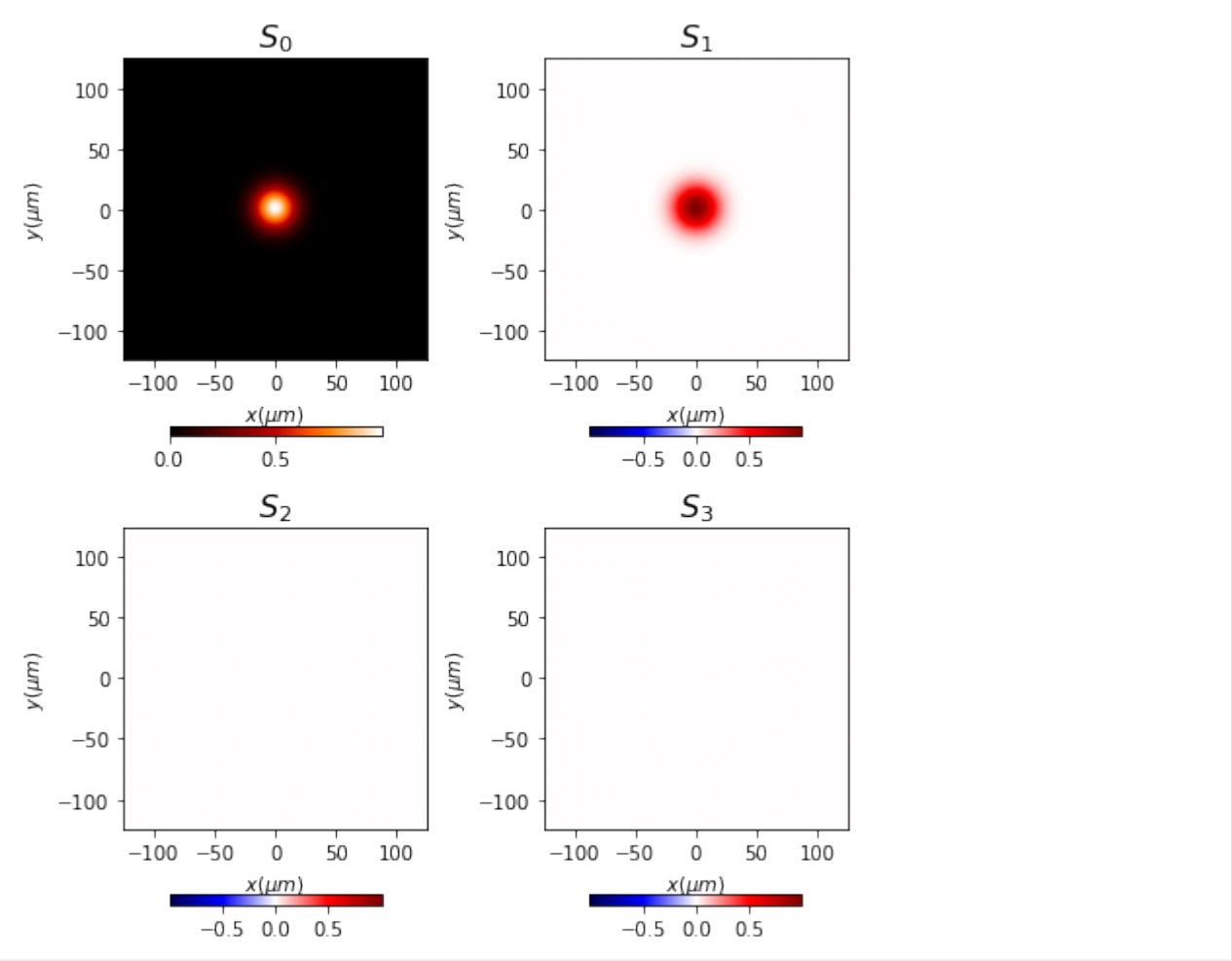
[29]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(25 * um, 25 * um),
              z0=0 * um,
              A=1,
              theta=0. * degrees,
              phi=0 * degrees)
```

[30]:
```
u0.draw()
```



[31]:
```
EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.constant_polarization(u=u0, v=(1, 0))
EM1.pupil()
```

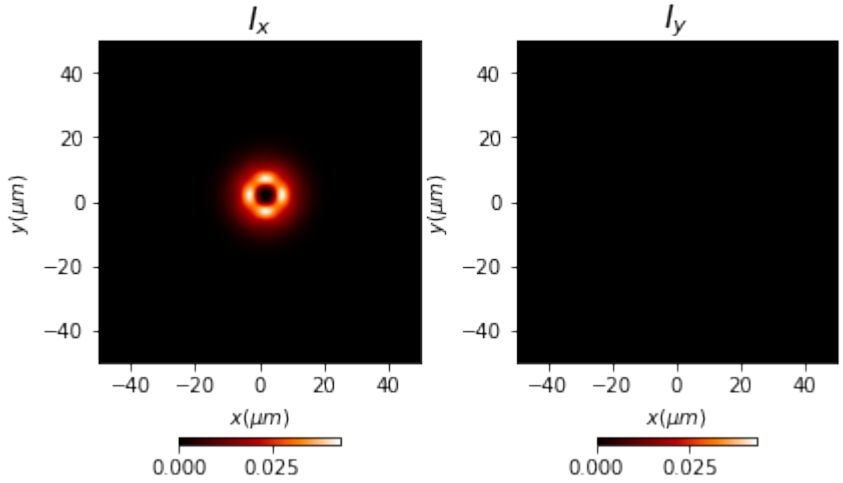
[32]:
```
EM1.draw('stokes')
```



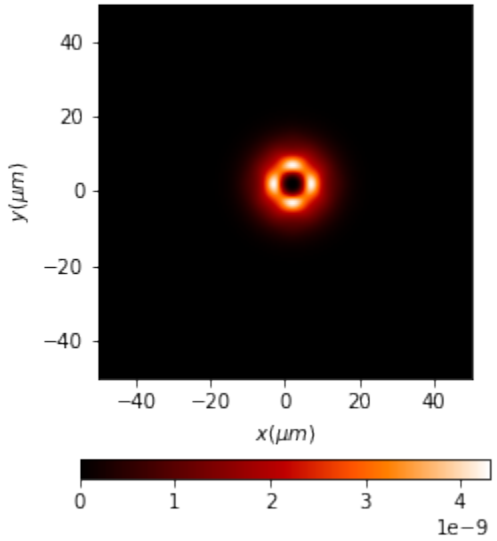
[33]:
```
EM2 = EM1.VFFT(radius=64 * um,
               focal=1 * mm,
               n=1,
               new_field=True,
               has_draw=False)
```

[34]:
```
x_resample = 50
EM2.cut_resample([-x_resample, x_resample], [-x_resample, x_resample],
                 num_points=(512, 512))
```

[35]:
```
EM2.draw('intensities', logarithm=False)
```



[36]:
```
EM2.draw('intensity', logarithm=1e-7)
```



Let us compare this result with that obtained with VRS. We need to include the lens in the mask.

[37]:
```
radius = 250 * um
t_lens = Scalar_mask_XY(x0, y0, wavelength)
t_lens.lens(r0=(0, 0), radius=radius, focal=1000 * um)
```

[38]:
```
u2 = t_lens * u0

EM3 = Vector_source_XY(x0, y0, wavelength)
EM3.constant_polarization(u=u2, v=(1, 0))
```

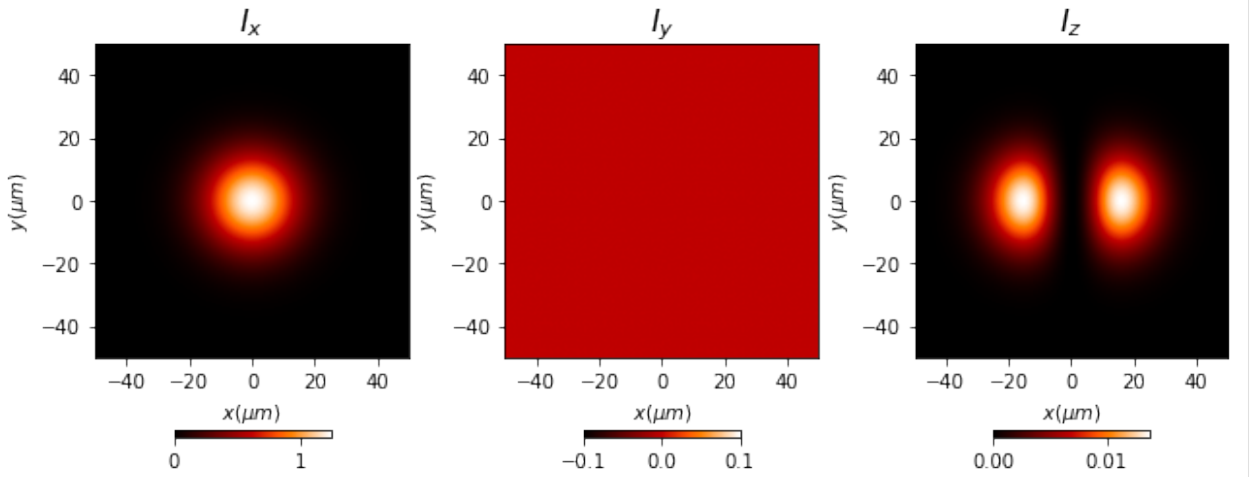
[39]:
```
EM4 = EM3.VRS(z=100 * um, n=1, new_field=True)
```

(continues on next page)

(continued from previous page)

[40]:
```
x_resample = 50
EM4.cut_resample([-x_resample, x_resample], [-x_resample, x_resample],
                 num_points=(512, 512))
```

[41]:
```
EM4.draw('intensities')
```



In comparison to VFFT, we can see that the I_x and I_z are reproduced in a similar way, but the VRS algorithm is not able to generate I_y field.

6.7.6 Example of light sources

6.7.6.1 Creating an instance

An instance must be created before starting to operate with light sources. The initialization accepts several arguments.

[1]:
```
from diffractio import degrees, mm, np, um
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_sources_XY import Vector_source_XY
```

6.7.6.2 Procedures to convert a scalar source into a vector source

When a light source is defined using Scalar_source_XY, it can be converted to vectorial using several functions with vector characteristics of the source:

- constant_polarization
- azimuthal_wave
- radial_wave
- radial_inverse_wave
- azimuthal_inverse_wave
- local_polarized_vector_wave
- local_polarized_vector_wave_radial
- local_polarized_vector_wave_hybrid
- spiral_polarized_beam

When the field *u* in the function is a float number, then it is considered as the amplitude of the wave. Also this functions are masked, with a circular mask, radius=(r_x, r_y), when r_x and r_y >0.

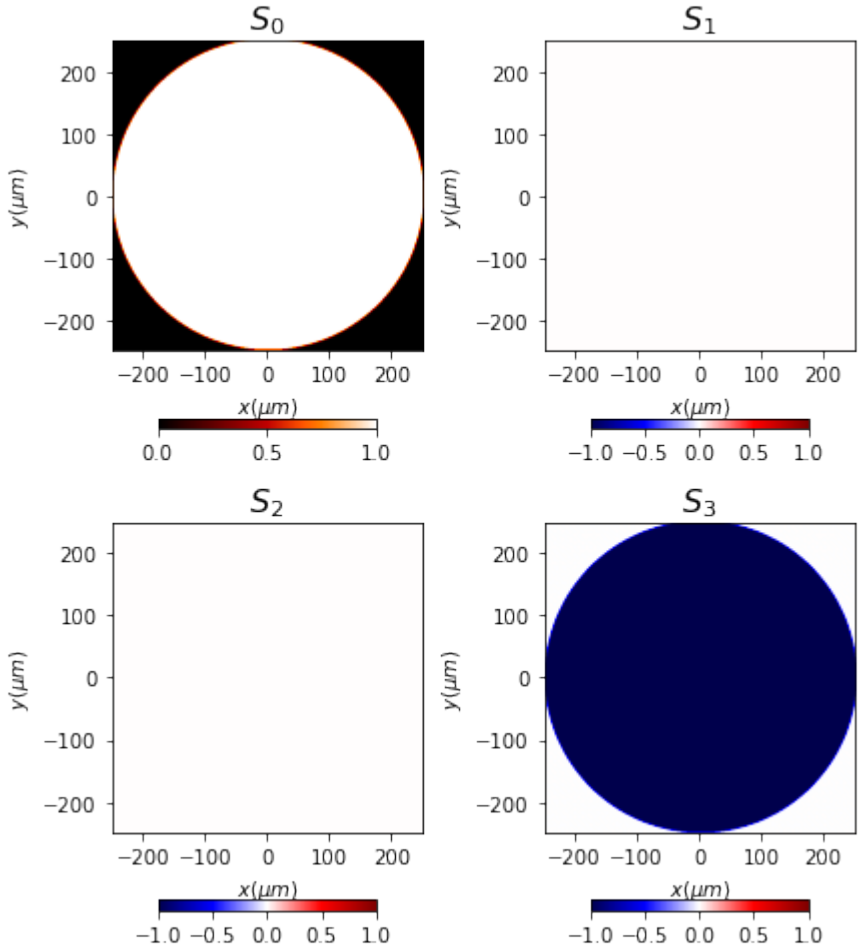
When the parameter results u=1, then the intensity distribution is 1 in both E_x and E_y fields.

constant polarization wave

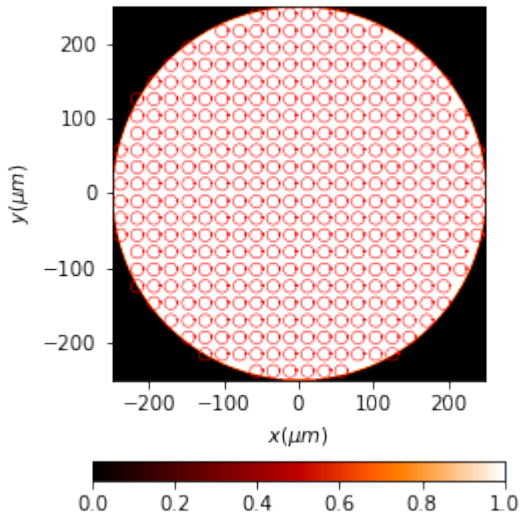
[2]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.constant_polarization(u=1, v=(1, 1.j), radius=250 * um)
EM.normalize()
EM.draw('stokes')
```



[3]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(22, 22),
                     amplification=0.75,
                     color_line='r',
                     line_width=.5,
                     draw_arrow=True,
                     head_width=1,
                     ax=False)
```

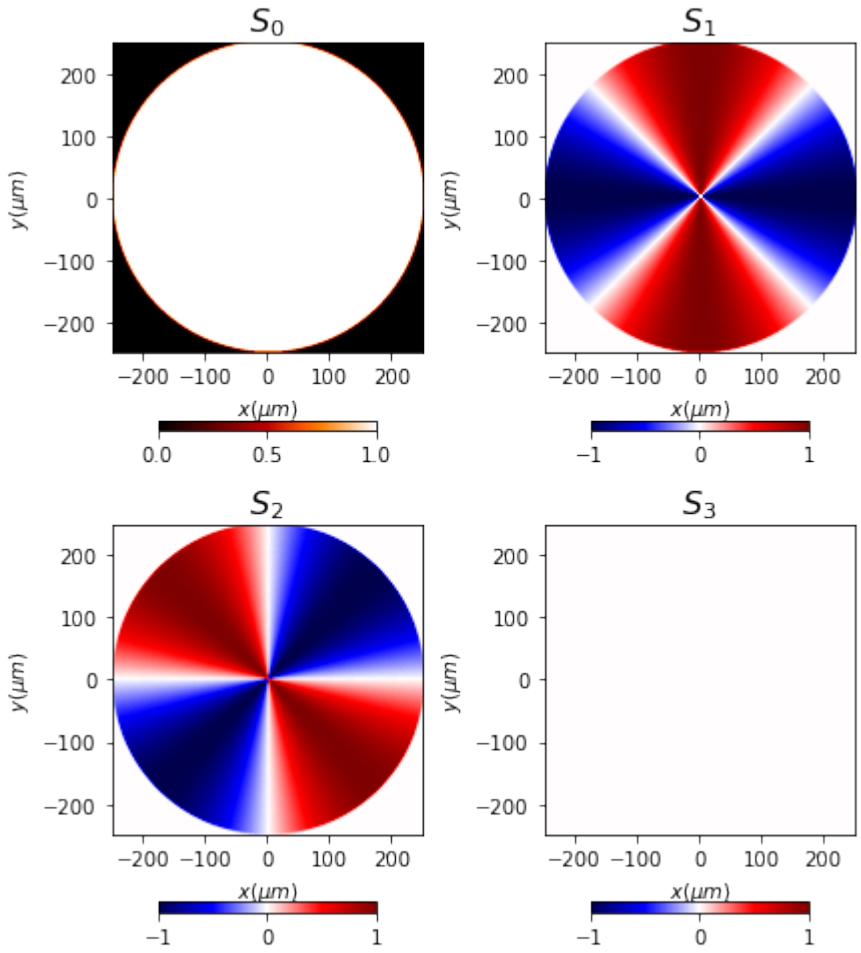


Azimuthal wave

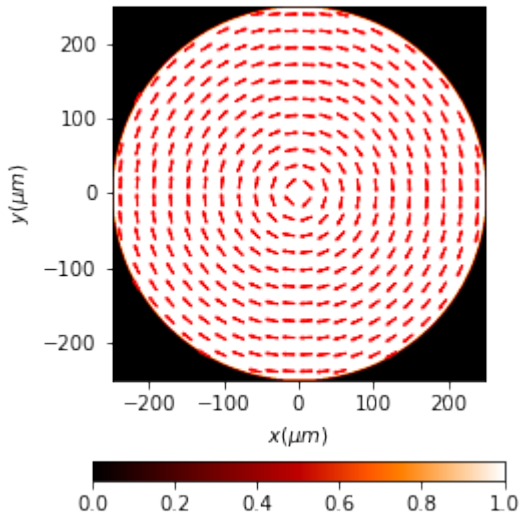
[4]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.azimuthal_wave(u=1, r0=(0, 0), radius=250 * um)
EM.draw('stokes')
```



[5]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(22, 22),
                     amplification=0.5,
                     color_line='r',
                     line_width=1,
                     draw_arrow=True,
                     head_width=3,
                     ax=False)
```

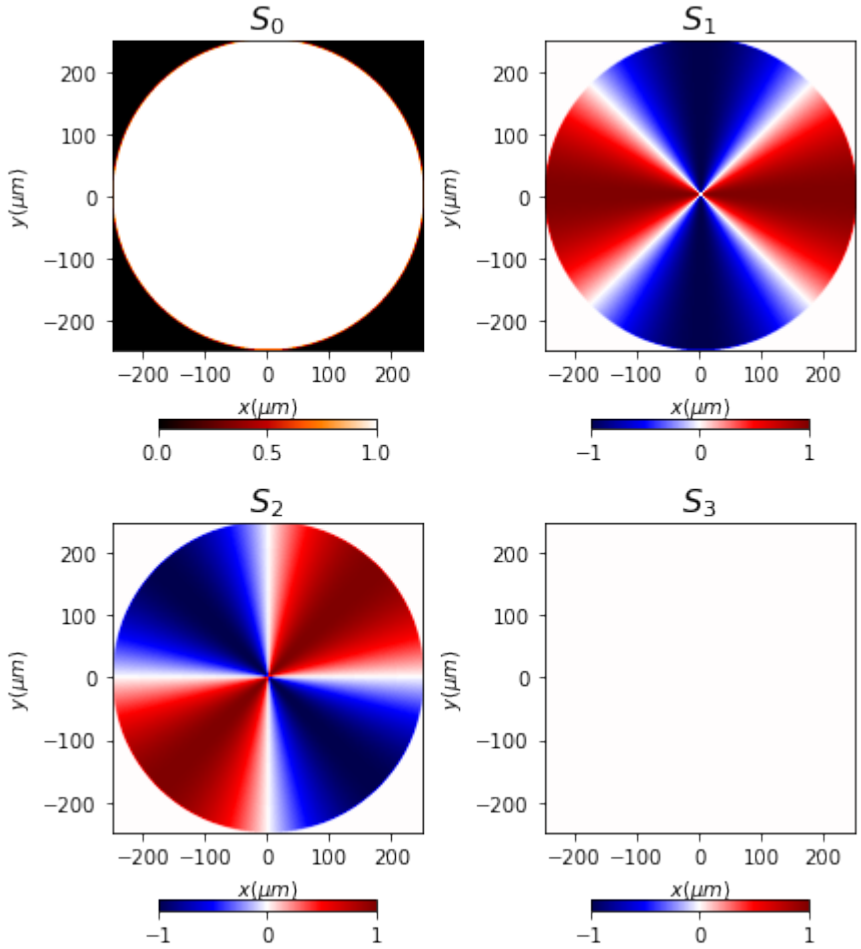


Radial wave

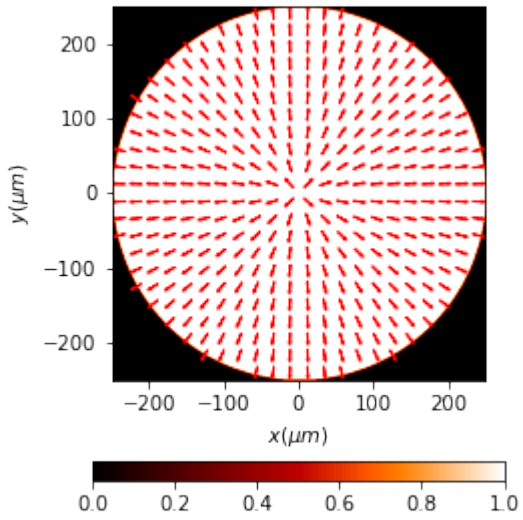
[6]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.radial_wave(u=1, r0=(0, 0), radius=250 * um)
EM.draw('stokes')
```



[7]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(22, 22),
                     amplification=0.5,
                     color_line='r',
                     line_width=1,
                     draw_arrow=True,
                     head_width=3,
                     ax=False)
```

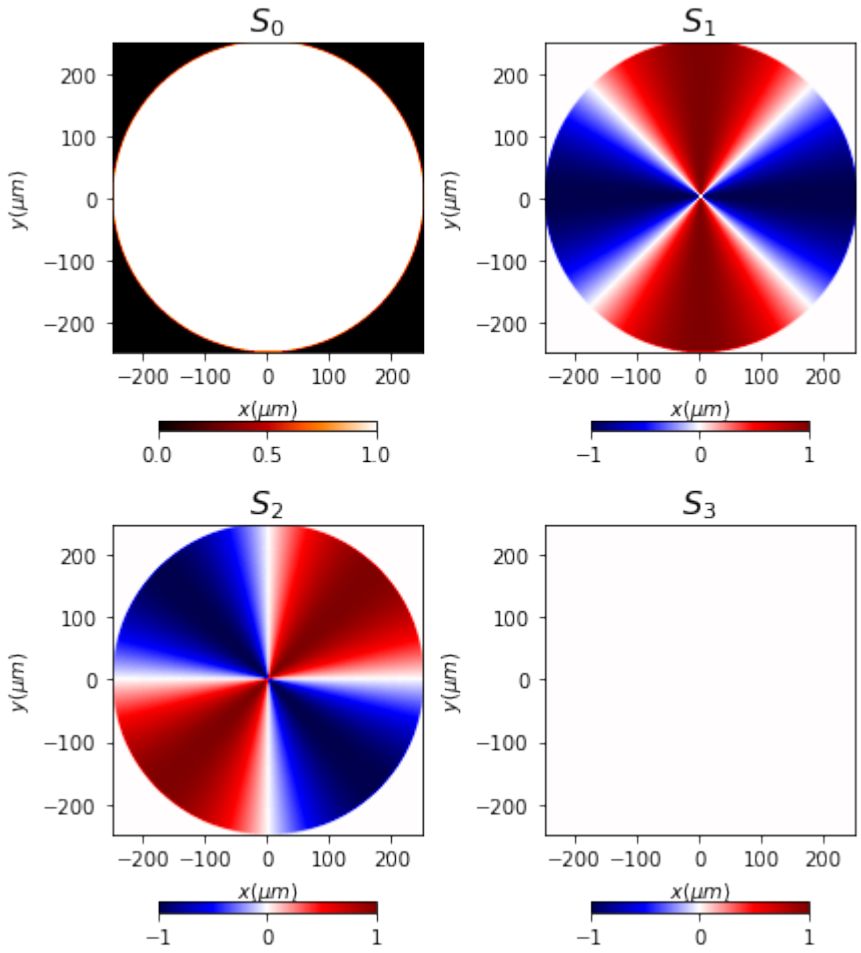


Azimuthal inverse wave

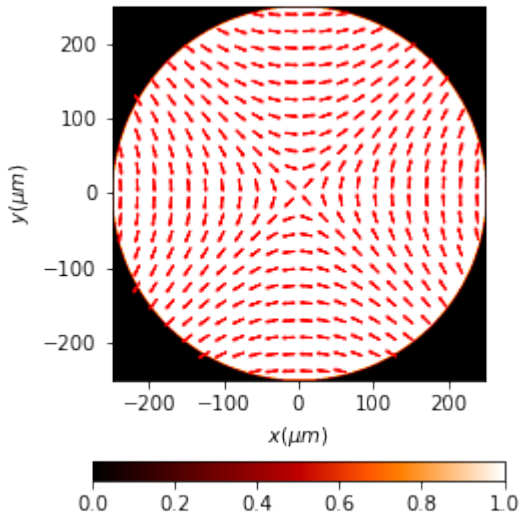
[8]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.azimuthal_inverse_wave(u=1, r0=(0, 0), radius=250 * um)
EM.draw('stokes')
```



[9]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(22, 22),
                     amplification=0.5,
                     color_line='r',
                     line_width=1,
                     draw_arrow=True,
                     head_width=3,
                     ax=False)
```

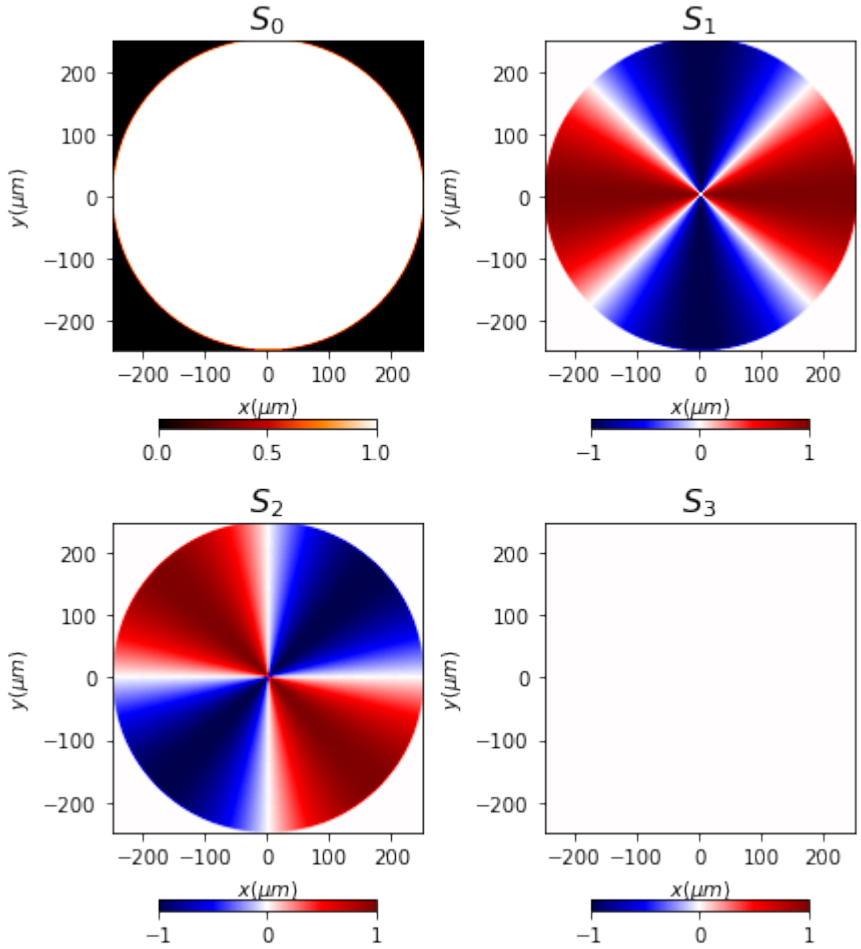


Radial inverse wave

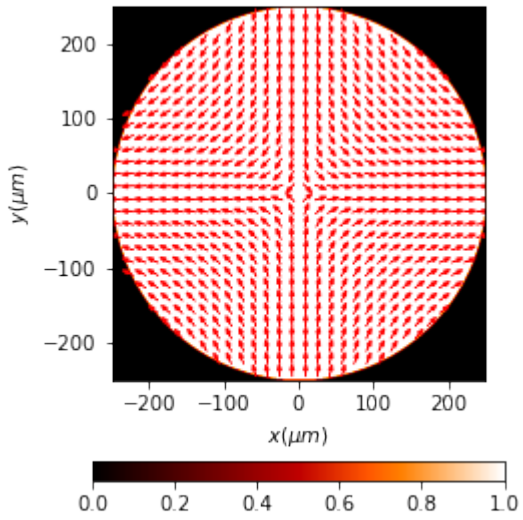
[10]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.radial_inverse_wave(u=1, r0=(0, 0), radius=250 * um)
EM.draw('stokes')
```



[11]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(30, 30),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=5,
                     ax=False)
```

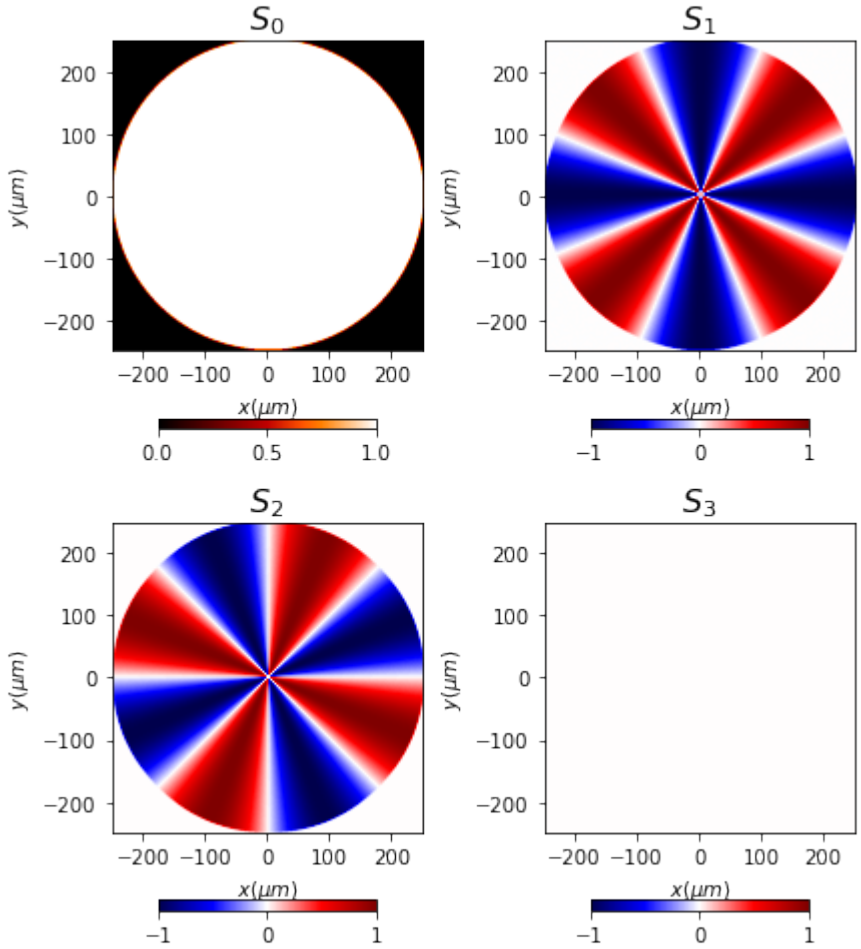


local polarized vector wave

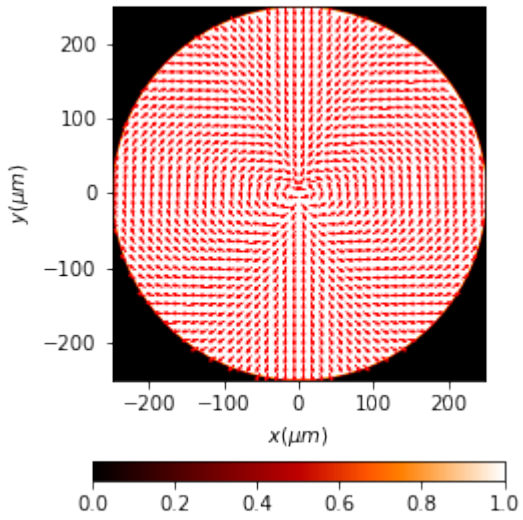
[12]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.local_polarized_vector_wave(u=1,
                               m=2,
                               fi0=np.pi / 2,
                               r0=(0, 0),
                               radius=250 * um)
EM.draw('stokes')
```



[13]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(42, 42),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=3,
                     ax=False)
```

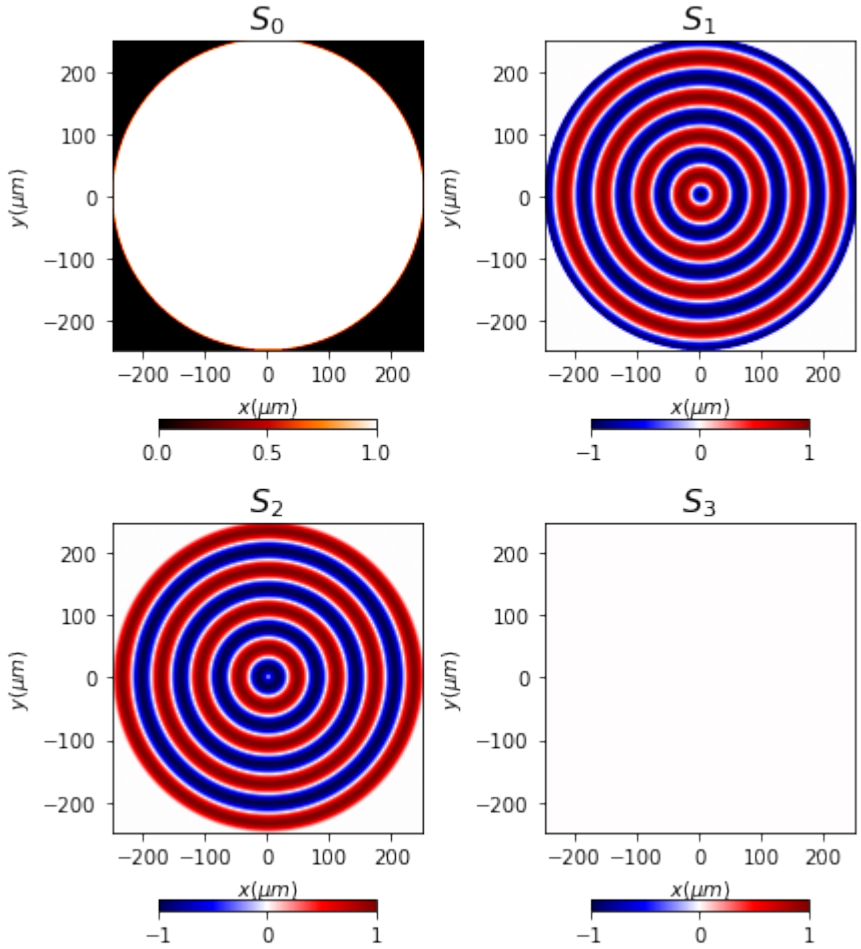


local polarized vector radial beam

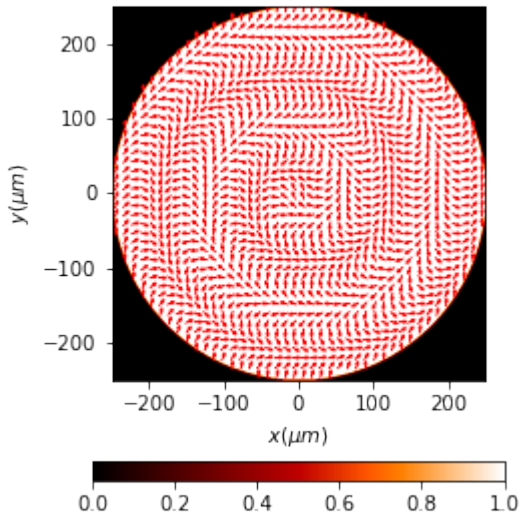
[14]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.local_polarized_vector_wave_radial(u=1,
                                      m=2,
                                      fi0=np.pi / 2,
                                      r0=(0, 0),
                                      radius=250 * um)
EM.draw('stokes')
```



[15]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(42, 42),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=3,
                     ax=False)
```

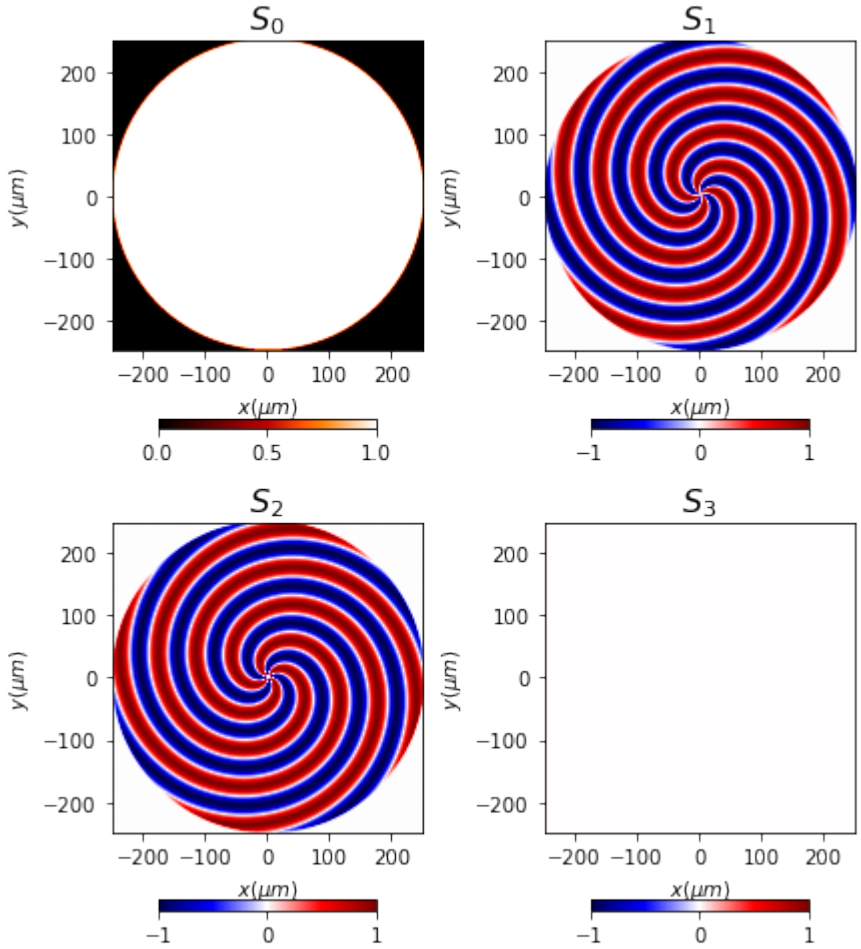


local polarized vector hybrid beam

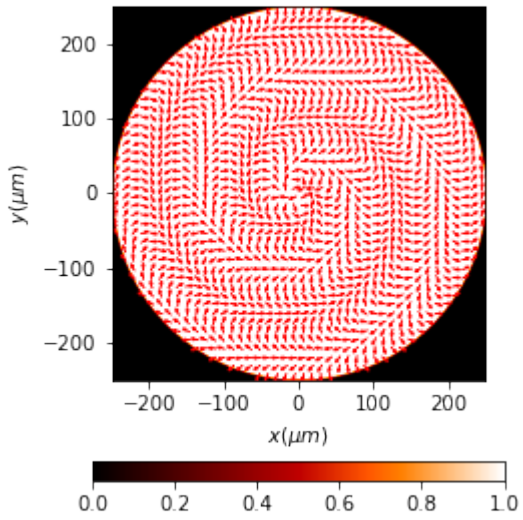
[16]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.local_polarized_vector_wave_hybrid(u=1,
                                      n=2,
                                      m=2,
                                      fi0=np.pi / 2,
                                      r0=(0, 0),
                                      radius=250 * um)
EM.draw('stokes')
```



[17]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(42, 42),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=3,
                     ax=False)
```

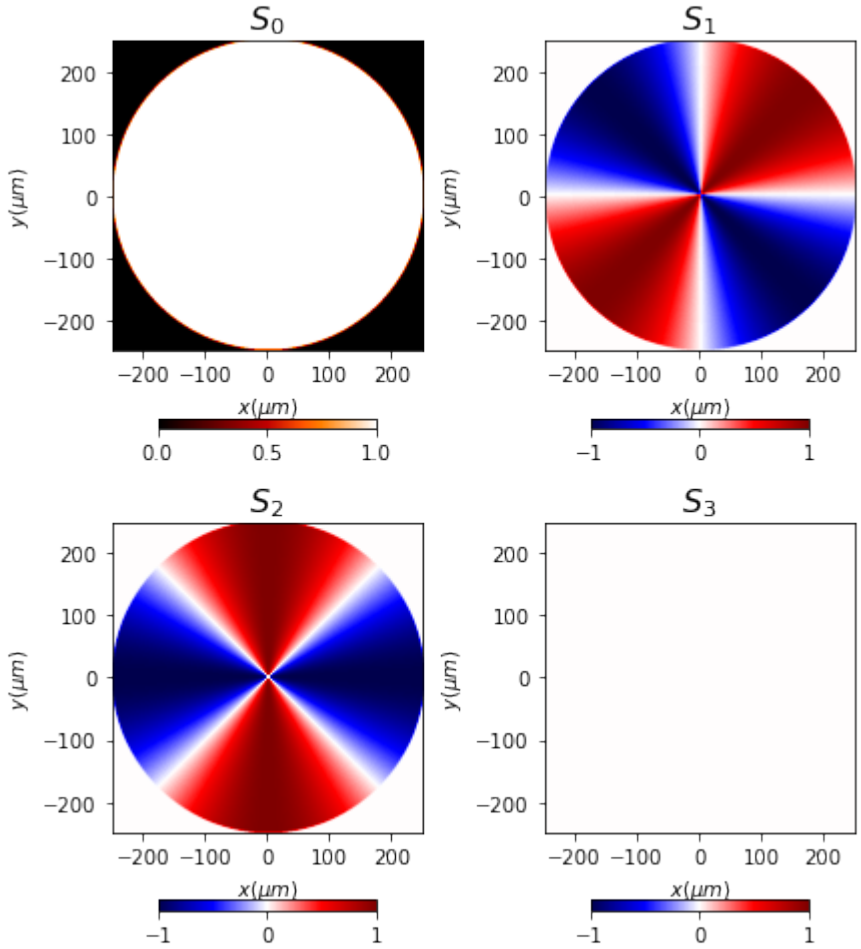


spiral polarized beam

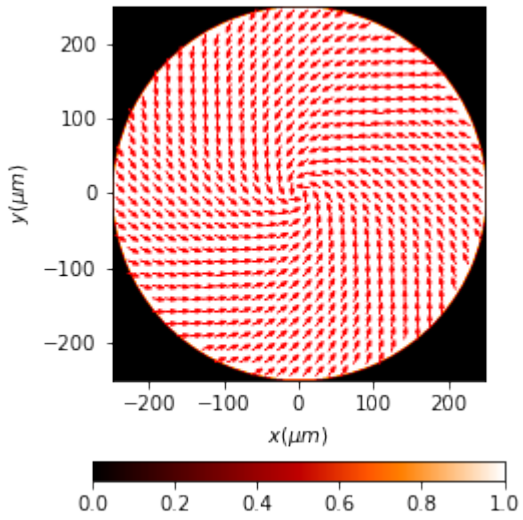
[18]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

EM = Vector_source_XY(x0, y0, wavelength)
EM.spiral_polarized_beam(u=1, r0=(0, 0), alpha=np.pi / 4, radius=250 * um)
EM.draw('stokes')
```



[19]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(30, 30),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=5,
                     ax=False)
```



6.7.6.3 Generation of a structured beam with polarization

The same functions used previously, can be used to generate vector fields with spatial intensity distribution. In this case, the u parameter is a Scalar_source_XY.

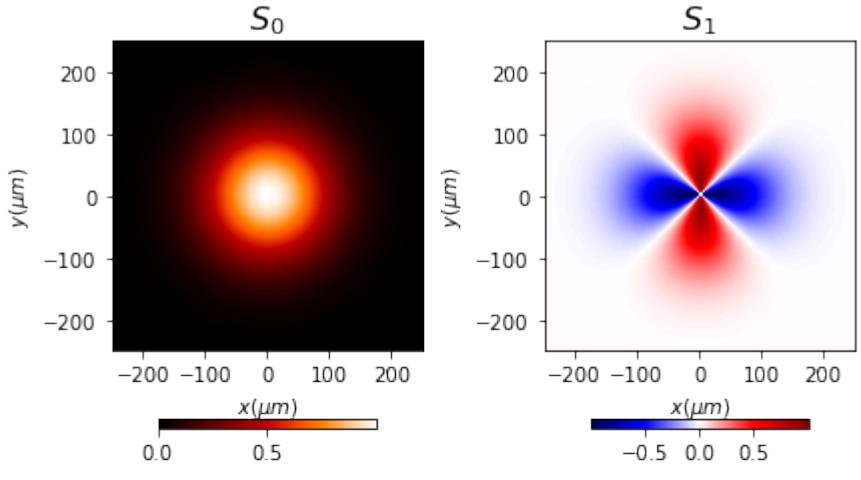
Here, let us see the Gauss beam example.

[20]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

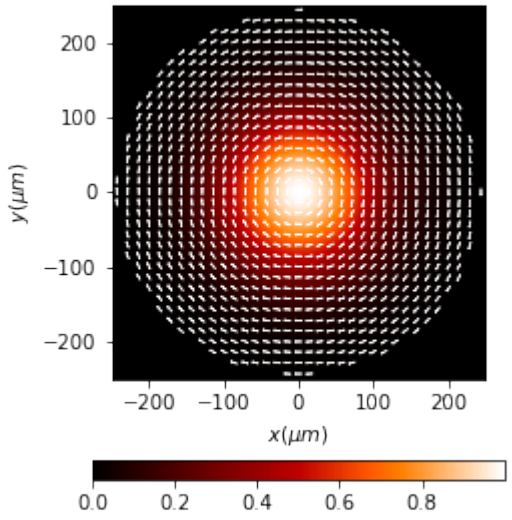
wavelength = 0.6328 * um

u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(150 * um, 150 * um),
              z0=0 * um,
              A=1,
              theta=0. * degrees,
              phi=0 * degrees)
```

[21]:
```
EM = Vector_source_XY(x0, y0, wavelength)
EM.azimuthal_wave(u=u0, r0=(0, 0), radius=250 * um)
EM.draw('stokes')
```



[22]:
```
EM.draw('ellipses', num_ellipses=(35, 35))
```



6.7.6.4 Vector wave from a scalar source

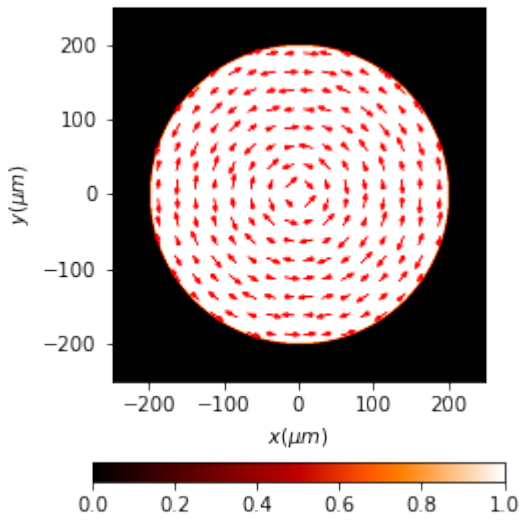
All these methods to provide vector polarization to a constant wave, can be used for any other scalar source. Then we include in the u parameters the scalar field:

[23]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1, theta=90 * degrees, phi=1 * degrees)
```

[24]:
```
EM = Vector_source_XY(x0, y0, wavelength)
EM.azimuthal_wave(u=u0, r0=(0, 0), radius=(200, 200))
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(20, 20),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=5)
```



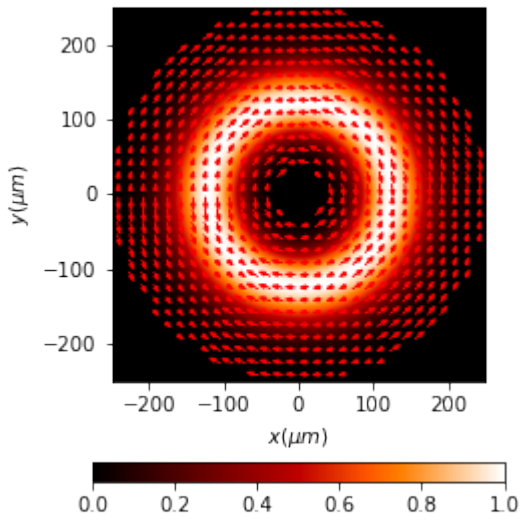
This can also been done, for example to a Gauss beam:

[25]:
```
x0 = np.linspace(-250 * um, 250 * um, 512)
y0 = np.linspace(-250 * um, 250 * um, 512)

wavelength = 0.6328 * um

u0 = Scalar_source_XY(x0, y0, wavelength)
u0.vortex_beam(A=1, r0=(0, 0), w0=100 * um, m=3)
```

[26]:
```
EM = Vector_source_XY(x0, y0, wavelength)
EM.azimuthal_wave(u0, r0=(0, 0))
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=None,
                     num_ellipses=(30, 30),
                     amplification=0.5,
                     color_line='r',
                     line_width=0.5,
                     draw_arrow=True,
                     head_width=5)
```



6.7.6.5 Gauss Polarization

Since Gauss beams are very used, we have defined the main polarization classes for these beams

[27]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
y0 = np.linspace(-200 * um, 200 * um, 512)
wavelength = 0.6328 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.gauss_beam(A=1,
              r0=(0 * um, 0 * um),
              w0=(50 * um, 50 * um),
              z0=0 * um,
              theta=0. * degrees,
              phi=0 * degrees)

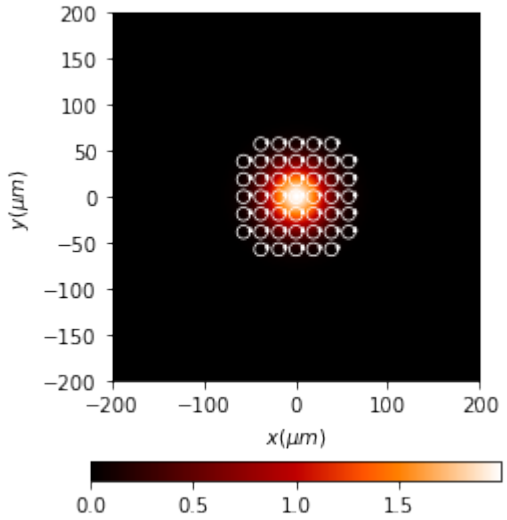
EM = Vector_source_XY(x0, y0, wavelength)
EM.constant_polarization(u1, v=(1, 1j))

EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
```

(continues on next page)

(continued from previous page)

```
                    cut_value=None,
                    num_ellipses=(21, 21),
                    amplification=0.75,
                    color_line='w',
                    line_width=.75,
                    draw_arrow=True,
                    head_width=3)
```



6.7.6.6 interferences

Vector fields also interfere. We can sum two vector beams using $E_M = E_{M1} + E_{M2}$

[28]:
```
length = 100 * um
num_data = 512
x0 = np.linspace(-length / 2, length / 2, num_data)
y0 = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328

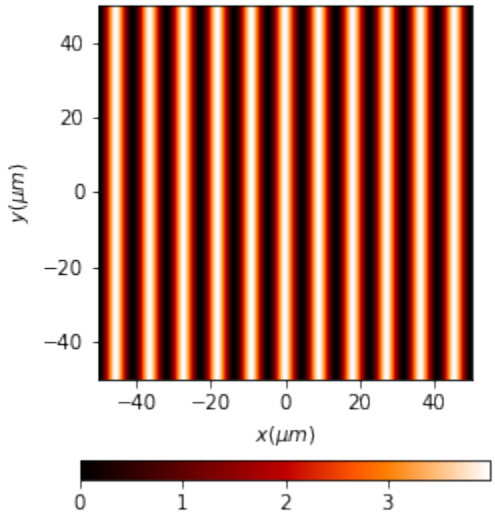
u1 = Scalar_source_XY(x0, y0, wavelength)
u1.plane_wave(A=1, theta=2 * degrees)

u2 = Scalar_source_XY(x0, y0, wavelength)
u2.plane_wave(A=1, theta=-2 * degrees)

EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.constant_polarization(u=u1, v=[1, 0])

EM2 = Vector_source_XY(x0, y0, wavelength)
EM2.constant_polarization(u=u2, v=[1, 0])

EM = EM1 + EM2
EM.draw(kind='intensity')
```



Obviously, when the two beams are ortogonal, no interference is produced:

[29]:
```
length = 100 * um
num_data = 512
x0 = np.linspace(-length / 2, length / 2, num_data)
y0 = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328

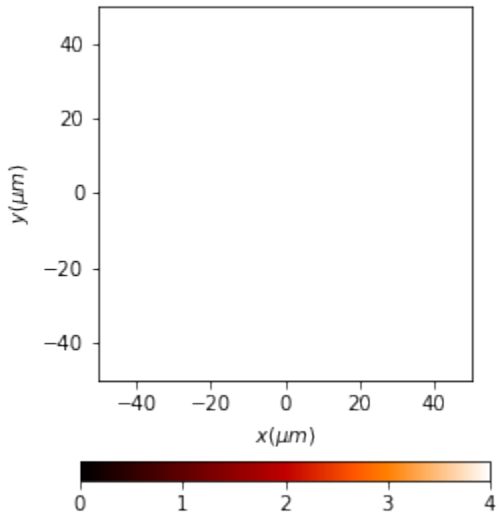
u1 = Scalar_source_XY(x0, y0, wavelength)
u1.plane_wave(A=1, theta=90 * degrees, phi=5 * degrees)

u2 = Scalar_source_XY(x0, y0, wavelength)
u2.plane_wave(A=1, theta=90 * degrees, phi=-5 * degrees)

EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.constant_polarization(u=u1, v=[1, 1j])

EM2 = Vector_source_XY(x0, y0, wavelength)
EM2.constant_polarization(u=u2, v=[1, -1j])

EM = EM1 + EM2
EM.draw(kind='intensity')
```



Partial polarization can also be possible.

[30]:
```
length = 100 * um
num_data = 512
x0 = np.linspace(-length / 2, length / 2, num_data)
y0 = np.linspace(-length / 2, length / 2, num_data)
wavelength = 0.6328

u1 = Scalar_source_XY(x0, y0, wavelength)
u1.plane_wave(A=1, theta=5 * degrees)

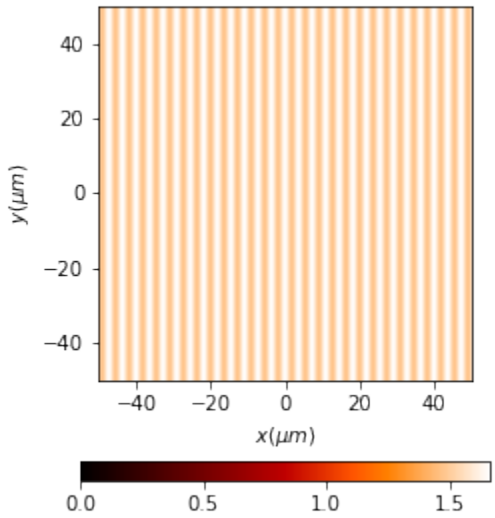
u2 = Scalar_source_XY(x0, y0, wavelength)
u2.plane_wave(A=1, theta=-5 * degrees)

EM1 = Vector_source_XY(x0, y0, wavelength)
EM1.constant_polarization(u=u1, v=[1, 0])

EM2 = Vector_source_XY(x0, y0, wavelength)
EM2.constant_polarization(u=u2, v=[0.05, .75])

EM = EM1 + EM2
EM.draw(kind='intensity')

intensity = EM.get(kind='intensity')
```



6.7.7 Example of masks

6.7.7.1 Creating an instance

An instance must be created before starting to operate with vector masks. The initialization accepts several arguments.

[1]:
```
from diffractio import np, plt, sp
from diffractio import degrees, mm, um
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.utils_tests import save_figure_test
from diffractio.vector_fields_XY import Vector_field_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_sources_XY import Vector_source_XY

from py_pol.jones_matrix import Jones_matrix
```

6.7.7.2 Polarization from standard elements

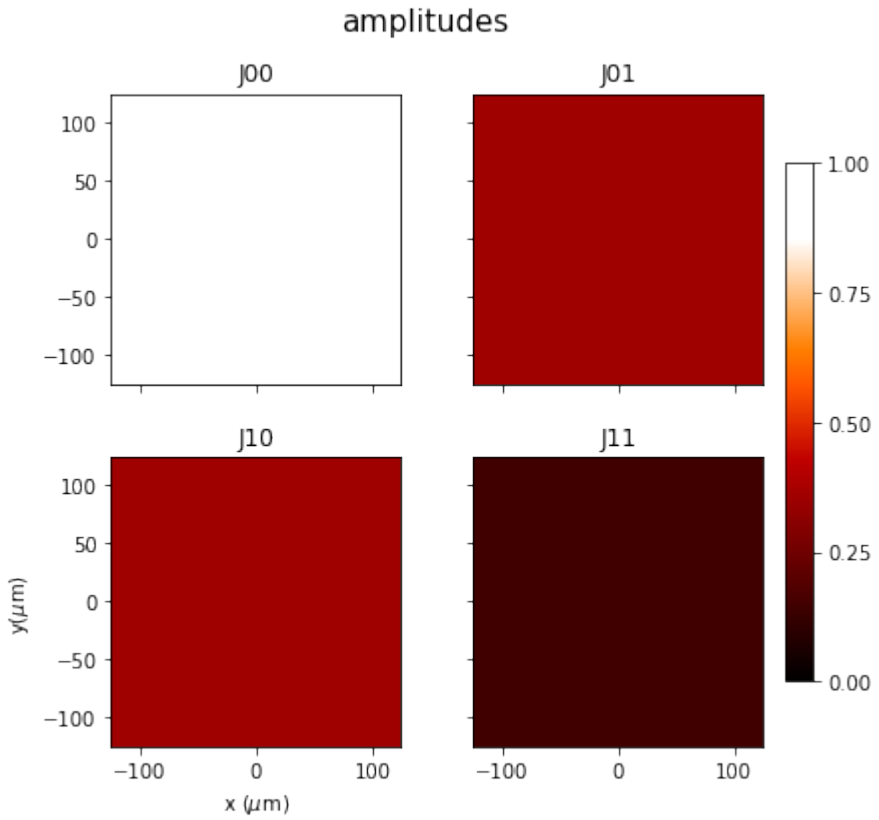
Using defined polarizers

There are several standard polarizers: - polarized_linear - quarter_waveplate - half_waveplate - polarizer_retarder

[2]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)
wavelength = 0.6328 * um

# mask vectorial
vc = Vector_mask_XY(x0, y0, wavelength)
vc.polarizer_linear(azimuth=22.5 * degrees)
vc.draw(kind='amplitudes')
```

```
<Figure size 432x288 with 0 Axes>
```



Using py_pol module

If more general polarizer is required, we han use py_pol module

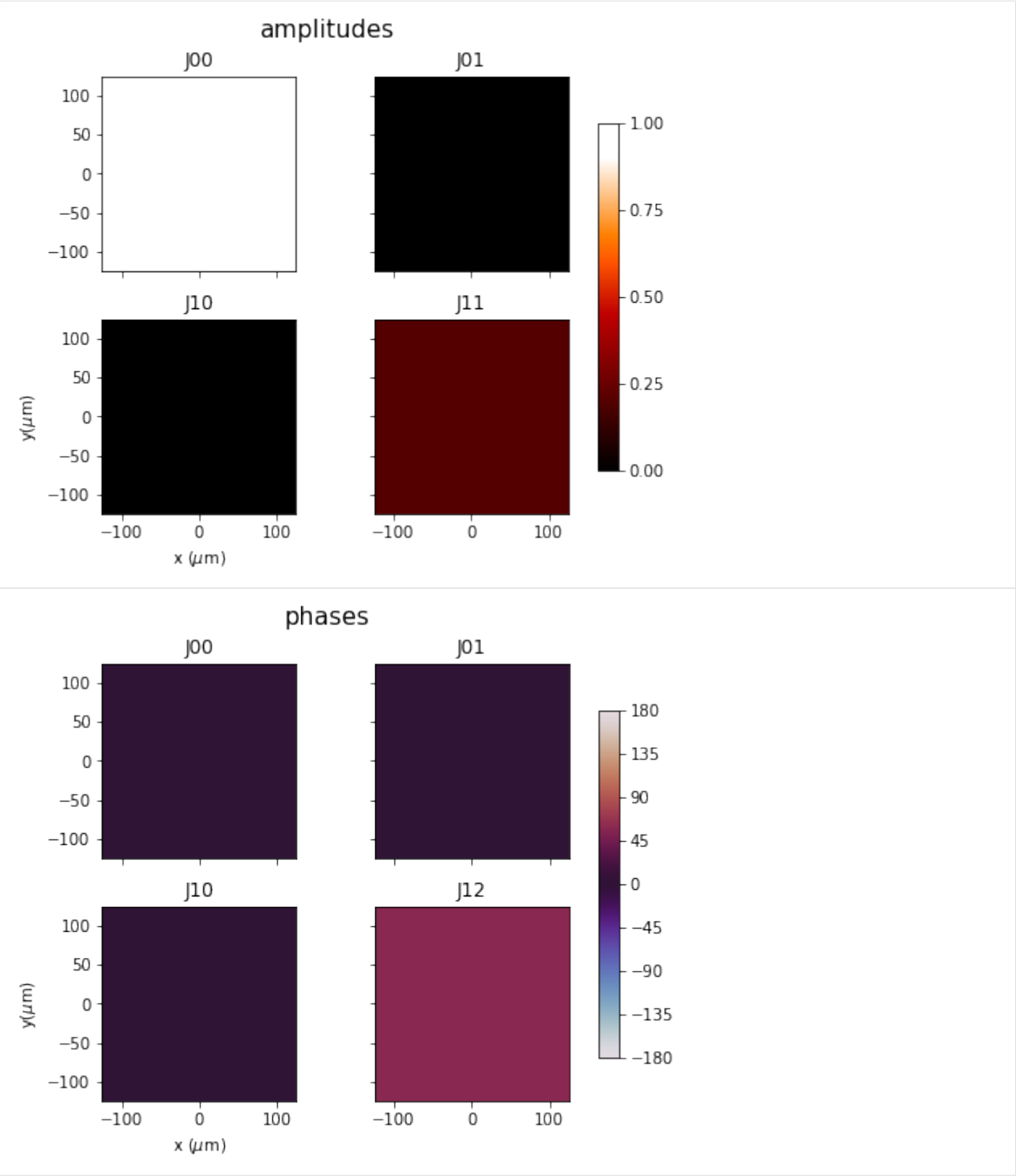
[3]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)
wavelength = 0.6328 * um

PL = Jones_matrix('m0')
PL.from_components([0.9, 0, 0, 0.2 * np.exp(1j)])

vp = Vector_mask_XY(x0, y0, wavelength)
vp.from_py_pol(PL)

vp.draw(kind='jones_ap')
```

```
<Figure size 432x288 with 0 Axes>
```


amplitudes
J00
J01
J10
J11
100
50
0
−50
−100
y(μm)
x (μm)
1.00
0.75
0.50
0.25
0.00
phases
J00
J01
J10
J12
180
135
90
45
0
−45
−90
−135
−180

6.7.7.3 Apply a mask to the polarizer

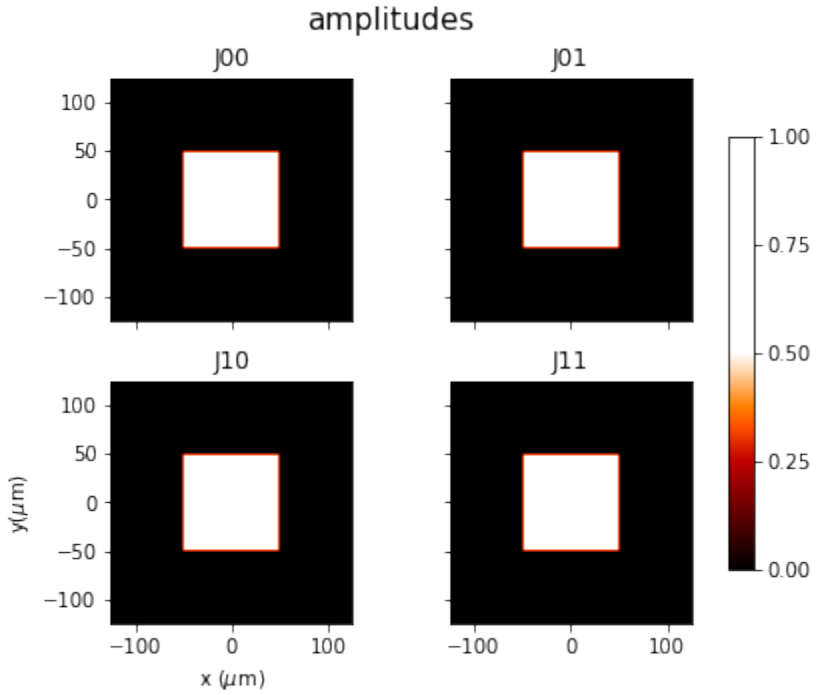
[4]:
```
j0 = Jones_matrix().diattenuator_perfect(azimuth=-45*degrees)
```

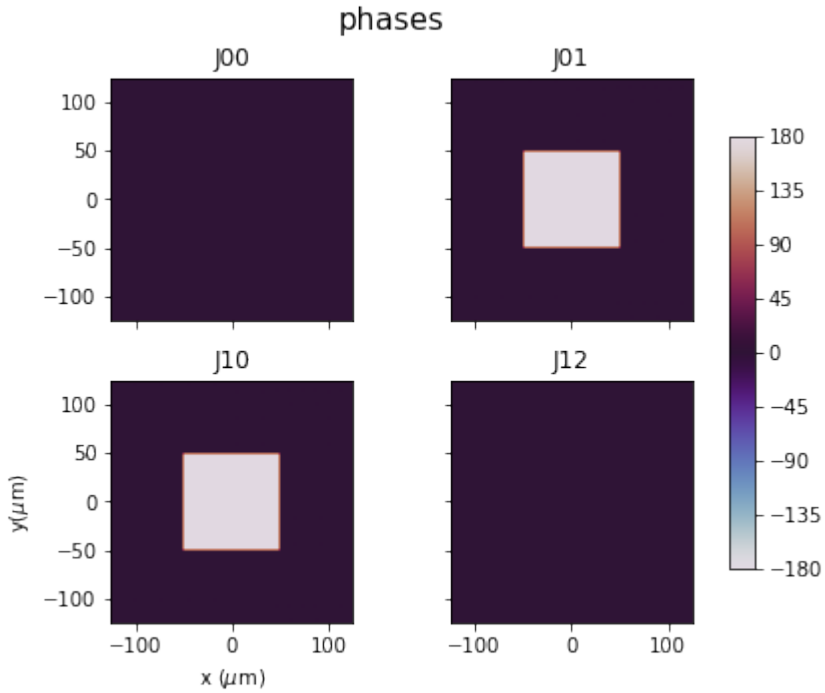
[5]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)
wavelength = 0.6328 * um

mask = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
mask.square(r0=(0 * um, 0 * um), size=(100 * um, 100 * um), angle=0 * degrees)

# mask vectorial
vc = Vector_mask_XY(x0, y0, wavelength)
vc.scalar_to_vector_mask(mask=mask, state=j0)
vc.draw(kind='jones_ap')
```

```
<Figure size 432x288 with 0 Axes>
```





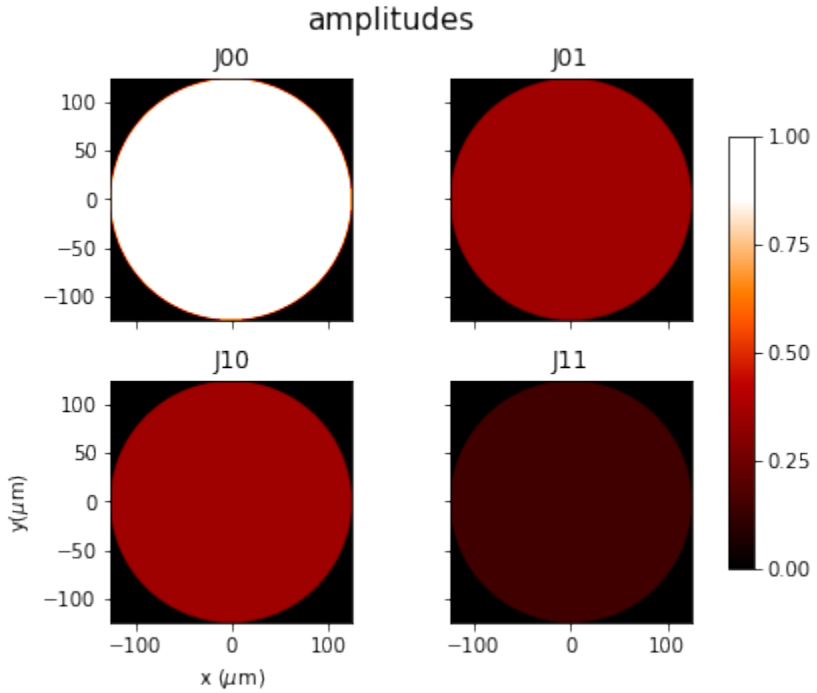
6.7.7.4 circle

Since normally a circular mask is used, a function for this is defined. Therefore, an Scalar_XY mask is not required.

[6]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)
wavelength = 0.6328 * um

# mask vectorial
vc = Vector_mask_XY(x0, y0, wavelength)
vc.polarizer_linear(azimuth=22.5 * degrees)
vc.apply_circle()
vc.draw(kind='amplitudes')
```

```
<Figure size 432x288 with 0 Axes>
```



6.7.7.5 Complementary mask

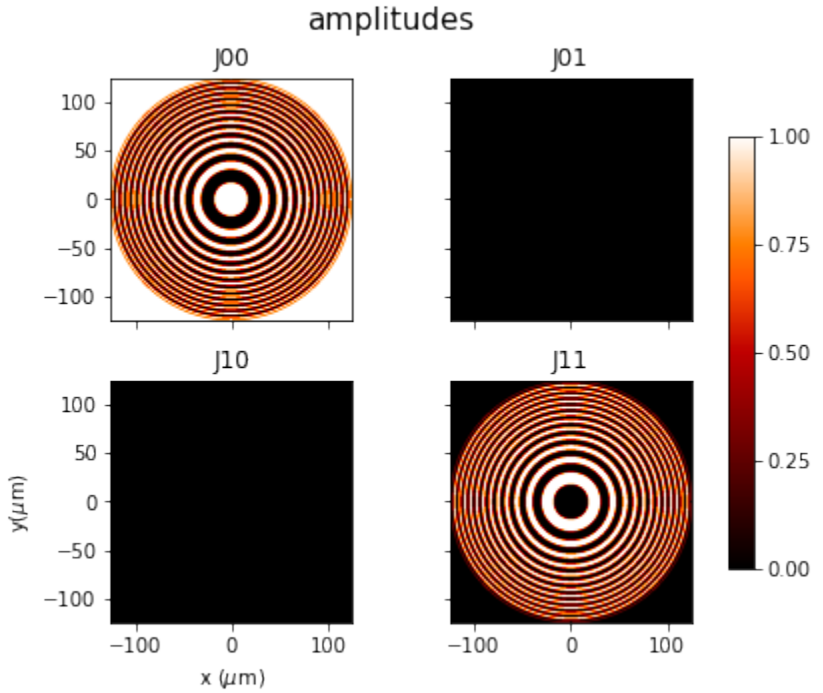
When we provide a binary mask, the 1 mask is converted to v1 polarization and 0 mask is converted to v2 polarization.

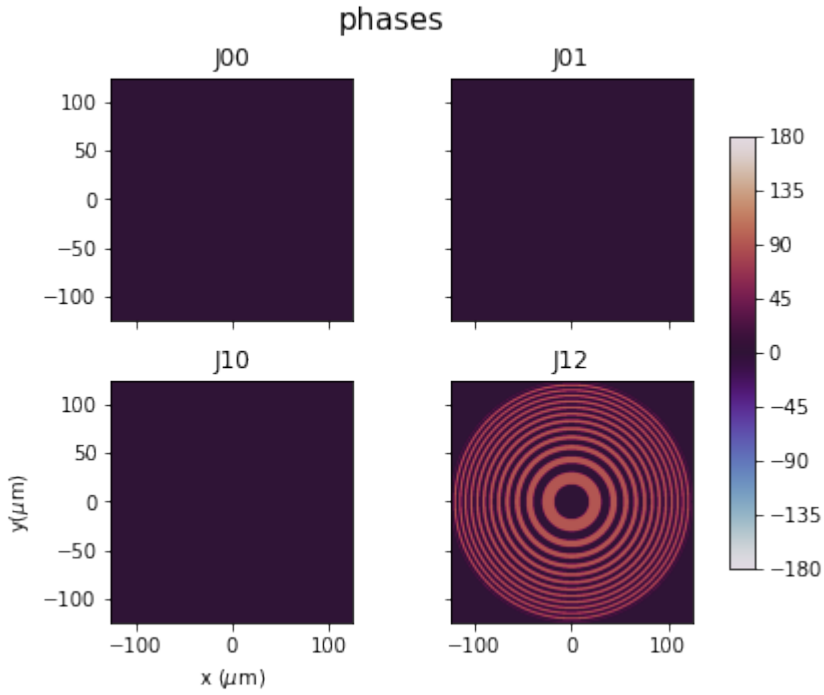
[7]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)
wavelength = 0.6328 * um

mask = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
mask.fresnel_lens(r0=(0 * um, 0 * um),
                  radius=(125 * um, 125 * um),
                  focal=(2 * mm, 2 * mm),
                  angle=0 * degrees,
                  kind='amplitudes')
mask.pupil()

EM0 = Vector_mask_XY(x0, y0, wavelength)
EM0.complementary_masks(mask=mask,
                        state_0=np.array([[1, 0], [0, 0]]),
                        state_1=np.array([[0, 0], [0, 1j]]))
EM0.draw(kind='jones_ap')
```

```
<Figure size 432x288 with 0 Axes>
```


amplitudes
J00
J01
J10
J11
100
50
0
−50
−100
y(μm)
x (μm)
1.00
0.75
0.50
0.25
0.00


phases
J00
J01
J10
J12
100
50
0
−50
−100
y(μm)
x (μm)
180
135
90
45
0
−45
−90
−135
−180

6.7.7.6 Pass to py_pol

[8]:
```
m0 = EM0.to_py_pol()
```

6.7.8 Drawing for vector fields

For vector fields, there are special drawings. Here we explain how to extract information from these drawings

[1]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_sources_XY import Vector_source_XY

from diffractio.utils_drawing import concatenate_drawings, draw_several_fields
```

[2]:
```
length = 500 * um
num_data = 256
wavelength = 0.6328 * um

x0 = np.linspace(-200 * um, 200 * um, 512)
y0 = np.linspace(-200 * um, 200 * um, 512)

u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(A=1, r0=(0, 0), w0=100 * um, z0=0)

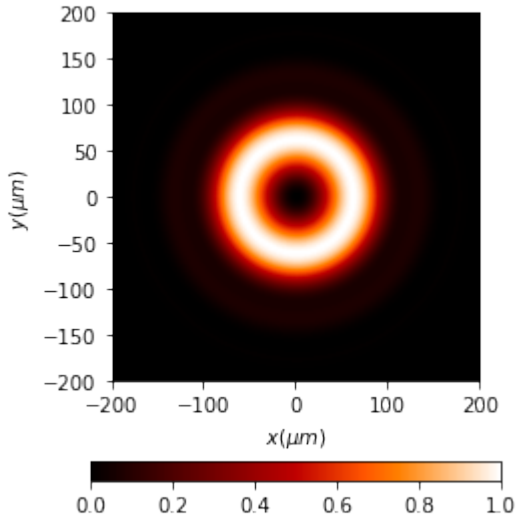
EM = Vector_source_XY(x0, y0, wavelength)
EM.azimuthal_wave(u=u0, r0=(0, 0), radius=(200, 200))
EM.VRS(z=10 * mm, new_field=False)
EM.normalize()

Ex, Ey, _ = EM.get()

EM.reduce_matrix = ''
```

Draw intensity

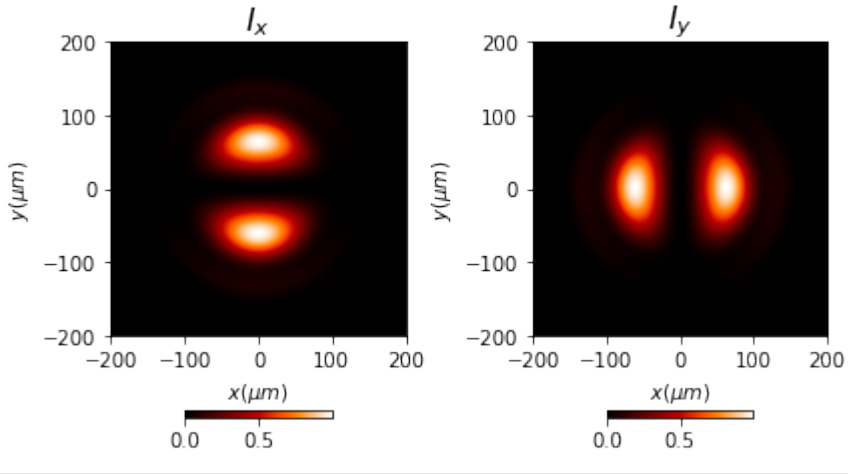
[3]:
```
EM.draw(kind='intensity')
```



Draw intensities

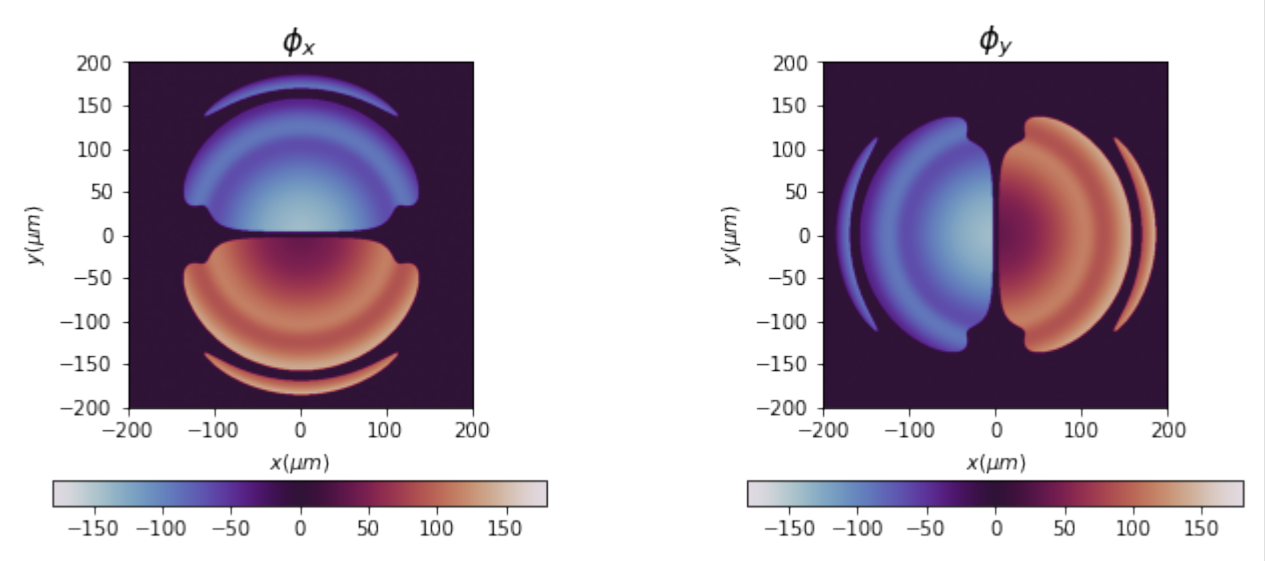
If the Ez field presents intensity, it is also shown

[4]:
```
EM.draw(kind='intensities')
```



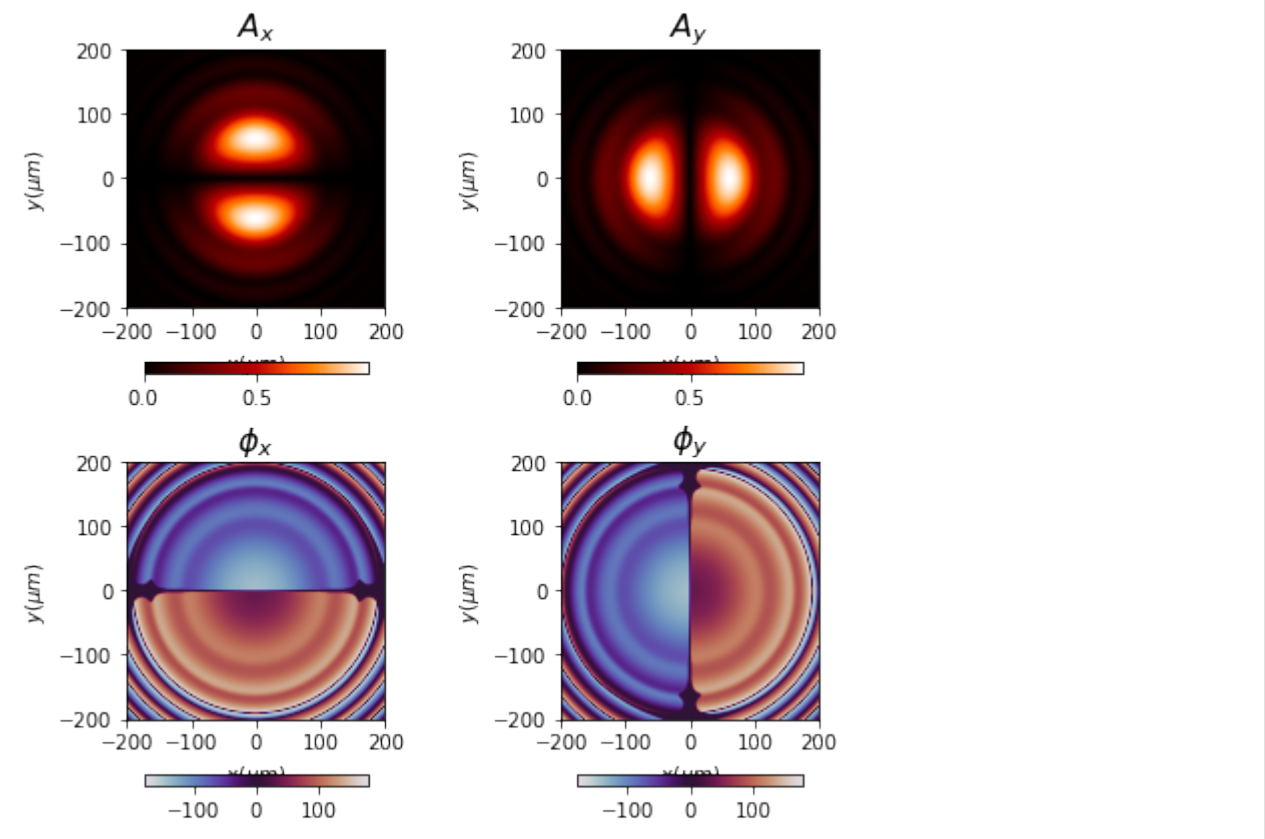
Draw phase

[5]:
```
EM.draw(kind='phases')
```



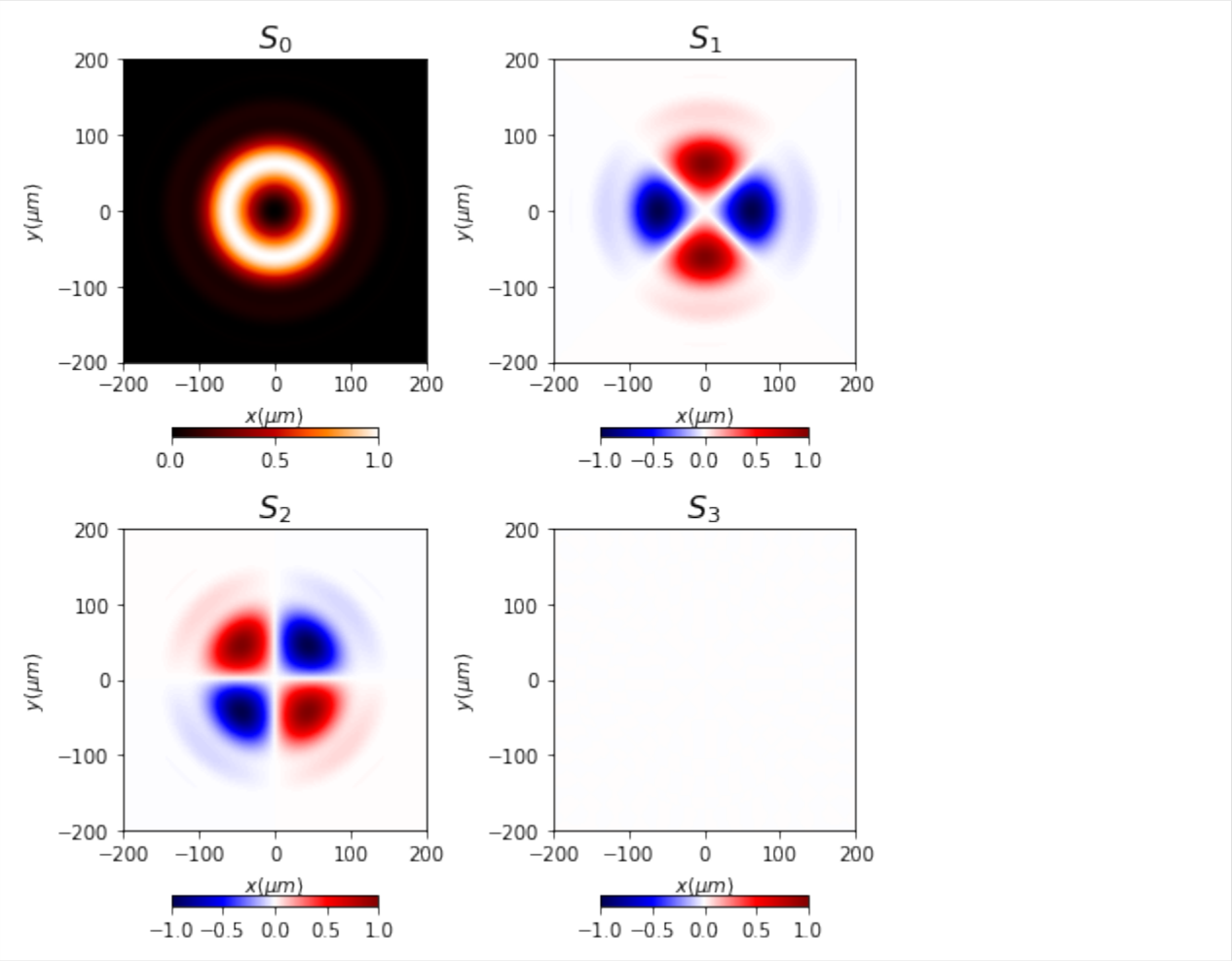
Draw field

[6]:
```
EM.draw(kind='fields')
```



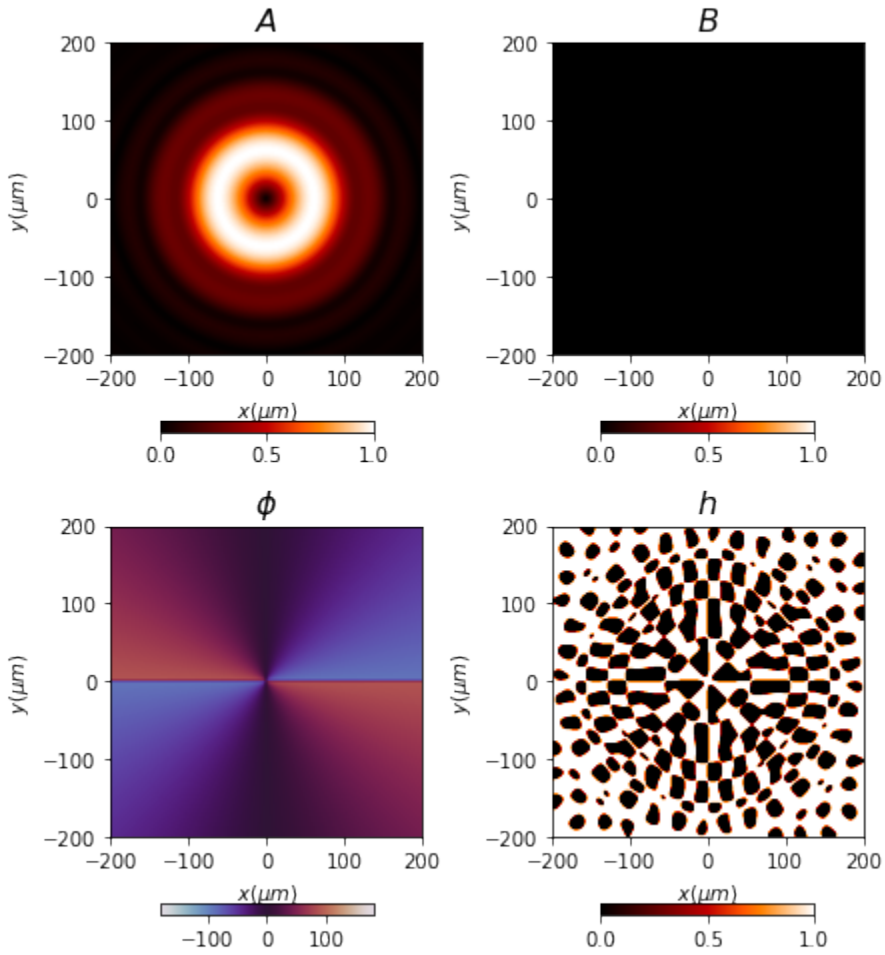
Draw Stokes

[7]:
```
EM.draw(kind='stokes')
```



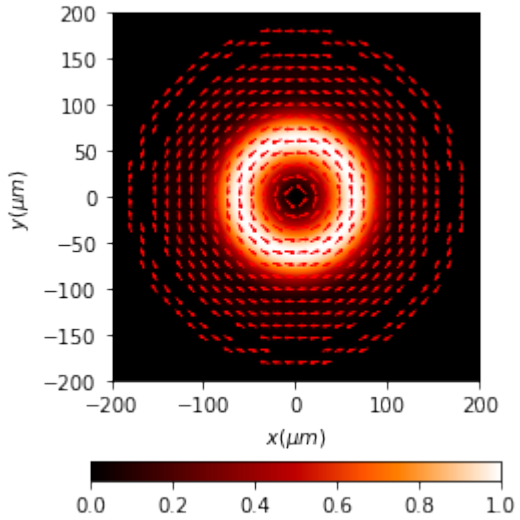
Draw polarization ellipses parameters

[8]:
```
EM.draw(kind='param_ellipse')
```



Draw intensity with polarization vectors

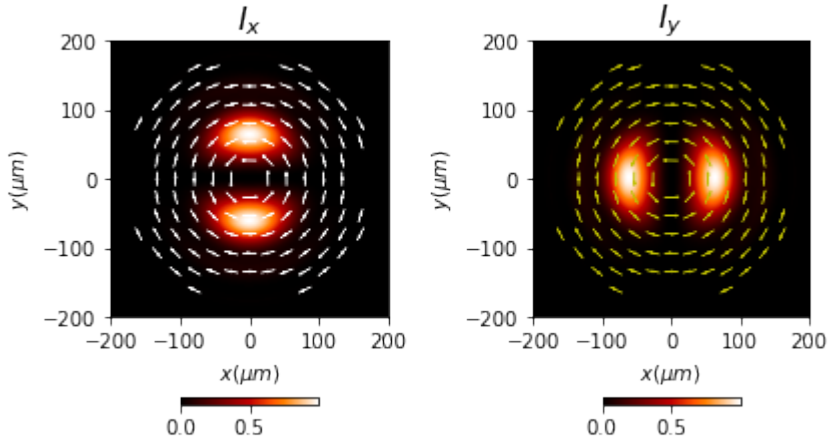
[9]:
```
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=False,
                     num_ellipses=(30, 30),
                     amplification=0.75,
                     color_line='r',
                     line_width=.5,
                     draw_arrow=True,
                     head_width=2,
                     ax=False)
```



This polarizaation ellipses can also be drawn at any other drawing, provided that we have the axis associated to this drawing

[11]:
```
h1, h2 = EM.draw(kind='intensities')
EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=False,
                     num_ellipses=(15, 15),
                     amplification=0.75,
                     color_line='w',
                     line_width=.5,
                     draw_arrow=True,
                     head_width=2,
                     ax=h1)

EM.__draw_ellipses__(logarithm=False,
                     normalize=False,
                     cut_value=False,
                     num_ellipses=(15, 15),
                     amplification=0.75,
                     color_line='y',
                     line_width=.5,
                     draw_arrow=True,
                     head_width=2,
                     ax=h2)
plt.tight_layout()
```



[ ]:

6.8 Other cases

6.8.1 Fresnel equations

Here, we check the functioning of Fresnel equations, that is, how a plane wave reflects and refracts on a plane surface. The refraction index of the media are n1 for the incicent media and n2 for the second media. The second media can be absorbent $n_c = n + i\kappa$.

For the computations, we have used two ways:

```
- fresnel_equations_kx: When the plane plane is defined as k = (kx, kz).

- fresnel_equations: When the plane wave is define by the angle.
```

We have also methods to determine the reflectance and transmittance.

```
- transmitances_reflectances_kx

- transmitances_reflectances
```

One of the parameters of the funcions is 'ouput'. It admits a list of 4 booleans to check whether we want to compute this parameter or not. The order is:

```
- t_TM, t_TE, r_TM, r_TE
```

Also, inside each method, we can obtain the drawing of the Fresnel equation or the reflectance. The parameters are:

```
- has_draw: boolean
- kind: 'amplitude_phase' or real_imag
```

[1]:
```
%load_ext autoreload
%autoreload 2
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

from diffractio import um, mm, degrees
from diffractio.utils_optics import fresnel_equations_kx, fresnel_equations,␣
↪transmitances_reflectances_kx, transmitances_reflectances
```

6.8.1.1 Fresnel with kx

We introduce the vector kx, which is useful mainly for algorithm programming.

[3]:
```
wavelength = 0.6328*um
n1 = 1
n2 = 1.5

n2c = 1 + 5j

k0 = 2 * np.pi / wavelength

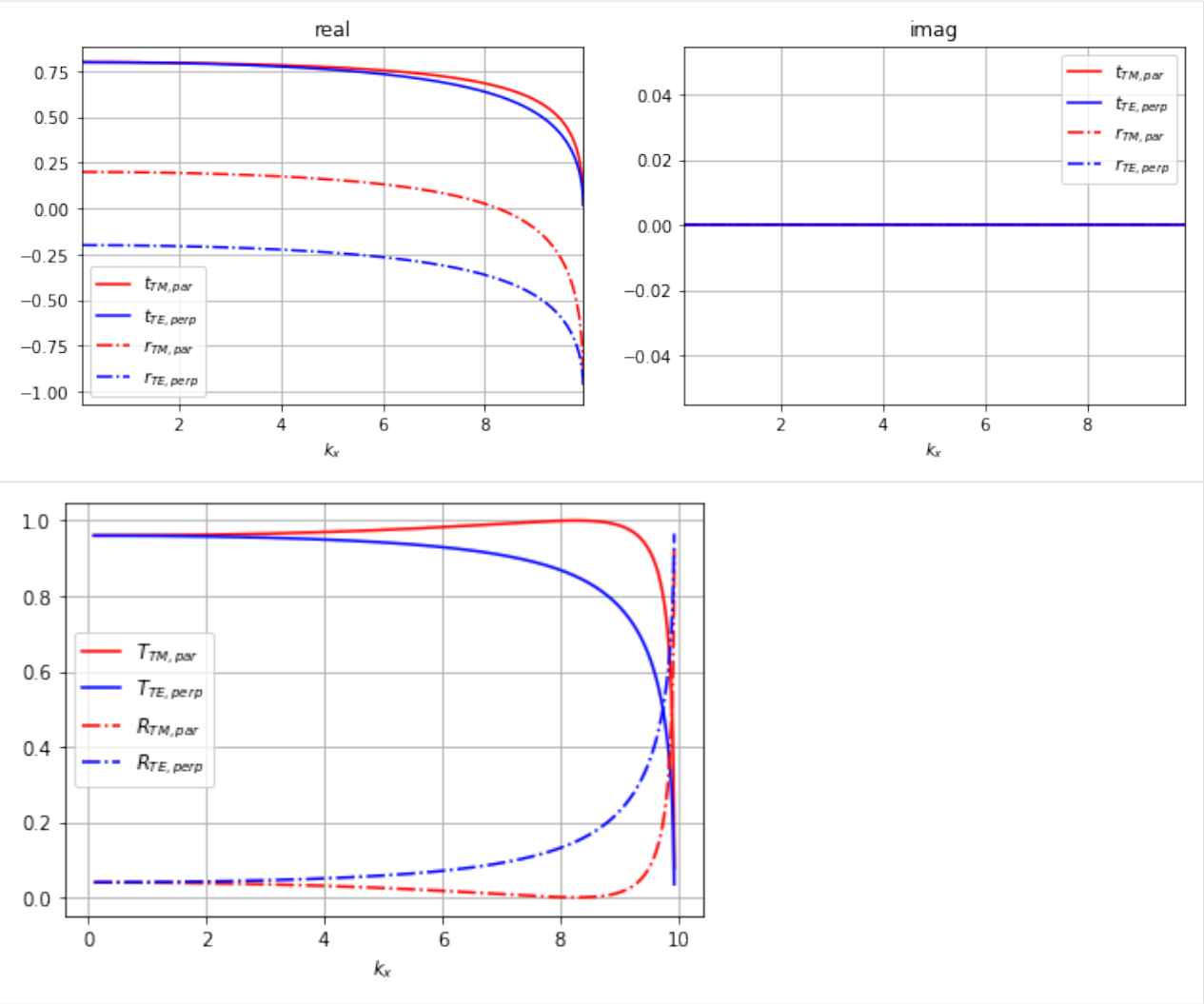

theta = np.linspace(0.01, np.pi/2-0.01, 361)
```

[4]:
```
# Here is the kx parameter, which need to be computed in the input side.

kx = k0 * n1 * np. sin(theta)


t_TM, t_TE, r_TM, r_TE = fresnel_equations_kx(kx,wavelength,  n1, n2, has_draw=True,␣
↪kind='real_imag')
t_TM, t_TE, r_TM, r_TE = transmitances_reflectances_kx(kx, wavelength, n1, n2, has_
↪draw=True)
```

```
/home/luismiguel/.local/lib/python3.10/site-packages/matplotlib/cbook/__init__.py:1298:␣
↪ComplexWarning: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



6.8.1.2 Fresnel with angles θ

This case is more suitable to understand what happens at the boundaries of a surface

[5]:
```
wavelength = 0.6328*um
n1 = 1
n2 = 1.5

n2c = 1 + 5j

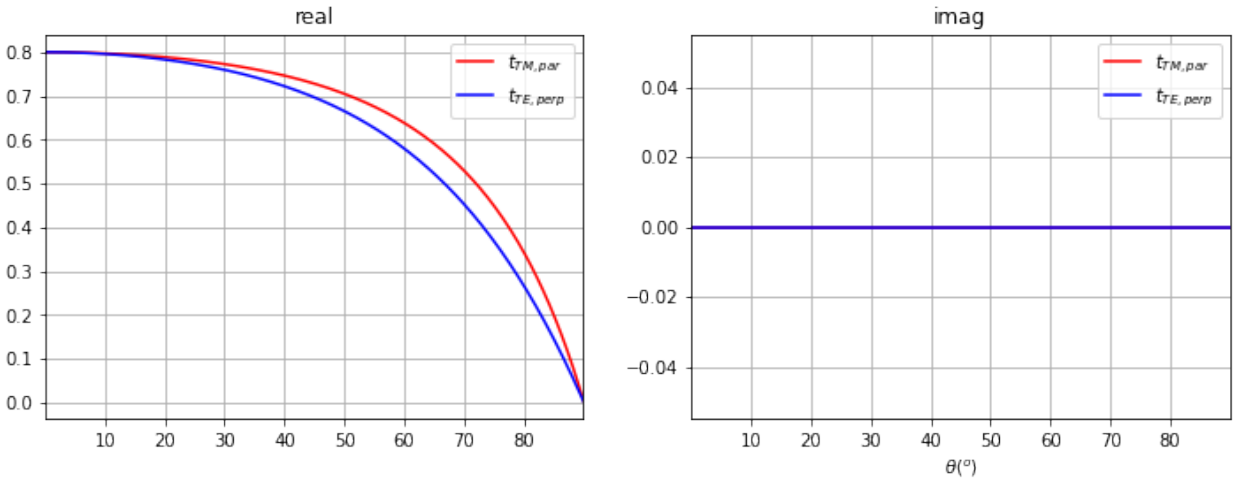
k0 = 2 * np.pi / wavelength

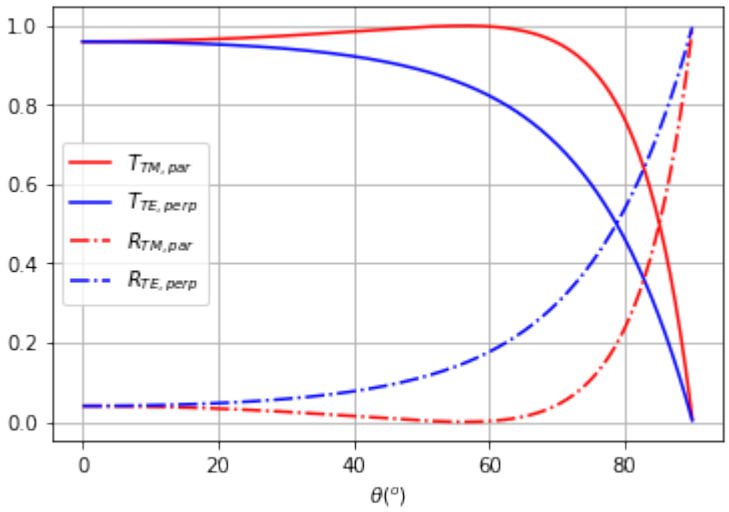

theta = np.linspace(0.001, np.pi/2-0.001, 361)
```

$n_2 > n_1$

[6]:
```
t_TM, t_TE, r_TM, r_TE = fresnel_equations(theta, wavelength, n1, n2, [1,1,0,0], has_
↪draw=True, kind='real_imag')

T_TM, T_TE, R_TM, R_TE = transmitances_reflectances(theta ,wavelength, n1, n2, has_
↪draw=True)
```



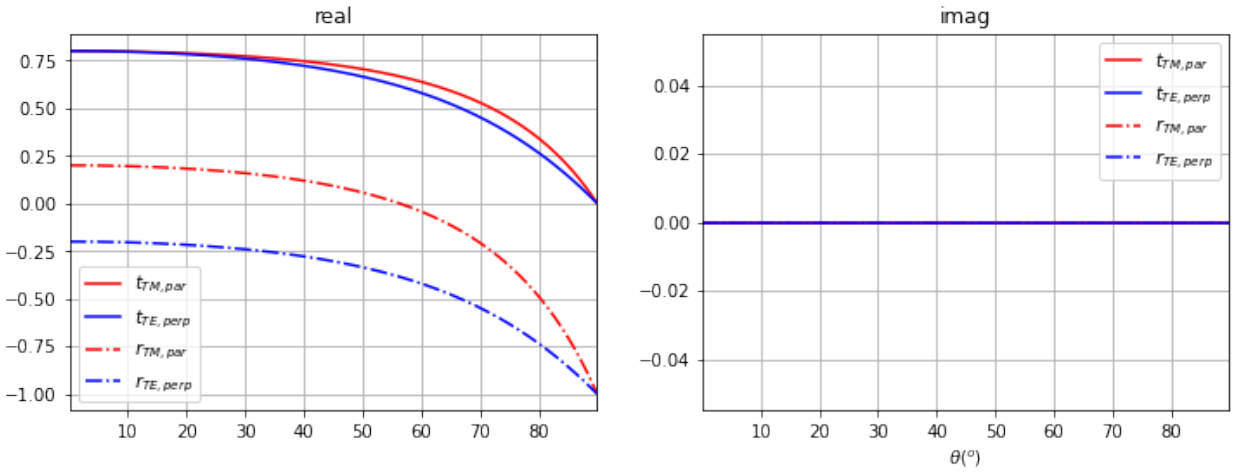


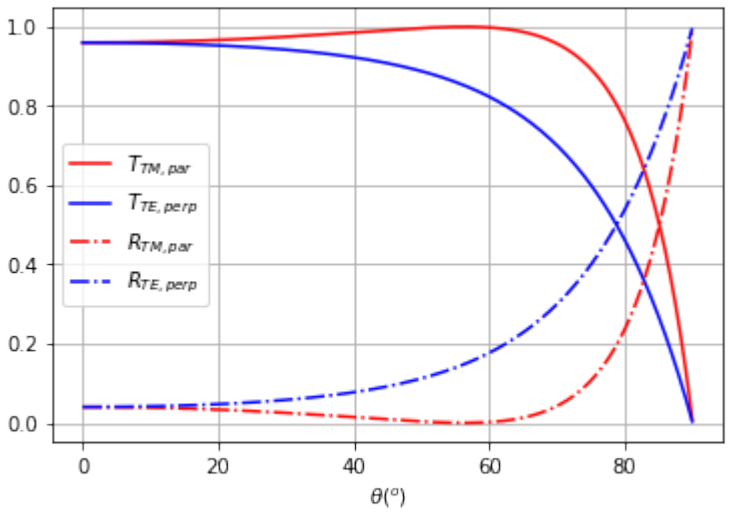
We can se Brewster angle.

$n_2 < n_1$

[7]:
```
t_TM, t_TE, r_TM, r_TE = fresnel_equations(theta, wavelength, n1, n2, has_draw=True,␣
↪kind='real_imag')

T_TM, T_TE, R_TM, R_TE = transmitances_reflectances(theta ,wavelength, n1, n2, has_
↪draw=True)
```

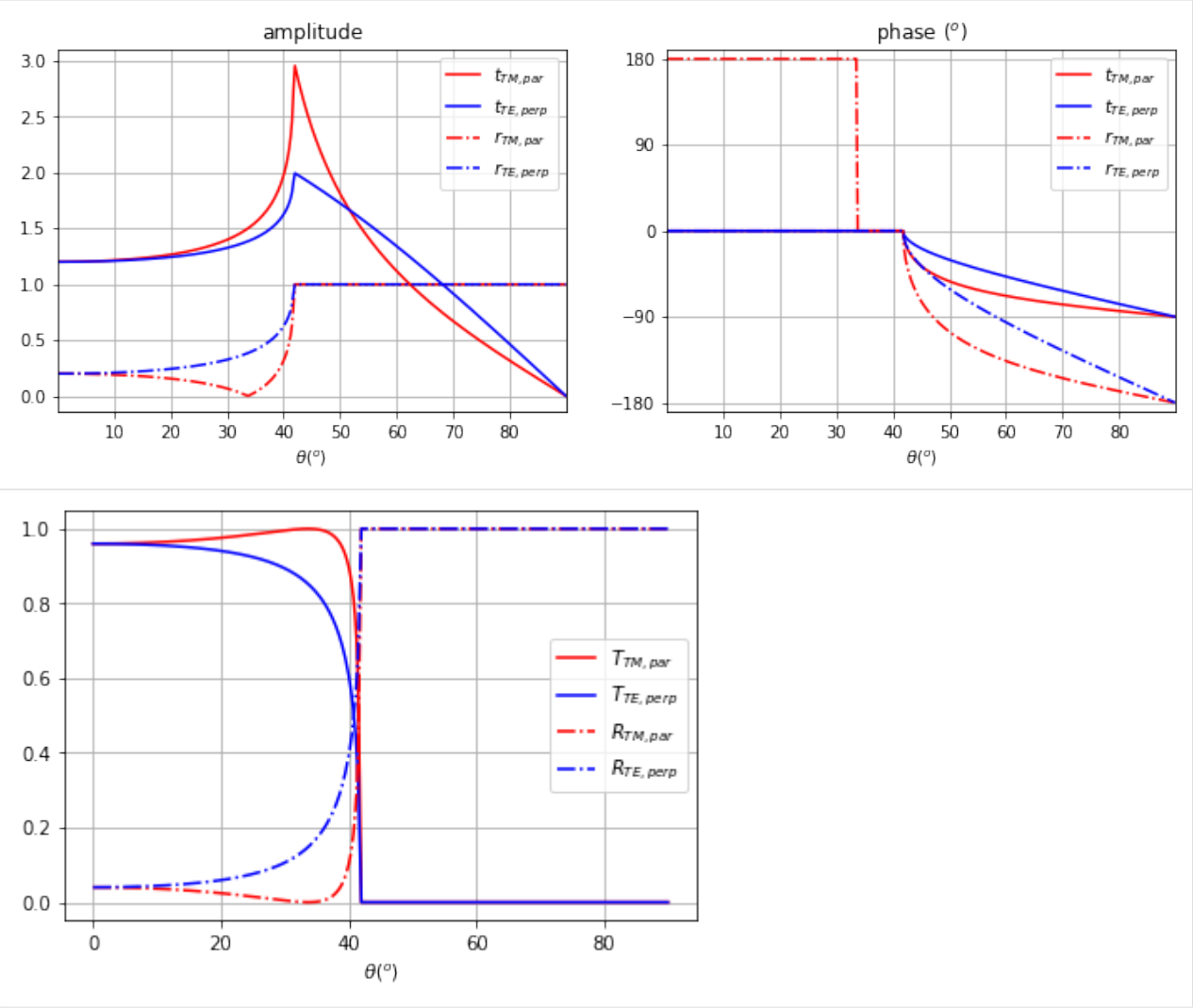




$n_2 > n_1$

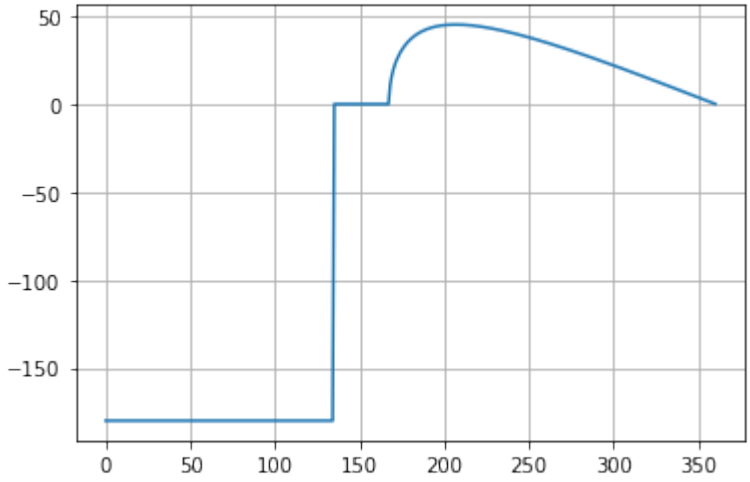
Here total internal refraction is clear. i nthe amplitude/phase representation.

[8]:
```
t_TM, t_TE, r_TM, r_TE = fresnel_equations(theta ,wavelength, n2, n1, has_draw=True,␣
↪kind='amplitude_phase')
T_TM, T_TE, R_TM, R_TE = transmitances_reflectances(theta ,wavelength,  n2, n1, has_
↪draw=True)
```



Phase difference between the two polarizations

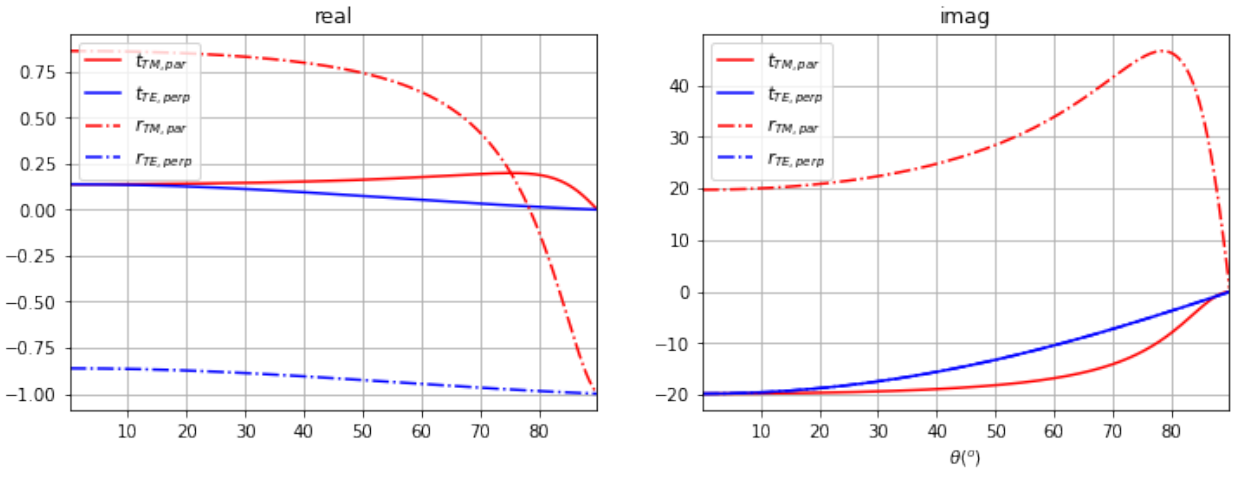
[9]:
```
plt.plot((np.angle(r_TE)-np.angle(r_TM))/degrees)
plt.grid()
```

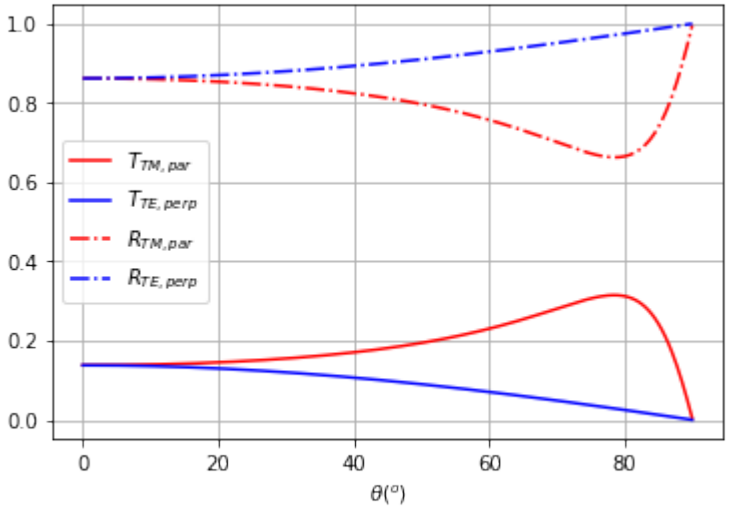


$n_2 = n + i\kappa$

This case is when the second medium present a complex refractive index, as in metals.

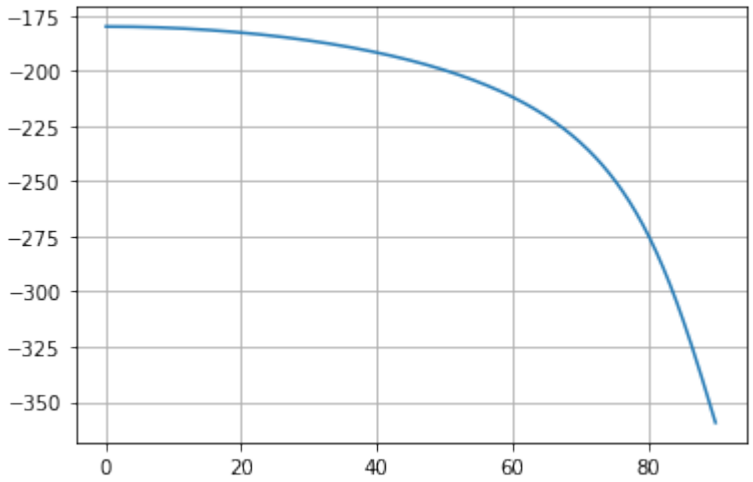
[10]:
```
t_TM, t_TE, r_TM, r_TE = fresnel_equations(theta, wavelength, n1, n2c, has_draw=True,␣
↪kind='real_imag')
T_TM, T_TE, R_TM, R_TE = transmitances_reflectances(theta, wavelength,  n1, n2c, has_
↪draw=True)
```





Phase difference between the two polarizations

[11]:
```
plt.plot(theta/degrees, (np.angle(r_TE)-np.angle(r_TM))/degrees)
plt.grid()
```

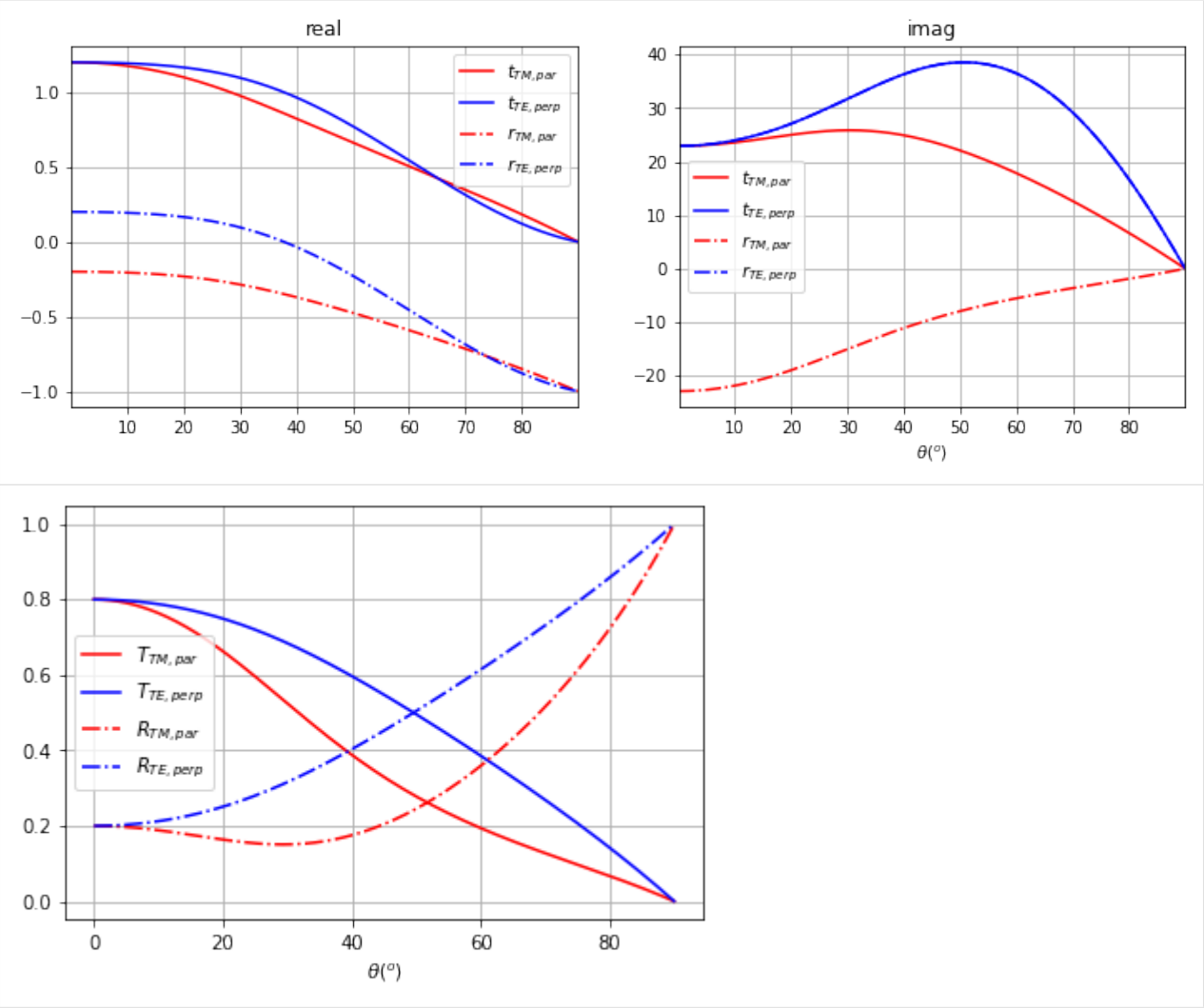


$n_1 = n + i\kappa$

This case is more strange, as the first medium is complex, it absorbs the light field, but it may be interesing when it is a thin metallic layer.

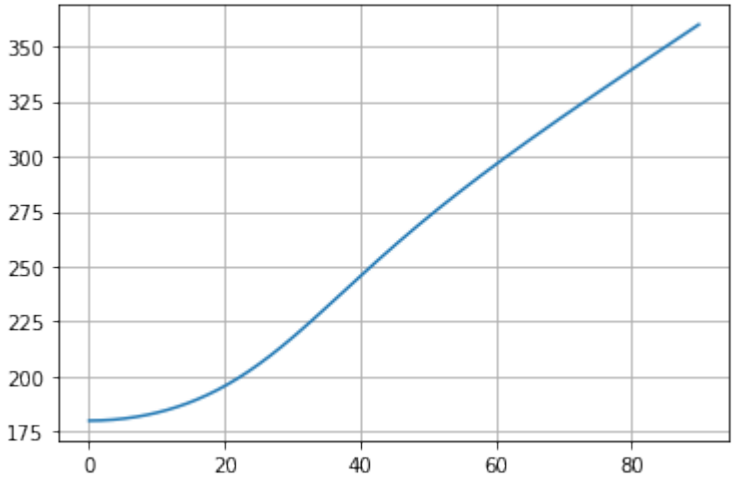
[12]:
```
n2c = 1 + 1j
```

[13]:
```
t_TM, t_TE, r_TM, r_TE = fresnel_equations(theta, wavelength, n2c, n1, has_draw=True,␣
↪kind='real_imag')
T_TM, T_TE, R_TM, R_TE = transmitances_reflectances(theta, wavelength,  n2c, n1, has_
↪draw=True)
```



Phase difference between the two polarizations

[14]:
```
plt.plot(theta/degrees, (np.angle(r_TE)-np.angle(r_TM))/degrees)
plt.grid()
```



6.9 Some info about drawing and videos

6.9.1 Drawing in logarithm scale

When showing intensity distribution, sometimes we need to analyze tiny intensity distributions that are hidden by a high intensity peak. Most of methods for visualization of intensity, present the *logarithm* parameter, which can be False, True, or a number. the function used to increase the color scale is:

```
u = np.log(logarithm * u + 1).
```

We show two examples where this parameter is important: The intensity distribution at the focal distance from a len, and the far field diffraction pattern of a small square aperture.

[1]:
```
from diffractio import degrees, mm, um, nm
from diffractio import np, plt, sp

from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

6.9.1.1 Intensity at the focal distance from a lens

[2]:
```
radius = 3 * mm

x0 = np.linspace(-radius, radius, 512)
y0 = np.linspace(-radius, radius, 512)
xout = np.linspace(-100 * um, 100 * um, 128)
yout = np.linspace(-100 * um, 100 * um, 128)

wavelength = 550 * nm

focal = 250 * mm
```

(continues on next page)

(continued from previous page)

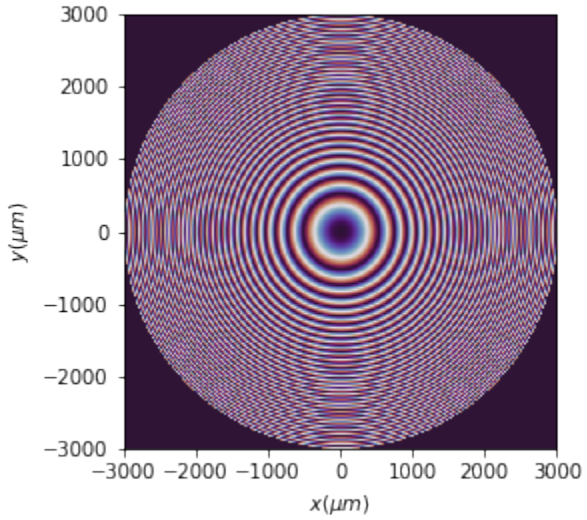
```
z = focal
```

[3]:
```
t0 = Scalar_mask_XY(x0, y0, wavelength)
t0.lens(r0=(0, 0), focal=focal, radius=radius)

u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)

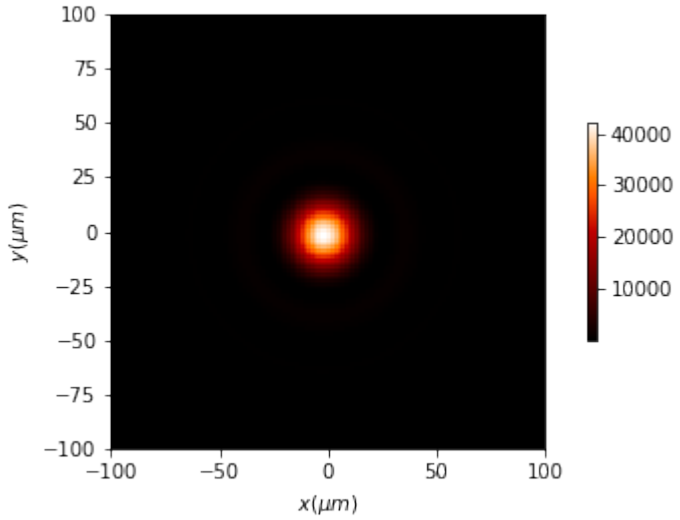
u1 = t0 * u0
```

[4]:
```
u1.draw('phase')
```

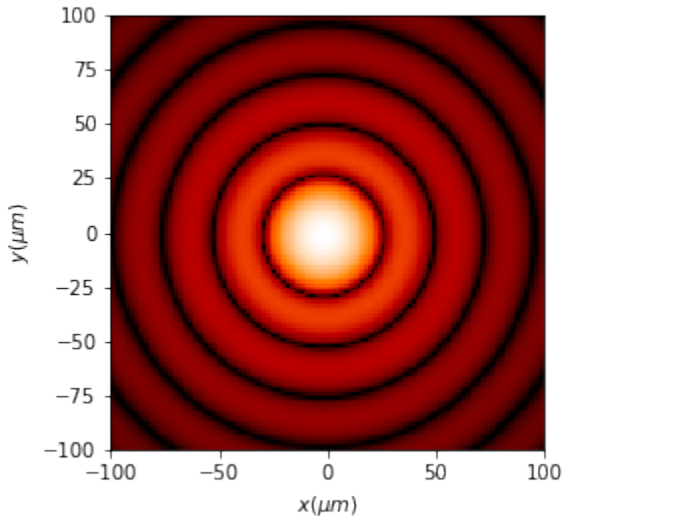


[5]:
```
u2 = u1.CZT(z, xout, yout)
```

[6]:
```
u2.draw(has_colorbar='vertical')
```



[7]:
```
u2.draw(logarithm=1)
```



6.9.1.2 Far field diffraction pattern of a small square aperture

[8]:
```
radius = 3 * mm

x0 = np.linspace(-radius, radius, 1024)
y0 = np.linspace(-radius, radius, 1024)
xout = np.linspace(-100 * um, 100 * um, 128)
yout = np.linspace(-100 * um, 100 * um, 128)
```

[9]:
```
focal = 2 * mm
z = focal

t0 = Scalar_mask_XY(x0, y0, wavelength)
```

(continues on next page)

(continued from previous page)

```
t0.lens(r0=(0, 0), focal=focal, radius=radius)

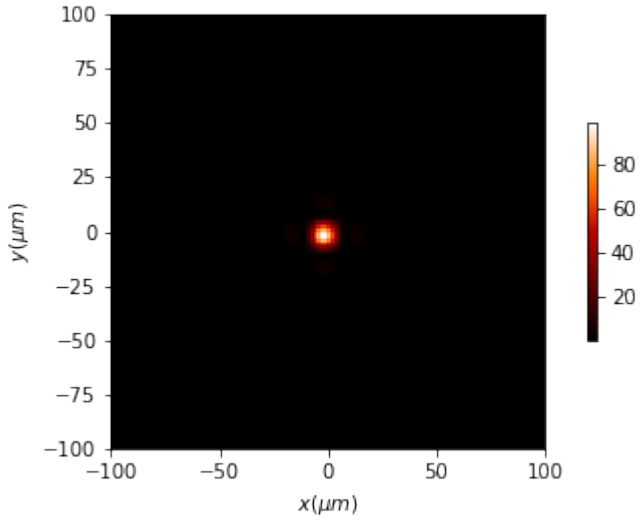
t1 = Scalar_mask_XY(x0, y0, wavelength)
t1.square(r0=(0, 0), size=100 * um, angle=0)

u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave()

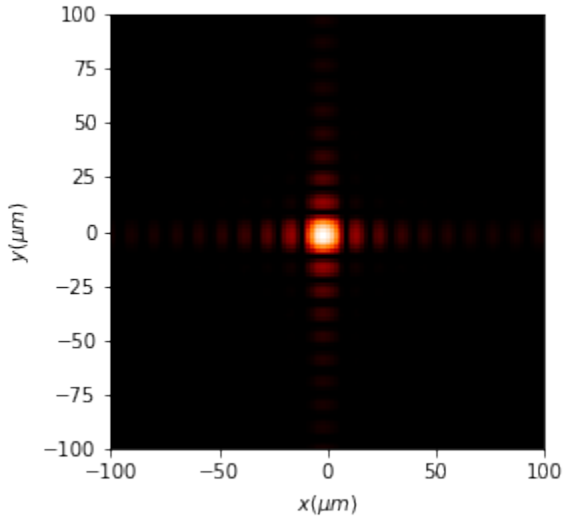
u1 = t0 * t1 * u0
```

[10]:
```
u2 = u1.CZT(z, xout, yout)
```

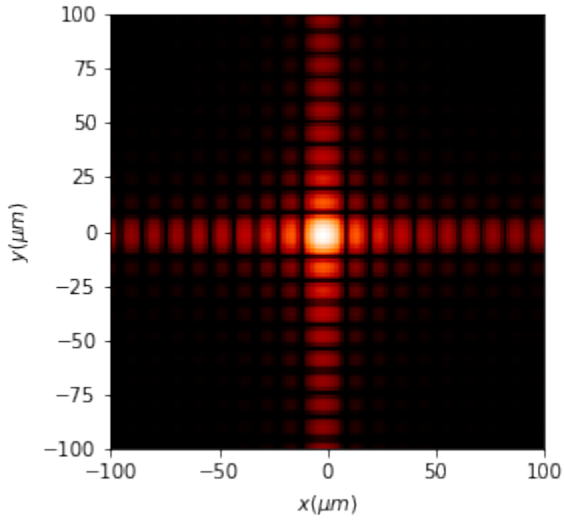
[12]:
```
u2.draw(has_colorbar='vertical')
```



[13]:
```
u2.draw(logarithm=1)
```



[14]:
```
u2.draw(logarithm=1e2)
```



6.9.2 Development of video using Matplotlib

[2]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from numpy.fft import fft, ifft
from scipy.signal import convolve2d
```

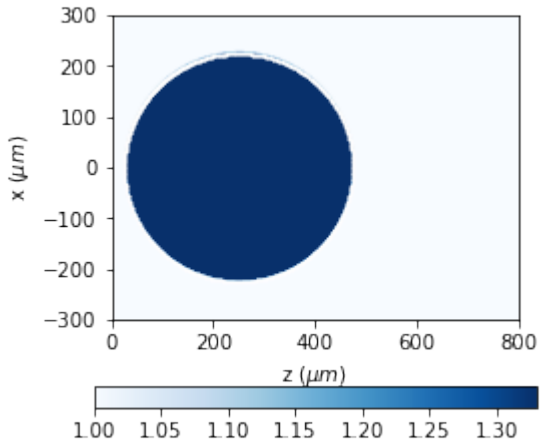
6.9.2.1 Generates a XZ field and show plots for each z

[3]:
```
# Initial parameters
x0 = np.linspace(-300 * um, 300 * um, 512)
z0 = np.linspace(-0.125 * um, 800 * um, 1024)
wavelength = 0.6238 * um * 10

# Definition of source
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1)
```

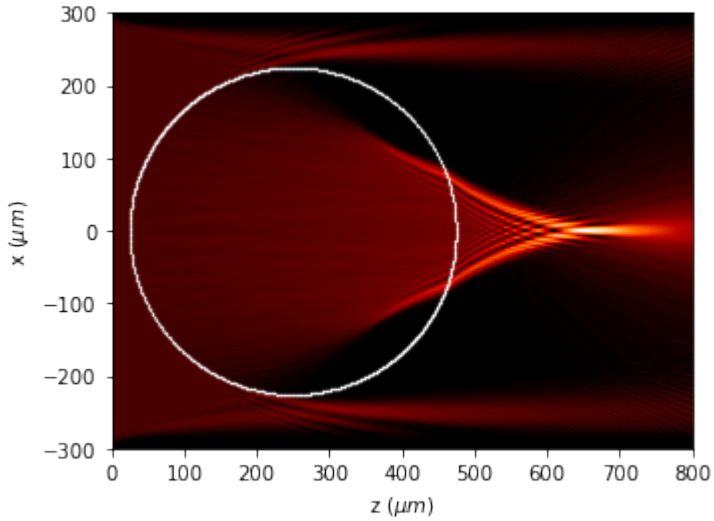
[4]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
u1.incident_field(u0)
```

[5]:
```
# insert sphere
u1.sphere(r0=(0, 250 * um),
          radius=(225 * um, 225 * um),
          refractive_index=1.33,
          angle=0)
u1.draw_refractive_index(scale='scaled', colorbar_kind='horizontal')
```



[6]:
```
# propagation
u1.WPM(verbose=False, has_edges=True)
```

[7]:
```
u1.draw(logarithm=True, scale='scaled', draw_borders=True)
```



[8]:
```
seconds = 1.
u1.video(kind='intensity',
         logarithm=True,
         normalize=False,
         time_video=10 * seconds,
         frames_reduction=5,
         filename='creating_video.mp4',
         dpi=100)
```

[9]:
```
%%HTML
<div align="middle">
```

(continues on next page)

(continued from previous page)

```
<video width="100%" controls>
      <source src="creating_video.mp4" type="video/mp4">
</video></div>
```

```
<IPython.core.display.HTML object>
```

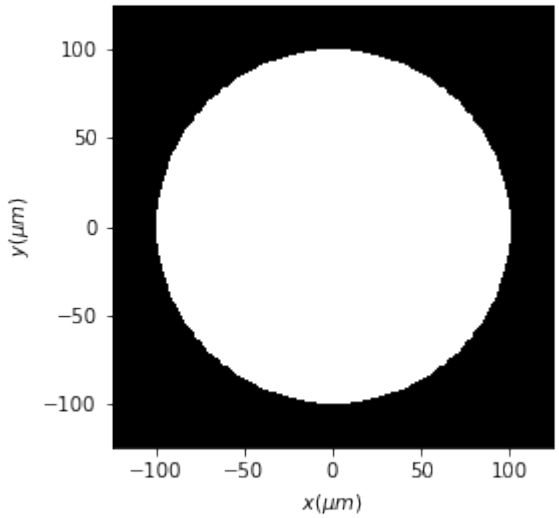
6.9.2.2 Generates a XYZ field and show plots XY for each z

[10]:
```
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

[16]:
```
wavelength0 = 0.6238 * um

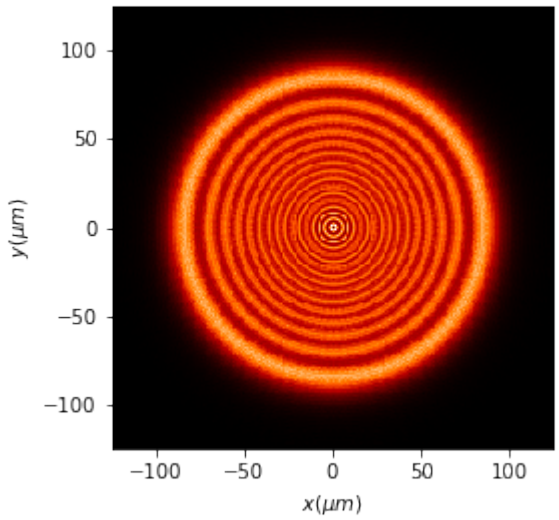
size = 250 * um
num_data = 256
x0 = np.linspace(-size / 2, size / 2, num_data)
y0 = np.linspace(-size / 2, size / 2, num_data)
z0 = np.linspace(500*um, 1.5*mm, 64)
```

[17]:
```
radius = 100 * um
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength0)
t1.circle(r0=(0 * um, 0 * um),radius=radius, angle=0 * degrees)
t1.draw()
```



[21]:
```
u2=t1.RS(z=500*um, verbose=True)
u2.draw()
```

```
Good result: factor 1.34
```



[19]:
```
u_xyz = Scalar_field_XYZ(x0, y0, z0, wavelength0)
u_xyz.incident_field(t1)
u_xyz.RS(num_processors=4)
```

```
time in RS= 14.639429330825806. num proc= 4
```

[20]:
```
u_xyz.video('video_circle.mp4', axis='auto', fps=10)
```

6.9.3 Interactive visualization of propagation of light in Jupyter notebook

Matplotlib present useful visualization tools for showing the intensity profiles at a distance z in an interactive fashion in jupyter notebooks. Here, we show how to use it.

In the first place, we well determine the intensity distribution XZ for an object.

[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from numpy.fft import fft, ifft
from scipy.signal import convolve2d

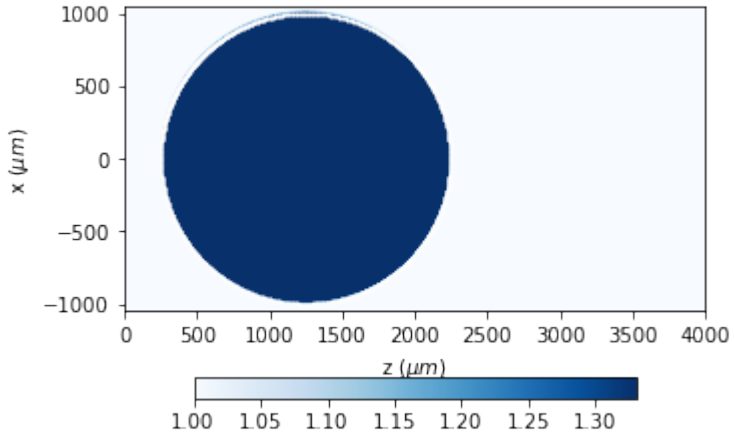
from ipywidgets import interactive
```

[2]:
```
# Initial parameters
x0 = np.linspace(-1050 * um, 1050 * um, 1024 * 2)
z0 = np.linspace(-0.125 * um, 4 * mm, 1024 * 4)
wavelength = 0.6238 * um * 10
```

[3]:
```
# Definition of source
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=2000 * um, theta=0. * degrees)
```

[4]:
```
# insert sphere
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)

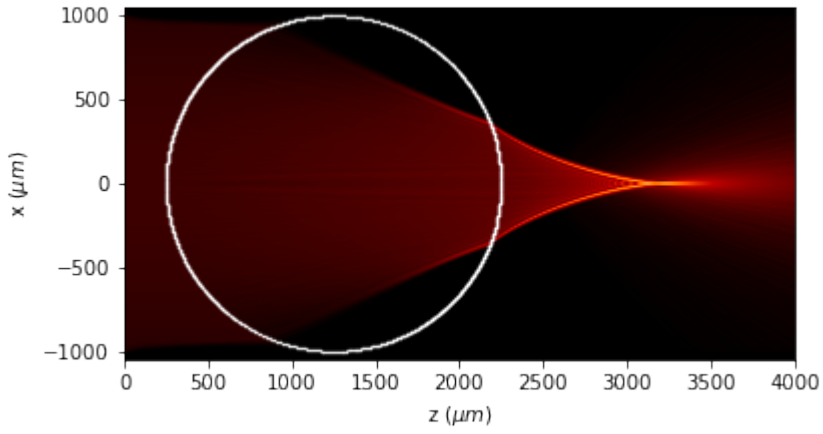
u1.sphere(r0=(0, 1.25 * mm),
          radius=(1 * mm, 1 * mm),
          refractive_index=1.33,
          angle=0)
u1.draw_refractive_index(scale='scaled', colorbar_kind='horizontal')
```



[5]:
```
#loading the incident field in the simulation
u1.incident_field(u0)
```

[6]:
```
# WPM propagation
u1.WPM(verbose=False)
```

[7]:
```
#simple drawing
u1.draw(logarithm=True, scale='scaled', draw_borders=True)
```



[8]:
```
Imax = u1.intensity().max()
```

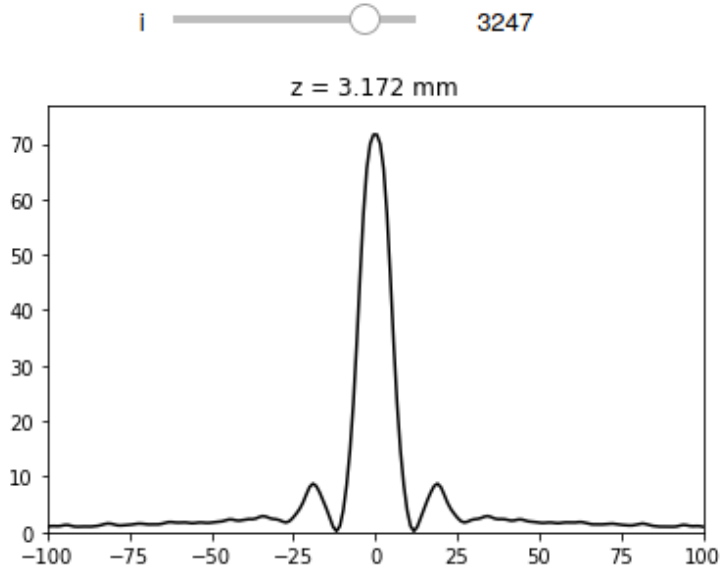
(continues on next page)

(continued from previous page)

[9]:
```
def __interact2__(i):
    plt.plot(u1.x, abs(intensity[:, i]), 'k')
    plt.title("z = {:2.3f} mm".format(u1.z[i] / 1000))
    plt.ylim(0, Imax)
    plt.xlim(-100 * um, 100 * um)
```

[11]:
```
%matplotlib widget

interactive(__interact2__, i=(0,len(u1.z)-1,1))
plt.show()
```



[ ]:

6.9.4 Interactive internal to diffractio

To include the figures into the jupyter notebook: - %matplotlib inline

To draw the figures in external windows: - %matplotlib qt - %matplotlib notebook - %matplotlib widgets

To install widgets: - pip3 install ipympl - pip3 install nodejs - jupyter labextension install @jupyter-widgets/jupyterlab-manager - jupyter labextension install jupyter-matplotlib

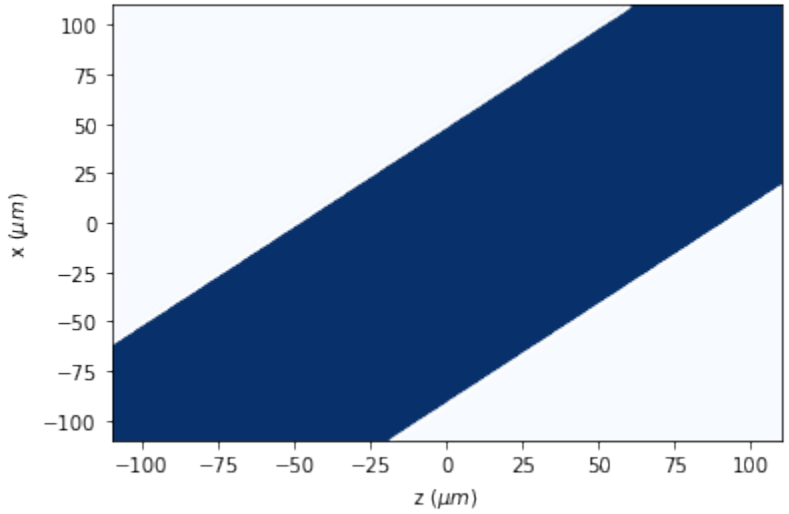
Windows: If you use windows: pip3 -> pip

[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[2]:
```
x0 = np.linspace(-110 * um, 110 * um, 2048)
z0 = np.linspace(-110 * um, 110 * um, 2048)
wavelength = 1 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=80 * um, w0=40 * um, theta=0 * degrees)

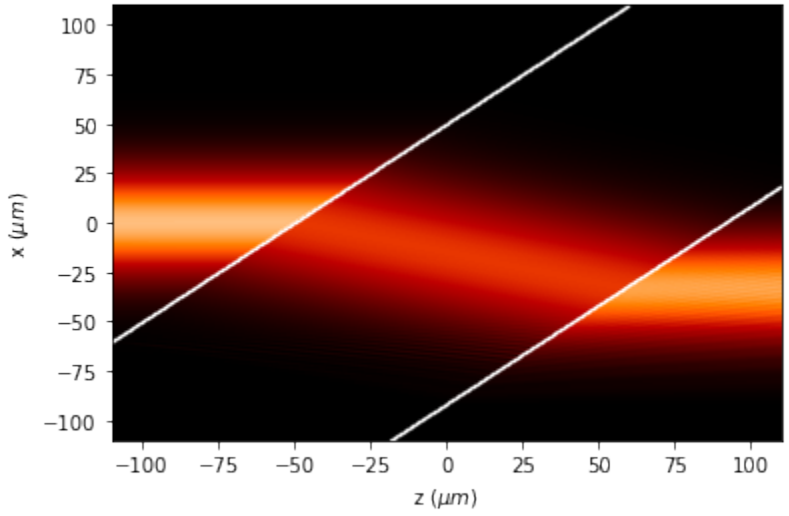
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.layer(r0=(50, 0), depth=100 * um, refractive_index=1.5, angle=45 * degrees)
```

[3]:
```
u1.draw_refractive_index()
```



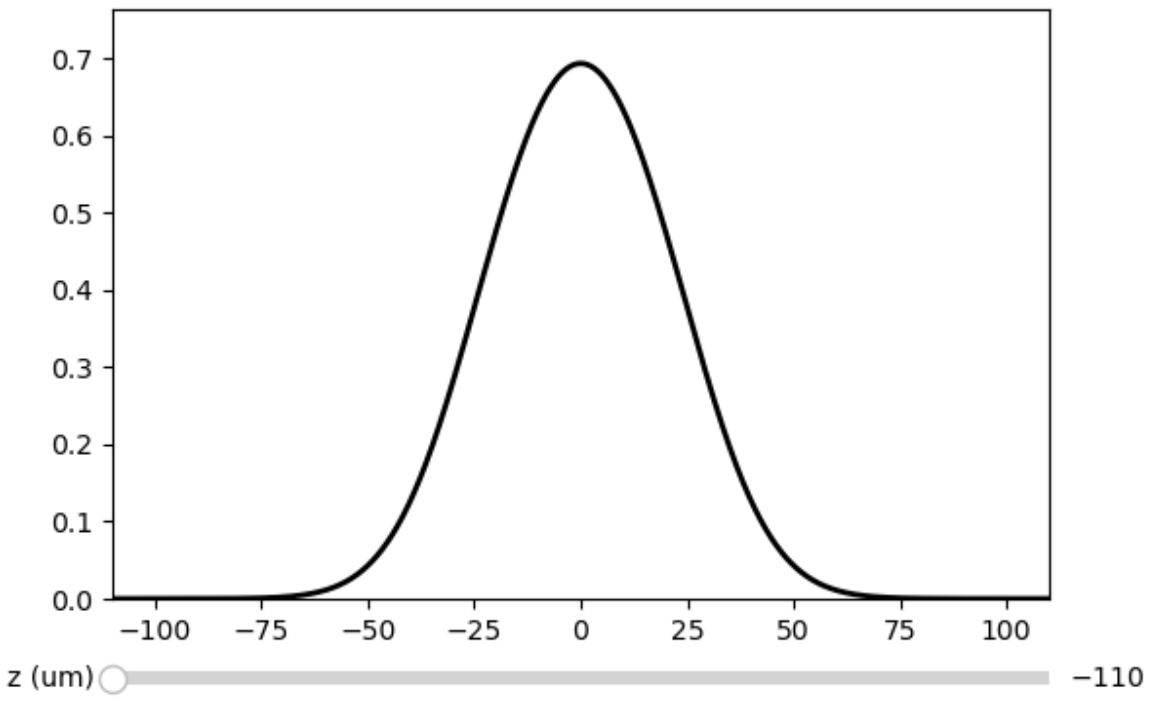
[4]:
```
u1.WPM(verbose=False)
```

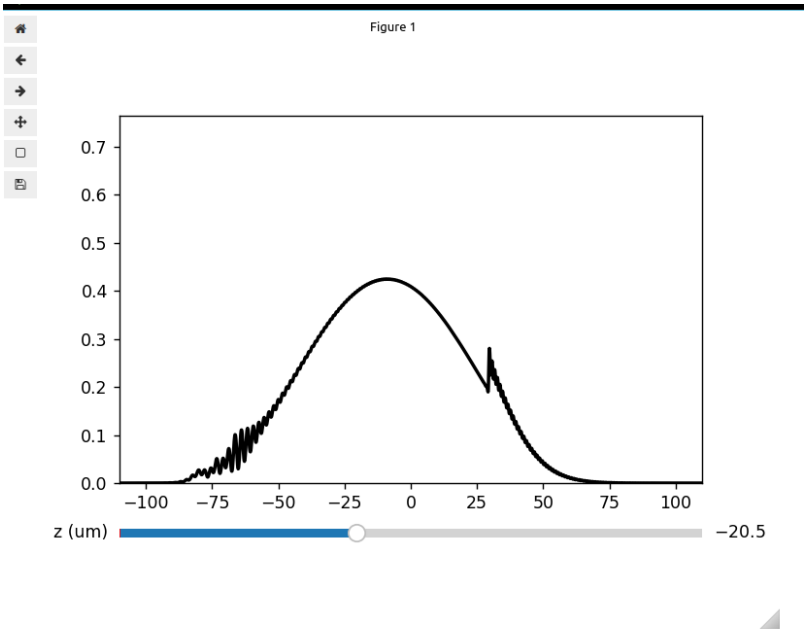
[5]:
```
u1.draw(kind='intensity', draw_borders=True)
```



[6]:
```
%matplotlib widget


u1.draw_profiles_interactive(
    kind='intensity', logarithm=True, normalize=False);
```





[ ]:

6.9.5 Images external to jupyter: %matplotlib notebook

[1]:
```
import time

from diffractio import degrees, mm, um
from diffractio import np, plt
from diffractio.scalar_fields_X import (Scalar_field_X,
                                        extended_polychromatic_source,
                                        extended_source_multiprocessing,
                                        polychromatic_multiprocessing)
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.utils_multiprocessing import (_pickle_method, _unpickle_method,
                                              execute_multiprocessing)
from multiprocessing import Pool
from diffractio.utils_optics import (gauss_spectrum, lorentz_spectrum,
                                     uniform_spectrum)

from diffractio.scalar_fields_XZ import Scalar_field_XZ

from diffractio import degrees, mm, plt, sp, um, np

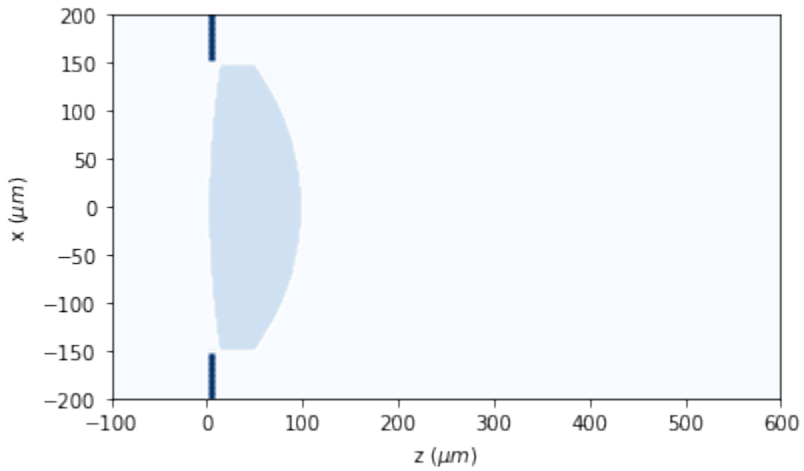
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

6.9.6 image in an external window

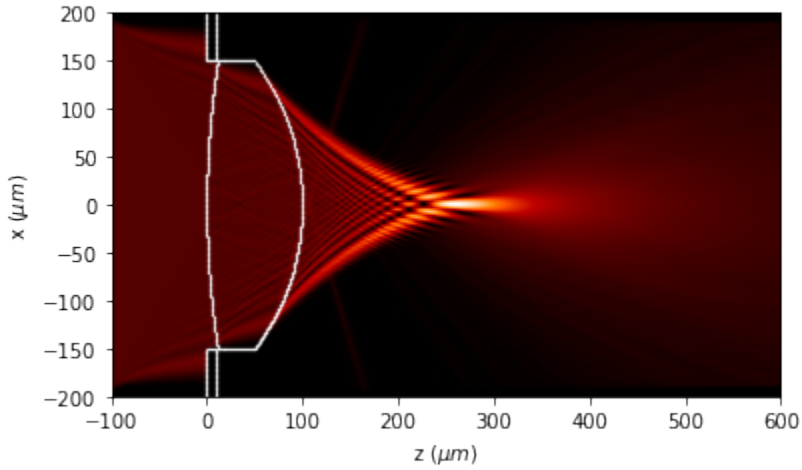
[2]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(-100 * um, 600 * um, 512 * 4)
wavelength = 10 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=300 * um,
                              radius=(1000 * um, -250 * um),
                              thickness=100 * um,
                              refractive_index=2,
                              angle=0 * degrees,
                              mask=(10 * um, 3 + 5j))
u1.clear_field()
```

[3]:
```
print(focal)
u1.draw_refractive_index(scale='scaled')
edge_matrix = u1.borders
```

```
192.30769230769232
```



[4]:
```
u1.BPM(verbose=False)
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
```



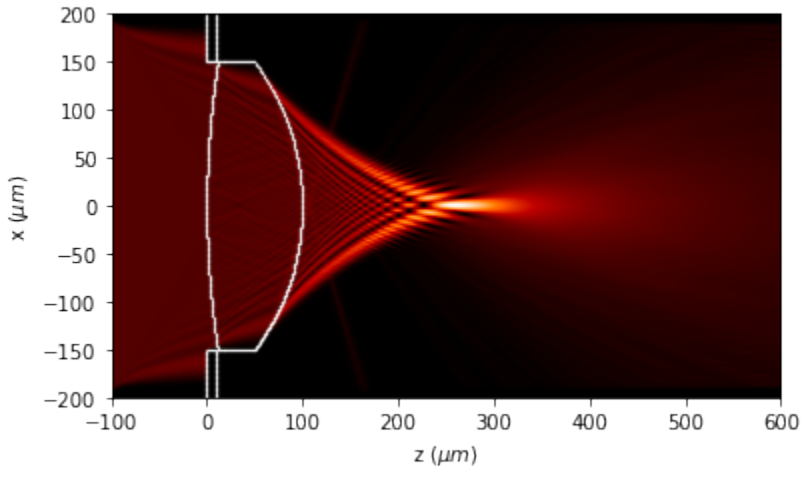
Now in external window

[5]:
```
%matplotlib qt
```

[6]:
```
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
```

[7]:
```
# Recovering inline
%matplotlib inline
```

[8]:
```
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
```



6.9.6.1 simulation of a video

[9]:
```
def __experiment_grating_movement__(dict_params):
    delta_x = dict_params['delta_x']
    period = dict_params['period']
    t1 = dict_params['t1']
    t2 = dict_params['t2']
    t2.ronchi_grating(period=period, x0=delta_x, fill_factor=0.5)
    u2 = t1 * t2
    return u2
```

[10]:
```
def creation_dictionary(deltas_x, period, t1, t2):
    # create Parameters: for multiprocessing
    dict_Parameters = []
    for i, delta_x in enumerate(deltas_x):
        dict_Parameters.append(
            dict(delta_x=delta_x, period=period, t1=t1, t2=t2))
    return dict_Parameters
```

[11]:
```
x0 = np.linspace(-400 * um, 400 * um, 1024 * 2)
wavelength = 0.85 * um
period = 50 * um
z_talbot = 2 * period**2 / wavelength
z0 = z_talbot / 2
delay = 0.001

t1 = Scalar_mask_X(x0, wavelength, info="__experiment_grating_movement__")
t1.ronchi_grating(period=period, x0=0 * um, fill_factor=0.5)
t1.RS(z=z0, new_field=False)

t2 = Scalar_mask_X(x0, wavelength, info="__experiment_grating_movement__")
t2.ronchi_grating(period=period, x0=0 * um, fill_factor=0.5)

deltas_x = np.linspace(-60 * um, 60 * um, 128)  # 512

dict_Parameters = creation_dictionary(deltas_x=deltas_x,
                                      period=period,
                                      t1=t1,
                                      t2=t2)

u_s, time_proc = execute_multiprocessing(__experiment_grating_movement__,
                                         dict_Parameters,
                                         num_processors=1,
                                         verbose=True)

x = u_s[0].x
```

```
Good result: factor 11.33
num_proc: 1, time=0.03563809394836426
```

[12]:
```
%matplotlib qt
```

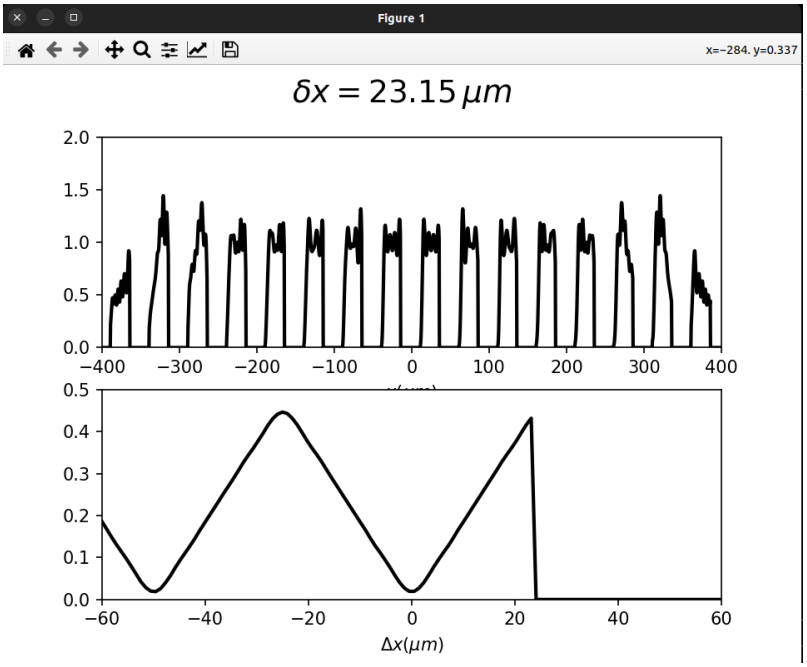
[13]:
```
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

perfil = np.zeros_like(deltas_x)

h1, = ax1.plot(x, np.zeros_like(x), 'k', lw=2)
ax1.set_xlim(x[0], x[-1])
ax1.set_ylim(0, 2)
ax1.set_xlabel(r'$x (\mu m)$')

h2, = ax2.plot(deltas_x, perfil, 'k', lw=2)
ax2.set_xlim(deltas_x[0], deltas_x[-1])
ax2.set_ylim(0, .5)
ax2.set_xlabel(r'$\Delta x (\mu m)$')

incr_frames = 1
for i in range(0, len(deltas_x), incr_frames):
    intensidad = abs(u_s[i].u)**2  # sacar fuera
    perfil[i] = intensidad.mean()
    plt.suptitle(r"$\delta x={:6.2f}\,\mu m$".format(deltas_x[i]), fontsize=18)
    h1.set_ydata(intensidad)
    h2.set_ydata(perfil)
    plt.draw()
    plt.pause(0.005)
```



[ ]:

CHAPTER

SEVEN

EXAMPLES

7.1 Focusing a Gauss beam

[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

7.1.1 Propagation of a Gauss beam

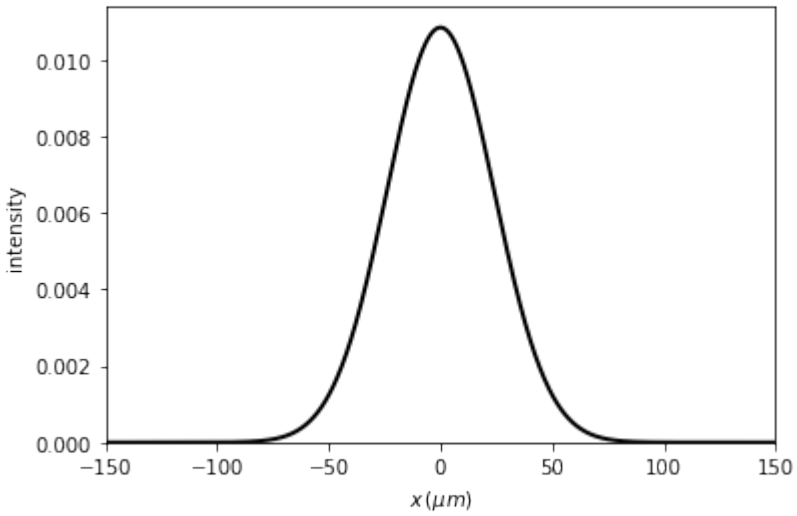
When we define the Gauss beam, we define the position of the beam waist, z0, and the the width at focus, w0. In ths case, z0 is the distance from the initial plane:

[2]:
```
x0 = np.linspace(-150 * um, 150 * um, 1024)
z0 = np.linspace(-100 * um, 100 * um, 1024)

wavelength = 7.5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=100 * um, w0=5 * um, theta=0 * degrees)
```

7.1.2 Vacuum

[3]:
```
u0.draw()
```



[4]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(-100 * um, 100 * um, 512)

wavelength = 10 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=100 * um, w0=10 * um, theta=0 * degrees)
```

[5]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
u1.incident_field(u0)
```

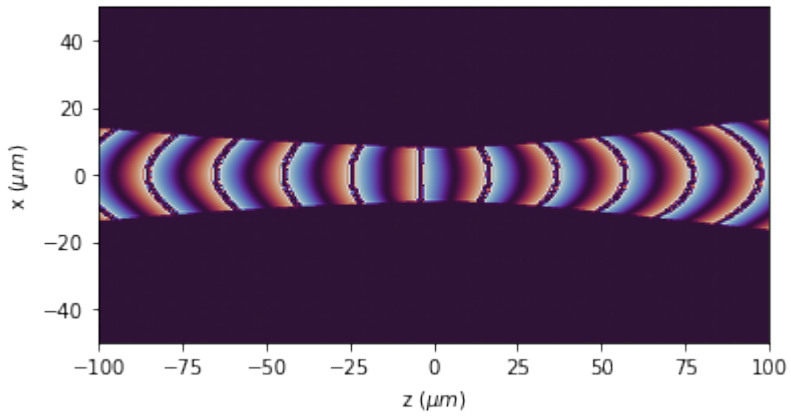
As we can see, the shape at the incident beam, has not w0=5 um.

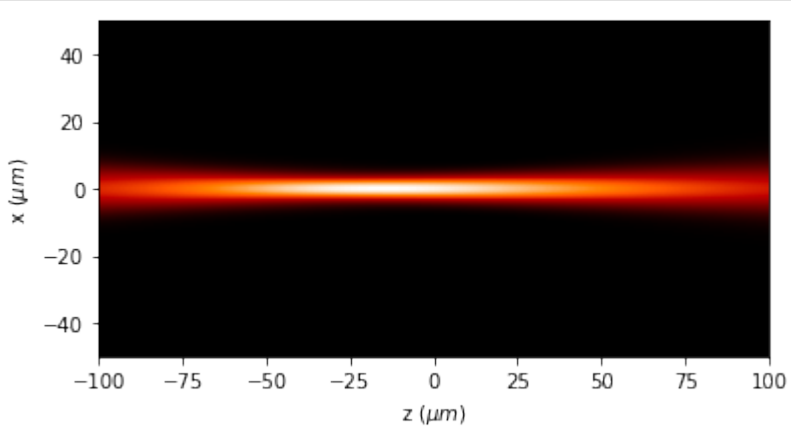
[6]:
```
u1.WPM(verbose=True)
u1.cut_resample(x_limits=(-50, 50), num_points=(512, 512))
```

```
Time = 0.41 s, time/loop = 0.8061 ms
```

Let us see the propagation

[7]:
```
u1.draw(kind='phase', scale='scaled')
u1.draw(kind='intensity', logarithm=1, scale='scaled')
```

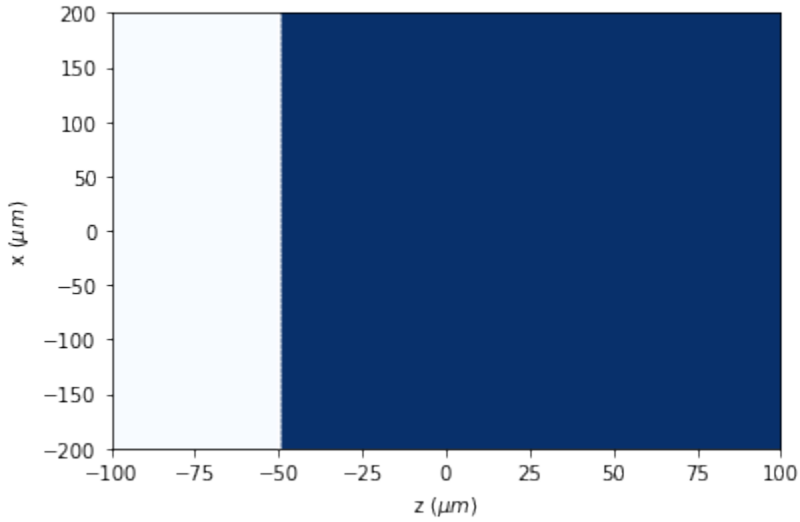




7.1.3 Layer

When the Gauss beam reaches a layer of dielectric, the wavefront and the beam profile changes

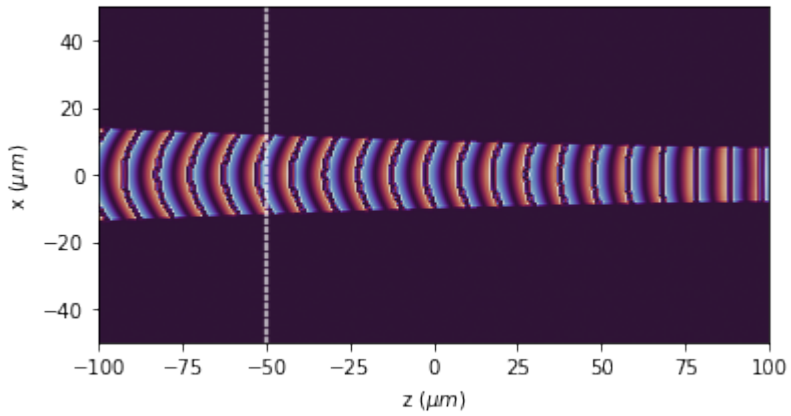
[8]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
u1.incident_field(u0)
u1.semi_plane(r0=(0, -50), refractive_index=2, angle=0 * degrees)
u1.draw_refractive_index()
```

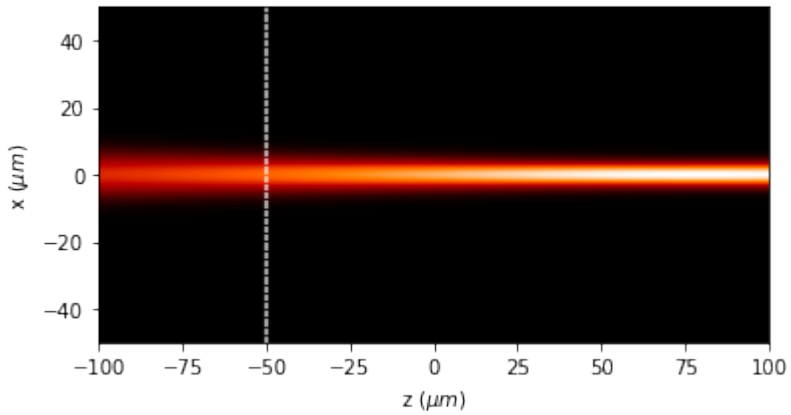


[9]:
```
u1.WPM(verbose=True)
u1.cut_resample(x_limits=(-50, 50), num_points=(512, 512))
```

```
Time = 0.43 s, time/loop = 0.8301 ms
```

[10]:
```
u1.draw(kind='phase', scale='scaled', draw_borders=True)
u1.draw(kind='intensity', logarithm=1, scale='scaled', draw_borders=True)
```





As can be seen, the beam waist is not at the same position.

7.2 Reflection and refraction

7.2.1 Refection and refraction at a surface

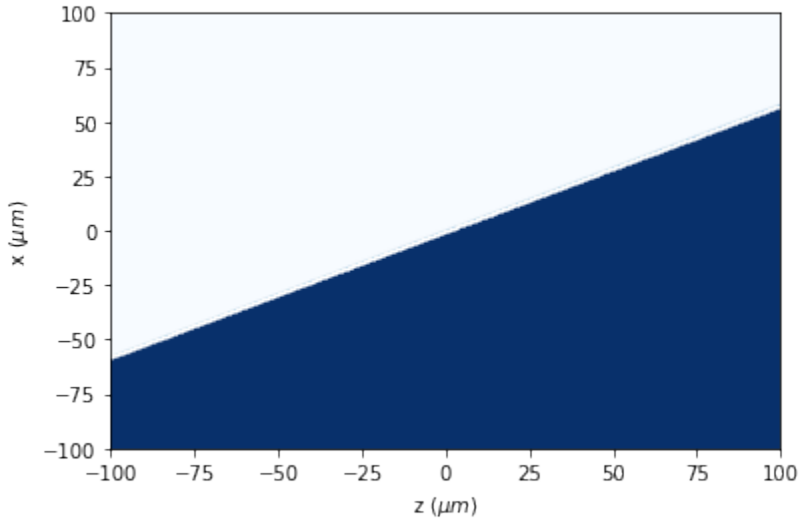
[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[2]:
```
x0 = np.linspace(-100 * um, 100 * um, 2048)
z0 = np.linspace(-100 * um, 100 * um, 2048)

wavelength = 5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=100 * um, w0=15 * um, theta=0 * degrees)

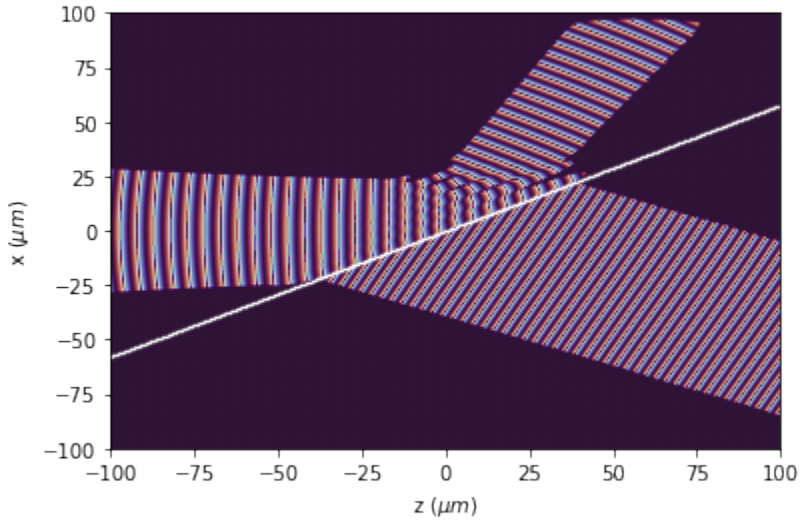
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1)
u1.incident_field(u0)
u1.semi_plane(r0=(0, 0), refractive_index=1.5, angle=60 * degrees)

u1.draw_refractive_index()
```

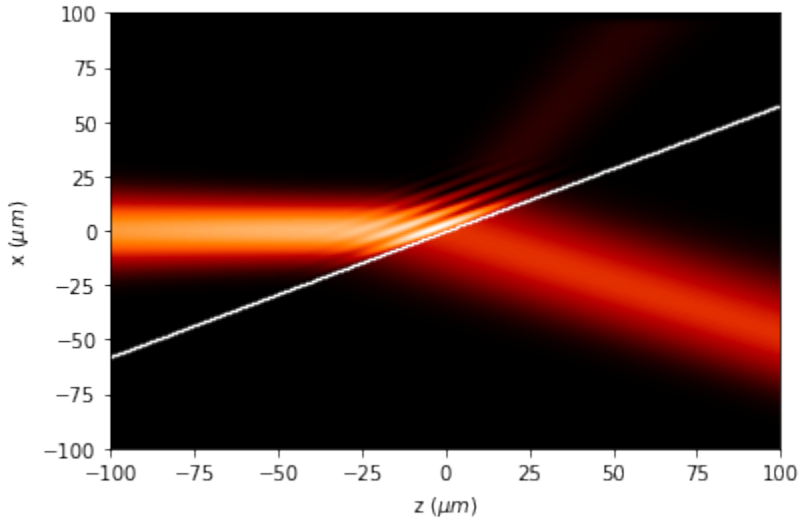


[3]:
```
u1.WPM(verbose=False)
```

[4]:
```
u1.draw(kind='phase', draw_borders=True)
```



[5]:
```
u1.draw(kind='intensity', logarithm=1e1, draw_borders=True)
```

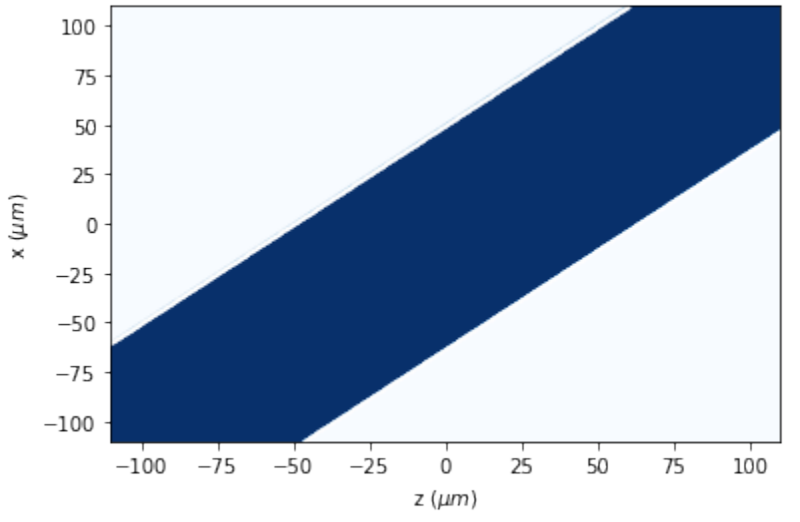


7.2.2 Refraction

[23]:
```
x0 = np.linspace(-110 * um, 110 * um, 2048)
z0 = np.linspace(-110 * um, 110 * um, 2048)
wavelength = 1 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=25 * um, z0=150 * um, w0=40 * um, theta=0 * degrees)

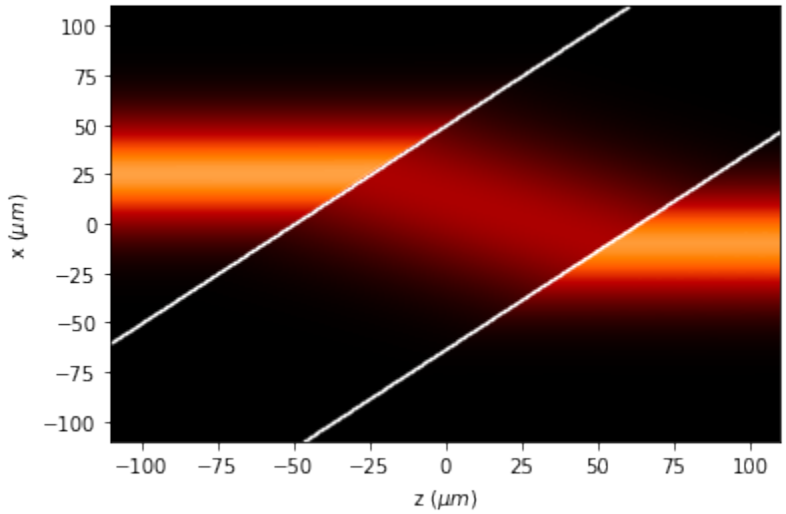
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.layer(r0=(50, 0), depth=80 * um, refractive_index=2, angle=45 * degrees)

u1.draw_refractive_index()
```



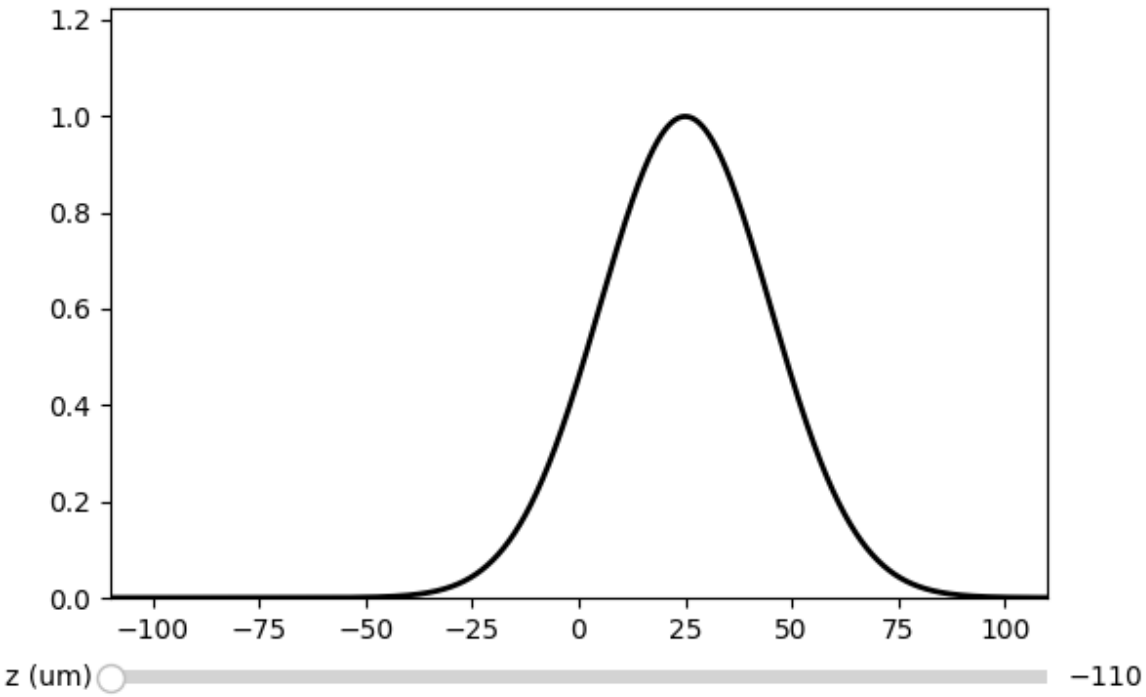
[24]:
```
u1.WPM(verbose=False)
```

[25]:
```
u1.draw(kind='intensity', draw_borders=True)
```



[26]:
```
%matplotlib widget
```

[27]:
```
u1.draw_profiles_interactive(
    kind='intensity', logarithm=False, normalize=False)
```



7.2.3 Total reflection

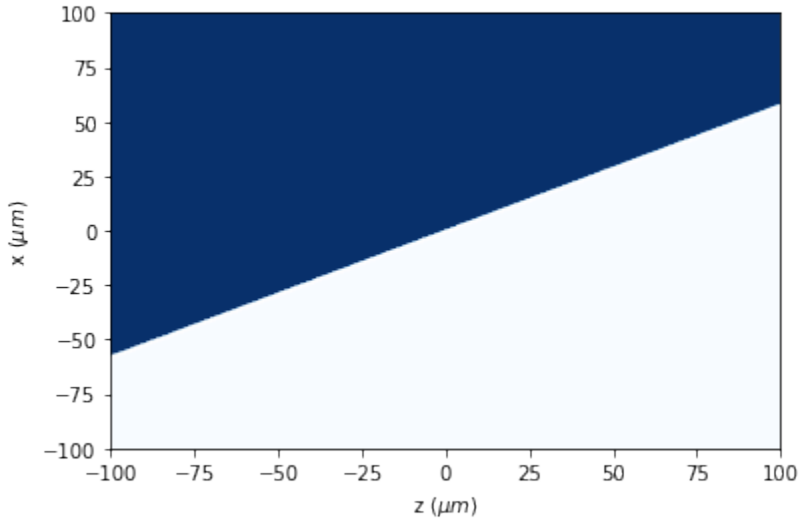
[11]:
```
%matplotlib inline
```

[12]:
```
x0 = np.linspace(-100 * um, 100 * um, 2048)
z0 = np.linspace(-100 * um, 100 * um, 2048)

wavelength = 5 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=100 * um, w0=15 * um, theta=0 * degrees)

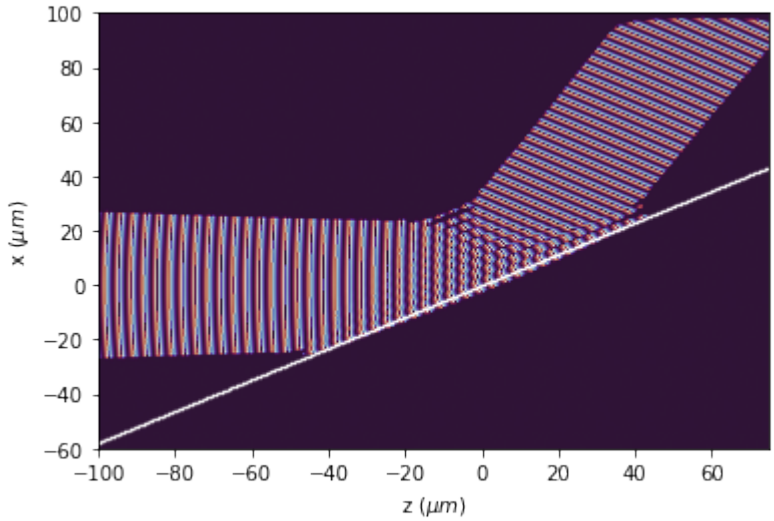
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.5)
u1.incident_field(u0)
u1.semi_plane(r0=(0, 0), refractive_index=1, angle=60 * degrees)

u1.draw_refractive_index()
```

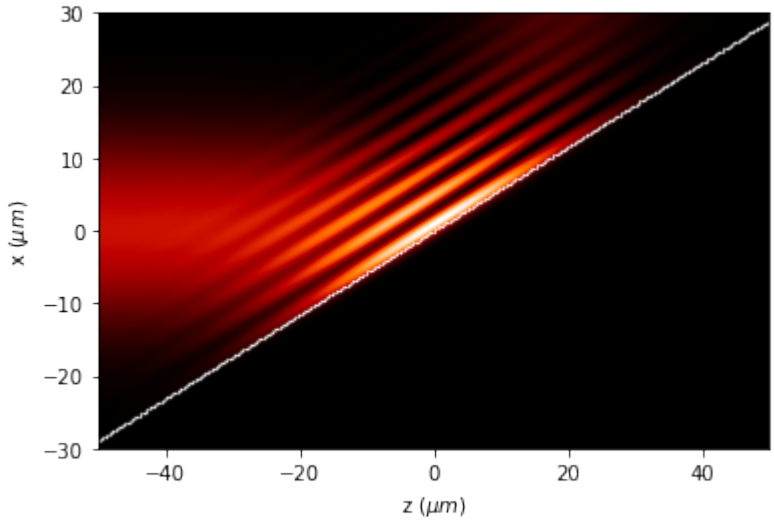


[13]:
```
u1.WPM(verbose=False)
```

[14]:
```
u1.draw(kind='phase', draw_borders=True)
plt.ylim(ymin=-60)
plt.xlim(xmax=75)
```



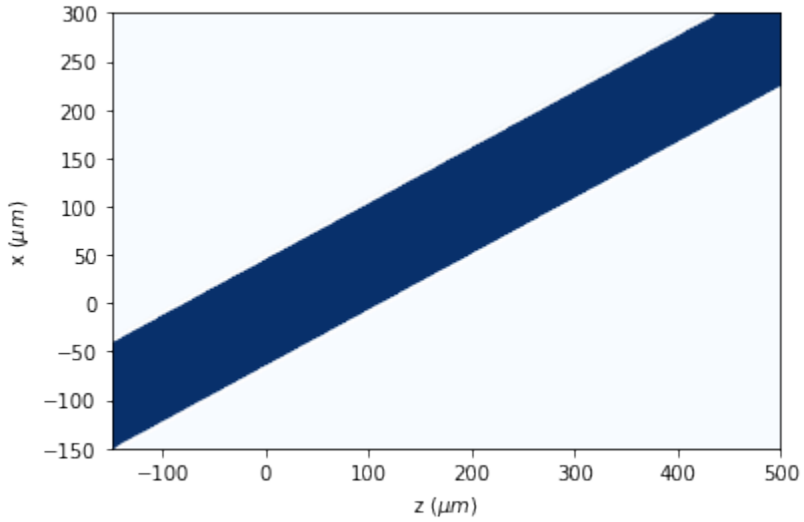
[15]:
```
u1.draw(kind='intensity', logarithm=True, draw_borders=True)
plt.xlim(-50, 50)
plt.ylim(-30, 30)
```



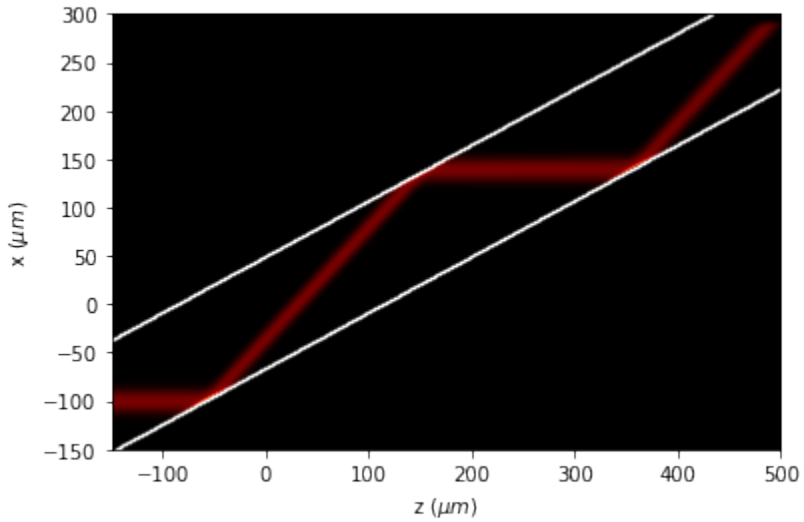
7.2.4 Total reflection at a layer

[16]:
```
x0 = np.linspace(-150 * um, 300 * um, 2048)
z0 = np.linspace(-150 * um, 500 * um, 2048)
wavelength = 1 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(
    A=1, x0=-100 * um, z0=150 * um, w0=15 * um, theta=0 * degrees)

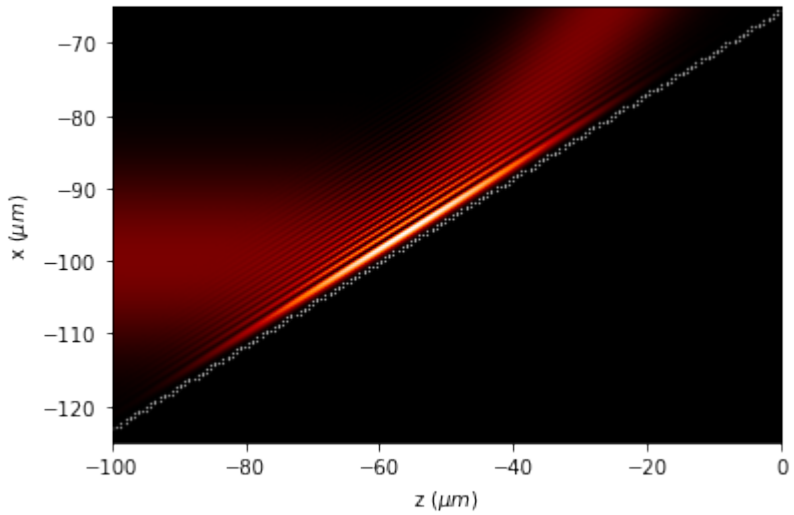
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.layer(
    r0=(50, 0), depth=100 * um, refractive_index=1.5, angle=60 * degrees)
u1.draw_refractive_index()
u1.smooth_refractive_index(type_filter=2, pixels_filtering=3)
```



[17]:
```
u1.BPM(verbose=False)
u1.draw(kind='intensity', logarithm=True, draw_borders=True)
```

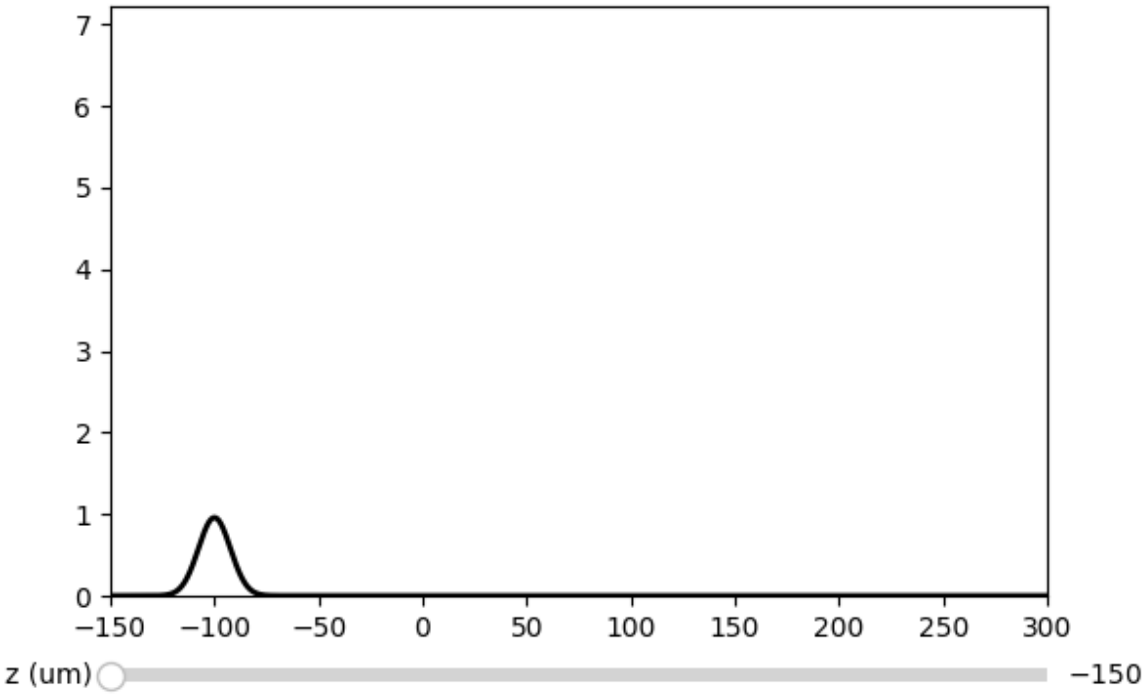


[18]:
```
u1.draw(kind='intensity', logarithm=True, draw_borders=True)
plt.xlim(-100, 0)
plt.ylim(-125, -65)
```



[19]:
```
%matplotlib widget
```

[20]:
```
u1.draw_profiles_interactive(
    kind='intensity', logarithm=False, normalize=False)
```



7.3 Diffraction by a single slit

Let us consider a light source passing through a simple slit. The technique is the following:

- Definition of light source at the plane of the mask $u_0(x)$
- Definition of mask $t(x)$
- Determination of light field after the mask $t_1(x) = t(x) * u_0(x)$
- Propagation using Rayleigh-Sommerfed `RS` `BPM` for a certain distance z
- Also, the Far Field intensity distribution using `FFT` can be determined.

We show how to analyze diffractive effects using the different frames X, XY, and XZ.

7.3.1 X

[64]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

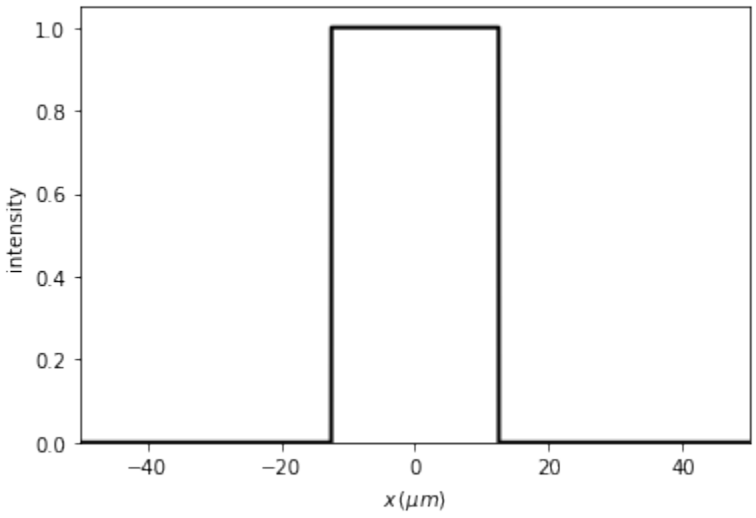
[65]:
```
from diffractio import degrees, mm, plt, np, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.utils_drawing import draw_several_fields
```

[66]:
```
x0 = np.linspace(-50 * um, 50 * um, 1024 * 32)
wavelength = 5 * um
```

[67]:
```
# plane wave
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0)

# slit
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.slit(x0=0, size=25 * um)
t0.draw()

u1 = u0 * t0
```



[68]:
```
# Propagation a certain distance and draw
u2 = u1.RS(z=25 * um, new_field=True, verbose=True)
u2.draw(kind='intensity')

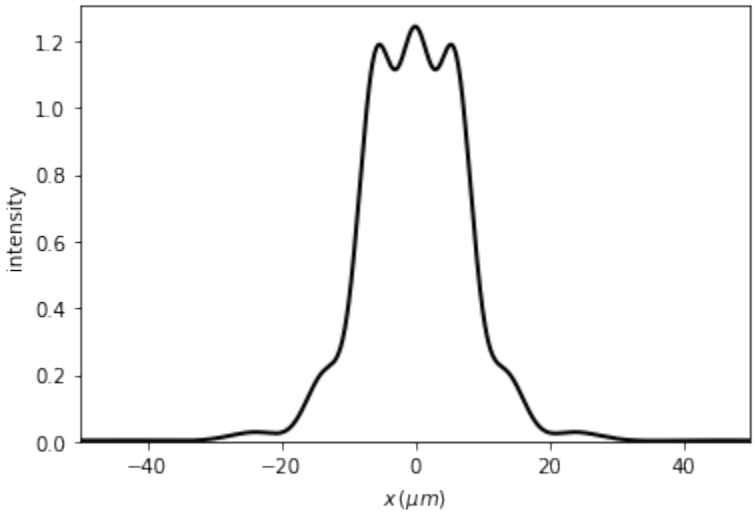
u3 = u1.RS(z=250 * um, new_field=True, verbose=True)
```

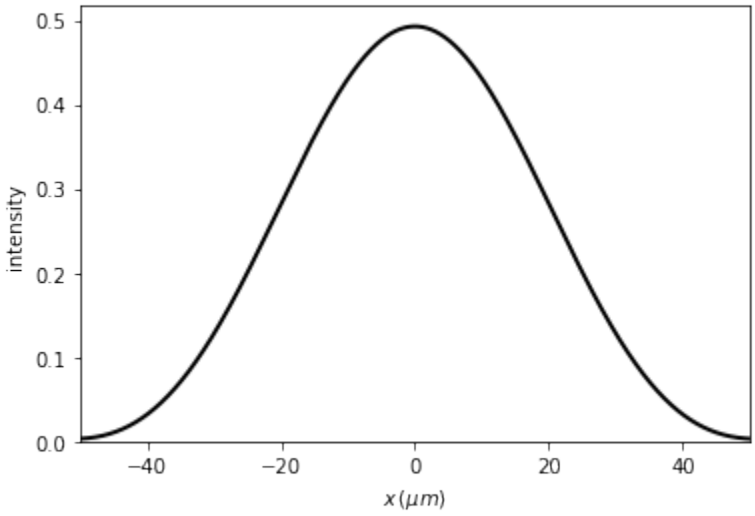
(continues on next page)

(continued from previous page)

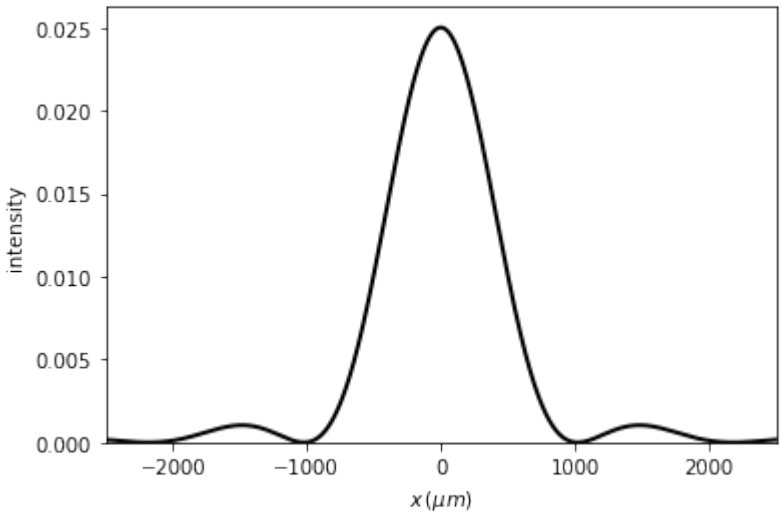
```
u3.draw(kind='intensity')
```

```
Good result: factor 906.65
Good result: factor 3479.13
```

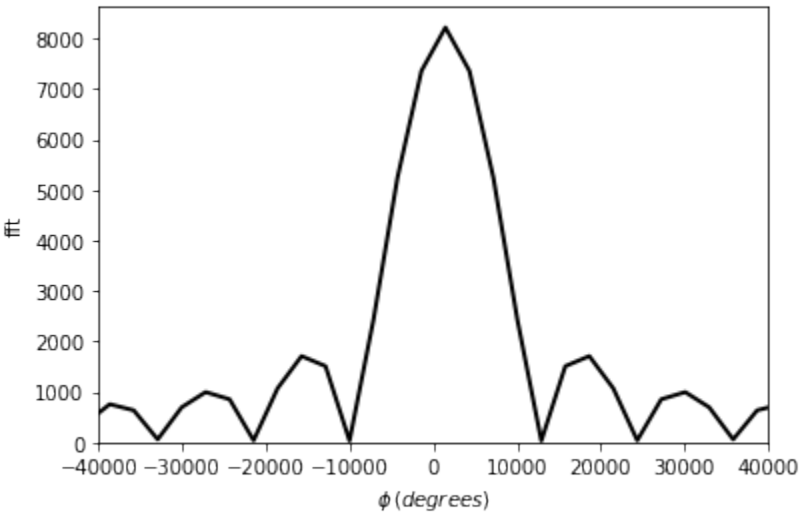




[69]:
```
u4 = u1.RS(z=5 * mm, amplification=50, new_field=True, verbose=False)
u4.draw(kind='intensity')
```



[70]:
```
u5 = u2.fft(z=1 * mm, remove0=False, new_field=True)
u5.draw(kind='fft', logarithm=False, normalize=True)
plt.xlim(-40000, 40000)
plt.ylim(bottom=0)
```



7.3.2 XY

[71]:
```
from diffractio import degrees, mm, plt, np, um
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.utils_drawing import draw_several_fields
```

[72]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024)
```

(continues on next page)

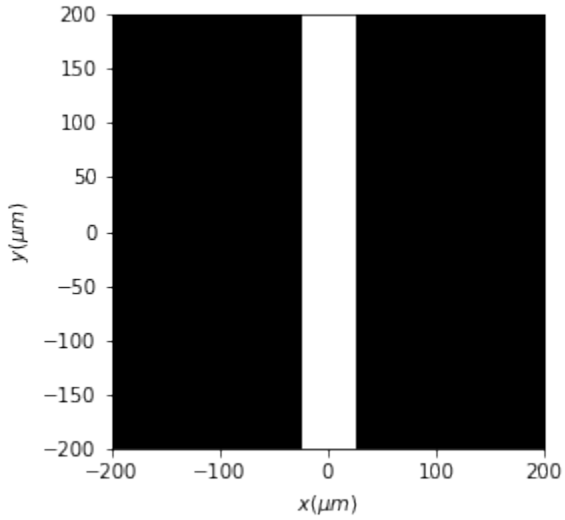
(continued from previous page)

```
y0 = np.linspace(-200 * um, 200 * um, 512)
```

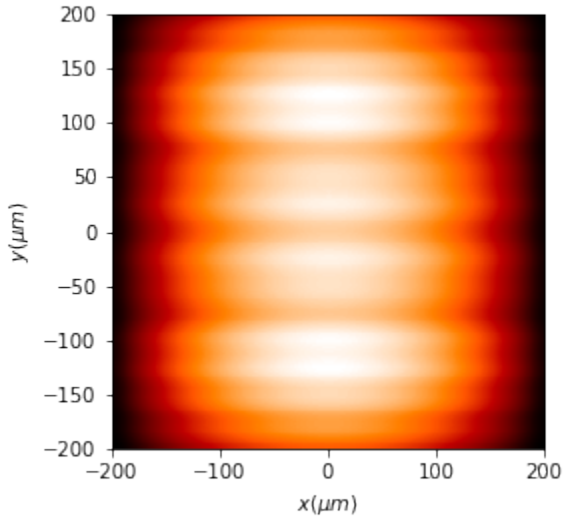
[73]:
```
u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.slit(x0=0, size=50 * um, angle=0 * degrees)

u2 = u1 * t1
u2.draw()
```



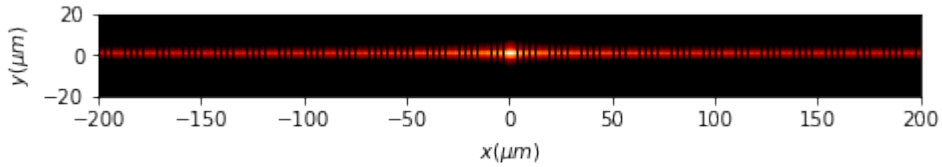
[74]:
```
u3 = u2.RS(z=2 * mm, new_field=True)
u3.draw(logarithm=1e3)
```



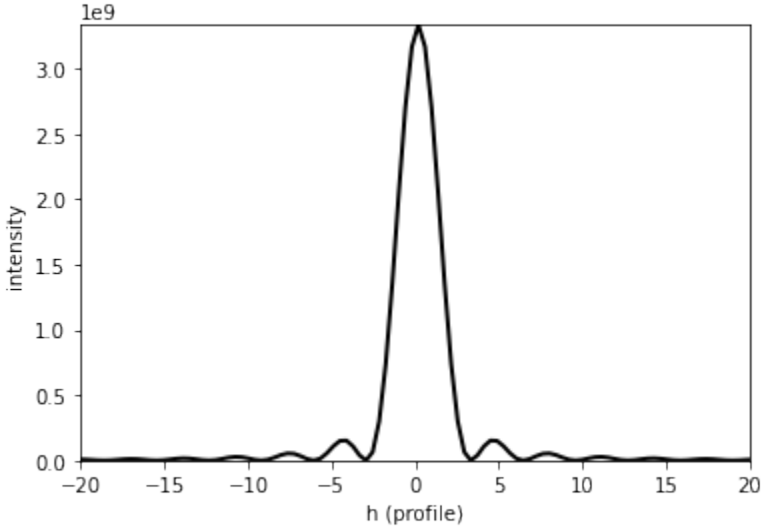
[75]:
```
u3 = u2.fft(remove0=False, new_field=True)
```

[76]:
```
u3.cut_resample(x_limits='', y_limits='', num_points=[1024, 1024])
```

[77]:
```
u3.draw(logarithm=1)
plt.ylim(-20,20)
```



[78]:
```
u3.draw_profile(point1=(u3.x[0], 0),
                point2=(u3.x[-1], 0),
                npixels=1024,
                kind='intensity',
                order=1)
plt.xlim(-20, 20)
```



7.3.3 XZ

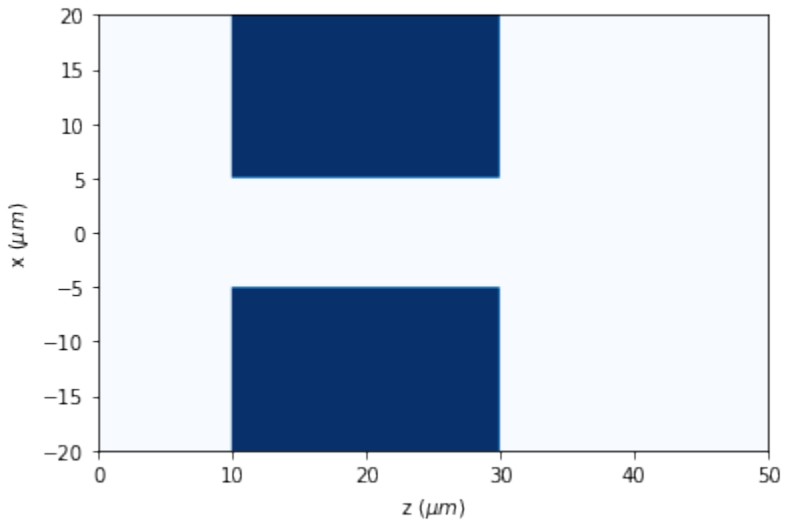
[79]:
```
from diffractio import degrees, mm, plt, np, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

[80]:
```
x = np.linspace(-20 * um, 20 * um, 1024)
z = np.linspace(0 * um, 50 * um, 512)
wavelength = 2 * um

u1 = Scalar_source_X(x=x, wavelength=wavelength)
u1.plane_wave(theta=0)

t0 = Scalar_mask_X(x=x, wavelength=wavelength)
t0.slit(x0=0, size=10 * um)
```

[89]:
```
t1 = Scalar_mask_XZ(x=x, z=z, wavelength=wavelength, n_background=1)
z0 = 10 * um
z1 = z0 + 20 * um
v_globals = dict(z0=z0, z1=z1)
t1.extrude_mask(t=t0,
                z0=z0,
                z1=z1,
                refractive_index=1.5 + 2j,
                v_globals=v_globals)
t1.draw_refractive_index(draw_borders=False, )
```

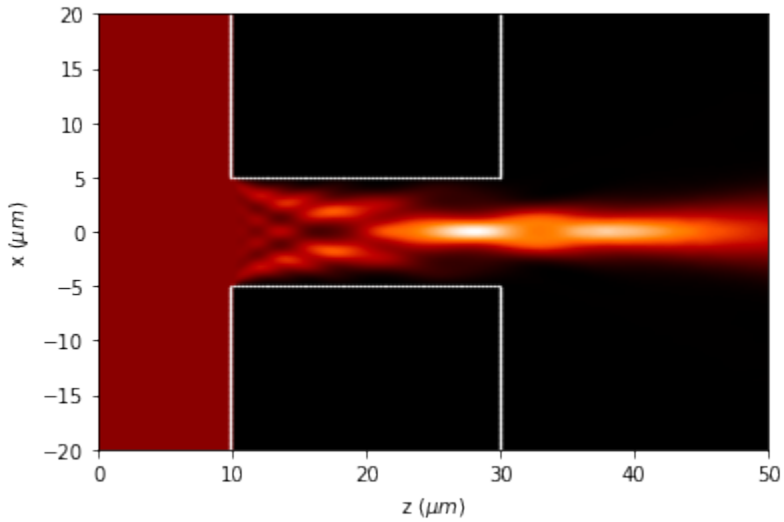


[90]:
```
t1.incident_field(u1)
t1.clear_field()

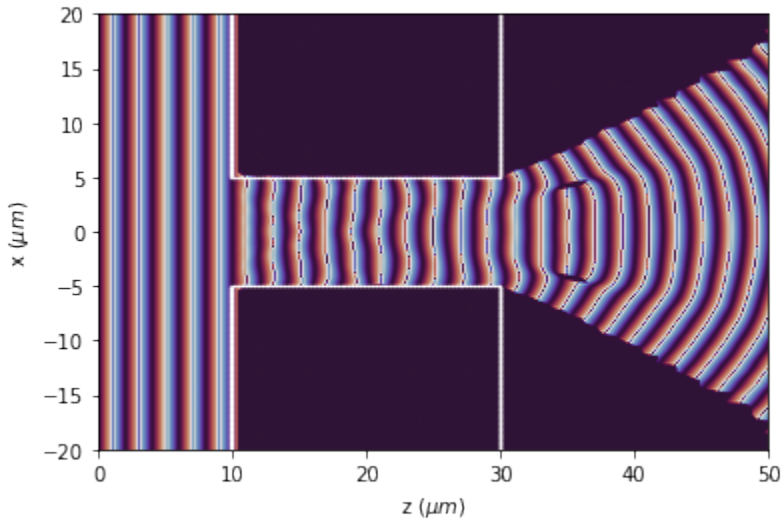
t1.WPM(has_edges=False)
```

[105]:
```
t1.draw(kind='intensity', draw_borders=True)
```

```
(511, 1024) (512, 1023) 512 1024
(512, 1024) (512, 1024)
```



[85]:
```
t1.draw(kind='phase', draw_borders=True, percentage_intensity=0.001)
```



[26]:
```
%load_ext autoreload
%autoreload 2
```

7.4 Arago point

Here, we analyze how Arago point (also called Poisson spot) is generated using in XY and XYZ framework.

More information: https://www.lighttrans.com/use-cases/application-use-cases/observation-of-the-poisson-spot.html

7.4.1 XY

[41]:
```
from diffractio import np, plt, um, mm, nm, degrees
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

[43]:
```
x0=np.linspace(-200*um, 200*um, 512)
y0=np.linspace(-200*um, 200*um, 1024)
wavelength=532*nm
```

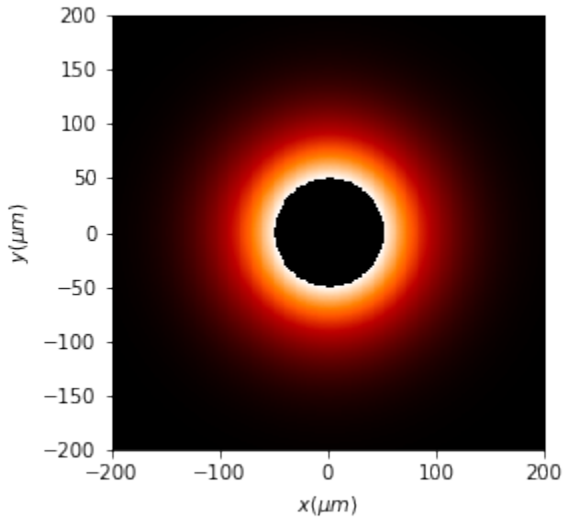
[44]:
```
u0=Scalar_source_XY(x0,y0,wavelength)
u0.gauss_beam(r0=(0,0), w0=130*um, z0=0, A=1, theta=0.0)
```

[45]:
```
obstacle=Scalar_mask_XY(x0,y0,wavelength)
obstacle.circle(r0=(0,0), radius=50*um)
obstacle.inverse_amplitude()
```

[46]:
```
obstacle=Scalar_mask_XY(x0,y0,wavelength)
obstacle.circle(r0=(0,0), radius=50*um)
obstacle.inverse_amplitude()

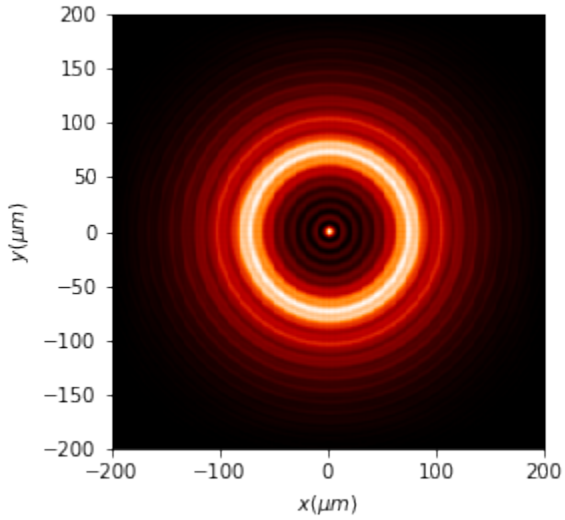
u1=u0*obstacle

u1.draw();
```

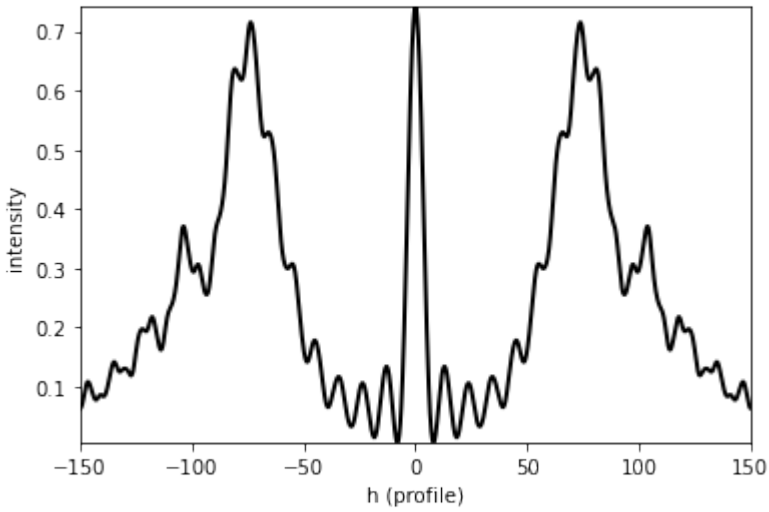


[47]:
```
arago_point=u1.RS(z=2*mm)
```

[48]:
```
arago_point.draw(logarithm=True);
```



[49]:
```
arago_point.draw_profile([-150,0], [150,0], npixels=2048, order=2);
```



7.4.2 XYZ

[50]:
```
from diffractio import np, plt, um, mm, nm, degrees
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

[51]:
```
x0=np.linspace(-200*um, 200*um, 512)
y0=np.linspace(-200*um, 200*um, 256)
z0=np.linspace(1*mm, 2*mm, 32)

wavelength=532*nm
```

[52]:
```
u0=Scalar_source_XY(x0,y0,wavelength)
u0.gauss_beam(r0=(0,0), w0=130*um, z0=0, A=1, theta=0.0)
```

[53]:
```
obstacle=Scalar_mask_XY(x0,y0,wavelength)
obstacle.circle(r0=(0,0), radius=50*um)
obstacle.inverse_amplitude()

u1=u0*obstacle
```

[54]:
```
u_xyz=Scalar_field_XYZ(x0,y0,z0,wavelength)
```

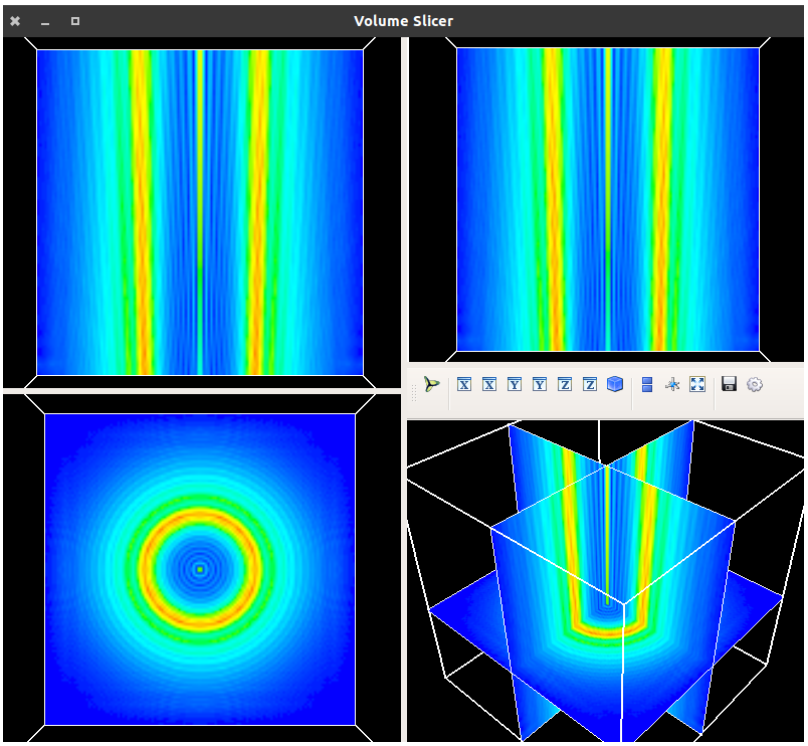
[55]:
```
u_xyz.incident_field(u1)
```

[56]:
```
u_xyz.RS()
```

```
time in RS= 15.285446882247925. num proc= 6
```

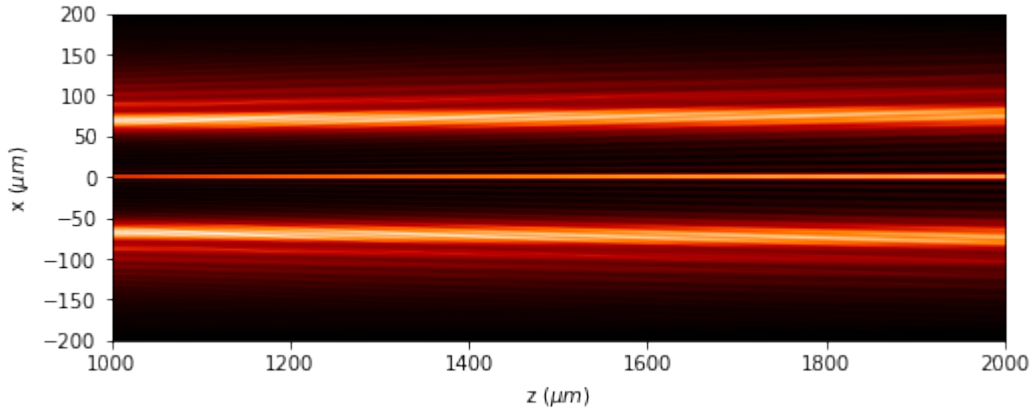
Warning: drawXYZ has been deprecated as I have not been able to re-install mayavi.

[57]:
```
# u_xyz.draw_XYZ(kind='intensity', logarithm=False)
```



[59]:
```
u_xyz.draw_XZ(y0=0)
fig=plt.gcf()
fig.set_size_inches(8,3)
```

```
<Figure size 432x288 with 0 Axes>
```



7.4.3 XZ

Warning: The XZ results are **not valid** since, as it is an interferential process, radial symmetry is required.

[60]:
```
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_sources_X import Scalar_source_X
```

[61]:
```
x0=np.linspace(-200*um, 200*um, 1024)
z0=np.linspace(1*mm, 2*mm, 512)
wavelength=532*nm
```

[62]:
```
source_x=Scalar_source_X(x0,wavelength)
source_x.gauss_beam(x0=0, w0=130*um, z0=0, A=1, theta=0.0)
```

[63]:
```
mask_x=Scalar_mask_X(x0,wavelength)
mask_x.slit(x0=0, size=100 * um);
mask_x.u=1-mask_x.u
```

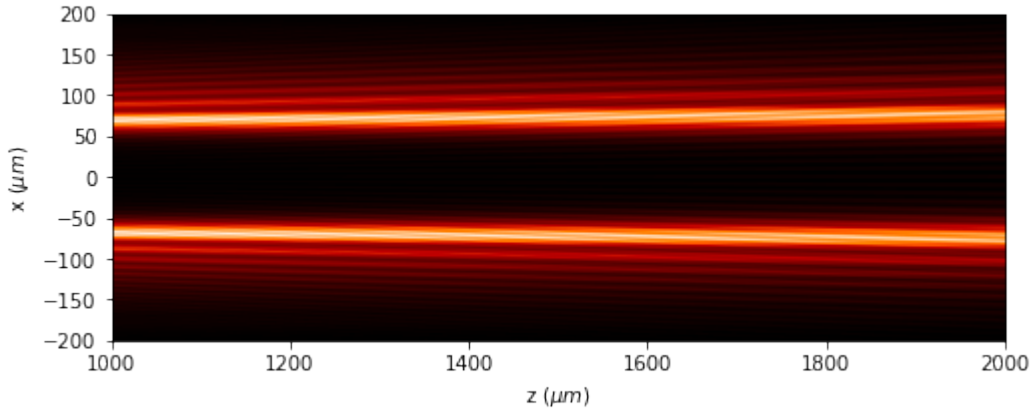
[64]:
```
u1=source_x*mask_x
```

[65]:
```
arago_xz=Scalar_mask_XZ(x0,z0,wavelength)
arago_xz.incident_field(u1)
```

```
Scalar_mask_XZ
```

[71]:
```
arago_xz.RS()
```

[69]:
```
arago_xz.draw()
fig=plt.gcf()
fig.set_size_inches(8,3);
```



7.5 Diffraction by different objects: xy frame

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

from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.utils_drawing import draw_several_fields
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

7.5.1 Slit

[2]:
```
num_pixels = 512

length = 100 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.slit(x0=0, size=10 * um, angle=0 * degrees)

u2 = u1 * t1

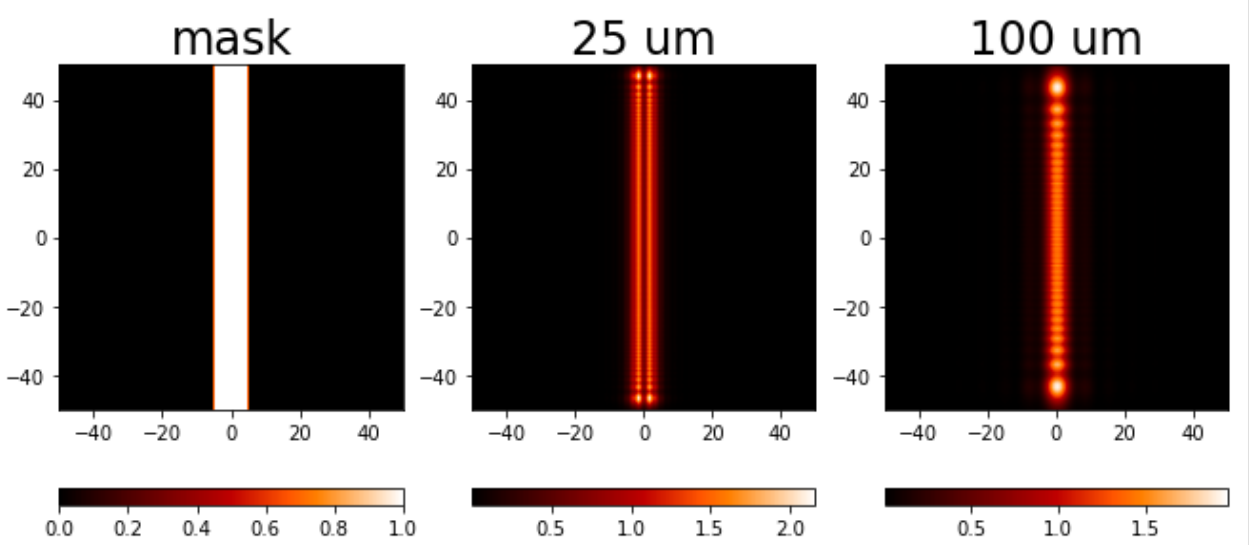
u3 = u2.RS(z=25 * um, new_field=True)

u4 = u2.RS(z=100 * um, new_field=True)
```

(continues on next page)

(continued from previous page)

```
draw_several_fields((u2, u3, u4), titles=('mask', '25 um', '100 um'))
```



7.5.2 double slit

[3]:
```
num_pixels = 512

length = 100 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

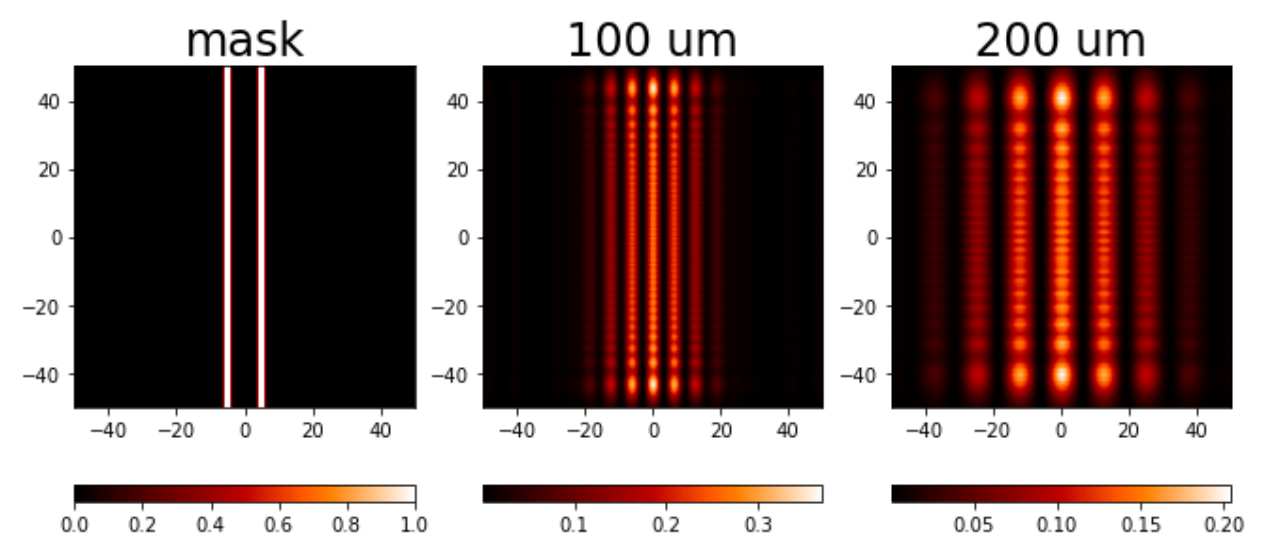
u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)
# u1.laguerre_beam(p=2, l=1, r0=(0 * um, 0 * um), w0=7 * um, z=0.01 * um)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.double_slit(x0=0, size=2 * um, separation=10 * um, angle=0 * degrees)

u2 = u1 * t1
u3 = u2.RS(z=100 * um, new_field=True)

u4 = u2.RS(z=200 * um, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '100 um', '200 um'))
```



7.5.3 square

[4]:
```
num_pixels = 512

length = 100 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)
# u1.laguerre_beam(p=2, l=1, r0=(0 * um, 0 * um), w0=7 * um, z=0.01 * um)

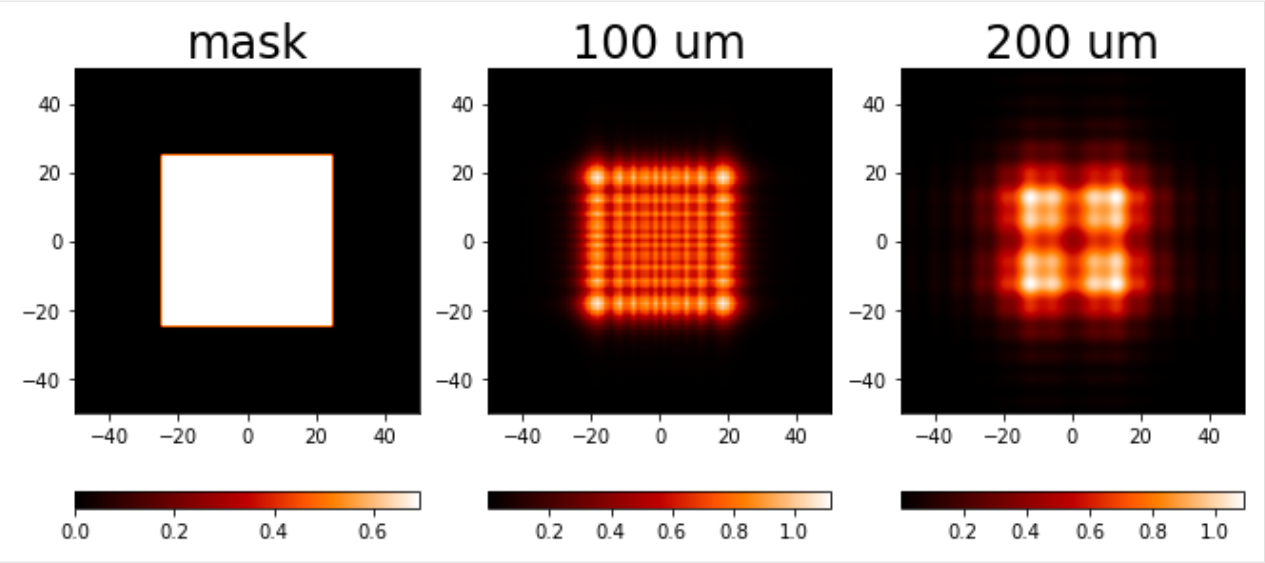
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.square(r0=(0 * um, 0 * um), size=(50 * um, 50 * um), angle=0 * degrees)

u2 = u1 * t1

u3 = u2.RS(z=100 * um, new_field=True)

u4 = u2.RS(z=500 * um, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '100 um', '200 um'), logarithm=True)
```



7.5.4 circle

[5]:
```
num_pixels = 512

length = 100 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

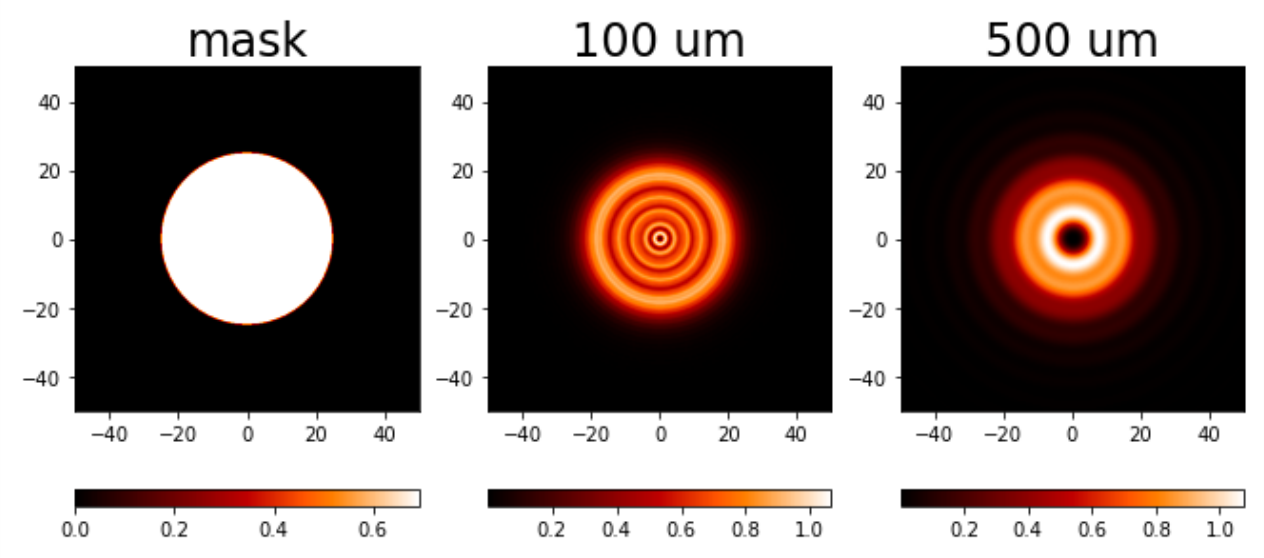
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.circle(
    r0=(0 * um, 0 * um), radius=(25 * um, 25 * um), angle=0 * degrees)

u2 = u1 * t1

u3 = u2.RS(z=100 * um, new_field=True)

u4 = u2.RS(z=500 * um, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '100 um', '500 um'), logarithm=True)
```



7.5.5 ring

[6]:
```
num_pixels = 512

length = 500 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.ring(
    r0=(0 * um, 0 * um),
    radius1=(25 * um, 50 * um),
    radius2=(100 * um, 150 * um),
    angle=45)

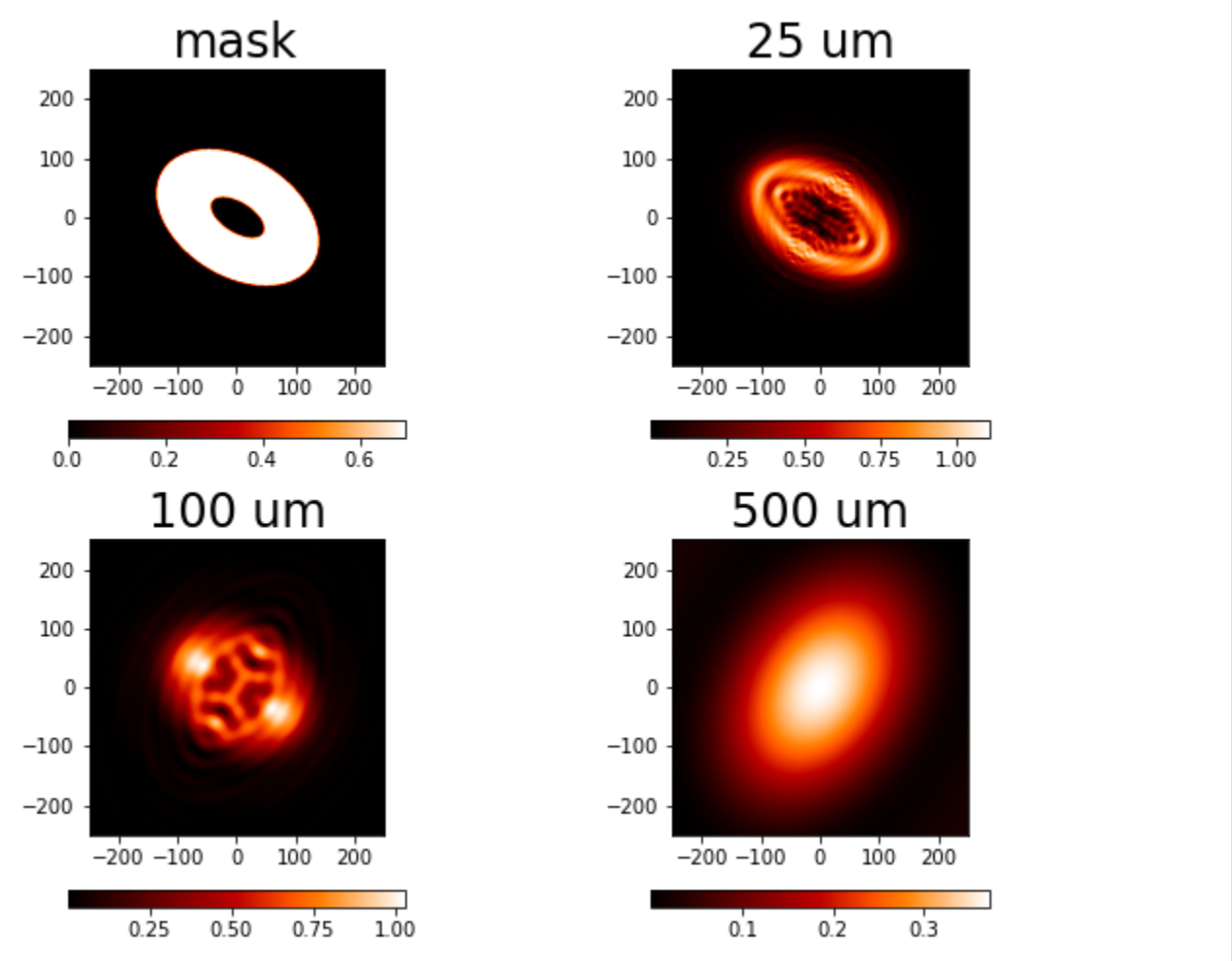
u2 = u1 * t1

u3 = u2.RS(z=2.5*mm, new_field=True)

u4 = u2.RS(z=10*mm, new_field=True)

u5 = u2.RS(z=100*mm, new_field=True)

draw_several_fields((u2, u3, u4, u5), titles=('mask', '25 um', '100 um', '500 um'),␣
↪logarithm=True)
plt.tight_layout()
```



7.5.6 cross

[7]:
```
num_pixels = 512

length = 200 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.cross(r0=(0 * um, 0 * um), size=(150 * um, 50 * um), angle=45 * degrees)

u2 = u1 * t1

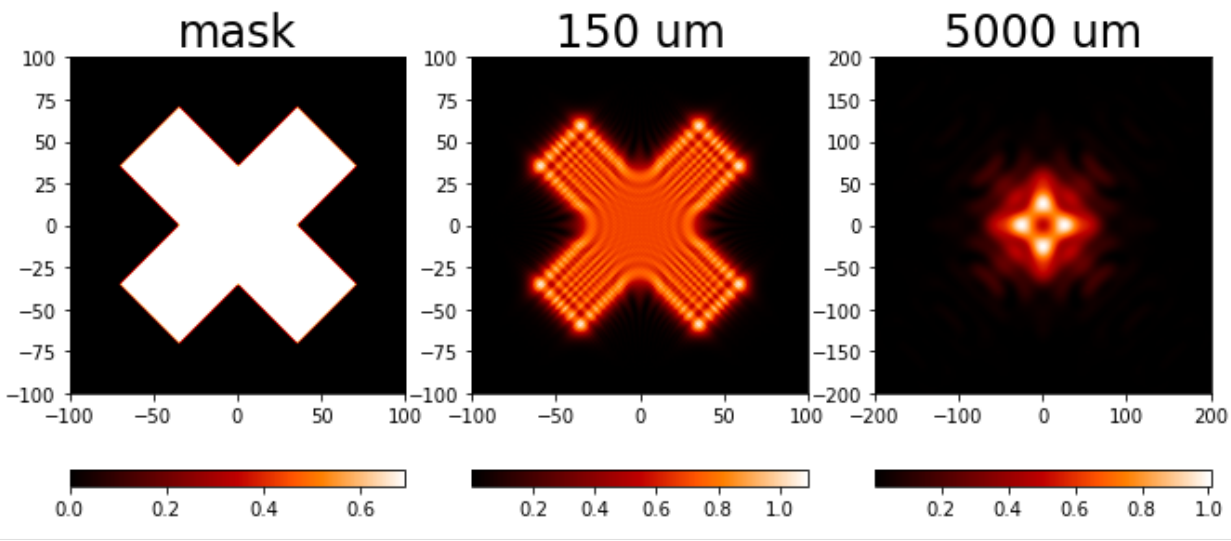
u3 = u2.RS(z=150 * um, new_field=True)
```

(continues on next page)

(continued from previous page)

```
u4 = u2.RS(z=5000 * um, amplification=(2, 2), new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '150 um', '5000 um'), logarithm=True)
```



7.5.7 edge

[8]:
```
num_pixels = 512

length = 200 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

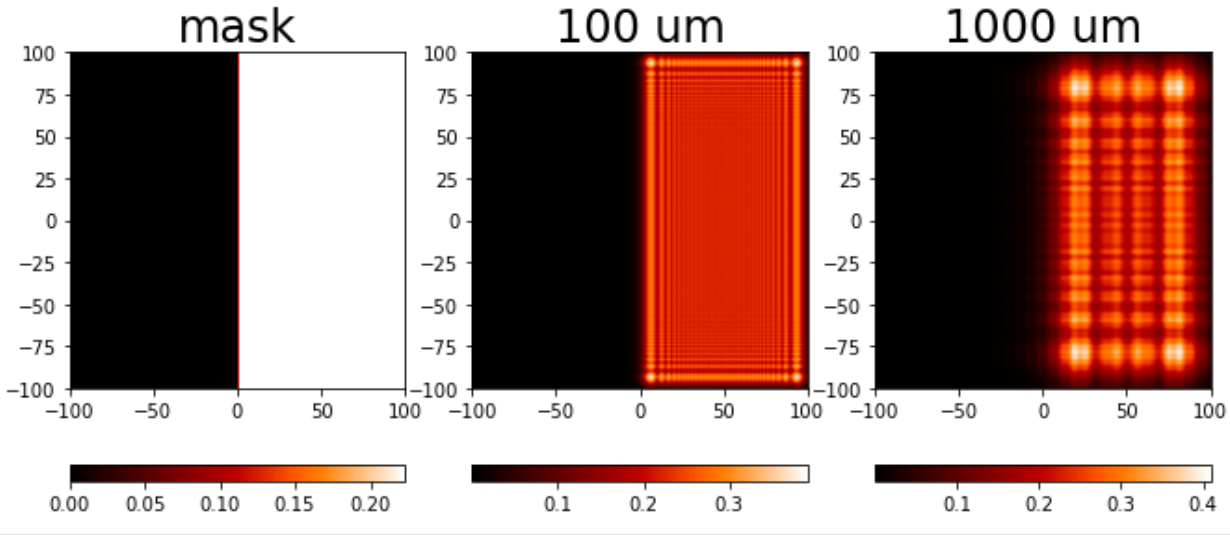
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.two_levels(level1=0, level2=.5, x_edge=0, angle=0)

u2 = u1 * t1

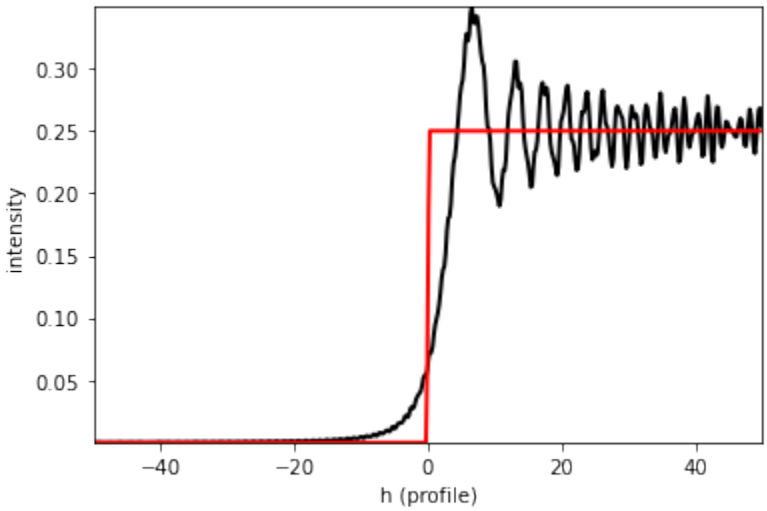
u3 = u2.RS(z=100 * um, new_field=True)

u4 = u2.RS(z=1000 * um, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '100 um', '1000 um'), logarithm=True)
```



[9]:
```
u3.draw_profile(point1=(-50, 0), point2=(50, 0), kind='intensity', order=2)
h, profile, p1, p2 = t1.profile(
    point1=(-50, 0), point2=(50, 0), kind='intensity', order=1)
plt.plot(h, profile, 'r', lw=2)
```



7.5.8 image

[10]:
```
num_pixels = 512

length = 100 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
```

(continues on next page)

(continued from previous page)

```
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

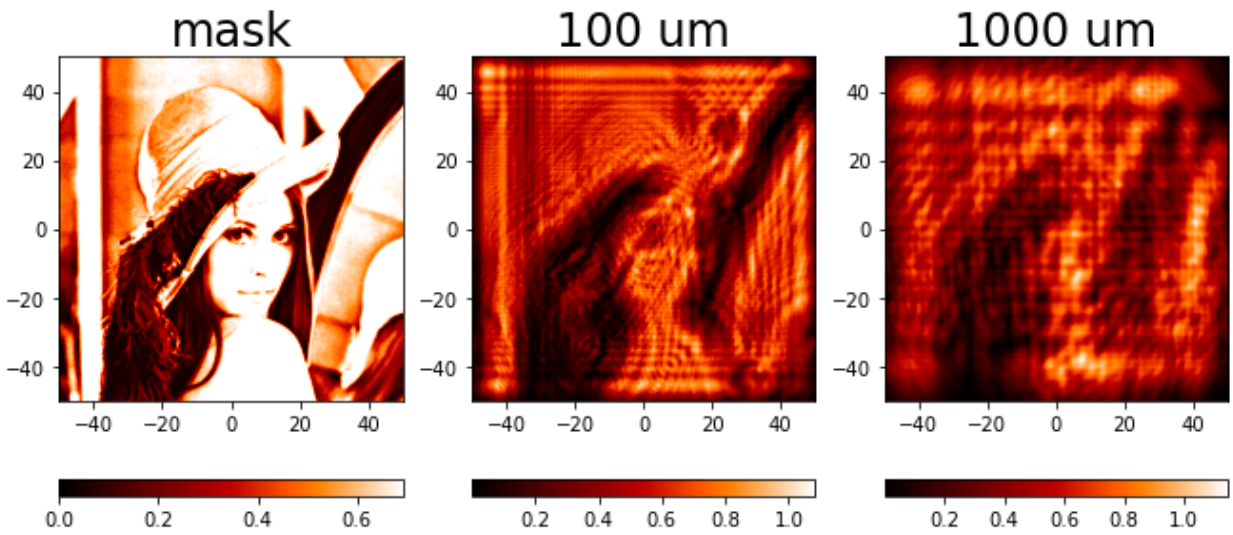
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)

t1.image(filename="lenaColor.png", invert=False)
u2 = u1 * t1

u3 = u2.RS(z=50 * um, new_field=True)

u4 = u2.RS(z=250 * um, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '100 um', '1000 um'), logarithm=True)
```



7.5.9 mask_from_function

[11]:
```
num_pixels = 512

length = 200 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

f1 = "R1-h1+np.sqrt(R1**4-(self.X-x0)**4-(self.Y-y0)**4)"
f2 = "R2-h2+np.sqrt(R2**2-(self.X)**2-(self.Y)**2)"
v_globals = {
    'R1': 6 * mm,
    'R2': 2 * mm,
    'x0': 0 * um,
    'y0': 25 * um,
```

(continues on next page)

(continued from previous page)

```
    'h1': 4 * mm,
    'h2': -1 * mm,
    'np': np,
}
index = 1.5
print(v_globals)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.mask_from_function(
    r0=(0 * um, 0 * um),
    index=index,
    f1=f1,
    f2=f2,
    v_globals=v_globals,
    radius=(100 * um, 100 * um))
t1.pupil()

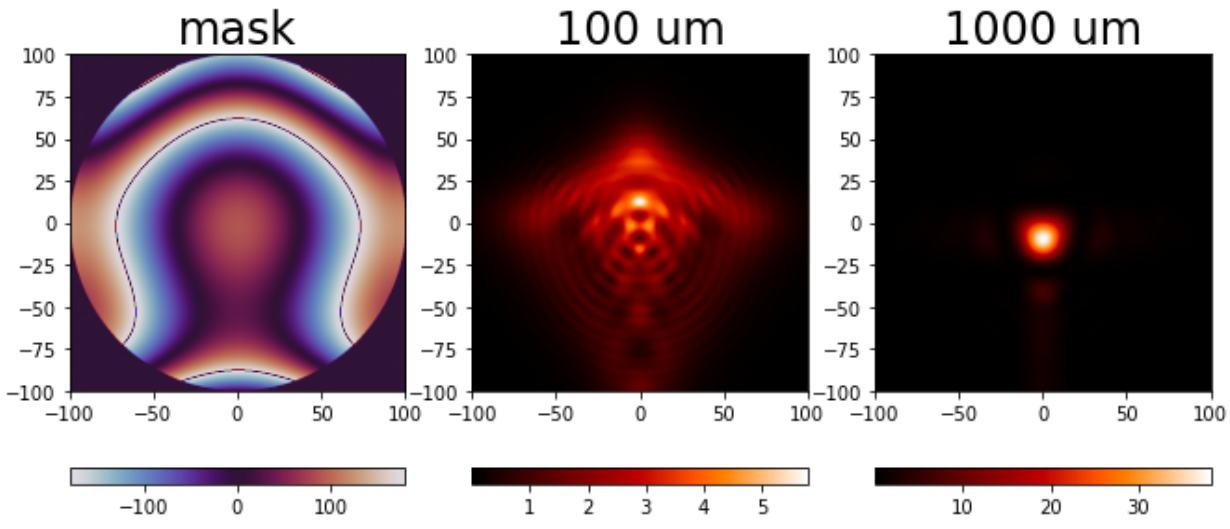
u2 = u1 * t1

u3 = u2.RS(z=2000 * um, new_field=True)

u4 = u2.RS(z=5000 * um, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '100 um', '1000 um'), logarithm=False,
                    kinds=['phase', 'intensity', 'intensity'])
```

```
{'R1': 6000.0, 'R2': 2000.0, 'x0': 0.0, 'y0': 25.0, 'h1': 4000.0, 'h2': -1000.0, 'np':
↪<module 'numpy' from '/usr/lib/python3/dist-packages/numpy/__init__.py'>}
```



7.5.10 astigmatism

[12]:
```
num_pixels = 512

length = 500 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.lens(
    r0=(0 * um, 0 * um),
    radius=(250 * um, 250 * um),
    focal=(5 * mm, 10 * mm),
    angle=0 * degrees)

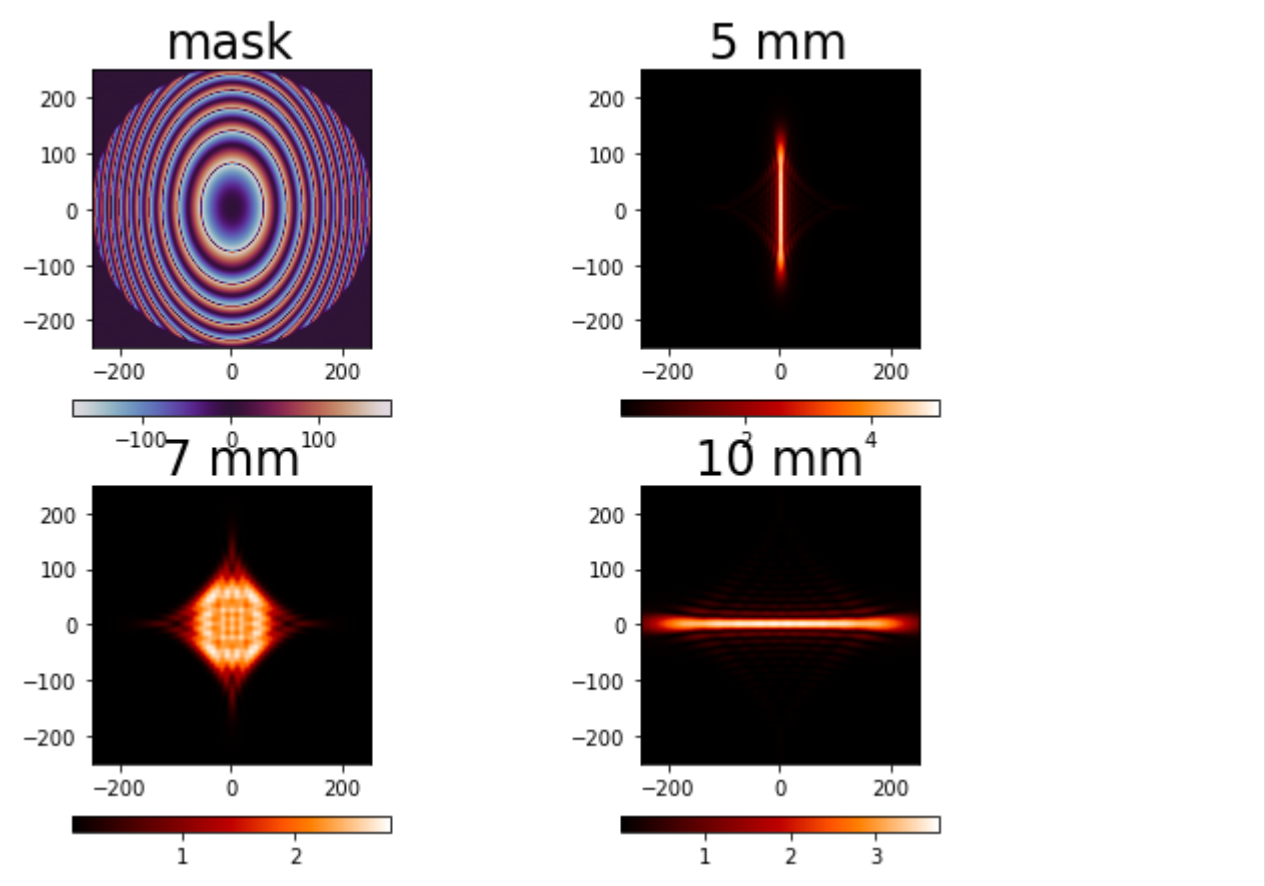
u2 = u1 * t1

u3 = u2.RS(z=5 * mm, new_field=True)

u4 = u2.RS(z=6.5 * mm, new_field=True)

u5 = u2.RS(z=10 * mm, new_field=True)

draw_several_fields((u2, u3, u4, u5), titles=('mask', '5 mm', '7 mm', '10 mm'),␣
↪logarithm=True,
                    kinds=['phase', 'intensity', 'intensity', 'intensity'])
```



7.5.11 Fresnel lens

[13]:
```
num_pixels = 512

length = 250 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.fresnel_lens(
    r0=(0 * um, 0 * um),
    radius=(125 * um, 125 * um),
    focal=(1 * mm, 1 * mm),
    angle=45 * degrees,
    kind='amplitude',
    phase=np.pi)

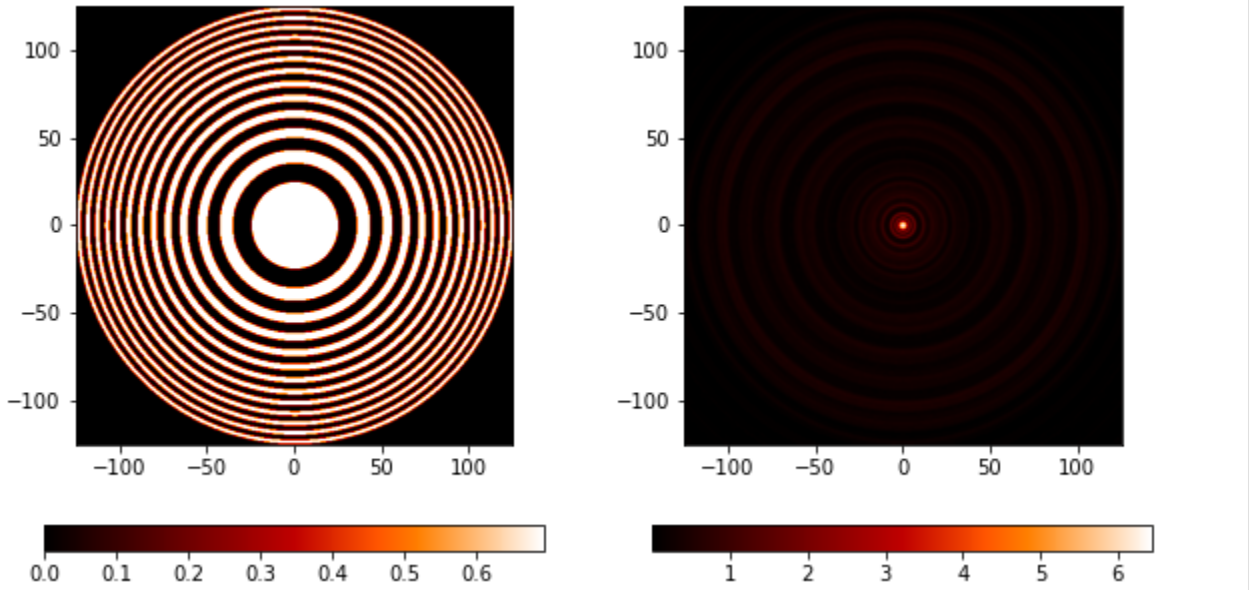
t1.pupil()
```

(continues on next page)

(continued from previous page)

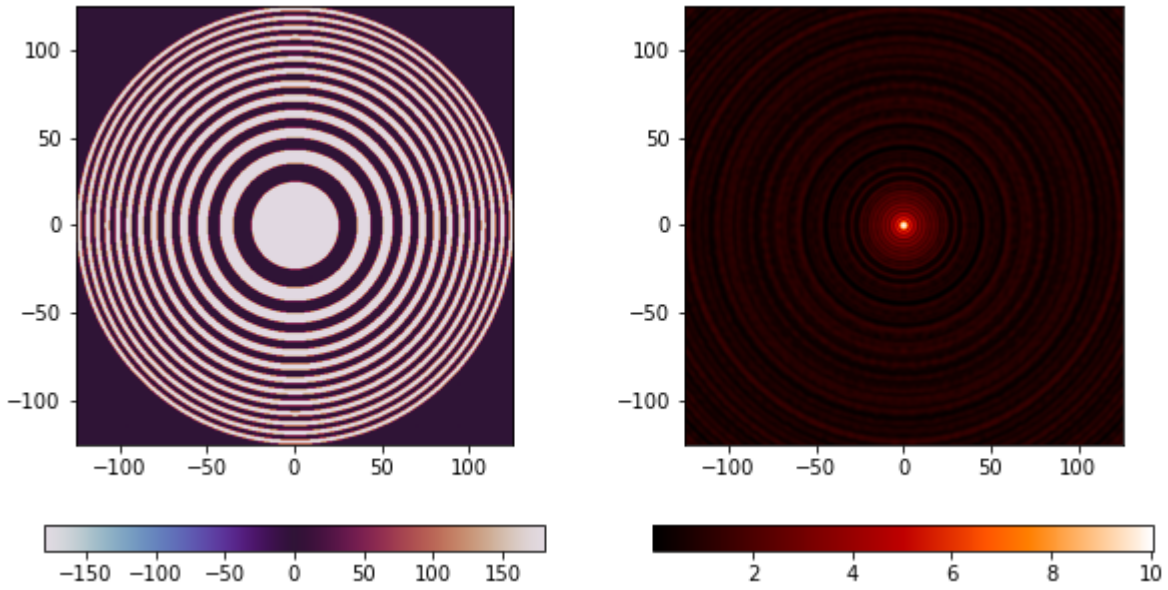
```
u2 = u1 * t1
u3 = u2.RS(z=1 * mm, new_field=True)

draw_several_fields((u2, u3), titles=('', ''), kinds=['intensity', 'intensity'],␣
↪logarithm=True)
```



[14]:
```
t1.fresnel_lens(
    r0=(0 * um, 0 * um),
    radius=(125 * um, 125 * um),
    focal=(1 * mm, 1 * mm),
    angle=0 * degrees,
    kind='phase',
    phase=np.pi)
t1.pupil()
u2 = u1 * t1
u3 = u2.RS(z=1 * mm, new_field=True)

draw_several_fields((t1, u3), kinds=['phase', 'intensity'], logarithm=1e1)
```



7.5.12 biprism

[15]:
```
num_pixels = 512

length = 250 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

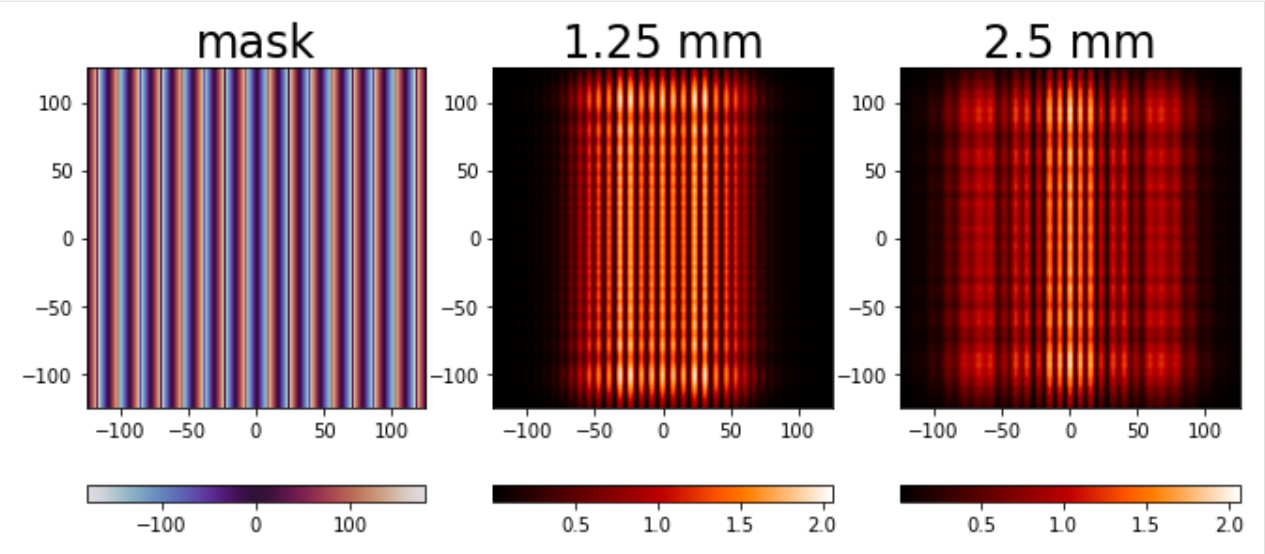
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)

t1.biprism_fresnel(
    r0=(0 * um, 0 * um), width=125 * um, height=5 * um, n=1.5)
u2 = u1 * t1

u3 = u2.RS(z=1.25 * mm, new_field=True)

u4 = u2.RS(z=2.5 * mm, new_field=True)

draw_several_fields((u2, u3, u4), titles=('mask', '1.25 mm', '2.5 mm'), logarithm=True,
                    kinds=['phase', 'intensity', 'intensity'])
```



7.5.13 axicon

[16]:
```
num_pixels = 512

length = 250 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

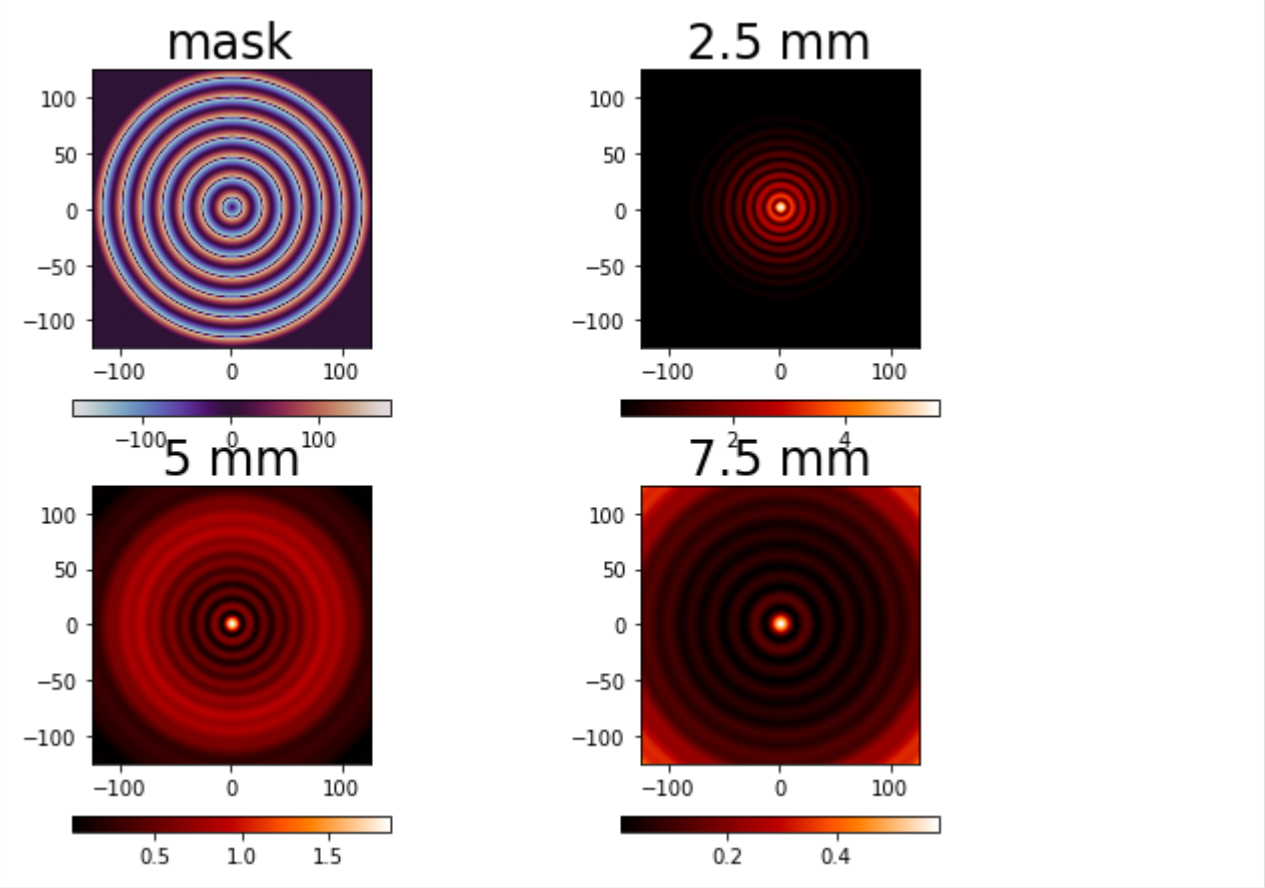
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.axicon(r0=(0 * um, 0 * um), radius=125 * um, angle=4*degrees, refractive_index=1.5)

u2 = u1 * t1
u3 = u2.RS(z=2.5 * mm, new_field=True)

u4 = u2.RS(z=5 * mm, new_field=True)

u5 = u2.RS(z=7.5 * mm, new_field=True)

draw_several_fields((u2, u3, u4, u5), titles=('mask', '2.5 mm', '5 mm', '7.5 mm'),␣
↪logarithm=True,
                    kinds=['phase', 'intensity', 'intensity', 'intensity'])
```



7.5.14 Laguerre spiral

[17]:
```
num_pixels = 512

length = 250 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.laguerre_gauss_spiral(
    kind='intensity', n=0, l=12, r0=(0 * um, 0 * um), w0=20 * um, z=1000 * um)

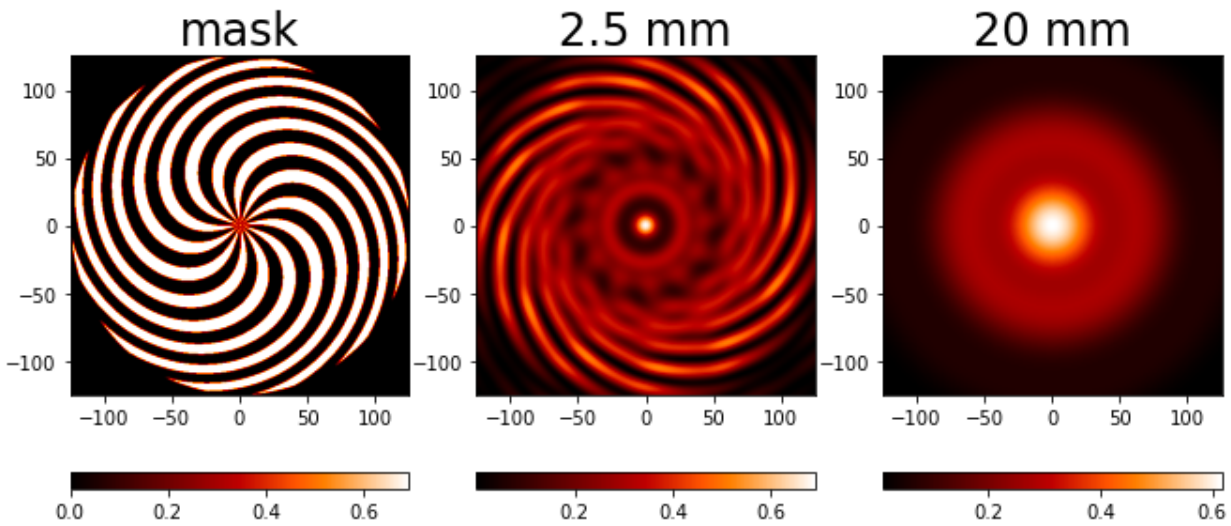
u2 = u1 * t1
u3 = u2.RS(z=5 * mm, new_field=True)

u4 = u2.RS(z=20 * mm, new_field=True)


draw_several_fields((u2, u3, u4), titles=('mask', '2.5 mm', '20 mm'), logarithm=True)
```

(continues on next page)

(continued from previous page)



7.5.15 forked grating

[18]:
```
period = 20*um
num_pixels = 1024

length = 250 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.forked_grating(kind='amplitude',
                  r0=(0 * um, 0 * um), period=period, l=3, alpha=2, angle=0 * degrees)

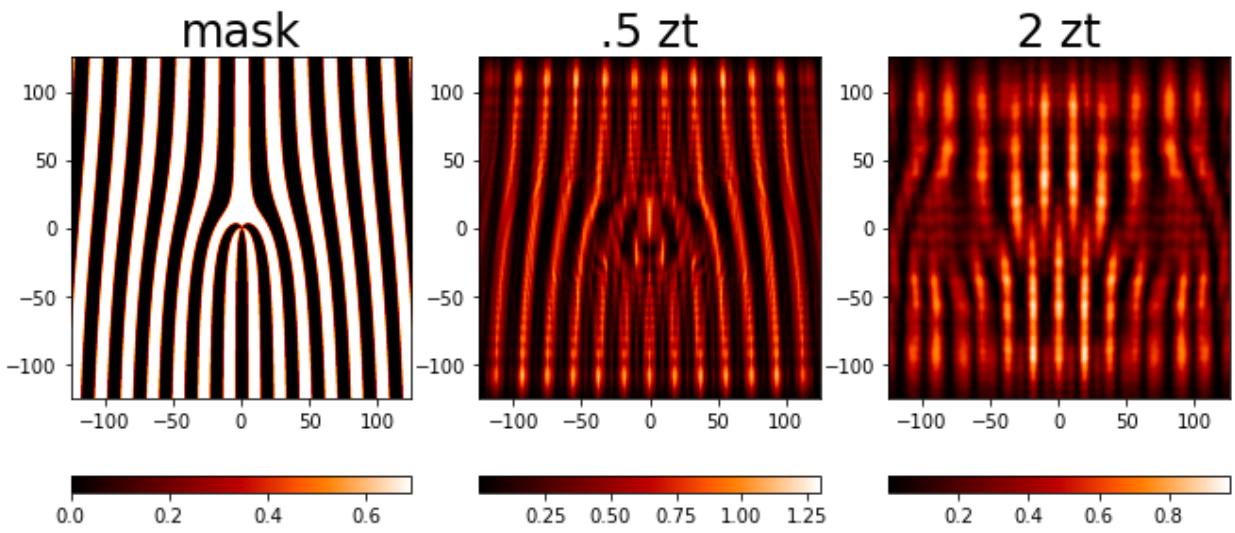
zt = 2 * period**2 / wavelength

u2 = u1 * t1

u4 = u2.RS(z=.5 * zt, new_field=True)

u5 = u2.RS(z=2 * zt, new_field=True)


draw_several_fields((u2, u4, u5), titles=('mask', '.5 zt', '2 zt'), logarithm=True)
```



7.5.16 Roughness and speckle

[19]:
```
num_pixels = 512

length = 250 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1, theta=0 * degrees, phi=0 * degrees)

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.roughness(t=(20 * um, 20 * um), s=1 * um)

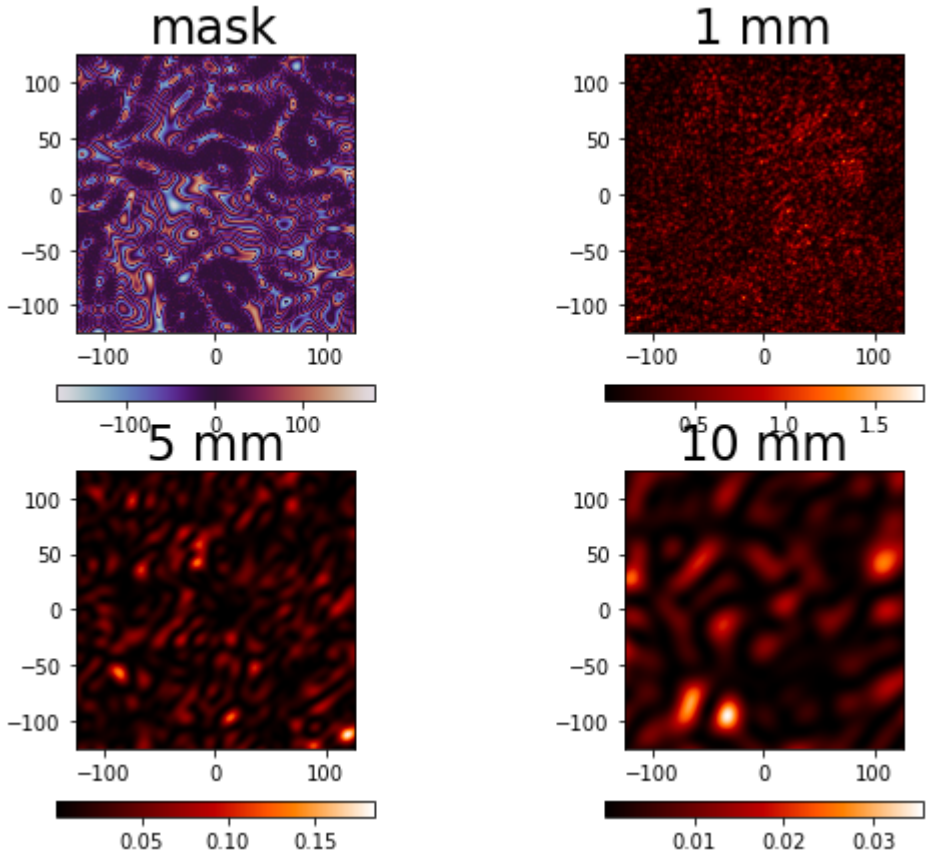
u2 = u1 * t1

u3 = u2.RS(z=1 * mm, new_field=True)

u4 = u2.RS(z=5 * mm, new_field=True)

u5 = u2.RS(z=10 * mm, new_field=True)

draw_several_fields((u2, u3, u4, u5), titles=('mask', '1 mm', '5 mm', '10 mm'),␣
↪logarithm=True,
                    kinds=['phase', 'intensity', 'intensity', 'intensity'])
```



7.5.17 Blazed grating

Right phase: 2pi

[20]:
```
num_pixels = 512

length = 500 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

focal = 5 * mm

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1)
u1.pupil()

t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.blazed_grating(period=20 * um,
                  phase_max=2 * np.pi,
                  x0=0 * um,
                  angle=0 * degrees)
```

(continues on next page)

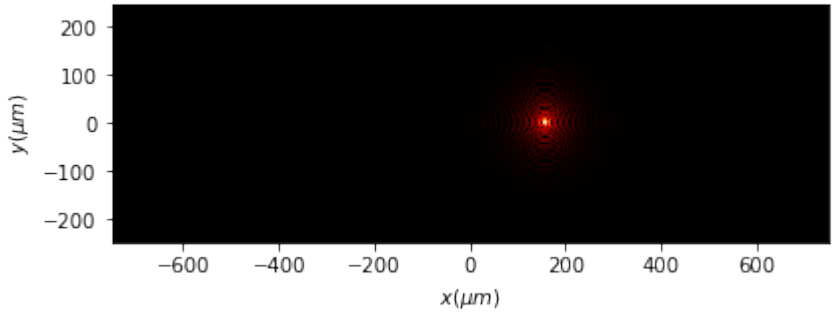
(continued from previous page)

```
t2 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t2.lens(r0=(0, 0), focal=focal)

u2 = u1 * t1 * t2

u3 = u2.RS(z=focal, new_field=True, amplification=(3, 1))

u3.draw(logarithm=1e1)
```



Wrong phase: 1.5pi

[21]:
```
num_pixels = 512

length = 500 * um
x0 = np.linspace(-length / 2, length / 2, num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

focal = 5 * mm

u1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u1.plane_wave(A=1)
u1.pupil()

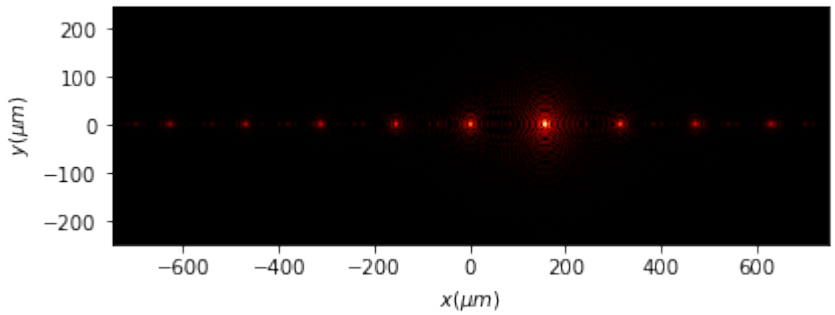
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.blazed_grating(period=20 * um,
                  phase_max=1.5 * np.pi,
                  x0=0 * um,
                  angle=0 * degrees)

t2 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t2.lens(r0=(0, 0), focal=focal)

u2 = u1 * t1 * t2

u3 = u2.RS(z=focal, new_field=True, amplification=(3, 1))

u3.draw(logarithm=1e1)
```



7.6 Scattering by a small cylinder

[8]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees

from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

[9]:
```
from matplotlib import cm
```

[10]:
```
def scattering_small_cylinder(n_out=1.33,
                              n_in=1.59,
                              r_sphere=1 * um,
                              wavelength=600 * nm):
    x0 = np.linspace(-2 * r_sphere, 2 * r_sphere, 1024)
    z0 = np.linspace(0, 4 * r_sphere, 1024)
    u0 = Scalar_source_X(x0, wavelength)
    u0.plane_wave(A=1, theta=0 * degrees)

    u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=n_out)
    u1.incident_field(u0)
    mascara = u1.sphere(r0=(0 * um, 1.25 * r_sphere),
                        radius=(r_sphere, r_sphere),
                        angle=0,
                        refractive_index=n_in)

    u1.WPM(has_edges=False, verbose=False)

    u1.draw(kind='intensity',
            logarithm=True,
            draw_borders=True,
            scale='scaled')

    plt.suptitle("$n_{{out}} = {:2.2f}\,\, /\,\, n_{{in}} = {:2.2f}$".format(
        n_out, n_in),
                 fontsize=15)
```

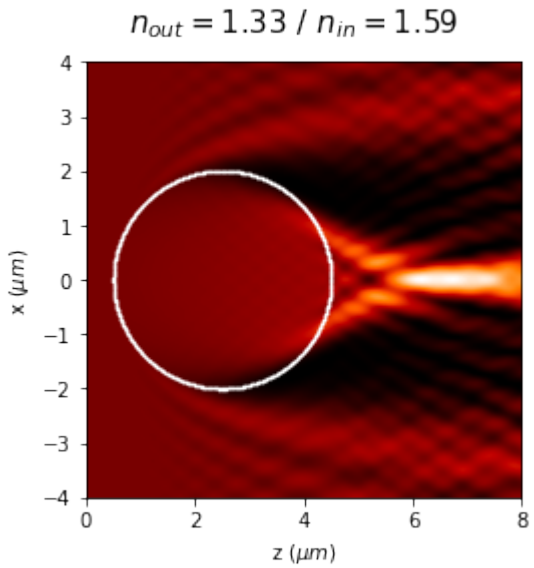
(continues on next page)

(continued from previous page)

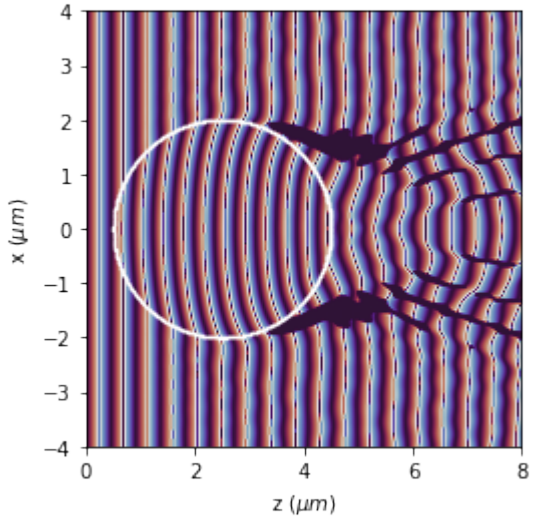
```
    return u1
```

7.6.1 Scattering by a small, transparent cylinder

[11]:
```
u1 = scattering_small_cylinder(n_out=1.33,
                               n_in=1.59,
                               r_sphere=2 * um,
                               wavelength=600 * nm)
```

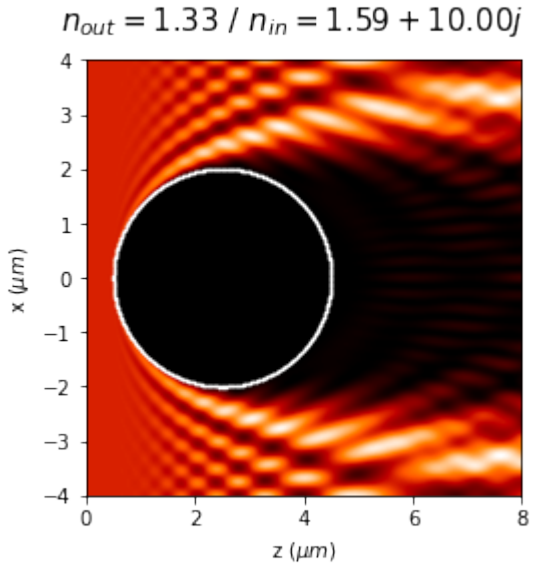


[12]:
```
u1.draw(kind='phase', logarithm=False, draw_borders=True, scale='scaled')
```

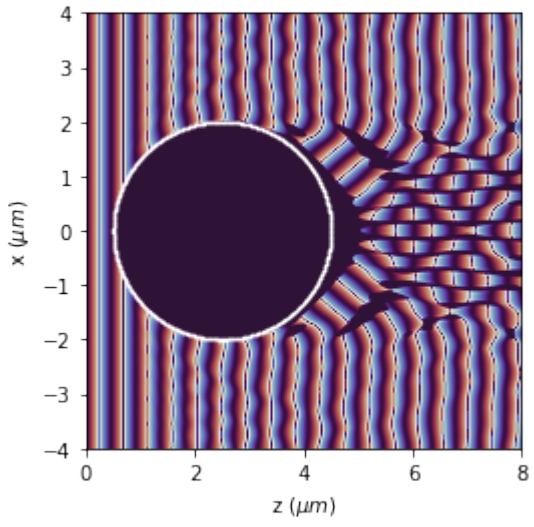


7.6.2 Scattering by a small, absorbent cylinder

[13]:
```
u1 = scattering_small_cylinder(n_out=1.33,
                               n_in=1.59 + 10j,
                               r_sphere=2 * um,
                               wavelength=600 * nm)
```



[14]:
```
u1.draw(kind='phase', logarithm=False, draw_borders=True, scale='scaled')
```



[1]:
```
%load_ext autoreload
%autoreload 2
```

7.7 Inteferences using Fresnel biprism

A classical experiment in Optics is the interference process by a Fresnel biprism. In order to obtain a interference, a monocromatic light beam needs to be divided into two coherent waves and join them again. This can be performed using a Fresnel biprism.

To analyze the inteference process, we have used the XY and XZ framework.

7.7.1 XY scheme

[2]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

In the first place, we create a plane wave and mask it with a circular aperture.

[3]:
```
x0 = np.linspace(-.2 * mm, .2 * mm, 1024)
y0 = np.linspace(-.2 * mm, .2 * mm, 1024)
wavelength = 0.6238 * um
```

[4]:
```
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.plane_wave()

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.circle(r0=(0 * um, 0 * um), radius=(.2 * mm, .2 * mm))
```

After that, we generate a Fresnel Biprism mask. The field just after the Fresnel biprism is just the multiplication of the field by the two masks.

[5]:
```
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.biprism_fresnel(r0=(0 * um, 0 * um), width=.2 * mm, height=5 * um, n=1.5)

u1 = t0 * t1 * u0
```

Then, we create a function in order to draw the intensity distribution at a distance z after the mask.

[6]:
```
def difraccion_biprism_Fresnel(u1, z_mask):
    """Computes and draws the intensity distribution of a light beam after a mask.

    Parameters:
    u1 (Scalar_field_XY): Field just after the mask.
    z_mask (float): Distance from the mask."""

    if z_mask > 0:
        u2 = u1.RS(z=z_mask, new_field=True)
    else:
        u2 = u1
```

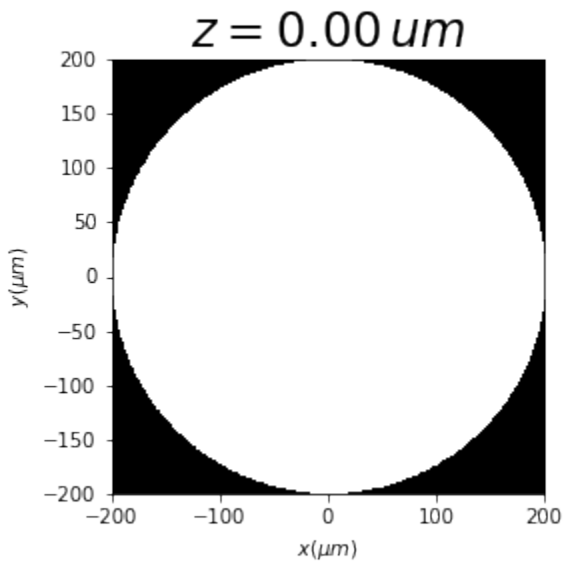
(continues on next page)

(continued from previous page)

```
    u2.draw(kind='intensity')
    plt.title("$z={:2.2f}\,um$".format(z_mask), fontsize=24)
```

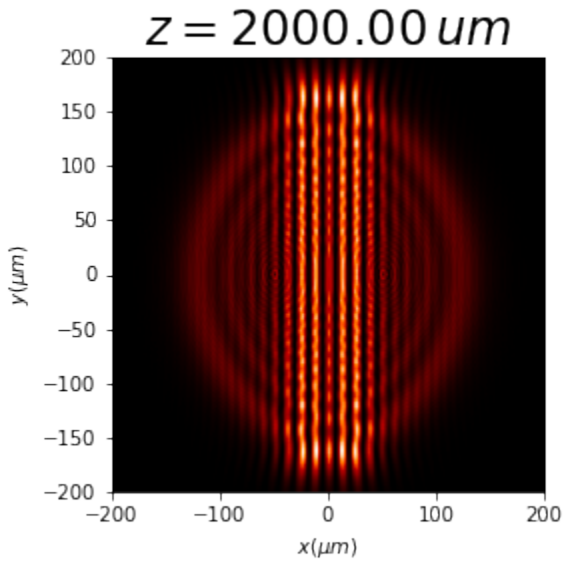
Now, let us show what is the light field just after the grating

[7]:
```
difraccion_biprism_Fresnel(u1, z_mask=0 * mm)
```



And now, at different increasing distances:

[8]:
```
difraccion_biprism_Fresnel(u1, z_mask=2 * mm)
```



7.7.2 XZ scheme

Now, let us show the same case for XZ scheme. In this case, we can see how light is modified by the briprism, as no Thin Element Approximation is necessary.

[9]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[79]:
```
num_x = 1024 * 2
num_z = 1024 * 1
x0 = np.linspace(-75, 75, num_x)
z0_near = np.linspace(0 * um, 25 * um, num_z)
z0_all = np.linspace(0 * um, 1000 * um, num_z)

wavelength = 0.6238 * um
```

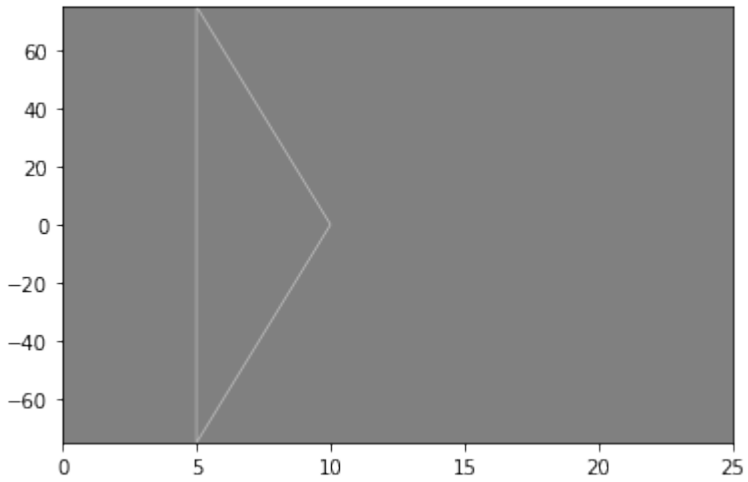
[80]:
```
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave()
```

[93]:
```
t0 = Scalar_mask_XZ(x=x0, z=z0_near, wavelength=wavelength)
t0.biprism(r0=(0 * um, 5 * um),
           length=150 * um,
           height=5 * um,
           refractive_index=1.5,
           angle=0 * degrees)
t0.incident_field(u0)
```

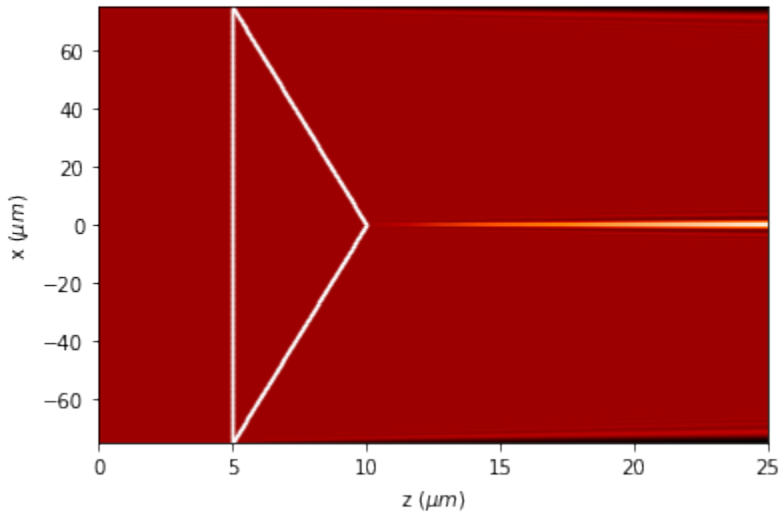
[94]:
```
t0.WPM(has_edges=False, verbose=True)
```

```
Time = 1.10 s, time/loop = 1.073 ms
```

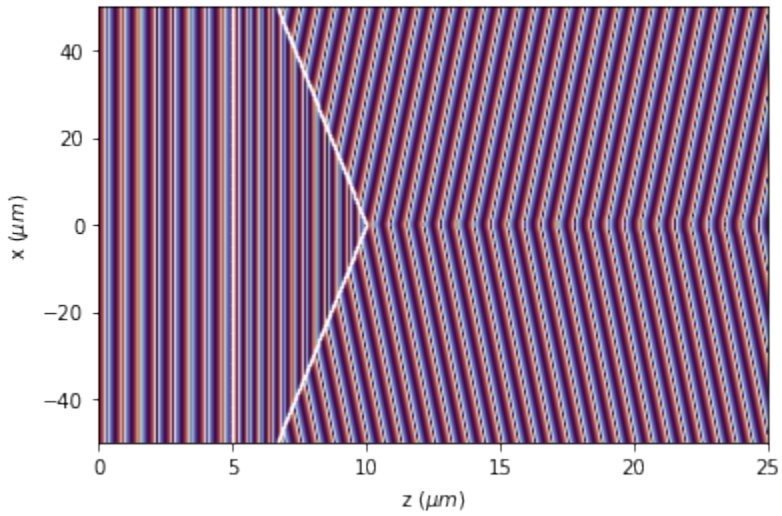
[95]:
```
t0.surface_detection(has_draw=True)
```



[96]:
```
t0.draw(kind='intensity', draw_borders=True)
```



[97]:
```
t0.draw(kind='phase', draw_borders=True)
plt.xlim(0, 25)
plt.ylim(-50, 50)
```

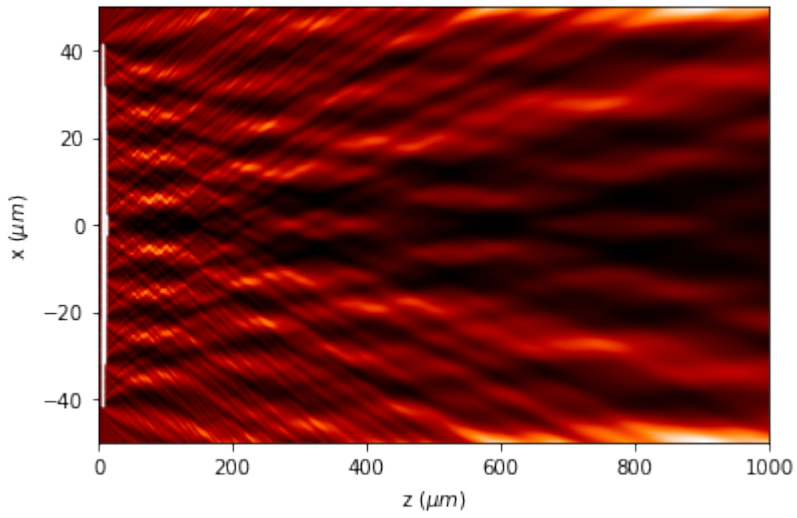


[98]:
```
u_far = Scalar_mask_XZ(x=x0, z=z0_all, wavelength=wavelength)
u_far.biprism(r0=(0 * um, 5 * um),
              length=100 * um,
              height=5 * um,
              refractive_index=1.5,
              angle=0 * degrees)
u_far.incident_field(u0)
```

[99]:
```
u_far.WPM(verbose=True)
```

```
Time = 0.88 s, time/loop = 0.8577 ms
```

[100]:
```
u_far.draw(kind='intensity', draw_borders=True)
plt.ylim(-50 * um, 50 * um)
```



Here, we see the intensity distribution at z=500μm, showing the interference process.

7.7.3 Polarization

[19]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

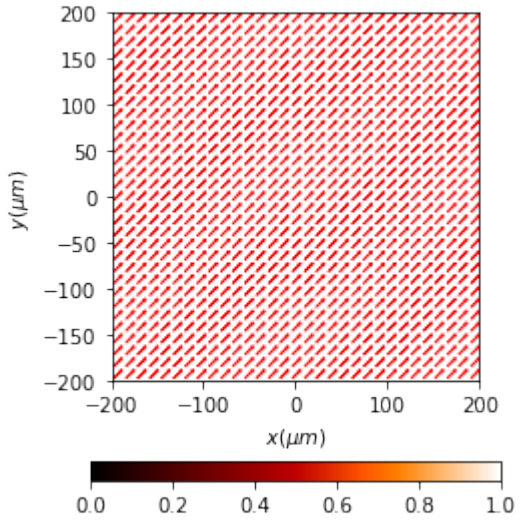
from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_fields_XY import Vector_field_XY
```

[20]:
```
x0 = np.linspace(-.2 * mm, .2 * mm, 512)
y0 = np.linspace(-.2 * mm, .2 * mm, 1024)
wavelength = 0.6238 * um
```

[21]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)
```

[22]:
```
EM0 = Vector_source_XY(x0, y0, wavelength)
EM0.constant_polarization(u0, v=[1 / np.sqrt(2), 1 / np.sqrt(2)])
```

[23]:
```
EM0.__draw_ellipses__(num_ellipses=(31, 31),
                      amplification=.75,
                      color_line='r',
                      head_width=2,
                      logarithm=False,
                      normalize=True,
                      cut_value='')
```



[24]:
```
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.circle(r0=(0 * um, 0 * um), radius=(200 * um, 200 * um))
```

[25]:
```
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.biprism_fresnel(r0=(0 * um, 0 * um), width=200 * um, height=5 * um, n=1.5)

u1 = t0 * t1
```

[26]:
```
state = np.array([[0.5, 0.5],[0.5, 0.5]])
EM1 = Vector_mask_XY(x0, y0, wavelength)
EM1.scalar_to_vector_mask(mask=u1, state= state)

# The state can also be generated using py_pol.Jones_matrix().
```

[27]:
```
t2 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t2.two_levels(level1=0, level2=1)
```

[28]:
```
EM2 = Vector_mask_XY(x0, y0, wavelength)
EM2.complementary_masks(t2,
                        state_0=np.array([[1, 0], [0, 0]]),
                        state_1=np.array([[0, 0], [0, 1]]))
```

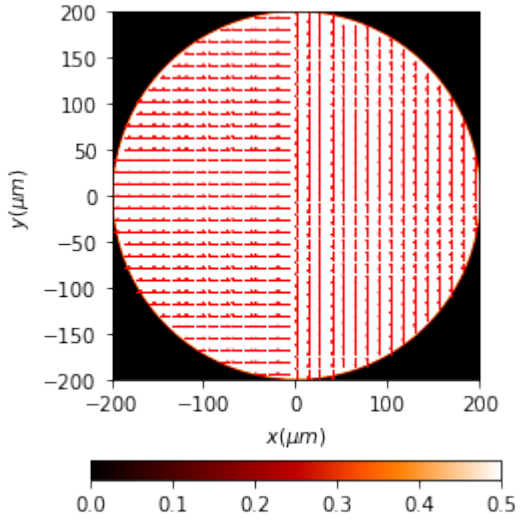
[29]:
```
EM3 = EM0 * EM1 * EM2
```

[30]:
```
EM3.draw('ellipses',
         num_ellipses=(31, 31),
         color_line='r',
         amplification=0.75,
```

(continues on next page)

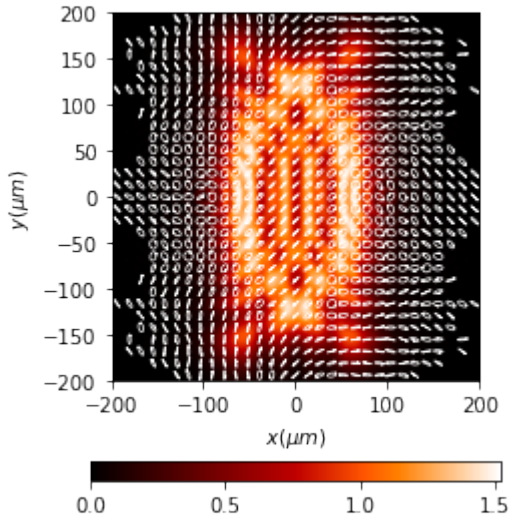
(continued from previous page)

```
        head_width=1)
```



[31]:
```
EM4 = EM3.VRS(z=4 * mm)
```

[32]:
```
EM4.__draw_ellipses__(num_ellipses=(31, 31),
                      color_line='w',
                      amplification=0.75,
                      head_width=1)
```



7.8 Focusing light with a lens

In this example, we will analyze how light is focused using a lens. We will use several algorithms. From an ondulatory point of view, a lens as a transmittance object

$$t(\xi,\eta) = P(\xi,\eta)\exp\left[-ik\frac{(\xi^2+\eta^2)}{2f'}\right]$$

where $P(\xi,\eta)$ is the shape of the lens, $k = 2\pi/\lambda$ and f' is the focal distance of the lens.

7.8.1 Lens in x mode

[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.utils_optics import beam_width_1D, FWHM1D, MTF_ideal, MTF_parameters
```

[2]:
```
focal = 25 * mm
diameter = 4 * mm

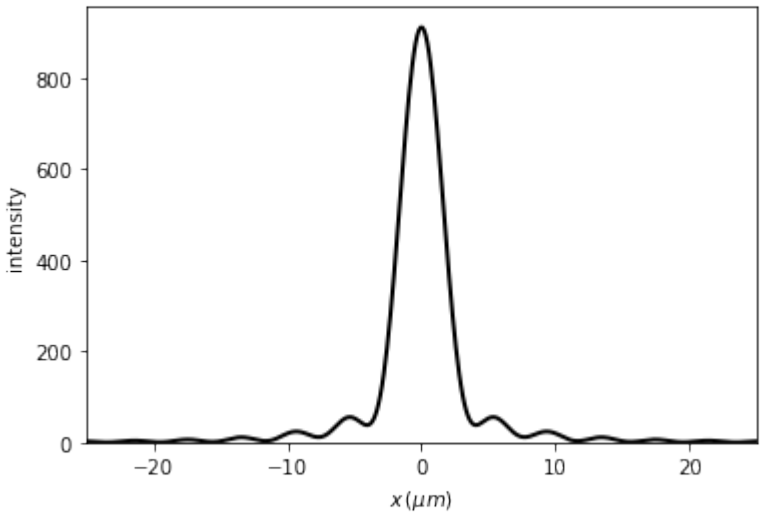
# Initial parameters
x0 = np.linspace(-diameter / 2 - 50 * um, diameter / 2 + 50 * um, 1024 * 32)
wavelength = 0.6238 * um

# Definition of source
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1)

t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.lens(x0=0.0, focal=focal, radius=diameter / 2)
```

[4]:
```
u1 = t0 * u0
u2 = u1.RS(z=focal, verbose=False)
```

[5]:
```
u2.draw()
plt.xlim(-25, 25)
```

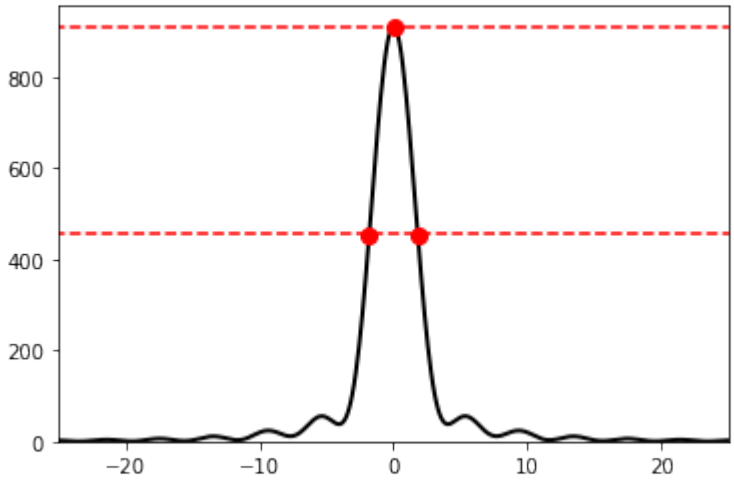


7.8.1.1 Beam width computation

[6]:
```
width, center = beam_width_1D(u2.u, u2.x)
```

[7]:
```
fwhm = FWHM1D(u2.x, np.abs(u2.u)**2, has_draw=True, percentage=0.5)
plt.xlim(-25, 25)
plt.ylim(bottom=0)
print(width, fwhm)
```

```
4.8440193446230575 3.610817918990506
```



7.8.1.2 MTF

[8]:
```
freq_real, mtf_norm = u2.MTF(has_draw=False)

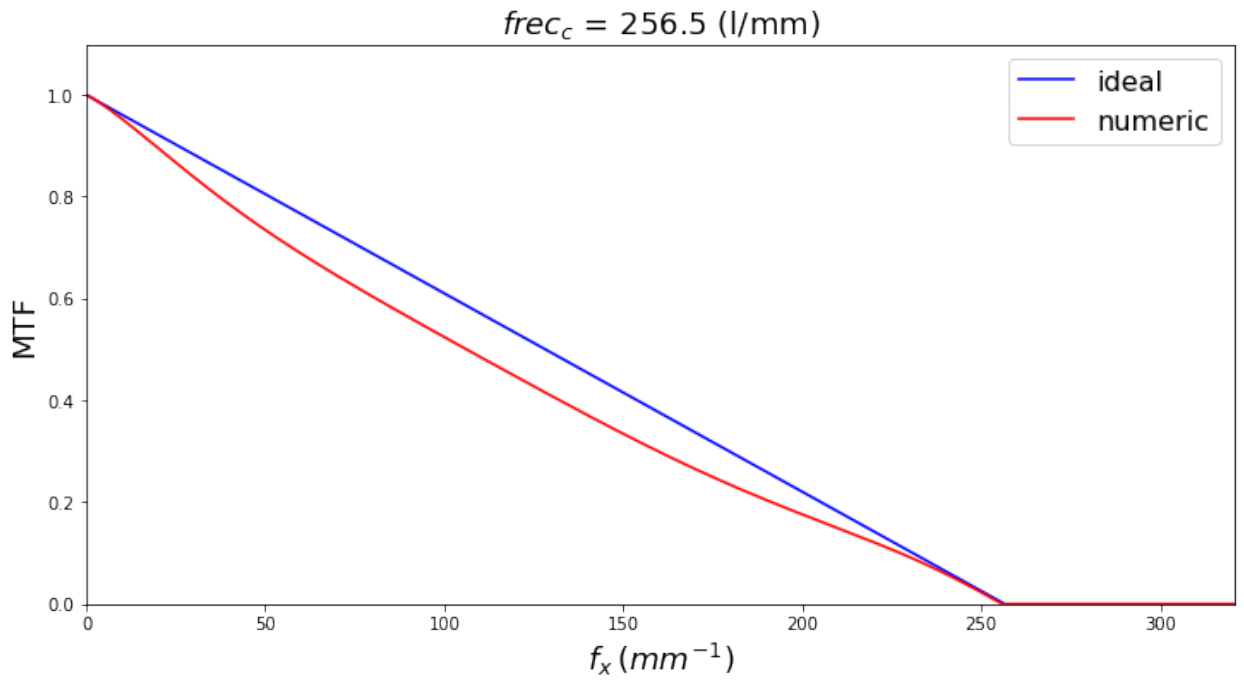
freq_ideal = np.linspace(0, 350, 600)
mtf_ideal, freq_corte = MTF_ideal(freq_ideal,
                                  wavelength=wavelength,
                                  diameter=diameter,
                                  focal=focal,
                                  kind='1D',
                                  has_draw=False,
                                  verbose=True)

plt.figure(figsize=(12, 6))
plt.plot(freq_ideal, mtf_ideal, 'b', label='ideal')
plt.xlabel("$f_x\,(mm^{-1})$", fontsize=18)
plt.ylabel("MTF", fontsize=18)
plt.xlim(left=0, right=1.25 * freq_corte)
plt.ylim(bottom=0, top=1.1)
plt.title("$frec_c$ = {:2.1f} (l/mm)".format(freq_corte), fontsize=18)

plt.plot(freq_real, mtf_norm, 'r', label='numeric')

plt.legend(fontsize=16)
```

```
frecuencia de bin_level = 256.49 lineas/mm
```

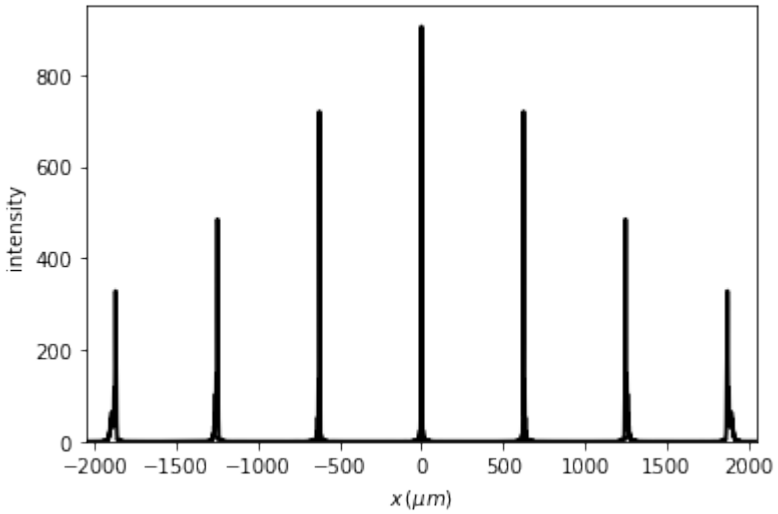


7.8.1.3 Several rays

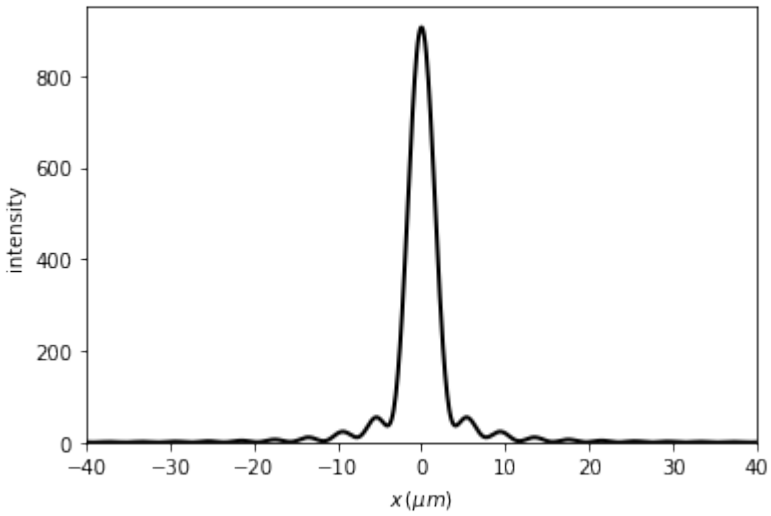
[9]:
```
# Definition of source
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_waves_several_inclined(A=1, num_beams=7, max_angle=10 * degrees)
```

[10]:
```
us1 = t0 * u0
us2 = us1.RS(z=focal, verbose=False)
```

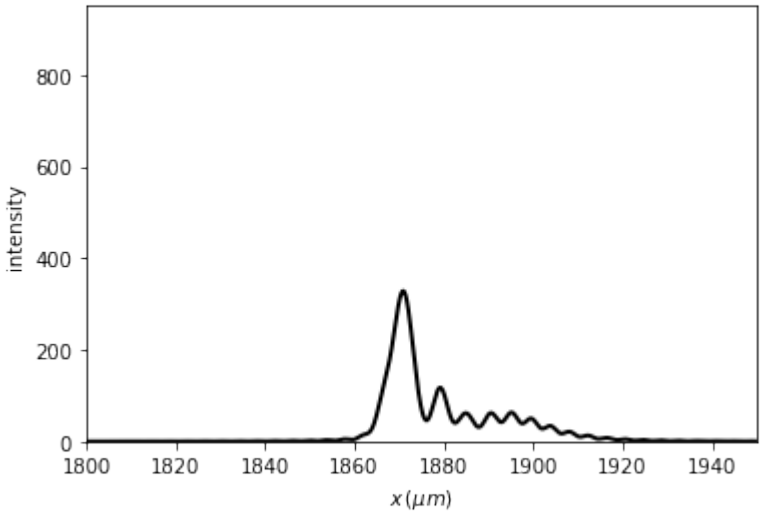
[11]:
```
us2.draw()
```



[12]:
```
us2.draw()
plt.xlim(-40, 40)
```



[13]:
```
us2.draw()
plt.xlim(1800, 1950)
```



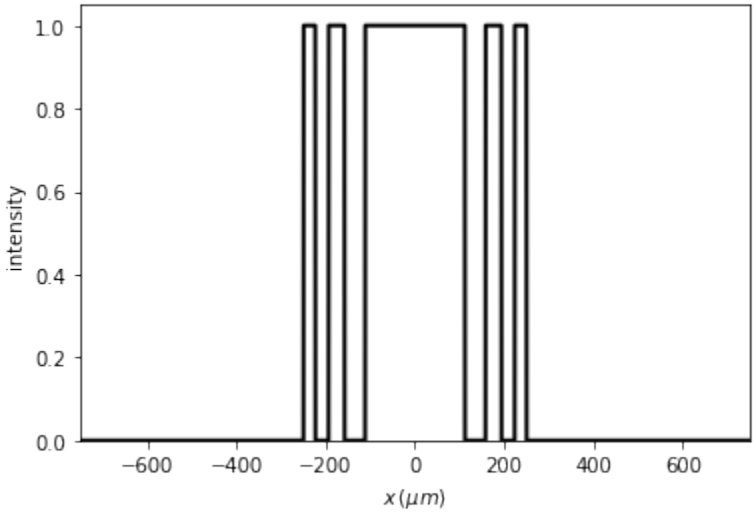
7.8.1.4 Binary Fresnel lens

With the modules diffractio.scalar_sources_X, scalar_fields_X and scalar_masks_X for propagating a field generated by a scalar light only the propagation to a given distances is performed. If you need to visualize the z propagation it is better the use of `diffractio.scalar_fields_XZ`, since it performs a for loop over a number of distances given by a linspace array. This for loop is executed, when possible, using multiprocessing, since there is not computational interaction between different locations z.

[14]:
```
# initial data
x0 = np.linspace(-750 * um, 750 * um, 1024 * 32)
wavelength = 0.6238 * um
focal = 20 * mm
radius = 250 * um

#definition of source
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

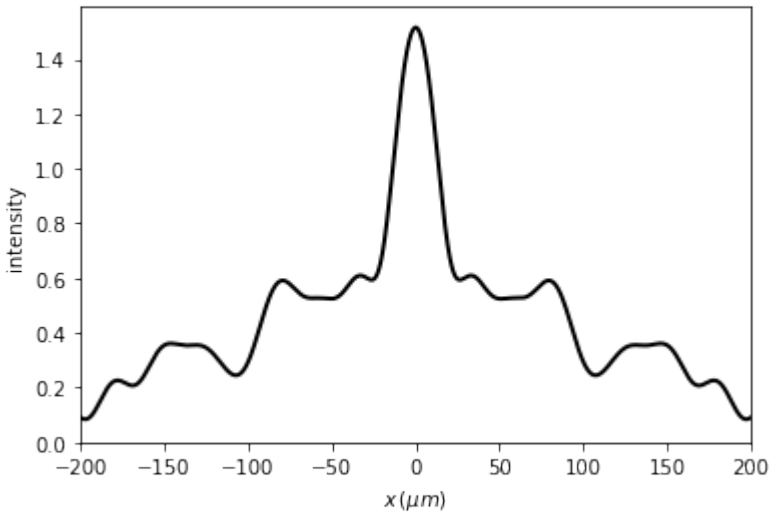
[15]:
```
#definition of mask
t0 = Scalar_mask_X(x0, wavelength)
t0.fresnel_lens(x0=0 * um,
                radius=radius,
                focal=focal,
                kind='amplitude',
                phase=np.pi)
t0.draw(kind='intensity')
```



[16]:
```
u1 = t0.RS(z=20 * mm)
```

```
Good result: factor 254.34
```

[17]:
```
u1.draw(logarithm=1, normalize='intensity')
plt.xlim(-200, 200)
```



7.8.2 XZ lens: CZT scheme

[18]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

[19]:
```
focal = 15 * mm
diameter = 4 * mm

x0 = np.linspace(-diameter / 2, diameter / 2, 1024)
wavelength = 0.6238 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)

t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.lens(x0=0.0, radius=diameter / 2, focal=focal)

u1 = u0 * t0
```

[25]:
```
z0 = np.linspace(14.8 * mm, 15.2 * mm, 512)
```

This procedure is deprecated. It is more ineficient than with CZT algorithm.

It requires plt.xlim(-20,20) after drawing. The drawing is badly sampled.

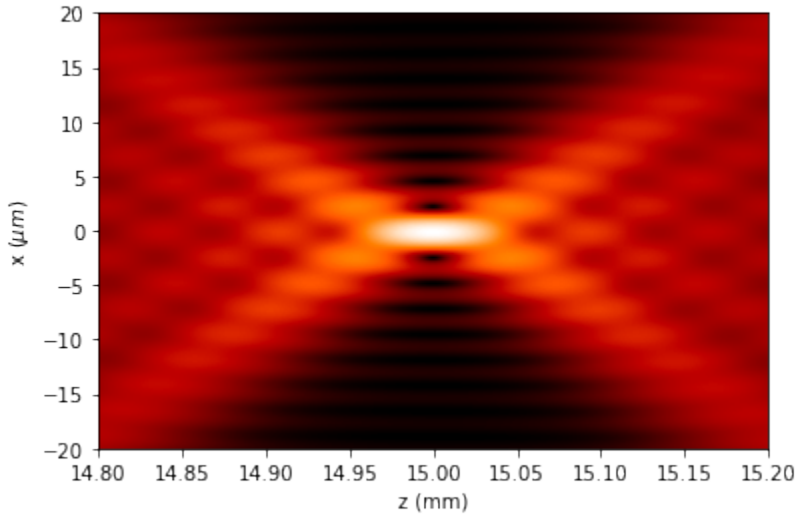
[30]:
```
%%time
u2 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u2.incident_field(u1)
u2.RS(num_processors=4);
```

```
CPU times: user 539 ms, sys: 303 ms, total: 842 ms
Wall time: 1.5 s
```

[31]:
```
%%time
xout=np.linspace(-20,20,256)
u2 = u1.CZT(z=z0, xout=xout)
```

```
CPU times: user 1.36 s, sys: 10.4 ms, total: 1.37 s
Wall time: 1.43 s
```

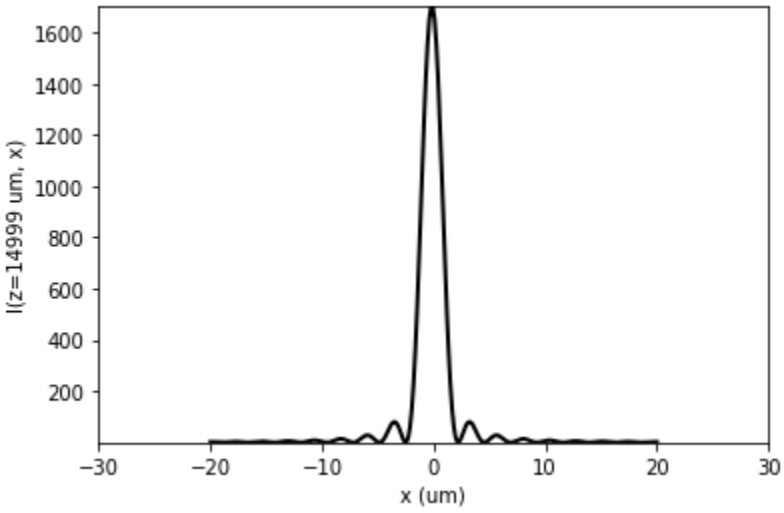
[32]:
```
u2.draw(logarithm=1e-1, z_scale='mm')
```



[33]:
```
x_f, z_f = u2.search_focus()
```

```
x = -0.235 um, z = 14999.609 um
```

[34]:
```
u2.profile_transversal(z0=z_f)
plt.xlim(-30, 30)
```



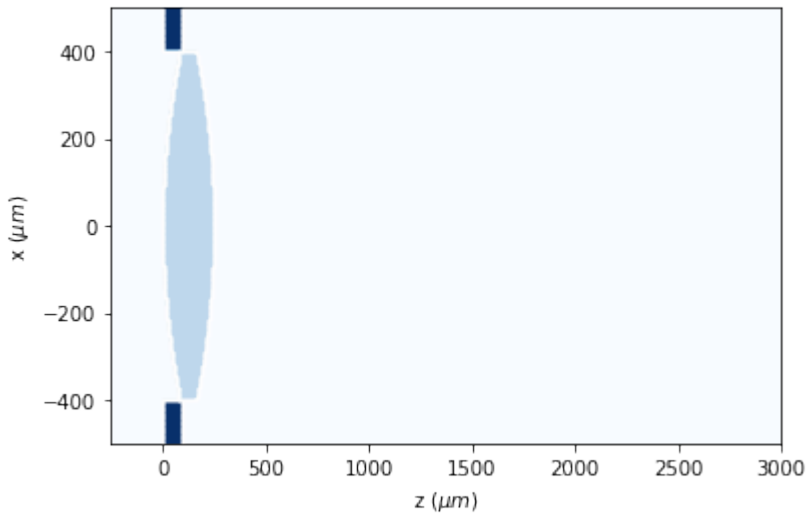
7.8.2.1 XZ lens: BPM scheme

[35]:
```
focal = 1 * mm
diameter = 1 * mm

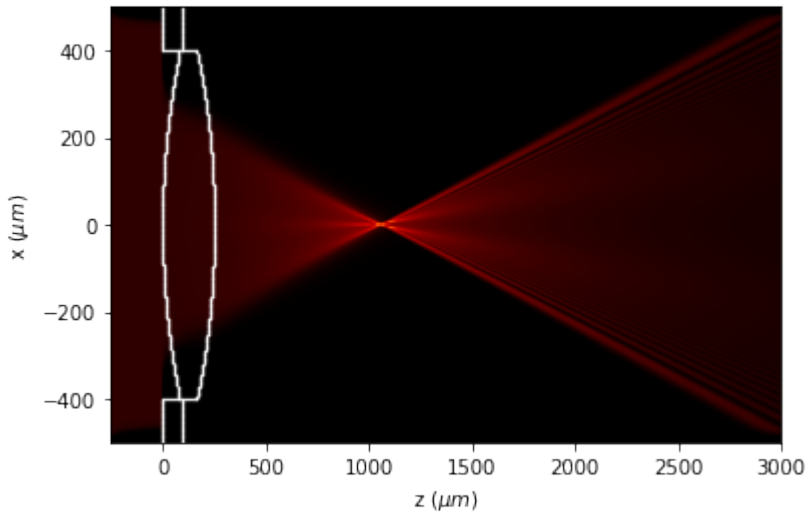
x0 = np.linspace(-diameter / 2, diameter / 2, 1024 * 4)
z0 = np.linspace(-250 * um, focal + 2 * mm, 1024)
wavelength = 0.6238 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=diameter / 1.25,
                              radius=(1 * mm, -1 * mm),
                              thickness=.25 * mm,
                              refractive_index=1.55,
                              angle=0 * degrees,
                              mask=(100 * um, 3 + 0.1j))
print(focal)
u1.draw_refractive_index()
edge_matrix = u1.borders

u1.smooth_refractive_index(type_filter=2, pixels_filtering=50)
```

```
870.4808704808705
```



[36]:
```
u1.BPM(verbose=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edge_matrix)
```

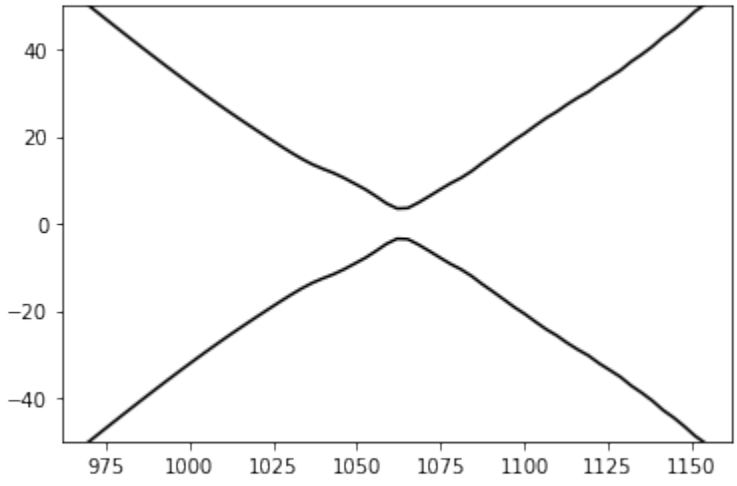


[37]:
```
x_focus, z_focus = u1.search_focus()
```

```
x = -0.122 um, z = 1062.072 um
```

[38]:
```
width, z_center = u1.beam_widths(kind='sigma4', has_draw=(0, 0))
```

[39]:
```
plt.plot(u1.z, width, 'k')
plt.plot(u1.z, -width, 'k')
plt.xlim(z_focus - 100 * um, z_focus + 100 * um)
plt.ylim(-50, 50)
```



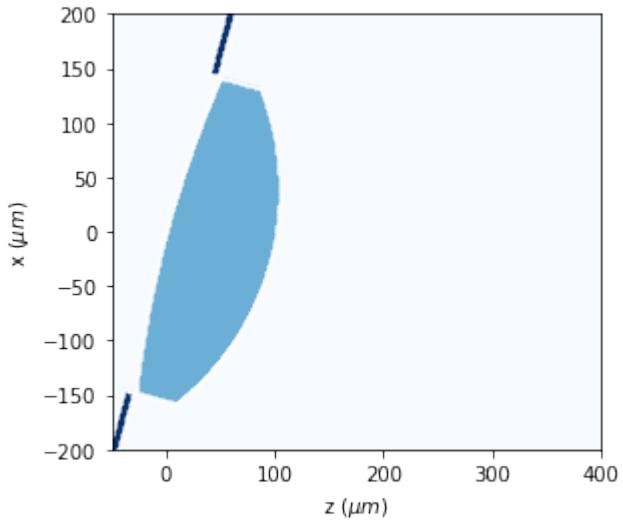
7.8.3 Effect of rotation

[40]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024 * 2)
z0 = np.linspace(-50 * um, 400 * um, 1024 * 4)
wavelength = 0.6238 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

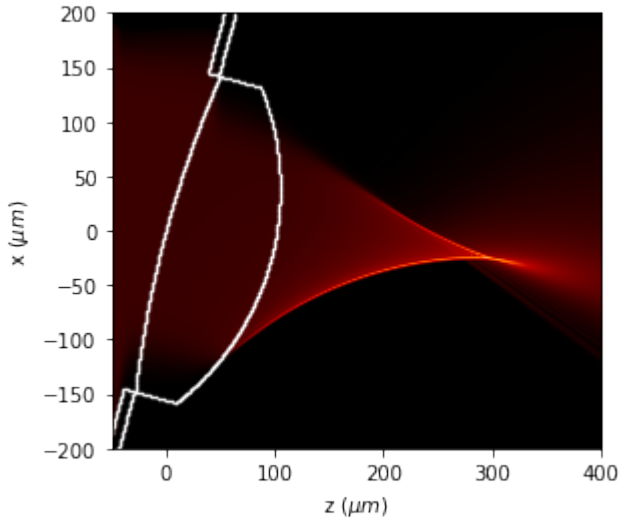
[41]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=300 * um,
                              radius=(1000 * um, -250 * um),
                              thickness=100 * um,
                              refractive_index=2,
                              angle=15 * degrees,
                              mask=(10 * um, 3 + 0.05j))
print(focal)
u1.draw_refractive_index(scale='scaled')
edge_matrix = u1.borders

u1.smooth_refractive_index(pixels_filtering=25, type_filter=2)
```

```
192.30769230769232
```



[42]:
```
u1.BPM(has_edges=True, verbose=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        scale='scaled',
        edge_matrix=edge_matrix)
```



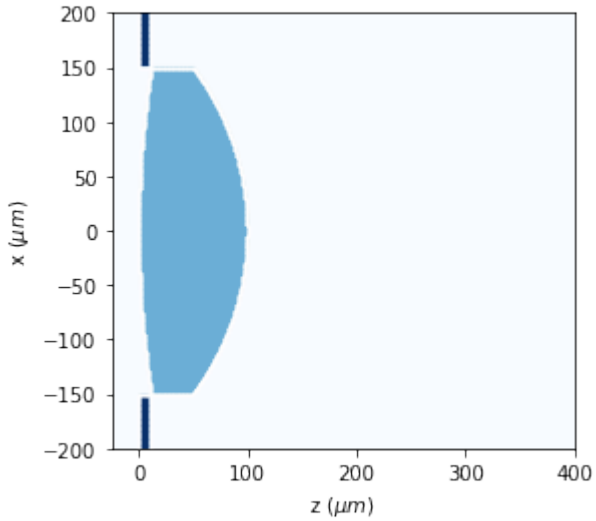
Here we see how several rays converge

[45]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024 * 1)
z0 = np.linspace(-25 * um, 400 * um, 1024 * 4)
wavelength = 0.6238 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beams_several_parallel(A=1,
                                num_beams=15,
                                w0=5,
                                z0=10,
                                x_central=0.0,
                                x_range=300.0,
                                theta=0.0)
```

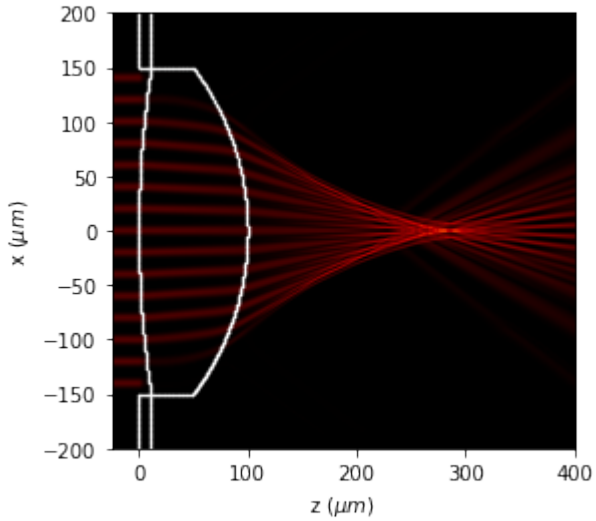
[46]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=300 * um,
                              radius=(1000 * um, -250 * um),
                              thickness=100 * um,
                              refractive_index=2,
                              angle=0 * degrees,
                              mask=(10 * um, 3 + 0.05j))
print(focal)
u1.draw_refractive_index(scale='scaled')
edge_matrix = u1.borders

u1.smooth_refractive_index(pixels_filtering=25, type_filter=2)
```

```
192.30769230769232
```



[47]:
```
u1.BPM(verbose=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        scale='scaled',
        edge_matrix=edge_matrix)
```



7.8.3.1 XY scheme

[48]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

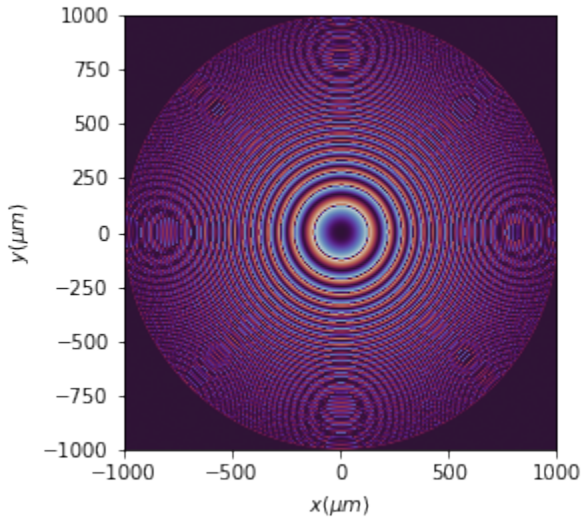
[49]:
```
diameter = 2 * mm
focal = 25 * mm

x0 = np.linspace(-diameter / 2, diameter / 2, 1024)
y0 = np.linspace(-diameter / 2, diameter / 2, 1024)
wavelength = 0.6238 * um
```

[50]:
```
u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0.plane_wave()

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.lens(r0=(0 * um, 0 * um),
        radius=(diameter / 2, diameter / 2),
        focal=(focal, focal))
```

[51]:
```
t0.draw('phase')
```



[52]:
```
u1 = u0 * t0
```

[53]:
```
u2 = u1.RS(z=focal)
```

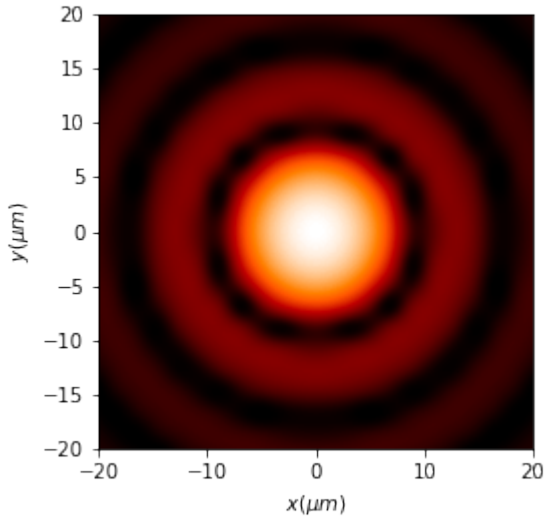
[55]:
```
u2.cut_resample(x_limits=(-20, 20),
                y_limits=(-20, 20),
```

(continues on next page)

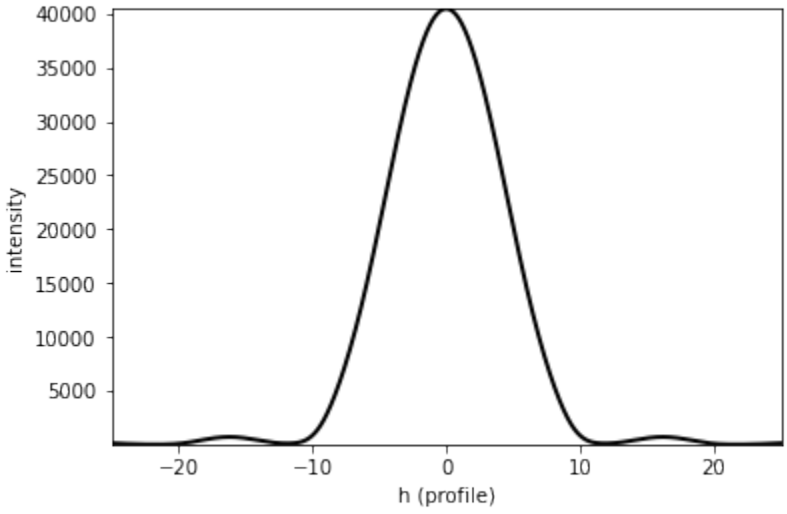
(continued from previous page)

```
                num_points=(1024, 1024),
                new_field=False,
                interp_kind=(3, 1))
```

[56]:
```
u2.draw(logarithm=1e-2)
```



[57]:
```
u2.draw_profile(point1=(-25, 0),
                point2=(25, 0),
                npixels=2048,
                kind='intensity',
                order=2)
```



[58]:
```
def intensity_area(u, r0, radius, power=1, has_draw=True):
    intensities = np.zeros_like(radius)
```

(continues on next page)

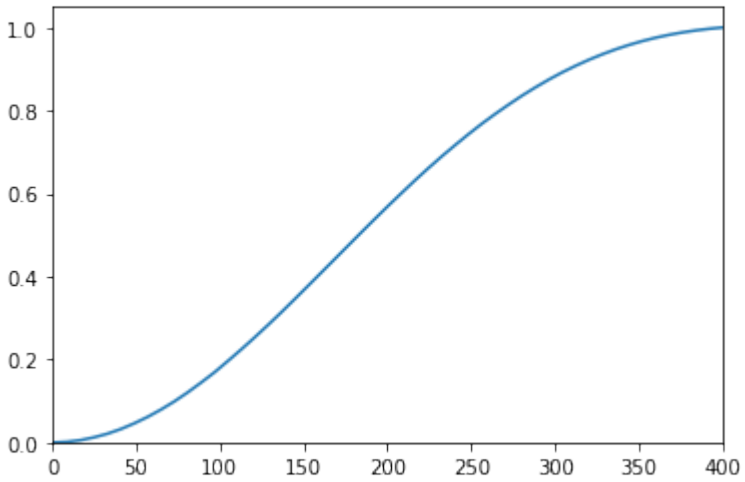
(continued from previous page)

```
    mask = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)

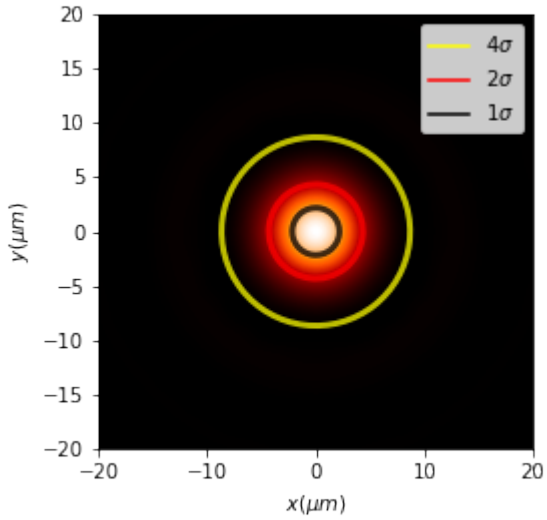
    for i, r in enumerate(radius):
        mask.circle(r0=r0, radius=(r, r), angle=0)
        masked_field = u * mask
        intensities[i] = masked_field.intensity().sum()**power

    intensities = intensities / intensities.max()
    if has_draw is True:
        plt.figure()
        plt.plot(rs, intensities)
        plt.ylim(bottom=0)
        plt.xlim(left=0, right=rs[-1])
    return intensities
```

[59]:
```
rs = np.linspace(1 * um, 400 * um, 51)
intensities = intensity_area(u2, r0=(0, 0), radius=rs, power=1)
```



[60]:
```
u2.beam_width_4s(has_draw=True)
```



7.8.4 Vector scheme

7.8.4.1 Linear polarization

[61]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_fields_XY import Vector_field_XY
```

[62]:
```
diameter = 2 * mm
x0 = np.linspace(-diameter / 2, diameter / 2, 1024)
y0 = np.linspace(-diameter / 2, diameter / 2, 1024)
wavelength = 0.6238 * um
```

[63]:
```
focal = 25 * mm
limit = 50 * um
```

[64]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.lens(r0=(0 * um, 0 * um),
        radius=(diameter / 2, diameter / 2),
        focal=(focal, focal))
```

(continues on next page)

(continued from previous page)

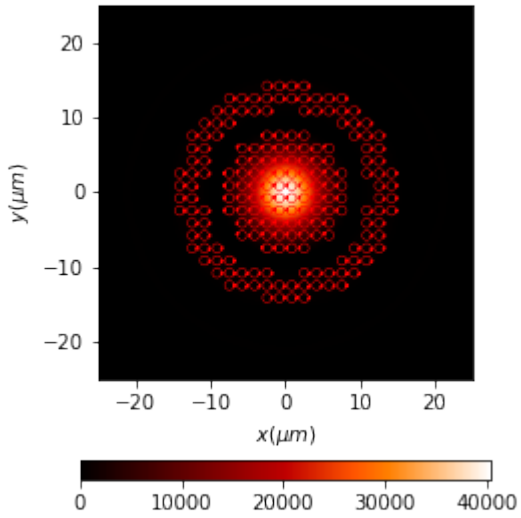
```
u1 = u0 * t0
```

[65]:
```
EM0 = Vector_source_XY(x0, y0, wavelength)
EM0.constant_polarization(u1, v=[(1 - 1j) / 2, (1 + 1j) / 2])
```

[66]:
```
EM1 = EM0.VRS(z=focal)
```

[67]:
```
EM1.cut_resample(x_limits=(-limit / 2, limit / 2),
                 y_limits=(-limit / 2, limit / 2),
                 num_points=(1024, 1024),
                 new_field=False,
                 interp_kind=(3, 1))
```

[68]:
```
EM1.__draw_ellipses__(num_ellipses=(30, 30),
                      color_line='r',
                      amplification=0.75,
                      head_width=.25,
                      line_width=0.5)
```



7.8.4.2 Radial polarization

[69]:
```
diameter = 2 * mm

x0 = np.linspace(-diameter / 2, diameter / 2, 1024)
y0 = np.linspace(-diameter / 2, diameter / 2, 1024)
wavelength = 0.6238 * um

focal = 25 * mm
```

[70]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)

t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.lens(r0=(0 * um, 0 * um),
        radius=(diameter / 2, diameter / 2),
        focal=(focal, focal))

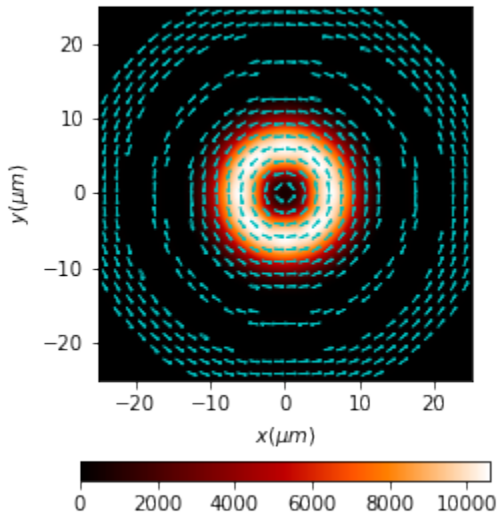
u1 = u0 * t0
```

[71]:
```
EM0 = Vector_source_XY(x0, y0, wavelength)
EM0.azimuthal_wave(u1)
```

[72]:
```
EM1 = EM0.VRS(z=focal)
```

[73]:
```
EM1.cut_resample(x_limits=(-limit / 2, limit / 2),
                 y_limits=(-limit / 2, limit / 2),
                 num_points=(1024, 1024),
                 new_field=False,
                 interp_kind=(3, 1))
```

[74]:
```
EM1.__draw_ellipses__(num_ellipses=(30, 30),
                      color_line='c',
                      amplification=0.75,
                      head_width=.25)
```



[ ]:

[3]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

7.9 Binary phase diffractive lens

Here we analyze how to determine the field produced by a phase binary diffractive lens.

Loading modules

[4]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ

import matplotlib.cm as cm
```

Simulation data

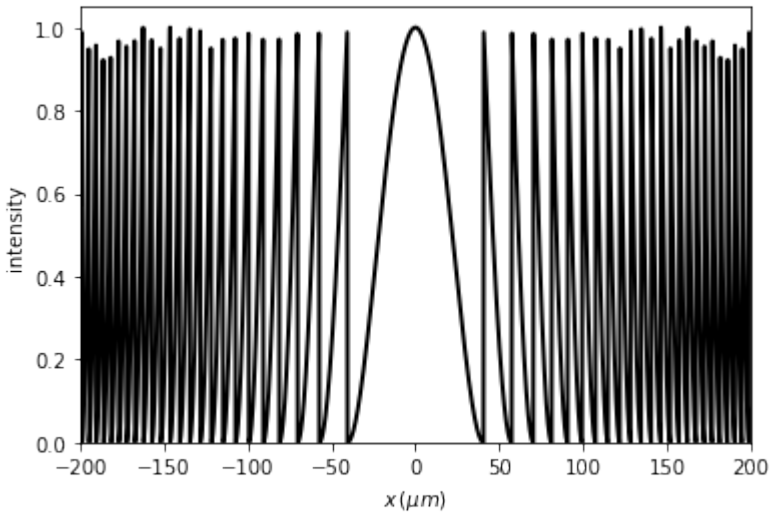
[5]:
```
x0 = np.linspace(-200 * um, 200 * um, 2048)
z0 = np.linspace(0 * um, 50 * um, 512)
wavelength = .55 * um
```

Light source

[6]:
```
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=1 * mm, theta=0. * degrees)
```

Definition of binary diffractive lens in x plane

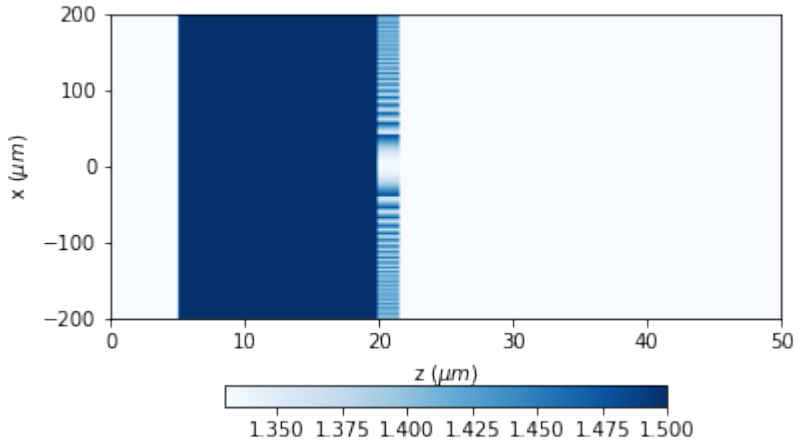
[7]:
```
size_lens = 2 * mm
t0 = Scalar_mask_X(x=x0, wavelength=wavelength)
t0.fresnel_lens(x0=0.0 * um,
                focal=1.5 * mm,
                kind='amplitude',
                phase=np.pi,
                radius=size_lens)
t0.draw()
```



[8]:
```
n_lens = 1.5
n_background = 1.33
k = 2 * np.pi / wavelength
desfase = np.pi
height_lens = desfase / (k * (n_lens - n_background))
height_rectangle = 15 * um
z_ini = 5 * um

DOE = Scalar_mask_XZ(x=x0,
                     z=z0,
                     wavelength=wavelength,
                     n_background=n_background)
DOE.rectangle(r0=(0, z_ini + height_rectangle / 2),
              angle=(0, (0, 0)),
              refractive_index=n_lens,
              size=(size_lens, height_rectangle))
DOE.extrude_mask(t=t0,
                 z0=z_ini + height_rectangle,
                 z1=z_ini + height_rectangle + height_lens,
                 refractive_index=n_lens)
DOE.draw_refractive_index(draw_borders=False, colorbar_kind='horizontal')

DOE.incident_field(u0)
```



7.9.1 Near field

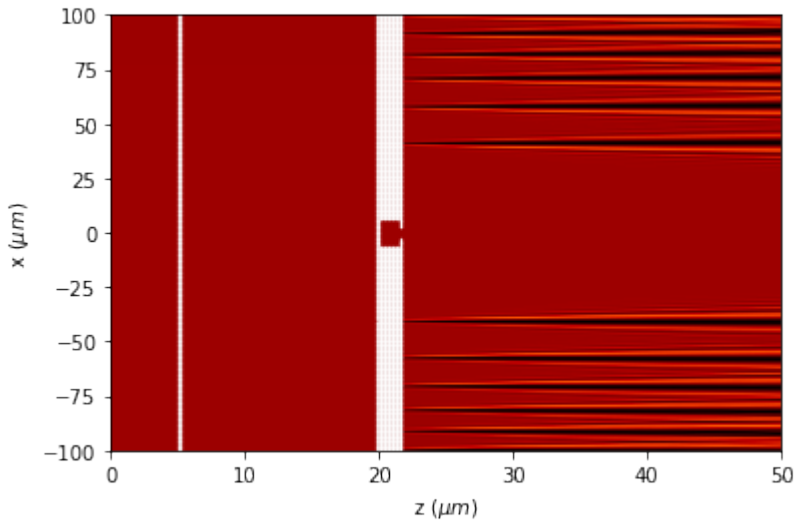
Once we have defined the structure XZ of the diffractive lens, now let us see the optical field around the DOE. We use BPM to determine the field inside the mask and also after it.

[9]:
```
DOE.clear_field()
DOE.BPM(verbose=False)
```

Now, we show the intensity distribution of the field in the mask. We use the same parameters, but whe change the simulation distance z.

[10]:
```
DOE.draw(kind='intensity',
         logarithm=True,
         normalize='maximum',
         draw_borders=True)
plt.ylim(-100, 100)
```

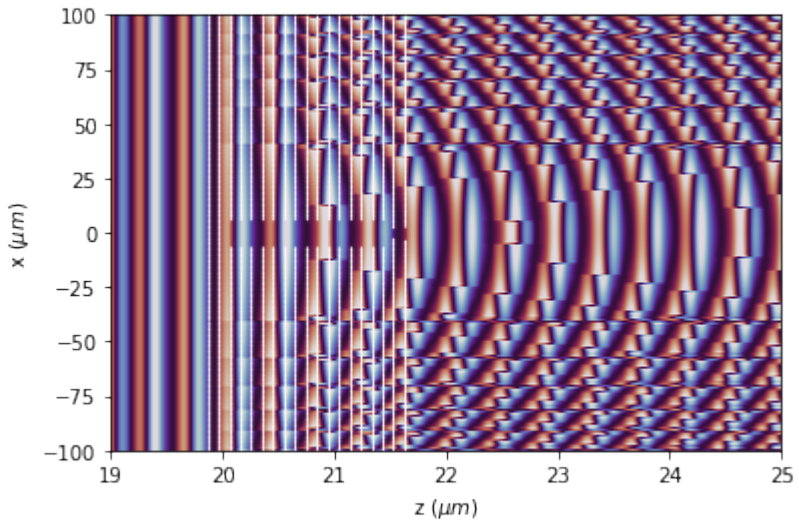
```
[ 51  51  51 ... 221 221 221] [   0    1    2 ... 1021 1022 1023] (512,) (1024,)
```



We see that at the edges of the binary lens, a shadow is produced.

Also, we can see the phase distribution after the lens.

[11]:
```
DOE.draw(kind='phase', draw_borders=True)
plt.xlim(19, 25)
plt.ylim(-100, 100)
```



we can see that the wavefront is divided in two parts, depending on the top or valley of the grating. A phase shift is produced.

7.9.2 Far field

At the focal distance of the grating, focusing of the light beam is produced. We propagate again with BPM technique, up to the focal distance.

[12]:
```
x0_far = np.linspace(-2 * mm, 2 * mm, 1024 * 8)
z0_far = np.linspace(0 * um, 25 * mm, 1024 * 4)
wavelength = .55 * um
```

[13]:
```
u0 = Scalar_source_X(x=x0_far, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=1 * mm, theta=0. * degrees)
t0 = Scalar_mask_X(x=x0_far, wavelength=wavelength)
```

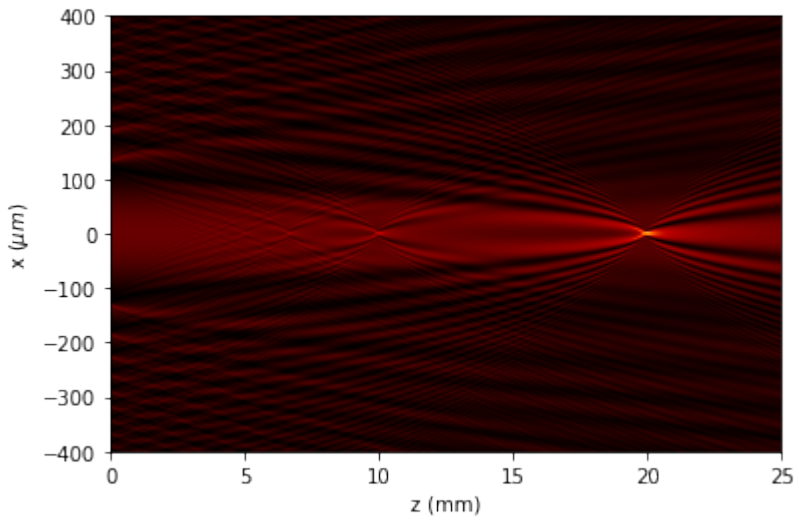
[14]:
```
size_lens = 3.8 * mm
t0.fresnel_lens(x0=0.0 * um,
                focal=15 * mm,
                kind='amplitude',
                phase=np.pi,
                radius=size_lens / 2)
```

[15]:
```
n_lens = 1.5
n_background = 1.33
k = 2 * np.pi / wavelength
desfase = np.pi
height_lens = abs(desfase / (k * (n_lens - n_background)))
z_ini = 1 * mm

DOE_far = Scalar_mask_XZ(x=x0_far,
                         z=z0_far,
                         wavelength=wavelength,
                         n_background=n_background)
DOE_far.incident_field(u0 * t0)
```

[16]:
```
DOE_far.WPM(verbose=False)
```

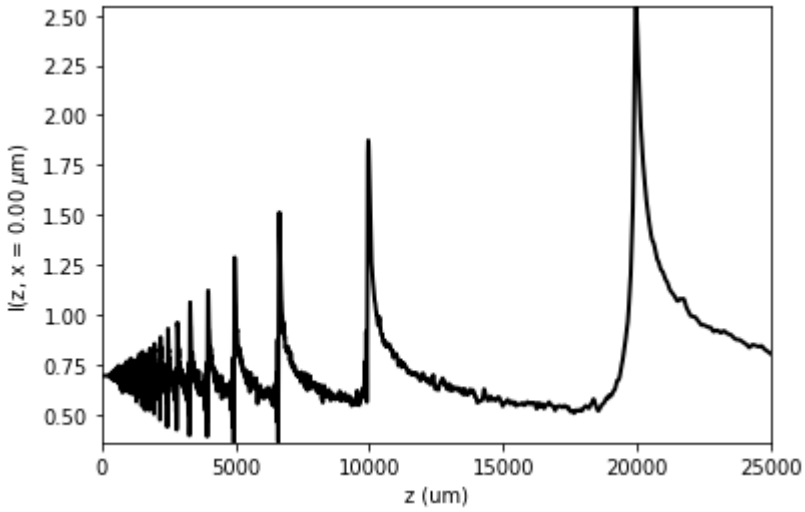
[23]:
```
DOE_far.draw(logarithm=True, draw_borders=True, z_scale='mm')
plt.ylim(-400 * um, 400 * um)
```



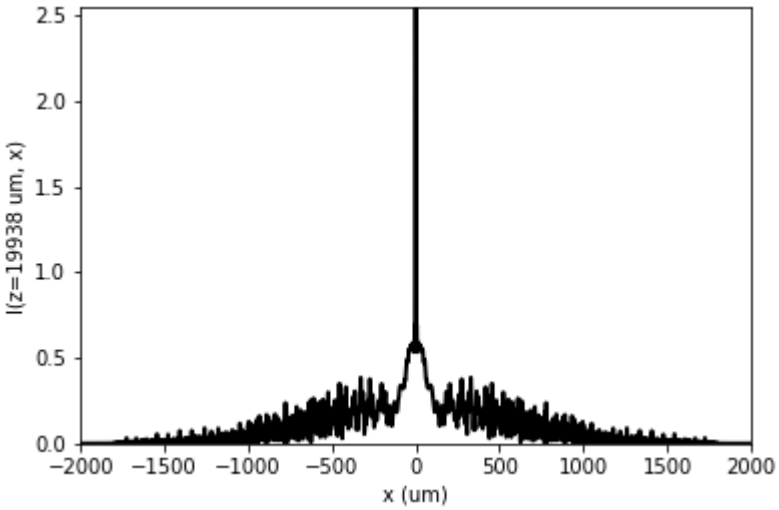
[18]:
```
x_f, z_f = DOE_far.search_focus()
```

```
x = 0.244 um, z = 19938.950 um
```

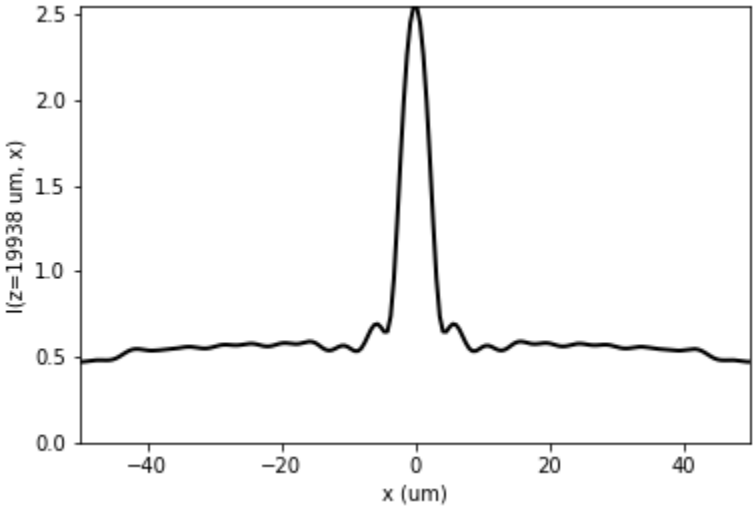
[19]:
```
DOE_far.profile_longitudinal(kind='intensity', logarithm=True, x0=0)
```



[20]:
```
DOE_far.profile_transversal(kind='intensity', logarithm=True, z0=z_f)
```



[21]:
```
DOE_far.profile_transversal(kind='intensity', logarithm=True, z0=z_f)
plt.xlim(-50 * um, 50 * um)
```



7.10 Talbot effect

[1]:
```
from diffractio import plt, sp, np, um, mm, degrees, num_max_processors
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.utils_multiprocessing import execute_multiprocessing
```

7.10.1 Standard Talbot effect

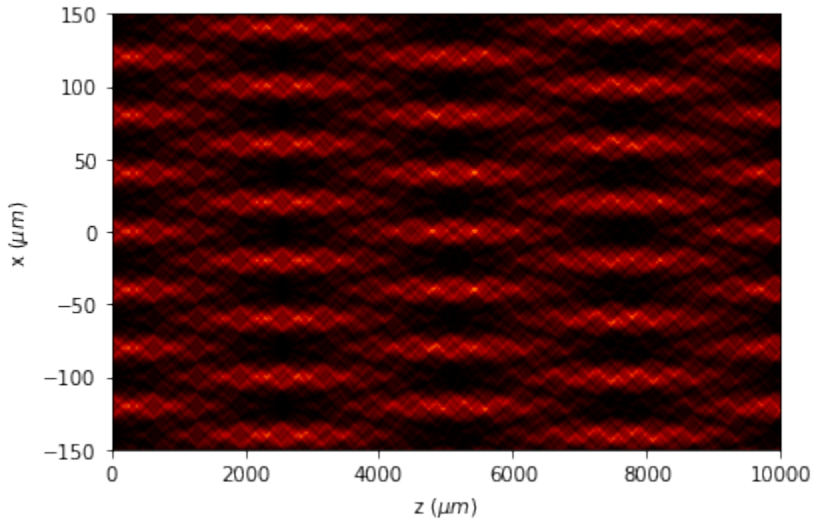
[2]:
```
x = np.linspace(-350 * um, 350 * um, 2048)
z = np.linspace(0 * um, 10 * mm, 512)
wavelength = 0.6238 * um
period = 40 * um
z_talbot = 2 * period**2 / wavelength
```

[3]:
```
u0 = Scalar_source_X(x, wavelength)
u0.plane_wave(A=1)

t = Scalar_mask_X(x, wavelength)
t.ronchi_grating(x0=0 * um, period=40 * um, fill_factor=0.5)

talbot_effect = Scalar_field_XZ(x, z, wavelength)
talbot_effect.incident_field(u0 * t)
talbot_effect.BPM()
```

[4]:
```
talbot_effect.draw(kind='intensity')
plt.ylim(-150 * um, 150 * um)
```



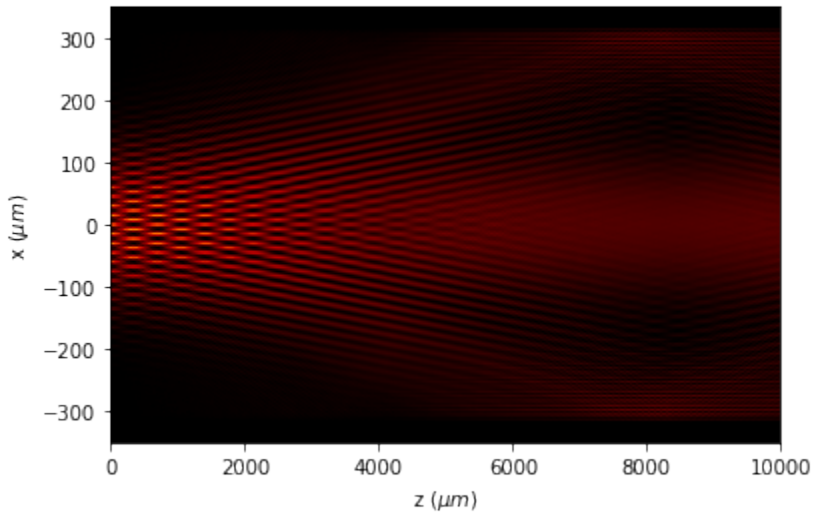
7.10.2 Gauss illumination: separation of orders

[5]:
```
u0 = Scalar_source_X(x, wavelength)
u0.gauss_beam(x0=0 * um, w0=150 * um, z0=0 * um, A=1, theta=0.0)

t = Scalar_mask_X(x, wavelength)
t.ronchi_grating(x0=0 * um, period=15 * um, fill_factor=0.5)

talbot_effect = Scalar_field_XZ(x, z, wavelength)
talbot_effect.incident_field(u0 * t)
talbot_effect.BPM()
```

[6]:
```
talbot_effect.draw(kind='intensity', logarithm=True)
```



7.10.3 Ronchi grating at focal distance of a lens

[7]:
```
x0 = np.linspace(-500 * um, 500 * um, 2048 * 2)
z0 = np.linspace(0 * um, 2 * mm, 1024)
wavelength = 2 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)

u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)

focal, _ = u1.lens_convergent(r0=(0 * um, 500 * um),
                              aperture=800 * um,
                              radius=(.6 * mm, -1.4 * mm),
                              thickness=.25 * mm,
                              refractive_index=1.5,
                              angle=0 * degrees,
                              mask=(50 * um, 3 + 0.05j))

refractive_index = 1.5
height_grating = 20 * (refractive_index - 1) * wavelength / 2
u1.ronchi_grating(period=25 * um,
                  fill_factor=0.5,
                  length=2 * mm,
                  height=height_grating,
                  r0=(0 * um, 100 * um),
                  Dx=0 * um,
                  refractive_index=refractive_index,
                  heigth_substrate=height_grating,
                  refractive_index_substrate=refractive_index,
                  angle=0 * degrees)
```

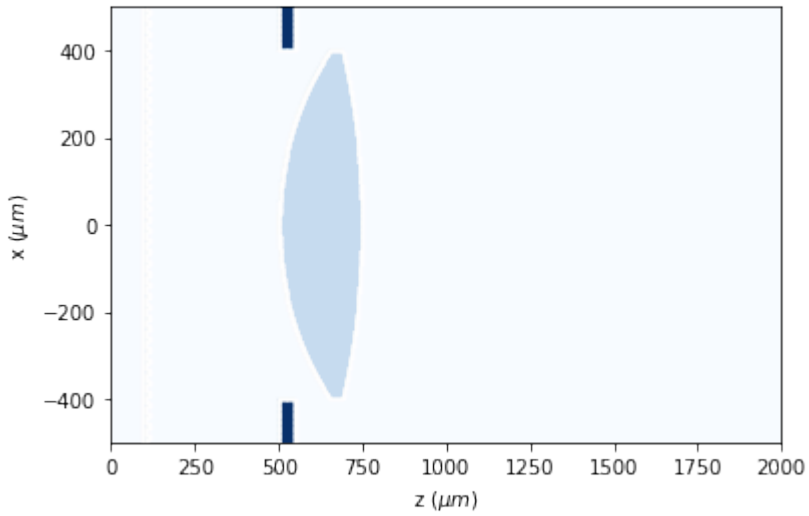
(continues on next page)

(continued from previous page)

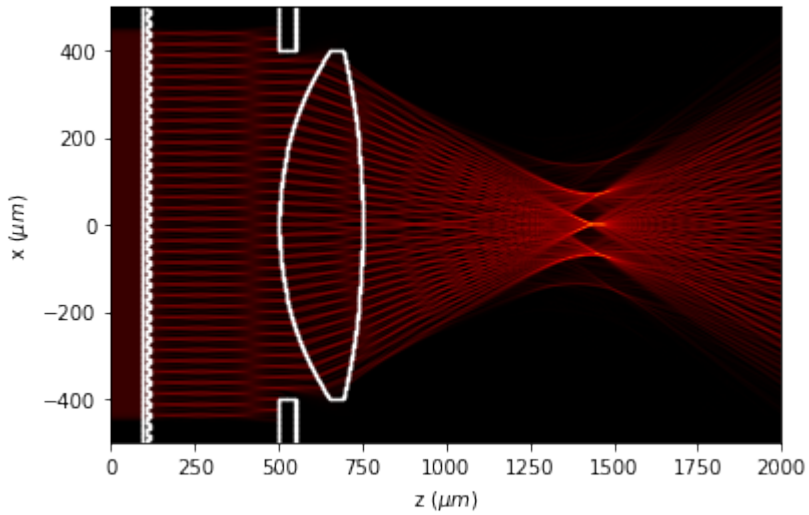
```
u1.draw_refractive_index()

edge_matrix = u1.borders

u1.smooth_refractive_index(pixels_filtering=8, type_filter=2)
```



[8]:
```
u1.BPM(verbose=False)
u1.draw(kind='intensity', logarithm=True, draw_borders=True)
```



7.10.3.1 Desplacement of a grating in a double grating system

[9]:
```
def __experiment_grating_movement__(dict_params):
    delta_x = dict_params['delta_x']
    period = dict_params['period']
    t1 = dict_params['t1']
    t2 = dict_params['t2']
    t2.ronchi_grating(period=period, x0=delta_x, fill_factor=0.5)
    u2 = t1 * t2
    return u2
```

[10]:
```
def creation_dictionary(deltas_x, period, t1, t2):
    # create Parameters: for multiprocessing
    dict_Parameters = []
    for i, delta_x in enumerate(deltas_x):
        dict_Parameters.append(
            dict(delta_x=delta_x, period=period, t1=t1, t2=t2))
    return dict_Parameters
```

[11]:
```
x0 = np.linspace(-400 * um, 400 * um, 1024 * 2)
wavelength = 0.85 * um
period = 50 * um
z_talbot = 2 * period**2 / wavelength
z0 = z_talbot / 2
delay = 0.001

t1 = Scalar_mask_X(x0, wavelength, info="__experiment_grating_movement__")
t1.ronchi_grating(period=period, x0=0 * um, fill_factor=0.5)
t1.RS(z=z0, new_field=False)

t2 = Scalar_mask_X(x0, wavelength, info="__experiment_grating_movement__")
t2.ronchi_grating(period=period, x0=0 * um, fill_factor=0.5)

deltas_x = np.linspace(-60 * um, 60 * um, 256)

dict_Parameters = creation_dictionary(deltas_x=deltas_x,
                                      period=period,
                                      t1=t1,
                                      t2=t2)

u_s, time_proc = execute_multiprocessing(__experiment_grating_movement__,
                                         dict_Parameters,
                                         num_max_processors - 2,
                                         verbose=True)

x = u_s[0].x
```

```
Good result: factor 8.01
num_proc: 6, time=0.24133062362670898
```

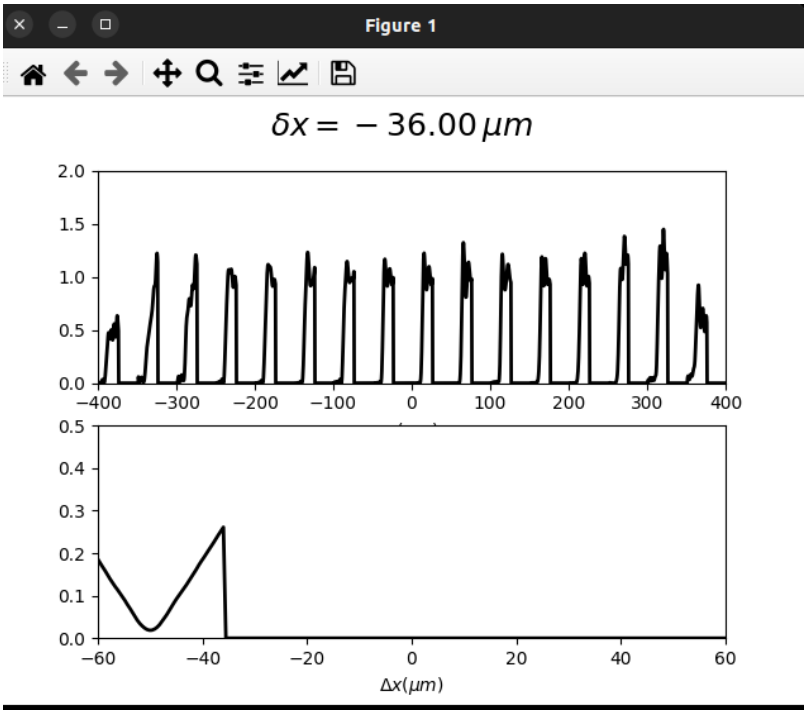
[12]:
```
%matplotlib qt
```

[ ]:
```
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)
perfil = np.zeros_like(deltas_x)

h1, = ax1.plot(x, np.zeros_like(x), 'k', lw=2)
ax1.set_xlim(x[0], x[-1])
ax1.set_ylim(0, 2)
ax1.set_xlabel(r'$x (\mu m)$')

h2, = ax2.plot(deltas_x, perfil, 'k', lw=2)
ax2.set_xlim(deltas_x[0], deltas_x[-1])
ax2.set_ylim(0, .5)
ax2.set_xlabel(r'$\Delta x (\mu m)$')

incr_frames = 1
for i in range(0, len(deltas_x), incr_frames):
    intensidad = abs(u_s[i].u)**2  # sacar fuera
    perfil[i] = intensidad.mean()
    plt.suptitle(r"$\delta x={:6.2f}\,\mu m$".format(deltas_x[i]), fontsize=18)
    h1.set_ydata(intensidad)
    h2.set_ydata(perfil)
    plt.draw()
    plt.pause(0.005)
```



[ ]:

7.11 Blazed gratings

In this example we show that blazed grating generates order 1, only when the angle of the grating is the correct one (maximum phase shift = 2π). On the other hand, for other grating angle, several diffraction orders are generated.

[1]:
```
from diffractio import np, plt
from diffractio import um, mm, degrees

from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XY import Scalar_mask_XY

from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_fields_XZ import Scalar_field_XZ
```

7.11.1 Grating in x mode

Grating with correct phase

[2]:
```
wavelength = 1 * um
focal = 5 * mm

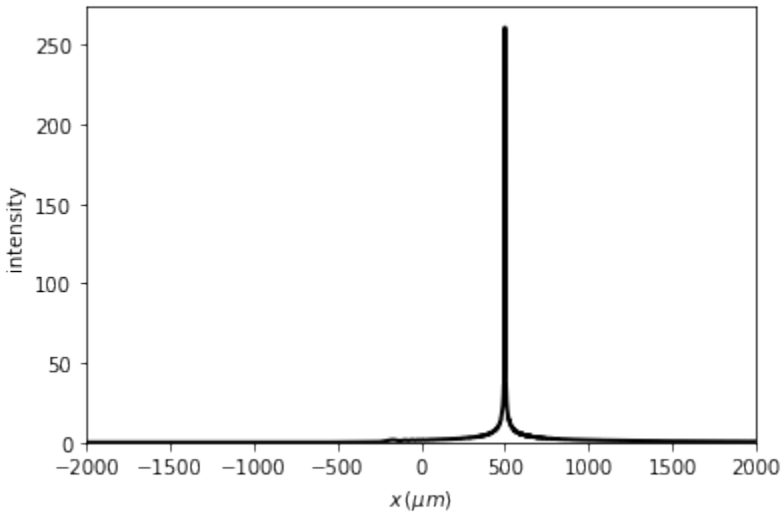
x = np.linspace(-3500,3500,2048*8)
z = np.linspace(focal - 2 * mm, focal + 2 * mm, 128)
```

[3]:
```
t_lens_x = Scalar_mask_X(x, wavelength)
t_lens_x.lens(x0=0, focal=focal, radius=0)
```

[4]:
```
t_blazed_x = Scalar_mask_X(x, wavelength)
t_blazed_x.blazed_grating(x0=125, period=10, phase_max=2 * np.pi)
```

[5]:
```
u1 = t_lens_x * t_blazed_x
u2 = u1.RS(z=focal)
u2.draw()
plt.xlim(-2000, 2000)
```

```
Good result: factor 2.04
```

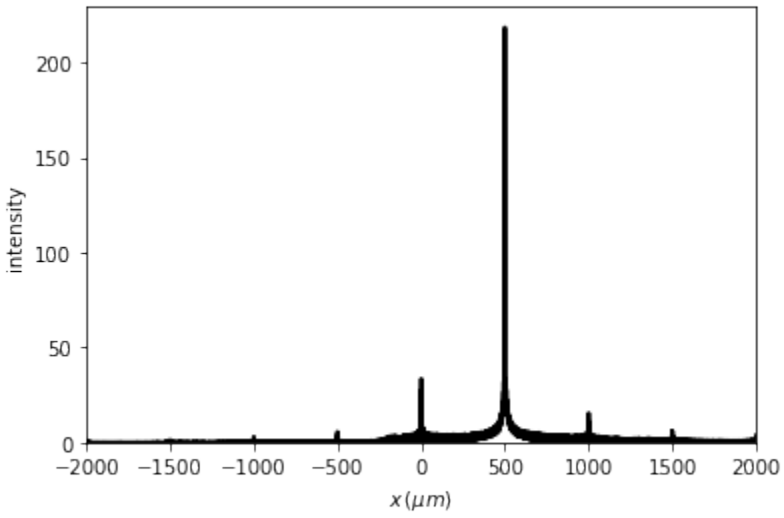


Grating with wrong phase

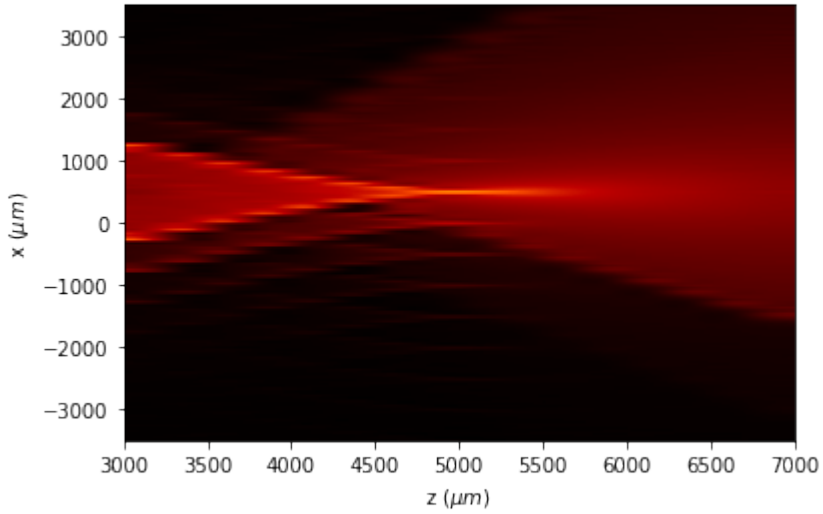
[6]:
```
t_blazed2_x = Scalar_mask_X(x, wavelength)
t_blazed2_x.blazed_grating(x0=125, period=10, phase_max=1.5 * np.pi)
```

[7]:
```
u1_x = t_lens_x * t_blazed2_x
u2_x = u1_x.RS(z=focal)
u2_x.draw()
plt.xlim(-2000, 2000)
```

```
Good result: factor 2.04
```



[8]:
```
txz=Scalar_field_XZ(x=x, z=z, wavelength=wavelength)
txz.incident_field(u1_x)
txz.RS()
txz.draw(logarithm=5)
```



7.11.2 Grating in XY mode

Grating with correct phase

[14]:
```
num_pixels = 512

length = 500 * um
x0 = np.linspace(-length / 2, length / 2, 2*num_pixels)
y0 = np.linspace(-length / 2, length / 2, num_pixels)
wavelength = 0.6238 * um

focal = 5 * mm
```

[15]:
```
u0_xy = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
u0_xy.plane_wave(A=1)
u0_xy.pupil()
```

[16]:
```
t_lens_xy = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t_lens_xy.lens(r0=(0, 0), focal=focal)
```

[17]:
```
t_blazed_xy = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t_blazed_xy.blazed_grating(period=20 * um,
                           phase_max=2 * np.pi,
                           x0=0 * um,
                           angle=0 * degrees)
```

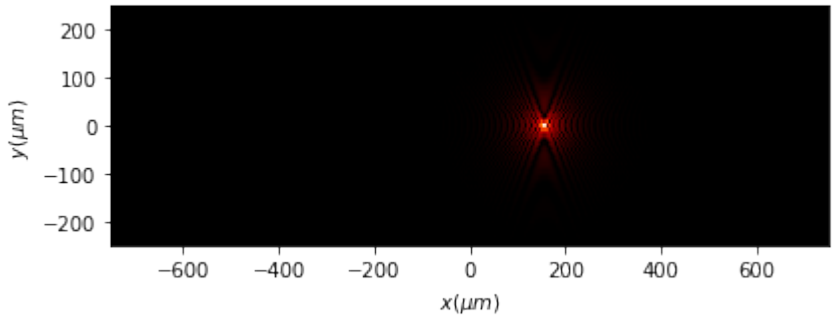
[18]:
```
u1_xy = u0_xy * t_lens_xy * t_blazed_xy

u2_xy = u1_xy.RS(z=focal, new_field=True, amplification=(3, 1))
```

(continues on next page)

(continued from previous page)

```
u2_xy.draw(logarithm=1e1)
```



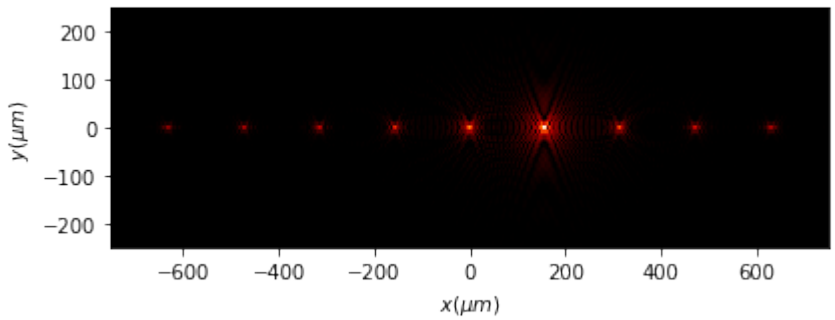
Grating with wrong phase

[19]:
```
t_blazed2_xy = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t_blazed2_xy.blazed_grating(period=20 * um,
                            phase_max=1.5 * np.pi,
                            x0=0 * um,
                            angle=0 * degrees)
```

[20]:
```
u1_xy = u0_xy * t_lens_xy * t_blazed2_xy

u2_xy = u1_xy.RS(z=focal, new_field=True, amplification=(3, 1))

u2_xy.draw(logarithm=1e1)
```



7.12 Dammann gratings

We check how Dammann gratings work, which are diffraction grating with uniform efficiencies in a number of diffraction orders.

The definition of the grating can be seen in: Bin Wu, Jiehu Kang, Wenqiang Han State “Design of Dammann grating based on the parallel recombination simulated annealing algorithm” Optik 124 (2013) 2928– 2931

[1]:
```
from diffractio import np, sp, plt
from diffractio import um, mm, nm, degrees

from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

7.12.1 X Scheme

One period

[2]:
```
period_size=40*um
x0=np.linspace(-period_size/2, period_size/2,128)
wavelength=0.6328*um

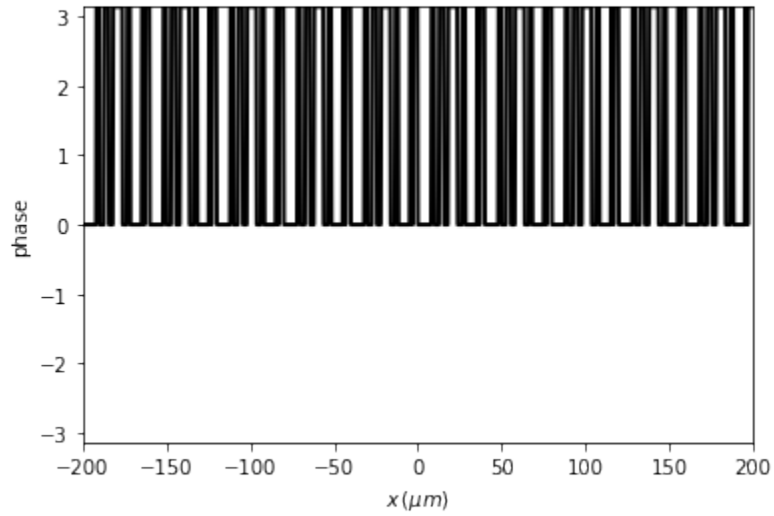
x_transitions = np.array([
    0, 0.201181, 0.250978, 0.326167, 0.370555, 0.372996, 0.396478, 0.453128,
    0.594731, 0.670591, 0.717718, 0.890632, 0.919921, 0.935546
])

x_transitions = (x_transitions - 0.5) * period_size

period = Scalar_mask_X(x=x0, wavelength=wavelength)
period.binary_code_positions(x_transitions=x_transitions,
                             start='down',
                             has_draw=False)
period.u = np.exp(1j * np.pi * period.u)
```

N periods

[3]:
```
t_dammann = period.repeat_structure(
                                    num_repetitions=10,
                                    position='center',
                                    new_field=True)
t_dammann.draw('phase')
```

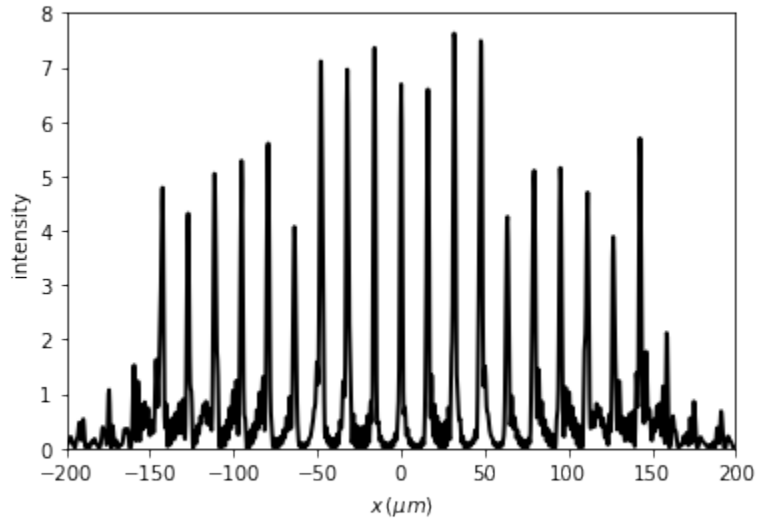


Propagation at far field

[4]:
```
focal = 1 * mm
lens=Scalar_mask_X(x=t_dammann.x, wavelength=wavelength)
lens.lens(x0=0, focal=focal, radius=1*mm);
```

[5]:
```
u_foco=t_dammann*lens
u_foco.RS(z=focal, new_field=False)
u_foco.draw()
```

```
Good result: factor 5.12
```



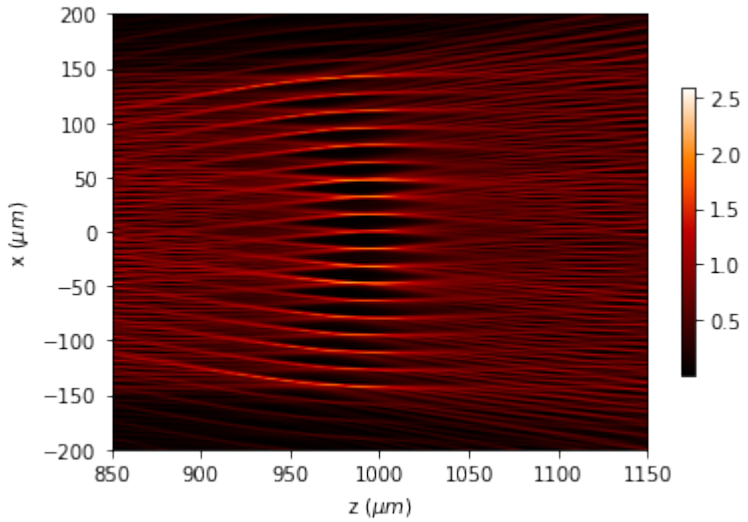
Around focal position

[6]:
```
z0=np.linspace(focal-150*um,focal+150*um,512)
u_prop = Scalar_field_XZ(x=t_dammann.x,z=z0, wavelength=wavelength)
u_prop.incident_field(t_dammann*lens)
```

(continues on next page)

(continued from previous page)

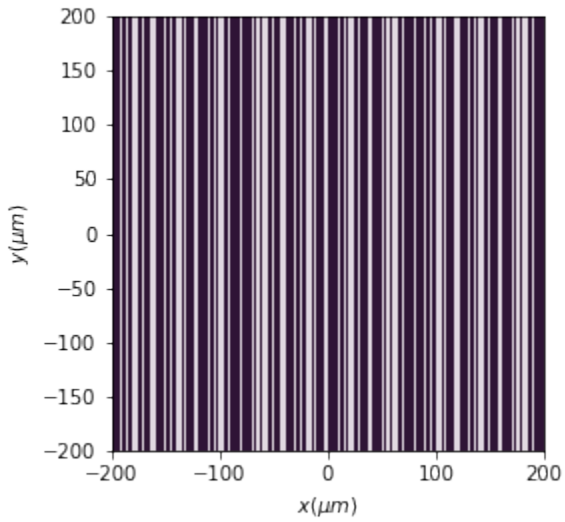
```
u_prop.RS()
u_prop.draw(kind='intensity', colorbar_kind='vertical', logarithm=True)
```



7.12.2 XY Scheme

Vertical grating

[7]:
```
dammann_xy = Scalar_mask_XY(x=t_dammann.x, y=t_dammann.x, wavelength=wavelength)
dammann_xy.extrude_mask_x(t_dammann)
dammann_xy.draw('phase')
```



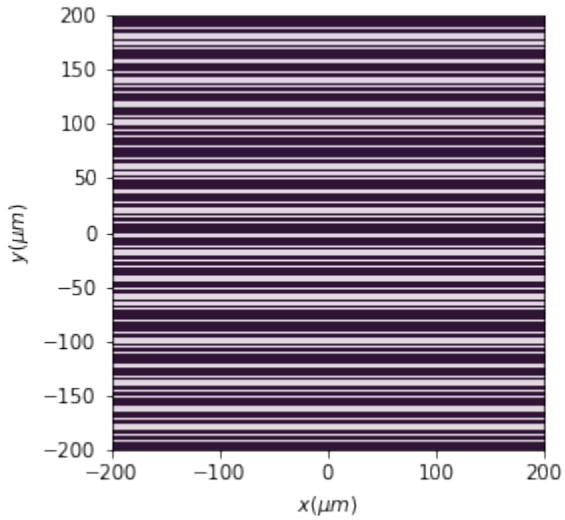
Horizontal grating

[8]:
```
dammann_xy2 = Scalar_mask_XY(x=t_dammann.x, y=t_dammann.x, wavelength=wavelength)
dammann_xy2.extrude_mask_x(t_dammann)
```

(continues on next page)

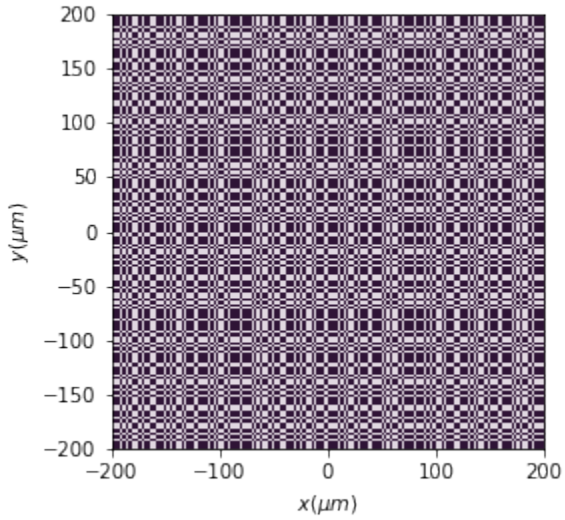
(continued from previous page)

```
dammann_xy2.rotate(angle=90*degrees)
dammann_xy2.draw('phase')
```



Composed XY grating

[9]:
```
dammann_xy_total = dammann_xy * dammann_xy2
dammann_xy_total.draw('phase')
```



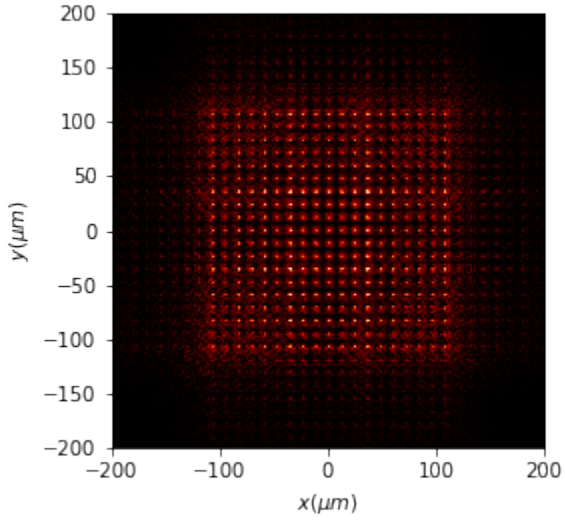
[10]:
```
focal=0.75*mm
lens_2D = dammann_xy.duplicate(clear=True)
lens_2D.lens(r0=(0,0), radius=200*um, focal=focal, angle=0.0);
```

[11]:
```
u_2d=dammann_xy_total*lens_2D
u_2d_focus=u_2d.RS(z=focal, verbose=True)
u_2d_focus.draw(logarithm=1)
```

(continues on next page)

(continued from previous page)

```
Good result: factor 4.04
```



7.13 Objects with a variable refraction index

7.13.1 Sphere

[1]:
```
from diffractio import degrees, mm, plt, sp, um, np
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[2]:
```
x0 = np.linspace(-120 * um, 120 * um, 1024 * 2)
z0 = np.linspace(0 * um, 250 * um, 1024 * 2)
wavelength = 1.5 * um
```

[3]:
```
w0 = 10 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=35 * um, z0=-250 * um, w0=w0, theta=0. * degrees)
u1 = Scalar_source_X(x=x0, wavelength=wavelength)
u1.gauss_beam(A=1, x0=0 * um, z0=-250 * um, w0=w0, theta=0. * degrees)

u2 = Scalar_source_X(x=x0, wavelength=wavelength)
u2.gauss_beam(A=1, x0=-35 * um, z0=-250 * um, w0=w0, theta=0. * degrees)

uf = u0 + u1 + u2
```

[4]:
```
t0 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.0)
t0.incident_field(uf)
```

(continues on next page)

(continued from previous page)

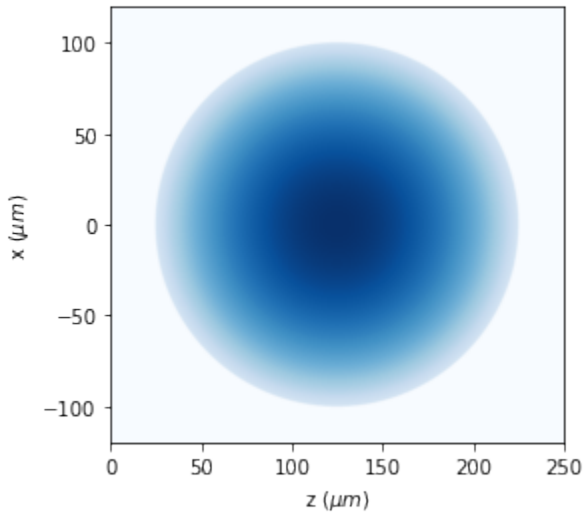
```
pn = dict(n_out=1.5, n_center=4, cx=0, cz=125, radius=100)

# ref_index = "{p[n_out]}".format(p=pn)
center = (pn['cx'], pn['cz'])
radius = pn['radius']
# ref_index = '2*(((X-0)**2+(Z-300)**2)/75**2-0)'
ref_index = "{p[n_out]}+({p[n_center]}-{p[n_out]})*(1-((X-{p[cx]})**2+(Z-{p[cz]})**2)/
↪{p[radius]}**2)".format(
    p=pn)
print("n={}".format(ref_index))

t0.sphere(r0=(0, 100),
          radius=(pn['radius'], pn['radius']),
          refractive_index=1.0,
          angle=0)
t0.sphere(r0=center,
          radius=(radius, radius),
          refractive_index=ref_index,
          angle=0)
```

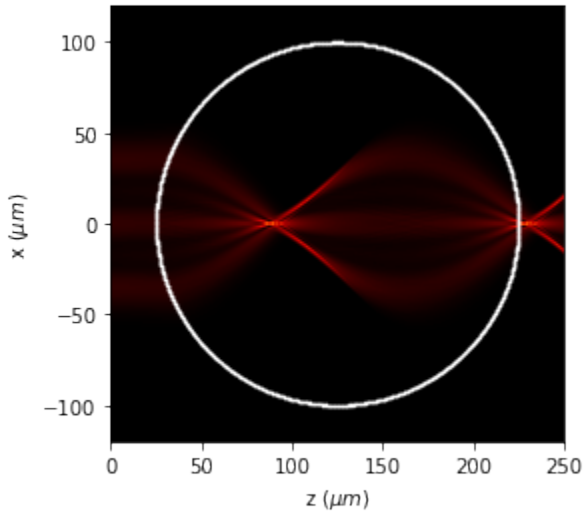
```
n=1.5+(4-1.5)*(1-((X-0)**2+(Z-125)**2)/100**2)
```

[5]:
```
t0.draw_refractive_index(draw_borders=False, scale='scaled')
```

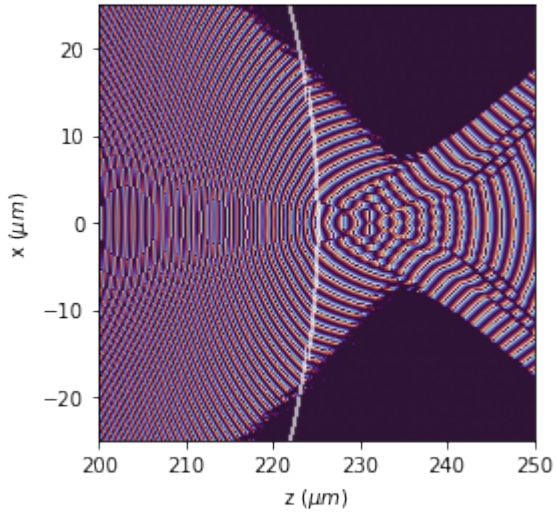


[6]:
```
t0.BPM(verbose=False)
```

[7]:
```
t0.draw(kind='intensity',
        logarithm=True,
        draw_borders=True,
        min_incr=0.05,
        scale='scaled')
```



[8]:
```
t0.draw(kind='phase',
        logarithm=True,
        draw_borders=True,
        min_incr=0.05,
        scale='scaled')
plt.ylim(-25, 25)
plt.xlim(200, 250)
```



7.13.2 Grin Lens

[9]:
```
x0 = np.linspace(-1000 * um, 1000 * um, 1024 * 2)
z0 = np.linspace(0 * um, 14500 * um, 1024 * 2)
wavelength = .85 * um
```

[10]:
```
w0 = 500 * um
z0_beam = 200 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=z0_beam, w0=w0, theta=0. * degrees)
```

[11]:
```
t0 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.0)
t0.incident_field(u0)

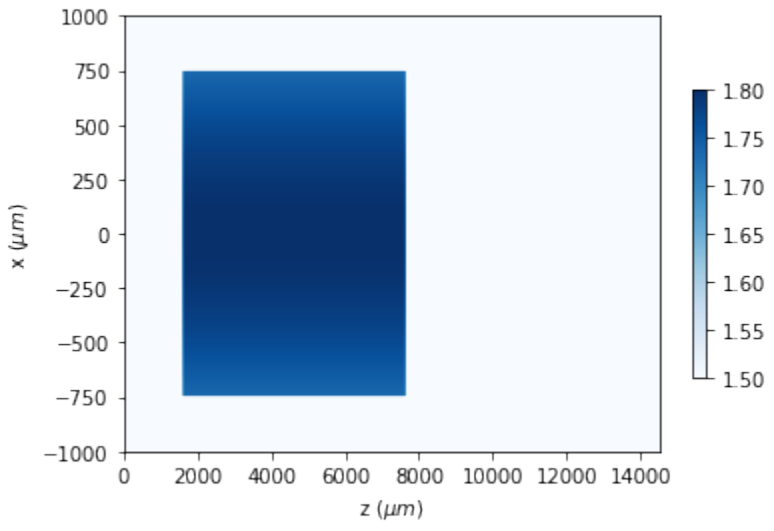
pn = dict(n_out=1.5, n_center=1.8, width=1500 * um, x_fiber=0 * um)
ref_index = "{p[n_out]}+({p[n_center]}-{p[n_out]})*(np.exp(-abs(X-{p[x_fiber]})**2/
↪{p[width]}**2))".format(
    p=pn)

print("n={}".format(ref_index))

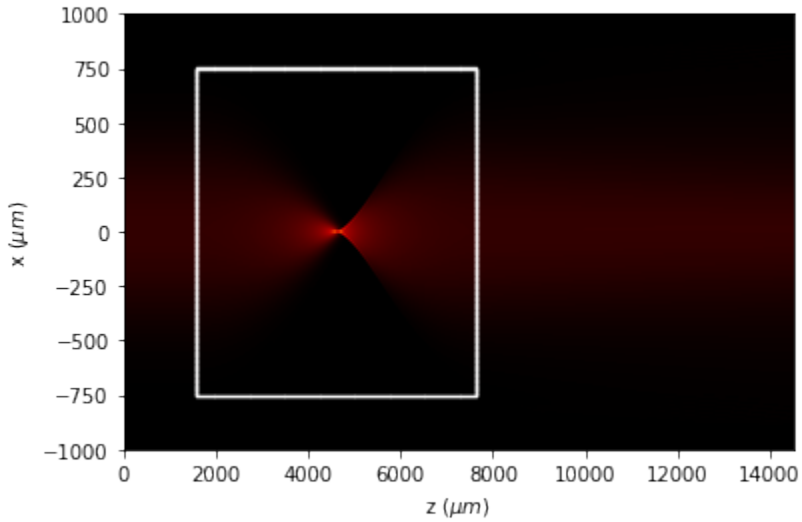
t0.rectangle(r0=(0, 4600 * um),
             size=(1500 * um, 6075 * um),
             refractive_index=ref_index,
             angle=0 * degrees)
```

```
n=1.5+(1.8-1.5)*(np.exp(-abs(X-0.0)**2/1500.0**2))
```

[12]:
```
t0.draw_refractive_index(draw_borders=False, colorbar_kind='vertical')
plt.clim(1.5, 1.8)
```



[13]:
```
t0.BPM(verbose=False)
t0.draw(kind='intensity', logarithm=True, draw_borders=True, min_incr=0.05)
```



Now, let us see the phase of the final field, which is quite plane at the center.

[14]:
```
u1 = t0.final_field()
```

[15]:
```
phase = np.unwrap(np.angle(u1.u))
phase = phase - phase.min()
```

7.13.3 Optical Fiber

In this example, we see how a optical fiber can be modelled using diffractio.scalar_mask_xy. First, we define a rectangle as a shape, and after, we provide a refraction index with a cuadratic shape.

[16]:
```
x0 = np.linspace(-15 * um, 15 * um, 1024)
z0 = np.linspace(0 * um, 700 * um, 4096)
wavelength = 0.5 * um
```

7.13.3.1 Aligned beam input

Beam definition

[17]:
```
w0 = 2 * um
z0_beam = 25 * um
aligment = 0 * um
```

[18]:
```
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=aligment, z0=50 * um, w0=2 * um, theta=0. * degrees)
```

Fiber definition

[19]:
```
t0 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength, n_background=1.0)
t0.incident_field(u0)

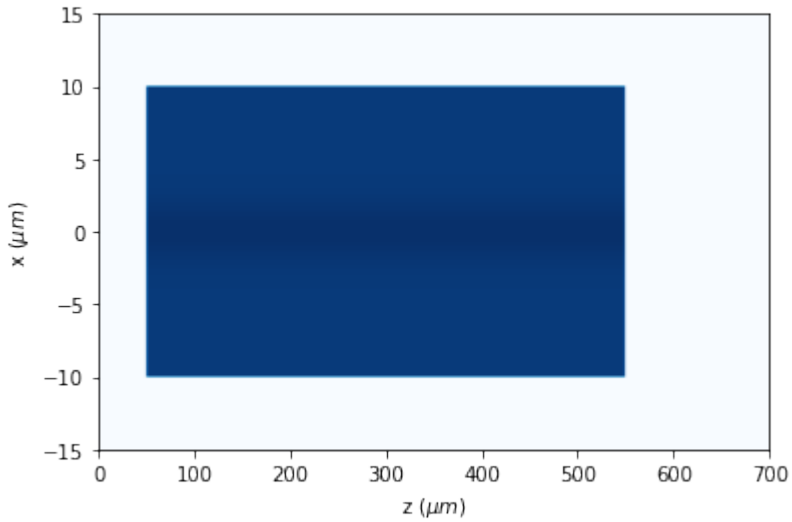
pn = dict(n_out=1.48, n_center=1.5, width=2 * um, x_fiber=0 * um)
# ref_index = 'nexp(-(x-x0)**2/w0**2)'
ref_index = "{p[n_out]}+({p[n_center]}-{p[n_out]})*(np.exp(-(X-{p[x_fiber]})**2/
↪{p[width]}**2))".format(
    p=pn)

print("n={}".format(ref_index))

t0.rectangle(r0=(0, 300 * um),
             size=(20 * um, 500 * um),
             refractive_index=ref_index)

t0.draw_refractive_index(draw_borders=False, scale='')
```

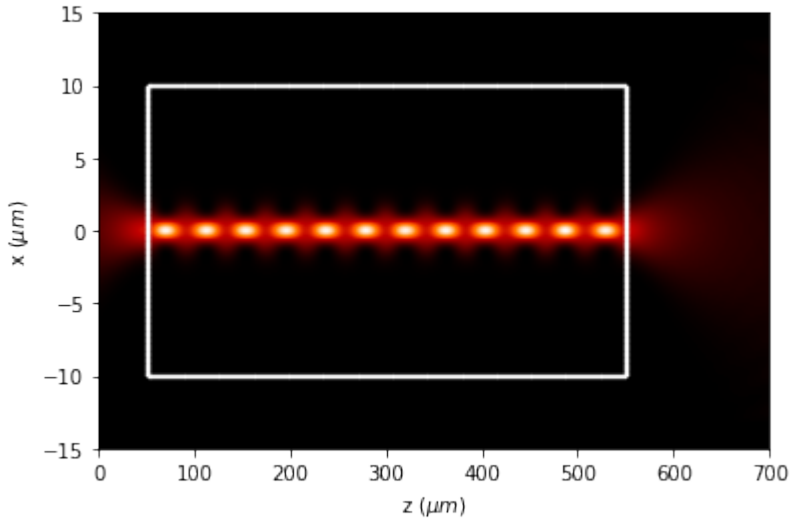
```
n=1.48+(1.5-1.48)*(np.exp(-(X-0.0)**2/2.0**2))
```



Propagation and representation

[20]:
```
t0.BPM(verbose=False)
```

[21]:
```
t0.draw(kind='intensity', logarithm=False, draw_borders=True, min_incr=0.05)
```



7.14 Vector Double slit experiment

Here we analize what happens in the double slit experiment when the polarization of the two slits is the same or ortogonal. In the first case, fringes are obtained but this is not the case for the second experiment.

7.14.1 Creating an instance

[1]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees

from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XY import Scalar_field_XY

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.vector_masks_XY import Vector_mask_XY
from diffractio.vector_fields_XY import Vector_field_XY

# for polarization components, diffractio also uses py_pol module.
from py_pol.jones_matrix import Jones_matrix
```

[2]:
```
x0 = np.linspace(-125 * um, 125 * um, 512)
y0 = np.linspace(-125 * um, 125 * um, 512)

wavelength = 0.6328 * um
```

7.14.1.1 Double Slit Experiment

7.14.2 Equal polarization

[4]:
```
j1 = Jones_matrix().diattenuator_perfect(azimuth=0)
```

[5]:
```
t1 = Scalar_mask_XY(x0, y0, wavelength)
t1.slit(x0=-40 * um, size=80 * um)
t1.square(r0=(-40, 0), size=(40, 160), angle=0)

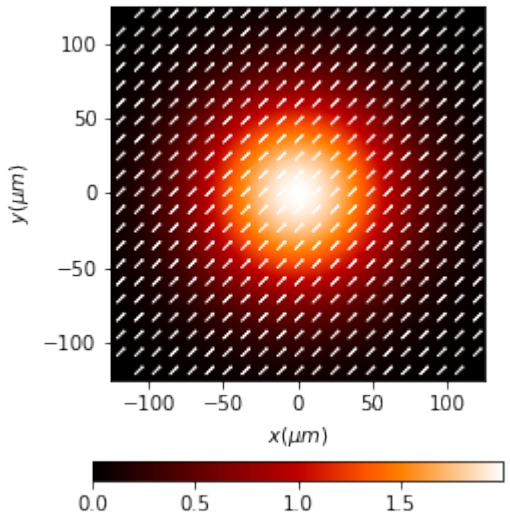
t2 = Scalar_mask_XY(x0, y0, wavelength)
t2.slit(x0=40 * um, size=80 * um)
t2.square(r0=(40, 0), size=(40, 160), angle=0)

M1 = Vector_mask_XY(x0, y0, wavelength)
M1.polarizer_linear(azimuth=0 * degrees)
M1.scalar_to_vector_mask(t1, j1)

M2 = Vector_mask_XY(x0, y0, wavelength)
M2.polarizer_linear(azimuth=0 * degrees)
M2.scalar_to_vector_mask(t2, j1)

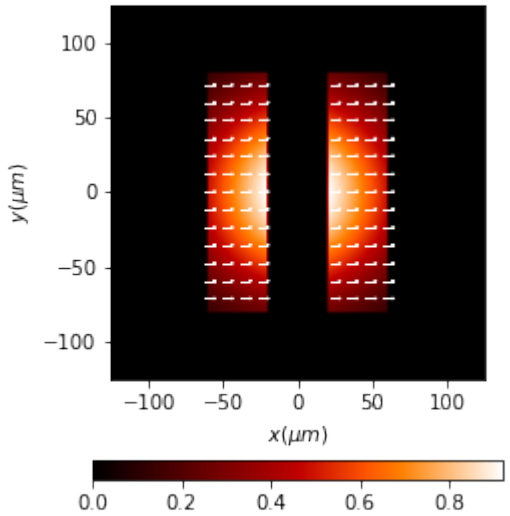
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(100 * um, 100 * um),
              z0=0 * um,
              A=1,
              theta=0. * degrees,
              phi=0 * degrees)

EM0 = Vector_source_XY(x0, y0, wavelength)
EM0.constant_polarization(u0, v=(1, 1))
EM0.draw('ellipses', num_ellipses=(21, 21))
```

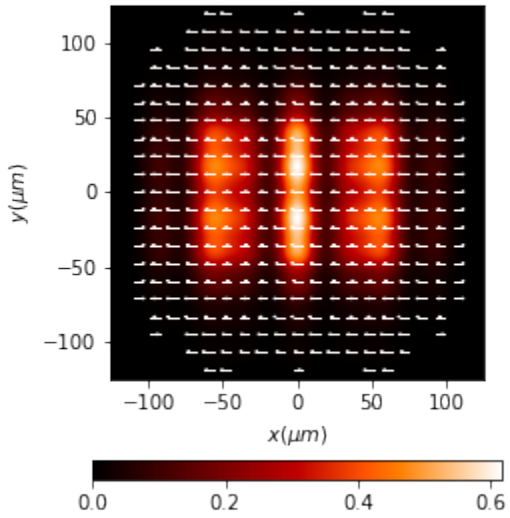


[6]:
```
Et1 = EM0 * M1
Et2 = EM0 * M2

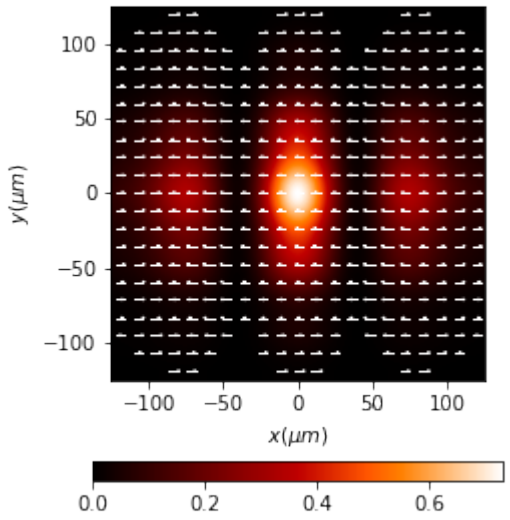
EM1 = Et1 + Et2
EM1.draw('ellipses', num_ellipses=(21, 21))
```



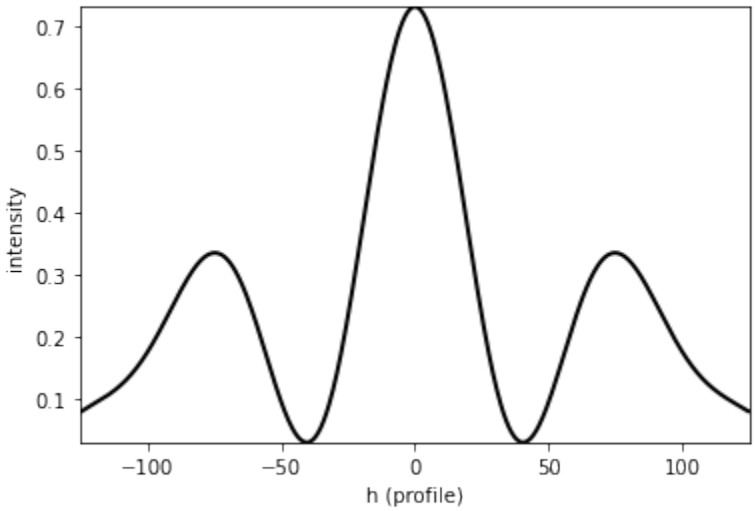
[7]:
```
EM2 = EM1.VRS(z=5 * mm)
EM2.draw('ellipses', num_ellipses=(21, 21))
```



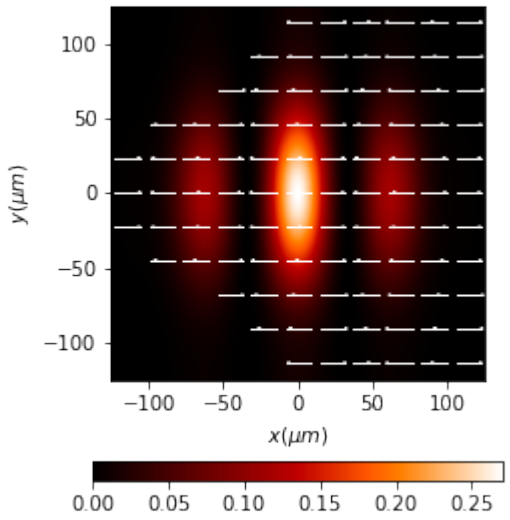
[8]:
```
EM2 = EM1.VRS(z=10 * mm, )
EM2.draw('ellipses', num_ellipses=(21, 21))
```



[9]:
```
Ex, Ey, _ = EM2.get('fields', is_matrix=False)
Ex.draw_profile(point1=[Ex.x[0], 0], point2=[Ex.x[-1], 0])
```



[10]:
```
EM2 = EM1.VRS(z=25 * mm, amplification=(3, 1))
EM2.draw('ellipses', num_ellipses=(11, 11), amplification=0.75)
```



7.14.3 Crossed polarization

[12]:
```
j0 = Jones_matrix().diattenuator_perfect(azimuth=0*degrees)
j1 = Jones_matrix().diattenuator_perfect(azimuth=90*degrees)
```

[13]:
```
t1 = Scalar_mask_XY(x0, y0, wavelength)
t1.slit(x0=-40 * um, size=80 * um)
t1.square(r0=(-40, 0), size=(50, 160), angle=0)

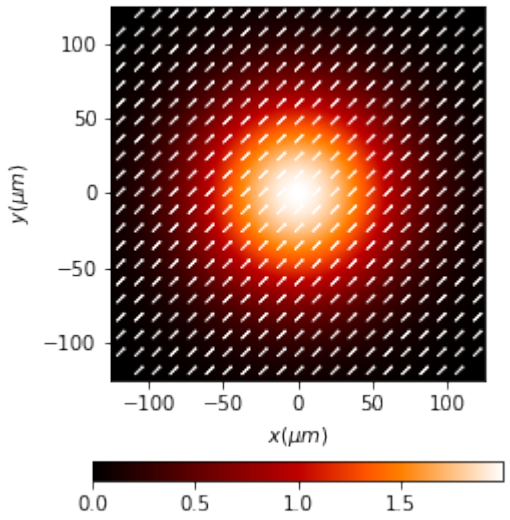
t2 = Scalar_mask_XY(x0, y0, wavelength)
t2.slit(x0=40 * um, size=80 * um)
t2.square(r0=(40, 0), size=(50, 160), angle=0)

M1 = Vector_mask_XY(x0, y0, wavelength)
M1.scalar_to_vector_mask(t1,j0)

M2 = Vector_mask_XY(x0, y0, wavelength)
M2.polarizer_linear(azimuth=90 * degrees)
M2.scalar_to_vector_mask(t2,j1)

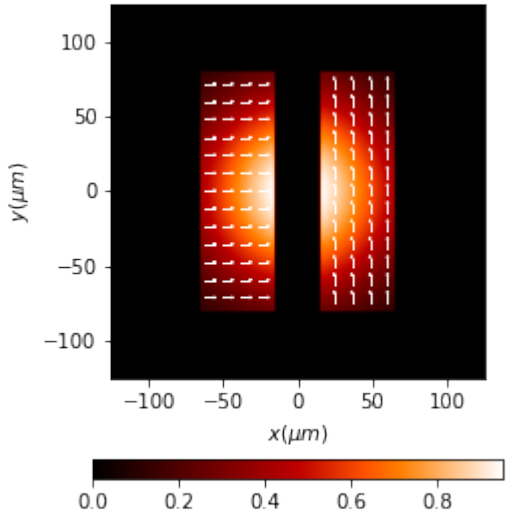
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(r0=(0, 0),
              w0=(100 * um, 100 * um),
              z0=0 * um,
              A=1,
              theta=0. * degrees,
              phi=0 * degrees)

EM0.constant_polarization(u0, v=(1, 1))
EM0.draw('ellipses', num_ellipses=(21, 21))
```

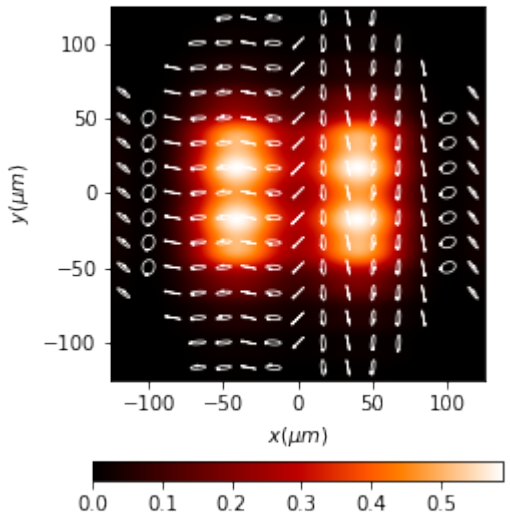


[14]:
```
Et1 = EM0 * M1
Et2 = EM0 * M2

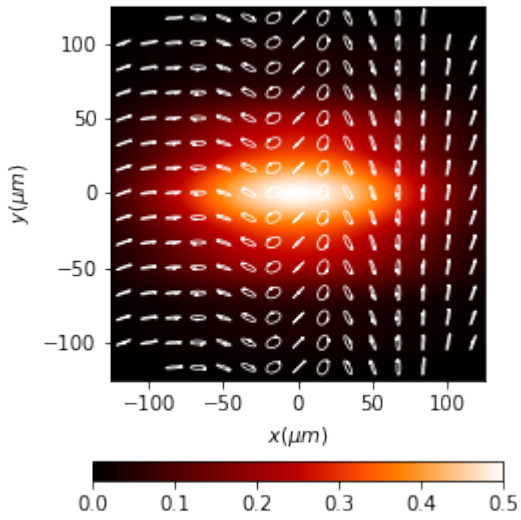
EM1 = Et1 + Et2
EM1.draw('ellipses', num_ellipses=(21, 21))
```



[15]:
```
EM2 = EM1.VRS(z=5 * mm)
EM2.draw('ellipses', num_ellipses=(15, 15), amplification=0.6)
```

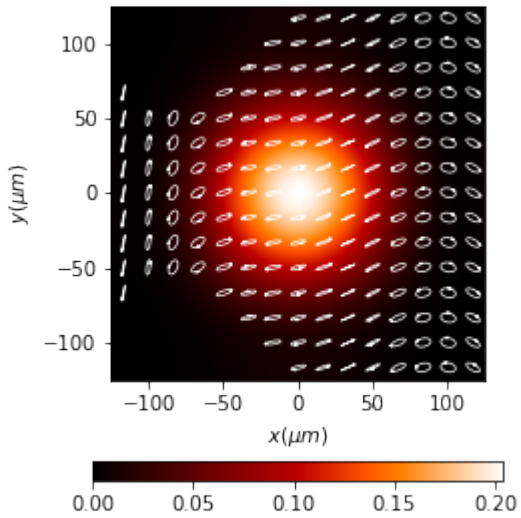


[16]:
```
EM2 = EM1.VRS(z=10 * mm)
EM2.draw('ellipses', num_ellipses=(15, 15), amplification=0.6)
```



[17]:
```
EM2 = EM1.VRS(z=25 * mm, amplification=(3, 1), verbose=True)
EM2.draw('ellipses', num_ellipses=(15, 15), amplification=0.6)
```

```
Good result: factor 55.196
```



7.15 Vector Rayleigh-Sommerfeld method

Here we analyze how to obtain the results at the article: H. Ye, C.-W. Qiu, K. Huang, J. Teng, B. Luk'yanchuk, y S. P. Yeo, «Creation of a longitudinally polarized subwavelength hotspot with an ultra-thin planar lens: vectorial Rayleigh–Sommerfeld method», Laser Phys. Lett., vol. 10, n.º 6, p. 065004, jun. 2013.

DOI: 10.1088/1612-2011/10/6/065004 (http://stacks.iop.org/1612-202X/10/i=6/a=065004?key=crossref.890761f053b56d7a9eeb8fc6da4d9b4e).

In this development notebook, we develope Vector Rayleigh-Sommerfeld method. The incident field is an vector field (Ex, Ey) and it is propagated and converted to (Ex,Ey,Ez)

[1]:
```
from diffractio import np, sp, plt
from diffractio import nm, um, mm, degrees

from diffractio.vector_sources_XY import Vector_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY

from diffractio.vector_fields_XY import Vector_field_XY

from matplotlib import cm
```

7.15.1 Diffraction by a hole

In Figure 2, a simulation of the electric fields at z=6.5 um from a circular aperture of diameter 5λ is obtained when λ = 0.648 μm.

[2]:
```
size = 5 * um
x0 = np.linspace(-size, size, 256)
y0 = np.linspace(-size, size, 256)
wavelength = .640 * um
radius = 5 * wavelength / 2
```

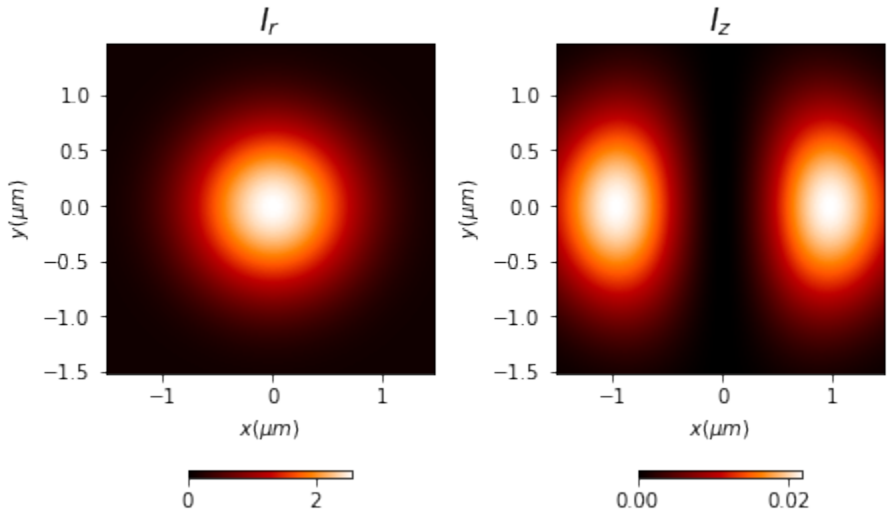
[3]:
```
t = Scalar_mask_XY(x0, y0, wavelength)
t.circle(r0=(0, 0), radius=radius)
```

[4]:
```
E0 = Vector_source_XY(x0, y0, wavelength)
E0.constant_polarization(u=t, v=(1, 0))
```

[5]:
```
%%time
E1=E0.VRS(z=6.5*um, n=1, new_field=True, verbose=False, amplification=(1,1))
E1.cut_resample([-1.5,1.5], [-1.5,1.5])
```

```
CPU times: user 343 ms, sys: 40.5 ms, total: 384 ms
Wall time: 382 ms
```

[6]:
```
E1.draw(kind='intensities_rz', logarithm=False)
```



7.15.2 Radial polarized Gaussian beam through a circular aperture

In Figure 3, the propagation of a radially polarized Gaussian beam is performed. The diameter of teh circular aperture is D = 10λ, and the beam is FWHM = 6 μm.

[7]:
```
size = 6.4 * um
x0 = np.linspace(-size, size, 512)
y0 = np.linspace(-size, size, 512)
```

[8]:
```
wavelength = .640 * um

radius = 10 * wavelength / 2
z_obs = 10 * um
beam_width = 6 * um / (2 * np.sqrt(2))
```

[9]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.gauss_beam(A=1, r0=(0, 0), w0=beam_width, z0=0)

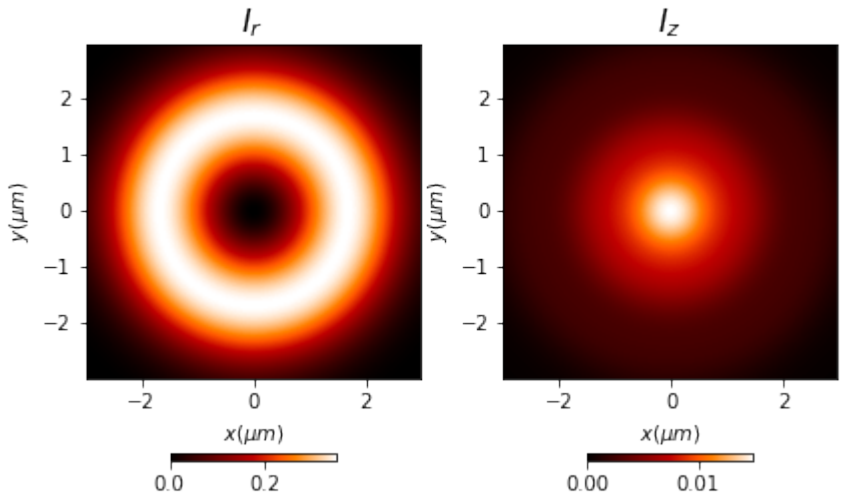
t0 = Scalar_mask_XY(x0, y0, wavelength)
t0.circle(r0=(0, 0), radius=radius)

t = u0 * t0
```

[10]:
```
E0 = Vector_source_XY(x0, y0, wavelength)
E0.radial_wave(u=t, r0=(0, 0))
```

[11]:
```
E1 = E0.VRS(z=z_obs, n=1, new_field=True)
E1.cut_resample([-3, 3], [-3, 3])
```

[12]:
```
E1.draw('intensities_rz')
```



[13]:
```
Ex, Ey, _ = E1.get('fields', is_matrix=False)
intensity = E1.get('intensity', is_matrix=True)
Ex.u = np.sqrt(intensity)
```

7.15.3 Focusing with a lens a beam with a high E_z component

In Figures 4 and 5 a planar lens formed by rings is developed in order to obtain a high E_z component. The data of the radii are shown in Table 1.

[14]:
```
size = 30 * um
x0 = np.linspace(-size, size, 1024)
y0 = np.linspace(-size, size, 1024)
```

[15]:
```
wavelength = .6328 * um
z_obs = 10.32 * um
```

[16]:
```
inner_radius = np.array([
    0.15, 1.3, 2.45, 3.3, 4.25, 5.35, 6.2, 7, 8.35, 9.65, 10.55, 11.9, 13.2,
    14.15, 15.25, 16.2, 17.6, 19.05, 19.8, 21.15, 22, 23.05, 24, 24.95, 26,
    26.8, 27.55, 28.25, 29.05
])

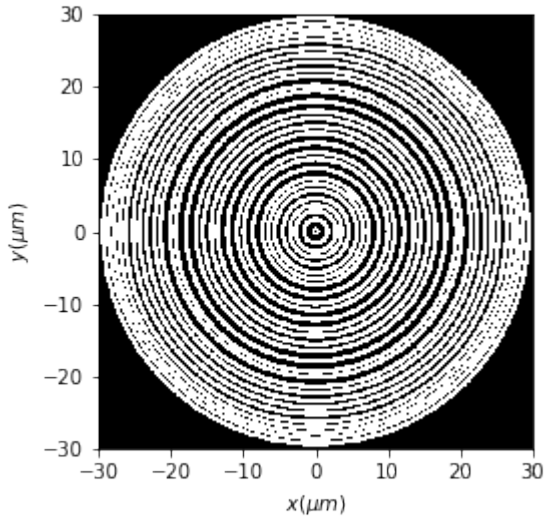
outer_radius = np.array([
    0.5, 1.95, 3.1, 3.95, 4.9, 6, 6.85, 7.65, 9, 10.3, 11.2, 12.55, 13.85,
    14.8, 15.9, 16.85, 18.25, 19.7, 20.45, 21.8, 22.65, 23.7, 24.65, 25.6,
    26.65, 27.45, 28.2, 28.9, 29.7
])
```

[17]:
```
u0 = Scalar_source_XY(x0, y0, wavelength)
u0.plane_wave(A=1)

t0 = Scalar_mask_XY(x0, y0, wavelength)
t0.rings(r0=(0, 0), inner_radius=inner_radius, outer_radius=outer_radius)
t0.pupil()

t = u0 * t0
```

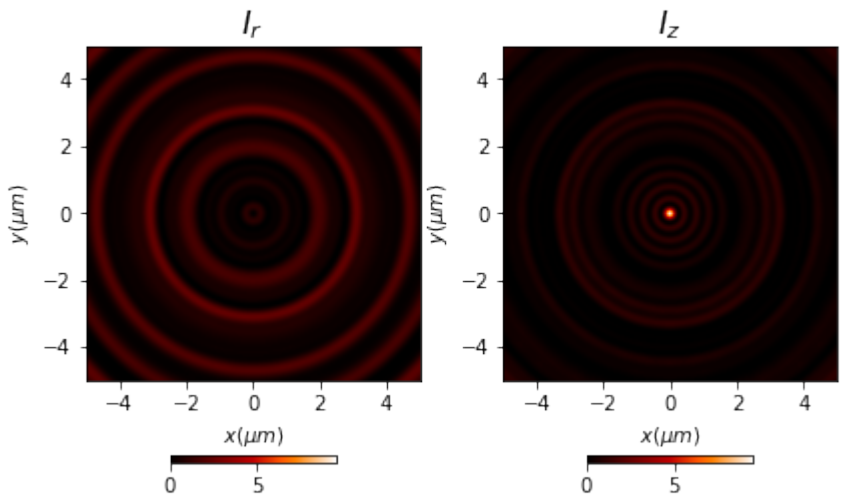
[18]:
```
t0.draw()
```



[19]:
```
E0 = Vector_source_XY(x0, y0, wavelength)
E0.radial_wave(u=t, r0=(0, 0))
```

[20]:
```
E1 = E0.VRS(z=z_obs, n=1, new_field=True)
E1.cut_resample([-5, 5], [-5, 5], num_points=[512, 512])
```

[21]:
```
E1.draw('intensities_rz', logarithm=False)
```



[22]:
```
Ex, Ey, Ez = E1.get('fields', is_matrix=False)

Iz = np.abs(Ez.u)**2
Ir = np.abs(Ex.u)**2 + np.abs(Ey.u)**2

I_total = Iz + Ir
```

(continues on next page)

(continued from previous page)

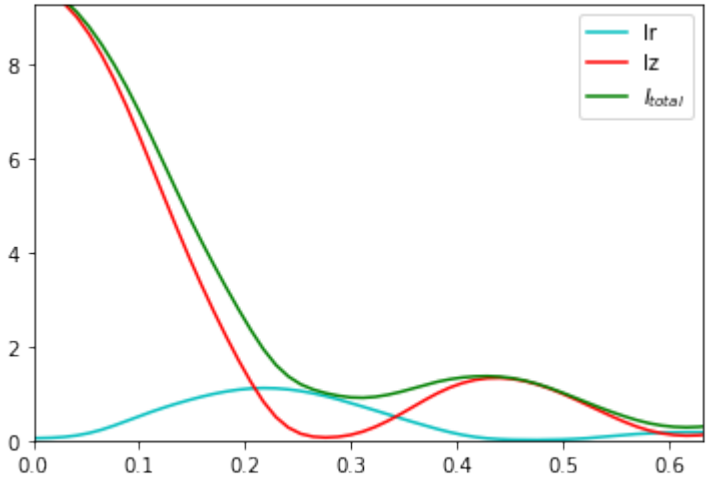
```
Ex.u = np.sqrt(Ir)
Ey.u = np.sqrt(Iz)

z, I_r, _, _ = Ex.profile([-3, 0], [3, 0], order=2)
z, I_z, _, _ = Ey.profile([-3, 0], [3, 0], order=2)

I_total = I_r + I_z
```

[23]:
```
plt.figure()
plt.plot(z, I_r, 'c', label='Ir')
plt.plot(z, I_z, 'r', label='Iz')
plt.plot(z, I_total, 'g', label='$I_{total}$')
plt.legend()

plt.xlim(0, wavelength)
plt.ylim(ymin=0, ymax=9.25)
```

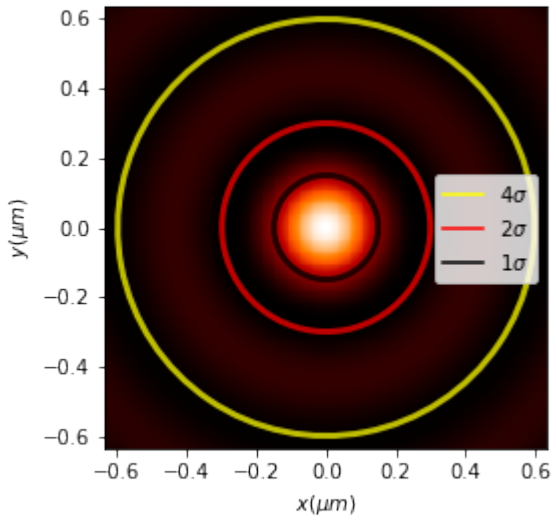


[24]:
```
Ey.u = Ey.u / Ey.u.max()
```

[25]:
```
Ey.cut_resample([-wavelength, wavelength], [-wavelength, wavelength],
                num_points=[256, 256])
```

[26]:
```
width, _, _, _ = Ey.beam_width_4s()
print("width = {} lambda".format(width / (2 * np.sqrt(2))))
```

```
width = 0.4219562175641547 lambda
```


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
y(μm)
−0.6 −0.4 −0.2 0.0 0.2 0.4 0.6
x(μm)
4σ
2σ
1σ

CHAPTER

EIGHT

ABOUT FUNCTIONG

8.1 Developing new functions

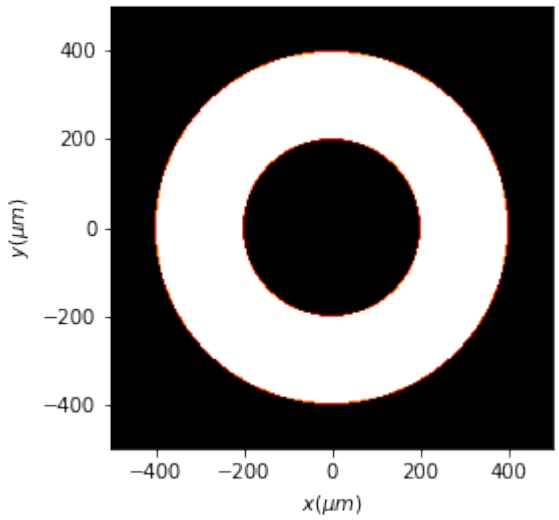
Sometimes we need to generate new functions for the modules. For development purposes, new functions can be generated in jupyter and included in a module as if they were inside the module: - Create the function to test. - Make the first argument of the function: self - Return the same value: self - When calling the function: u1=function_new(u1,...)

[1]:
```
from diffractio import np, plt, sp, um, mm, degrees, nm
from diffractio.scalar_masks_XY import Scalar_mask_XY
```

[2]:
```
x0 = np.linspace(-500 * um, 500 * um, 256)
y0 = np.linspace(-500 * um, 500 * um, 256)
wavelength = 0.6328 * um
```

8.1.1 Standard procedure

[3]:
```
u0 = Scalar_mask_XY(x0, y0, wavelength)
u0.ring(r0=(0, 0), radius1=200 * um, radius2=400 * um, angle=0 * degrees)
u0.draw()
```



This function, inside diffractio is:

[ ]:
```
def ring(self, r0, radius1, radius2, angle=0 * degrees):

        """ Ring.
         Parameters:
             r0 (float, float): center of ring
             radius1 (float, float) or (float): inner radius
             radius2 (float, float) or (float): outer radius
             angle (float): angle of rotation in radians
         """


         ring1 = Scalar_mask_XY(self.x, self.y, self.wavelength)
         ring2 = Scalar_mask_XY(self.x, self.y, self.wavelength)
         ring1.circle(r0, radius1, angle)
         ring2.circle(r0, radius2, angle)

         self.u = ring2.u - ring1.u
```

8.1.2 Development Mode

[4]:
```
def ring_new(self, r0, radius1, radius2, angle=0 * degrees):
    """ Ring.

    Parameters:
        r0 (float, float): center of ring
        radius1 (float, float) or (float): inner radius
        radius2 (float, float) or (float): outer radius
        angle (float): angle of rotation in radians
    """

    ring1 = Scalar_mask_XY(self.x, self.y, self.wavelength)
```

(continues on next page)

(continued from previous page)

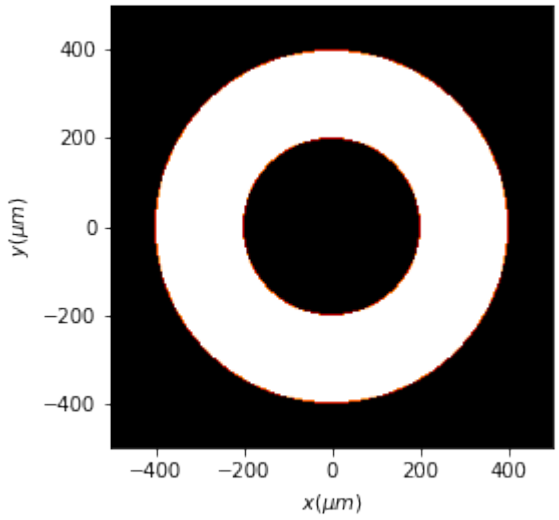
```
    ring2 = Scalar_mask_XY(self.x, self.y, self.wavelength)
    ring1.circle(r0, radius1, angle)
    ring2.circle(r0, radius2, angle)

    self.u = ring2.u - ring1.u
    print("I am in the new function")
    return self
```

[5]:
```
# Now it works as if it were inside Scalar_mask_XY

u1 = Scalar_mask_XY(x0, y0, wavelength)
u1 = ring_new(u1,
              r0=(0, 0),
              radius1=200 * um,
              radius2=400 * um,
              angle=0 * degrees)
u1.draw()
```

```
I am in the new function
```



[ ]:
```
```

8.2 Add surfaces to an object

Here, we show a procedure to add an structured surface to an object. The surafaces are to lenses. The first one is an Frsnel lens, and the second surface is an stardard lens.

Two examples are performed. Add the surfaces to a layer and to a biprism

[1]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_sources_X import Scalar_source_X
```

(continues on next page)

(continued from previous page)

```
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
```

[2]:
```
x0 = np.linspace(-.15 * mm, .15 * mm, 2048)
z0 = np.linspace(-25 * um, 400 * um, 2048)
wavelength = 10 * um
n_background = 1
```

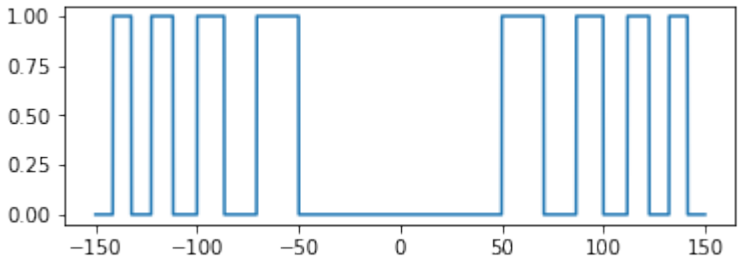
[3]:
```
u0 = Scalar_source_X(x0, wavelength)
u0.plane_wave()
```

8.2.1 Surfaces

[4]:
```
surface_1 = Scalar_mask_X(x0, wavelength, n_background=1, info='')

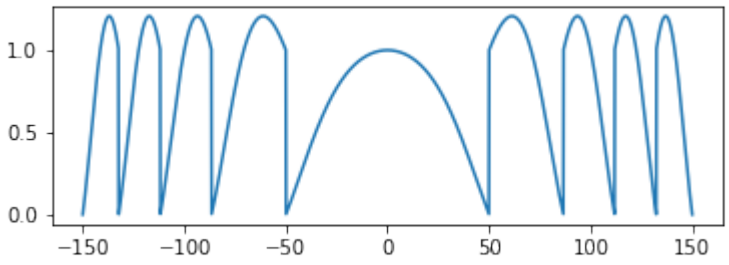
h1 = surface_1.fresnel_lens(x0=0.0,
                            focal=.25 * mm,
                            binary=True,
                            kind='phase',
                            phase=sp.pi,
                            radius=1 * mm)

plt.figure(figsize=(6, 2))
plt.plot(surface_1.x, h1.real)
```



[5]:
```
surface_2 = Scalar_mask_X(x0, wavelength, n_background=1, info='')
h2 = surface_2.fresnel_lens(x0=0.0,
                            focal=-.25 * mm,
                            binary=False,
                            kind='phase',
                            phase=sp.pi,
                            radius=1 * mm)

plt.figure(figsize=(6, 2))
plt.plot(surface_2.x, h2.real)
```



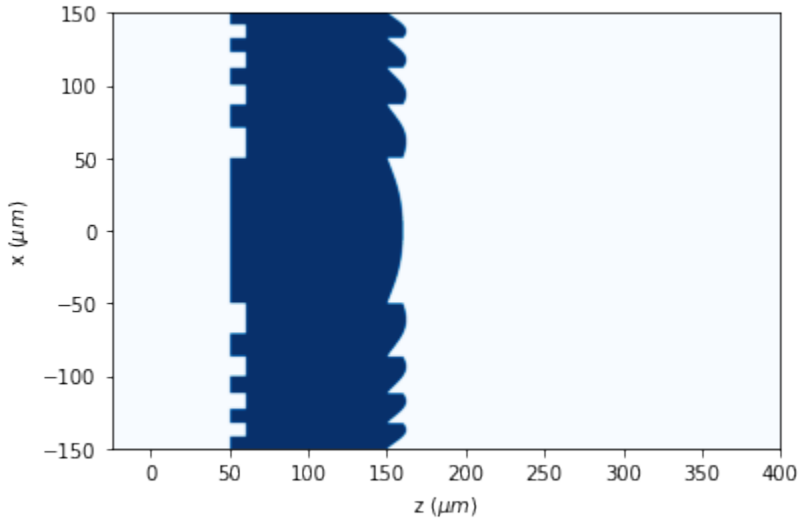
[6]:
```
fx1 = surface_1.x, h1 * wavelength
fx2 = surface_2.x, h2 * wavelength
```

8.2.2 Layer

[7]:
```
layer = Scalar_mask_XZ(x0, z0, wavelength, n_background=n_background, info='')
#lente3.biprism(r0=(0,0), length=.3*mm, height=.25*mm, refractive_index=1.5);
layer.layer(r0=(0 * mm, 50 * um),
            refractive_index=1.5,
            depth=100 * um,
            angle=0)
```

[9]:
```
layer.add_surfaces(fx=(fx1, fx2),
                   refractive_index=1.5,
                   x_sides=[-1 * mm, 1 * mm],
                   min_incr=0.1,
                   angle=0 * degrees)
```

[10]:
```
layer.draw_refractive_index(draw_borders=False)
layer.smooth_refractive_index(pixels_filtering=2 * um,
                              draw_check=False,
                              max_diff_filter=0.1)
```



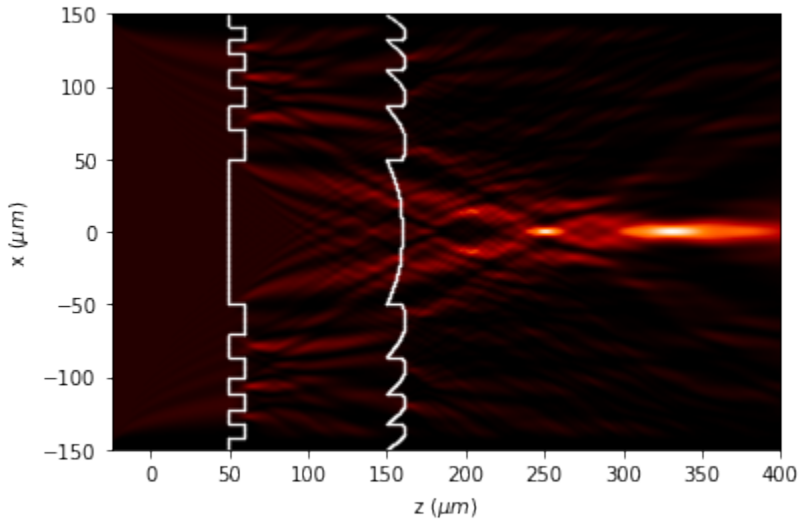
Propagation

[11]:
```
layer.incident_field(u0)
```

[12]:
```
layer.clear_field()
layer.BPM(verbose=True)
```

```
Time = 1.18 s, time/loop = 0.577 ms
```

[13]:
```
layer.draw(logarithm=False, normalize='maximum', draw_borders=True)
```

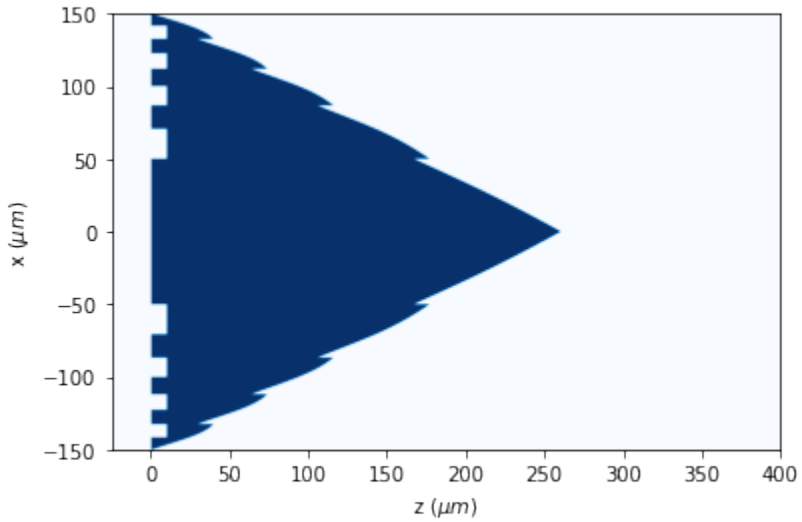


8.2.3 Biprism

[14]:
```
biprism = Scalar_mask_XZ(x0,
                         z0,
                         wavelength,
                         n_background=n_background,
                         info='')
biprism.biprism(r0=(0, 0),
                length=.3 * mm,
                height=.25 * mm,
                refractive_index=1.5)
```

[16]:
```
biprism.add_surfaces(fx=(fx1, fx2),
                     refractive_index=1.5,
                     x_sides=[-1 * mm, 1 * mm],
                     min_incr=0.1,
                     angle=0 * degrees)
```

[17]:
```
biprism.draw_refractive_index(draw_borders=False)
```



[18]:
```
biprism.smooth_refractive_index(pixels_filtering=2 * um,
                                draw_check=False,
                                max_diff_filter=0.1)
```

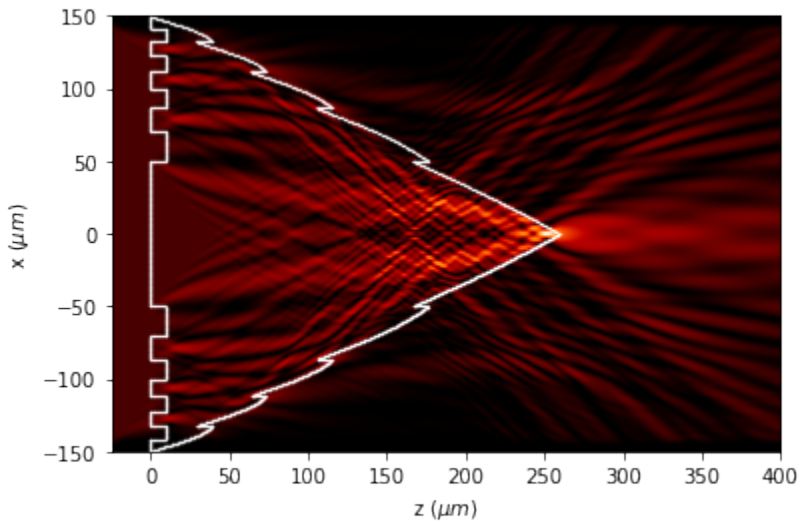
Propagation

[19]:
```
biprism.incident_field(u0)
```

[20]:
```
biprism.clear_field()
biprism.BPM(verbose=True)
```

```
Time = 0.86 s, time/loop = 0.4193 ms
```

[21]:
```
biprism.draw(logarithm=1e0, normalize='maximum', draw_borders=True)
```



8.3 Filtering: removing artifacts in BPM and WPM

For Rayleigh-Sommerfeld approach, masks are considered using Thin Element Approximation (TEA), that is, they are infinitely thin. This can be useful in many situations, but does not allow determining the field inside elements.

With Beam Propagation Method (BPM) and Wave Propagation Method (WPM), we define an XZ plane with a background index (n=1 for vaccum, for example) and insert optical elements considering its shape and refraction index. We also need to include the input scalar field and propagate. Since the field is computed sequentially, no multiprocessing is possible for determining the u(x,z) field in monocromatic, single field.

One disventage of this technique is that no reflections are considered, although perhaps they can be included in the future.

One of the limits of BPM and WPM is the resolution in the grid of x,z. If the grid is not small enough, for example, in a phase mask like a lens, the shape is not considered continuous but with edges. In this case, edges produce a discontinuity that proagates.

[1]:
```
%load_ext autoreload
%autoreload 2
```

[2]:
```
from diffractio import degrees, mm, plt, sp, um, np

from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
```

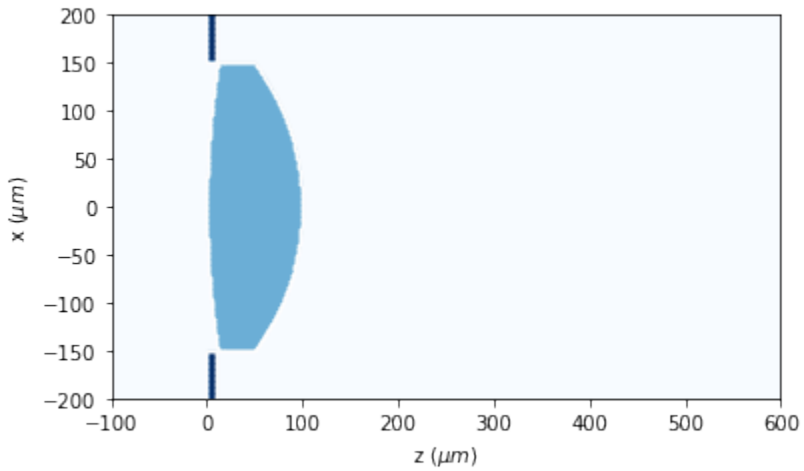
8.3.1 Bad computation with BPM

[3]:
```
x0 = np.linspace(-200 * um, 200 * um, 512)
z0 = np.linspace(-100 * um, 600 * um, 512 * 4)
wavelength = 0.6238 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=200 * um, theta=0. * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=300 * um,
                              radius=(1000 * um, -250 * um),
                              thickness=100 * um,
                              refractive_index=2,
                              angle=0 * degrees,
                              mask=(10 * um, 3 + 0.05j))
u1.clear_field()

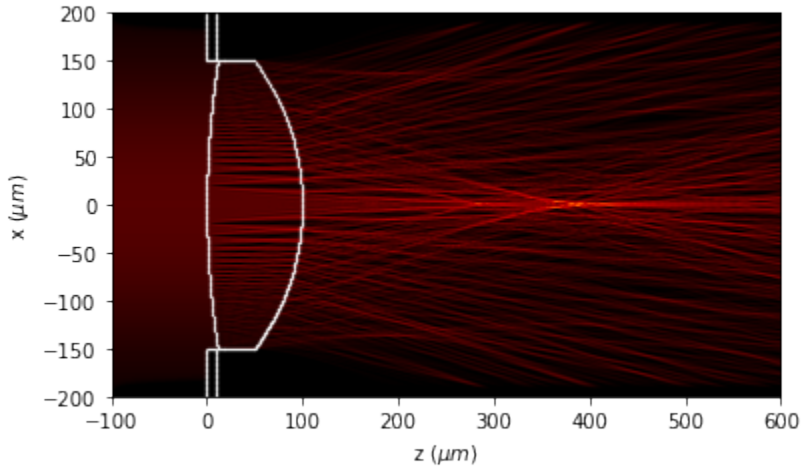
print(focal)
```

```
192.30769230769232
```

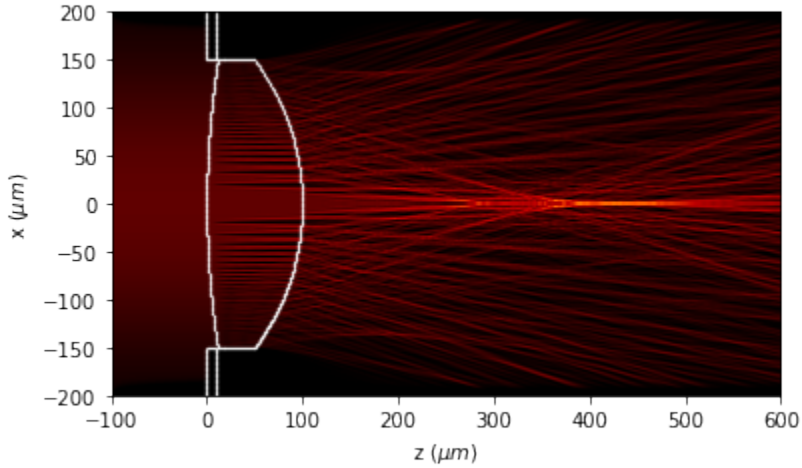
[4]:
```
u1.draw_refractive_index(scale='scaled')
edge_matrix = u1.borders
```



[5]:
```
u1.BPM(verbose=False)
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
```



[6]:
```
u1.clear_field()
u1.WPM(verbose=False)
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
```



8.3.2 Better resolution

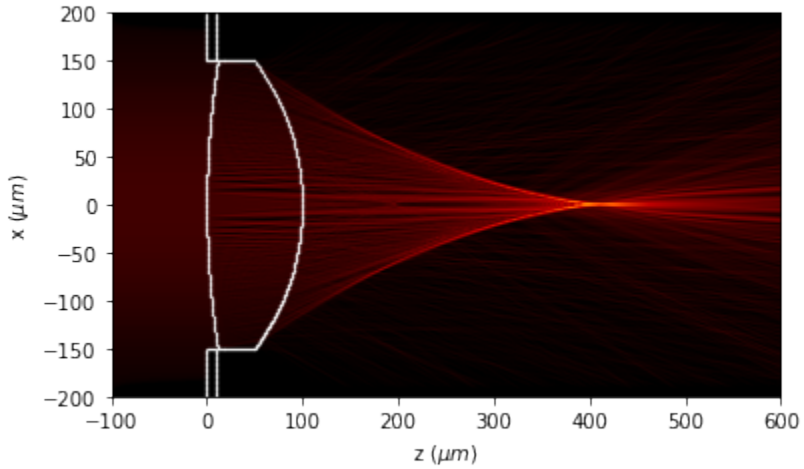
[7]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024 * 1)
z0 = np.linspace(-100 * um, 600 * um, 1024 * 2)
wavelength = 0.6238 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=200 * um, theta=0. * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=300 * um,
                              radius=(500 * um, -250 * um),
                              thickness=100 * um,
                              refractive_index=1.5,
```

(continues on next page)

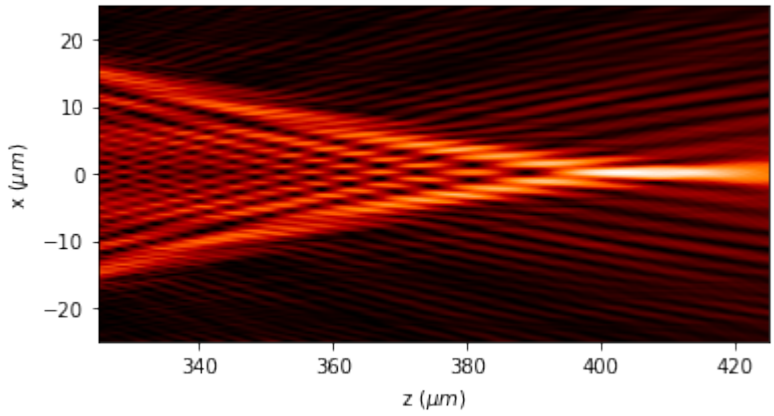
(continued from previous page)

```
                                angle=0 * degrees,
                                mask=(10 * um, 3 + 0.05j))
```

[8]:
```
u1.clear_field()
u1.BPM(verbose=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        scale='scaled',
        edge_matrix=edge_matrix)
```



[9]:
```
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
plt.ylim(-25, 25)
plt.xlim(325, 425)
```

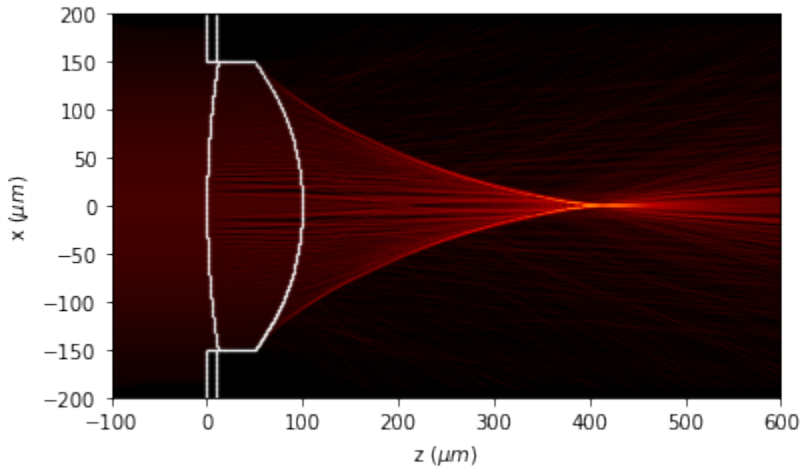


[10]:
```
u1.clear_field()
u1.WPM(verbose=False)
u1.draw(logarithm=True,
```

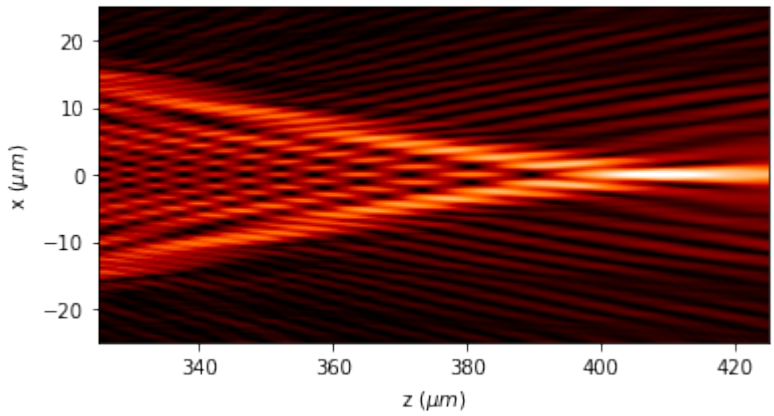
(continues on next page)

(continued from previous page)

```
        normalize='maximum',
        draw_borders=True,
        scale='scaled',
        edge_matrix=edge_matrix)
```



[11]:
```
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
plt.ylim(-25, 25)
plt.xlim(325, 425)
```



8.3.3 Effect of filtering

[12]:
```
x0 = np.linspace(-200 * um, 200 * um, 1024 * 1)
z0 = np.linspace(-100 * um, 600 * um, 1024 * 2)
wavelength = 0.6238 * um
u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.gauss_beam(A=1, x0=0 * um, z0=0 * um, w0=200 * um, theta=0. * degrees)
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
focal, _ = u1.lens_convergent(r0=(0, 0),
                              aperture=300 * um,
```

(continues on next page)

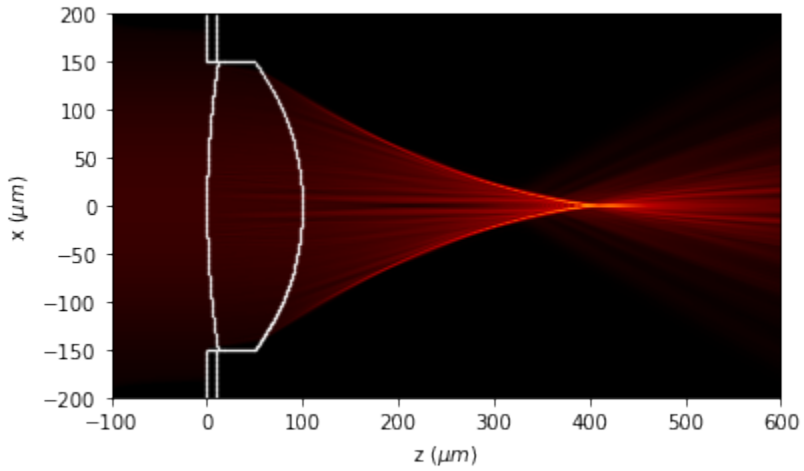
(continued from previous page)

```
                                radius=(500 * um, -250 * um),
                                thickness=100 * um,
                                refractive_index=1.5,
                                angle=0 * degrees,
                                mask=(10 * um, 3 + 0.05j))
```

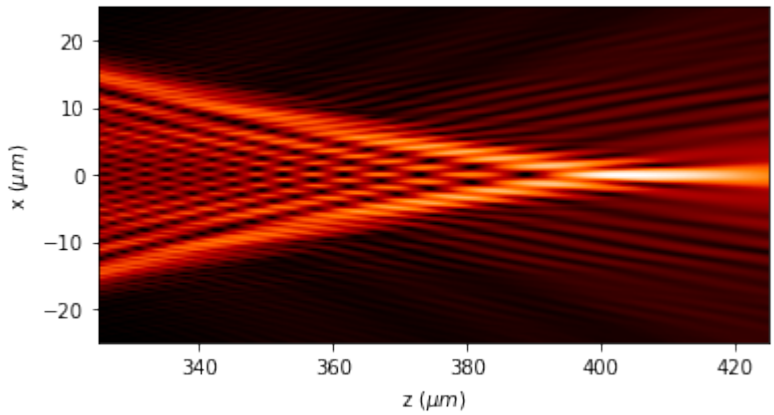
[13]:
```
u1.smooth_refractive_index(type_filter=1, pixels_filtering=10)
```

[14]:
```
u1.clear_field()
u1.BPM(verbose=False)
```

[15]:
```
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        scale='scaled',
        edge_matrix=edge_matrix)
```



[16]:
```
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
plt.ylim(-25, 25)
plt.xlim(325, 425)
```

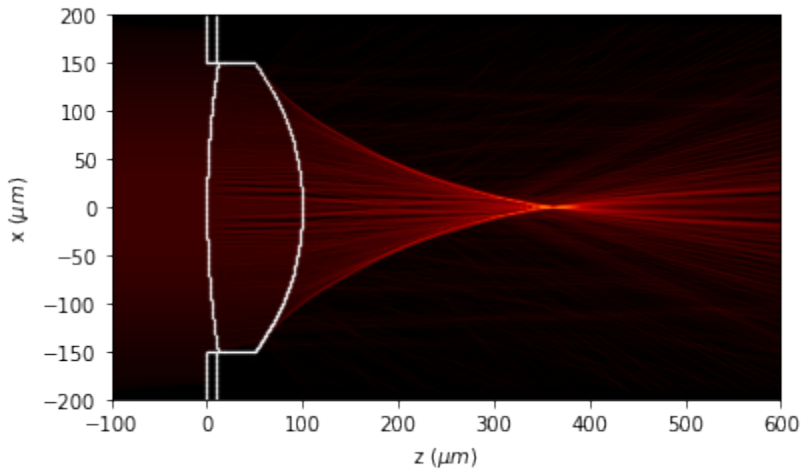


Now, it is possible to perform a filtering for WPM method, but it is necessary to discretize the refraction index, since the computing time in WPM is proportional to the number of refraction indexes at each iteration.

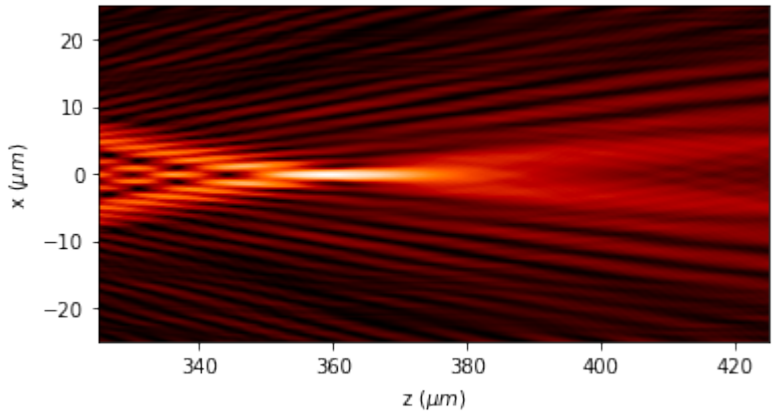
[17]:
```
u1.discretize_refractive_index(num_layers=8, new_field=False)
```

[18]:
```
u1.clear_field()
u1.WPM(verbose=False)
```

[19]:
```
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        scale='scaled',
        edge_matrix=edge_matrix)
```



[20]:
```
u1.draw(logarithm=True, normalize='maximum', draw_borders=True, scale='scaled')
plt.ylim(-25, 25)
plt.xlim(325, 425)
```



As a conclusion, WPM and BPM does not produce the same results. The reason is that BPM is developed for low variations in refraction indexes. In this problem we cannot be considered the variation low. [S. Schmidt "Wave-optical modeling beyond the thin-element-approximation" Optics Express 24(26) 30188 (2016) http://dx.doi.org/10.1364/OE.24.030188]

Nevertheless, BPM accepts arbitrary refraction indexes, and the algorithm of WPM is effective when assuming that only a limited number of different refractive indices are present in the problem. [Brenner, Karl-Heinz. "A high-speed version of the wave propagation method applied to micro-optics." 16th Workshop on Information Optics (WIO). IEEE, (2017)].

8.4 Multiprocessing examples

In this example we will analyze several cases were we can use multiprocessing for accelerating our computations. Some of this cases are implemented in the modules. Other cases are implemented in specific functions.

The cases that we will se include:

- Rayleigh-Sommerfed computations in XZ and XYZ modules.
- Polychromatic light
- Extended sources
- Polychromatic and extended sources
- Simultaneous simulations, as moving parts

[3]:
```
import time

from diffractio import degrees, mm, np, num_max_processors, plt, um
from diffractio.scalar_fields_X import (Scalar_field_X,
                                        extended_polychromatic_source,
                                        extended_source_multiprocessing,
                                        polychromatic_multiprocessing)
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.utils_multiprocessing import (_pickle_method, _unpickle_method,
                                              execute_multiprocessing)
from multiprocessing import Pool
from diffractio.utils_optics import (gauss_spectrum, lorentz_spectrum,
```

(continues on next page)

(continued from previous page)

```
                                        uniform_spectrum)

from diffractio.scalar_fields_XZ import Scalar_field_XZ
```

8.4.1 Extended Source

[4]:
```
x = np.linspace(-500 * um, 500 * um, 1024 * 8)
wavelength = 0.850 * um
num_fuentes = 101
S = 3000 * um
z0 = -500 * mm
period = 50 * um
focal = 5 * mm

red = Scalar_mask_X(x, wavelength)
red.ronchi_grating(x0=0 * um, period=period, fill_factor=0.5)

lens = Scalar_mask_X(x, wavelength)
lens.lens(x0=0, focal=focal, radius=30 * mm)

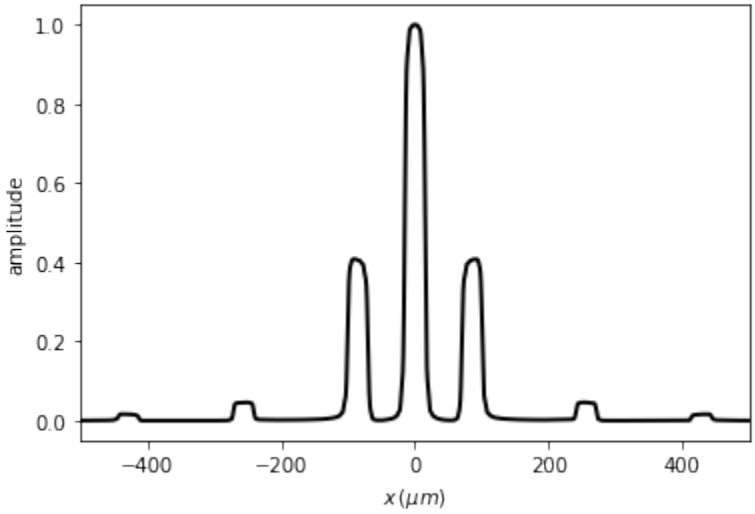
u1 = Scalar_source_X(x, wavelength)

# posiciones de la fuente
x0s = np.linspace(-S / 2, S / 2, num_fuentes)
intensities = Scalar_field_X(x, wavelength)

time1 = time.time()
for x0 in x0s:
    u1.spherical_wave(A=1, x0=x0, z0=z0, normalize=True)
    u2 = u1 * lens * red
    u2.RS(z=focal, new_field=False, verbose=False)
    intensities.u = intensities.u + abs(u2.u)**2

intensities.u = intensities.u / intensities.u.max()
time_proc = time.time() - time1
print("num_proc: {}, time={}".format(1, time_proc))
intensities.draw(kind='amplitude')
```

```
num_proc: 1, time=2.0856175422668457
```



[5]:
```
def __experiment_extended_source__(x0):

    x = np.linspace(-500 * um, 500 * um, 1024)
    wavelength = 0.850 * um
    z0 = -500 * mm
    period = 50 * um
    focal = 5 * mm

    red = Scalar_mask_X(x, wavelength)
    red.ronchi_grating(x0=0 * um, period=period, fill_factor=0.5)

    lens = Scalar_mask_X(x, wavelength)
    lens.lens(x0=0, focal=focal, radius=30 * mm)

    u1 = Scalar_source_X(x, wavelength)

    u1.spherical_wave(A=1., x0=x0, z0=z0,  normalize=True)
    u2 = u1 * red * lens
    u2.RS(z=focal, new_field=False, verbose=False)

    return u2
```

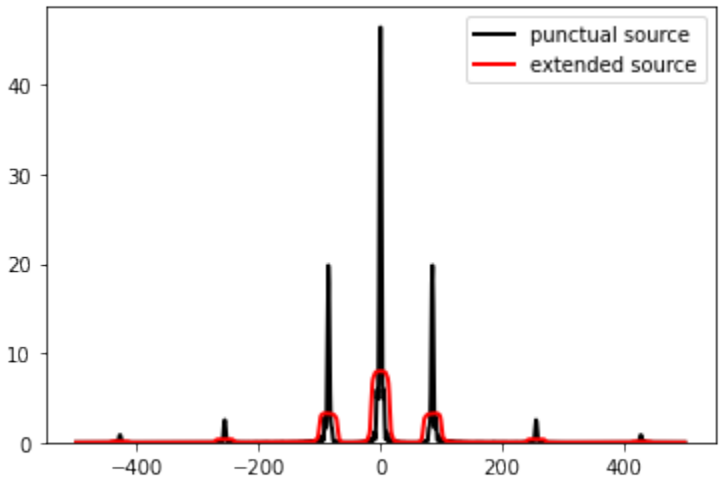
[6]:
```
x0s = np.linspace(-1500 * um, 1500 * um, 101)
x0_central = 0 * um

intensity0, u_s0, time_proc0 = extended_source_multiprocessing(
    __experiment_extended_source__, x0_central, num_processors=1, verbose=True)

intensity, u_s, time_proc = extended_source_multiprocessing(
    __experiment_extended_source__,
    x0s,
    num_processors=num_max_processors,
    verbose=True)
```

```
num_proc: 1, time=0.006186962127685547
num_proc: 8, time=0.4044191837310791
```

[7]:
```
plt.figure()
plt.plot(u_s0.x, intensity0, 'k', lw=2, label='punctual source')
plt.plot(u_s[0].x, intensity, 'r', lw=2, label='extended source')
plt.legend()
plt.ylim(bottom=0)
```



8.4.2 Polychromatic light sources

[8]:
```
wavelengths = np.linspace(.3 * um, .9 * um, 1001)

w_central = wavelengths.mean()

spectrum_gauss = gauss_spectrum(wavelengths=wavelengths,
                                w_central=w_central,
                                Dw=0.4,
                                normalize=True)
```

[9]:
```
def __function_polychromatic__(wavelength):
    x0 = np.linspace(-75 * um, 75 * um, 1024 * 8)

    t1 = Scalar_mask_X(x0, wavelength)
    t1.slit(x0=0, size=100 * um)

    f1 = Scalar_source_X(x0, wavelength)
    f1.gauss_beam(x0=0 * um, w0=200 * um, z0=0 * um, A=1, theta=0. * degrees)

    u1 = f1 * t1
    u1.RS(z=10 * um, new_field=False, verbose=False)

    return u1
```

[10]:
```
time1 = time.time()
intensity_0, u_s0, time_monochromatic_proc_0 = polychromatic_multiprocessing(
    __function_polychromatic__, w_central, 1, num_processors=1, verbose=True)
time_proc_0 = time.time() - time1
print("Monochromatic: time={:2.5f}".format(time_proc_0))
```

```
Monochromatic: time=0.06294
```

[11]:
```
time1 = time.time()
intensity_1, u_s1, time_proc_1 = polychromatic_multiprocessing(
    __function_polychromatic__,
    wavelengths,
    spectrum_gauss,
    num_processors=1,
    verbose=True)
time_proc1 = time.time() - time1

print("Polychromatic - num_proc: 1,  time={}".format(time_proc_1))
```

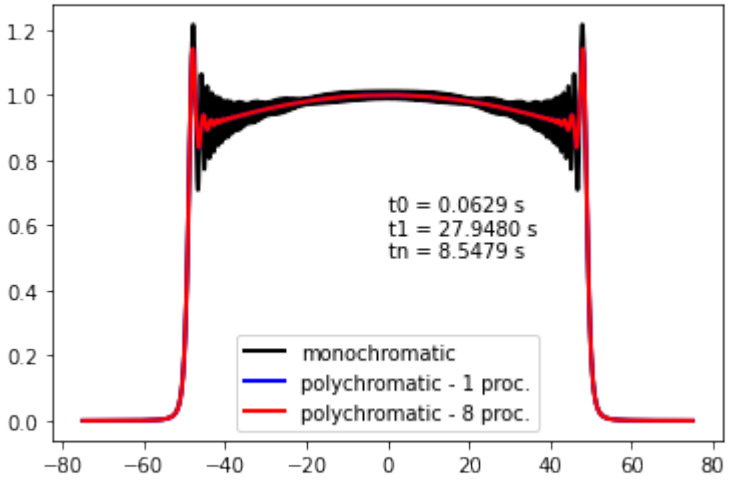
```
num_proc: 1, time=27.947970867156982
1001
Polychromatic - num_proc: 1,  time=27.947970867156982
```

[12]:
```
intensity_n, u_s, time_proc_n = polychromatic_multiprocessing(
    __function_polychromatic__,
    wavelengths,
    spectrum_gauss,
    num_processors=num_max_processors,
    verbose=True)
```

```
num_proc: 8, time=8.547862768173218
1001
```

[13]:
```
plt.figure()
plt.plot(u_s0.x, intensity_0, 'k', lw=2, label='monochromatic')
plt.plot(u_s0.x, intensity_1, 'b', lw=2, label='polychromatic - 1 proc.')
plt.plot(u_s[0].x,
         intensity_n,
         'r',
         lw=2,
         label='polychromatic - {} proc.'.format(num_max_processors))
plt.legend()

plt.text(
    0, 0.5, 't0 = {:2.4f} s\nt1 = {:2.4f} s\ntn = {:2.4f} s'.format(
        time_proc_0, time_proc_1, time_proc_n))
```



8.4.3 Simultaneous experiments: moving a grating in a double grating experiment

[14]:
```
def __experiment_grating_movement__(dict_params):
    delta_x = dict_params['delta_x']
    period = dict_params['period']
    t1 = dict_params['t1']
    t2 = dict_params['t2']
    t2.ronchi_grating(x0=delta_x, period=period, fill_factor=0.5)
    u2 = t1 * t2
    return u2
```

[15]:
```
def creation_dictionary(deltas_x, period, t1, t2):
    # create Parameters: for multiprocessing
    dict_Parameters = []
    for i, delta_x in enumerate(deltas_x):
        dict_Parameters.append(
            dict(delta_x=delta_x, period=period, t1=t1, t2=t2))
    return dict_Parameters
```

[16]:
```
x0 = np.linspace(-400 * um, 400 * um, 1024 * 2)
wavelength = 0.85 * um
period = 50 * um
z_talbot = 2 * period**2 / wavelength
z0 = z_talbot / 2
delay = 0.001

t1 = Scalar_mask_X(x0, wavelength, info="__experiment_grating_movement__")
t1.ronchi_grating(x0=0 * um, period=period, fill_factor=0.5)
t1.RS(z=z0, new_field=False)

t2 = Scalar_mask_X(x0, wavelength, info="__experiment_grating_movement__")
t2.ronchi_grating(x0=0 * um, period=period, fill_factor=0.5)

deltas_x = np.linspace(-60 * um, 60 * um, 128)  # 512
```

(continues on next page)

(continued from previous page)

```
num_processors = num_max_processors

dict_Parameters = creation_dictionary(deltas_x=deltas_x,
                                      period=period,
                                      t1=t1,
                                      t2=t2)

u_s, time_proc = execute_multiprocessing(__experiment_grating_movement__,
                                         dict_Parameters,
                                         num_processors,
                                         verbose=True)

x = u_s[0].x
```

```
Good result: factor 8.01
num_proc: 8, time=0.8066287040710449
```

[17]:
```
%matplotlib qt
```

[18]:
```
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

perfil = np.zeros_like(deltas_x)

h1, = ax1.plot(x, np.zeros_like(x), 'k', lw=2)
ax1.set_xlim(x[0], x[-1])
ax1.set_ylim(0, 2)
ax1.set_xlabel(r'$x (\mu m)$')

h2, = ax2.plot(deltas_x, perfil, 'k', lw=2)
ax2.set_xlim(deltas_x[0], deltas_x[-1])
ax2.set_ylim(0, .5)
ax2.set_xlabel(r'$\Delta x (\mu m)$')

incr_frames = 1
for i in range(0, len(deltas_x), incr_frames):
    intensidad = abs(u_s[i].u)**2  # sacar fuera
    perfil[i] = intensidad.mean()
    plt.suptitle(r"$\delta x={:6.2f}\,\mu m$".format(deltas_x[i]), fontsize=18)
    h1.set_ydata(intensidad)
    h2.set_ydata(perfil)
    plt.draw()
    plt.pause(0.005)
```

```
QSocketNotifier: Can only be used with threads started with QThread
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
```

(continues on next page)

(continued from previous page)

```
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
```

(continues on next page)

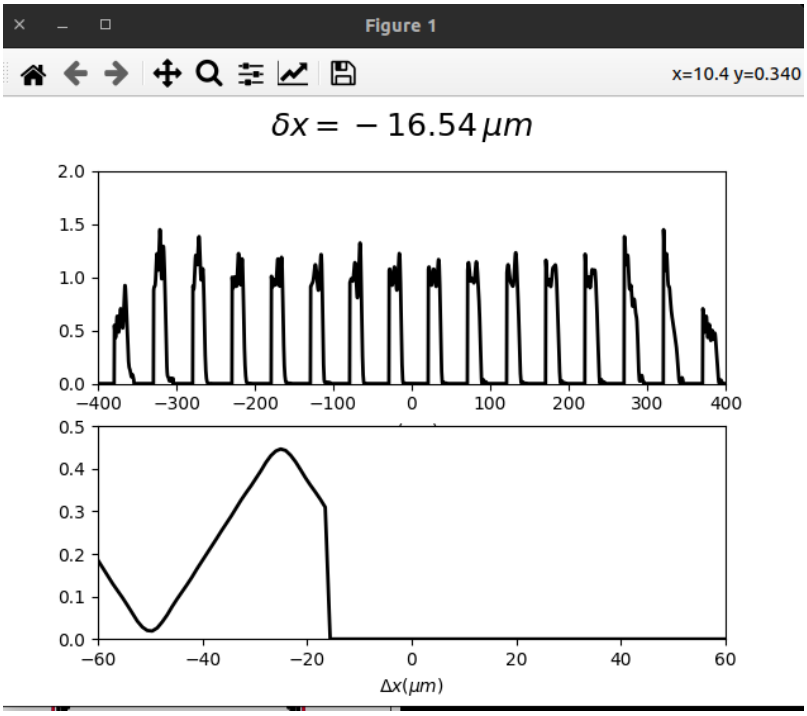
(continued from previous page)

```
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
```

(continues on next page)

(continued from previous page)

```
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
qt.qpa.wayland: Wayland does not support QWindow::requestActivate()
```



[19]:
```
%matplotlib inline
```

8.4.4 Drawing at several distances

[20]:
```
def __experiment_double_slit_dictionary__(dict_params):
    x0 = dict_params['x0']
    wavelength = dict_params['wavelength']
    z = dict_params['z']
    slit_size = dict_params['slit_size']
    separation = dict_params['separation']
    t1 = Scalar_mask_X(x0, wavelength)
    t1.slit(x0=0, size=20 * um)
    t1.double_slit(x0=0, size=slit_size, separation=separation)

    f1 = Scalar_source_X(x0, wavelength)
    f1.gauss_beam(x0=0 * um, w0=200 * um, z0=0 * um, A=1, theta=0. * degrees)

    u1 = f1 * t1
    u1.RS(z, new_field=False, verbose=False)

    return u1
```

[21]:
```
def creation_dictionary(wavelengths, x0, z, slit_size, separation):
    # create Parameters: for multiprocessing
    dict_Parameters = []
    for i, wavelength in enumerate(wavelengths):
        dict_Parameters.append(
            dict(x0=x0,
                 wavelength=wavelength,
                 z=z,
                 slit_size=slit_size,
                 separation=separation))
    return dict_Parameters


x0 = np.linspace(-100 * um, 100 * um, 1024 * 16)
wavelengths = np.linspace(0.3 * um, 0.8 * um, 101)
spectrum_gauss = gauss_spectrum(wavelengths=wavelengths,
                                w_central=0.6,
                                Dw=0.1,
                                normalize=True)
num_processors = num_max_processors

dict_Parameters0 = dict(x0=x0,
                        wavelength=0.6 * um,
                        z=.05 * mm,
                        slit_size=50 * um,
                        separation=75 * um)

dict_Parameters = creation_dictionary(wavelengths=wavelengths,
                                      x0=x0,
                                      z=.05 * mm,
                                      slit_size=50 * um,
                                      separation=75 * um)
```

[22]:
```
t1 = time.time()
I0 = __experiment_double_slit_dictionary__(dict_Parameters0)
t2 = time.time()
print("num_proc= 1, time={:2.2}s".format(t2 - t1))
```

```
num_proc= 1, time=0.061s
```

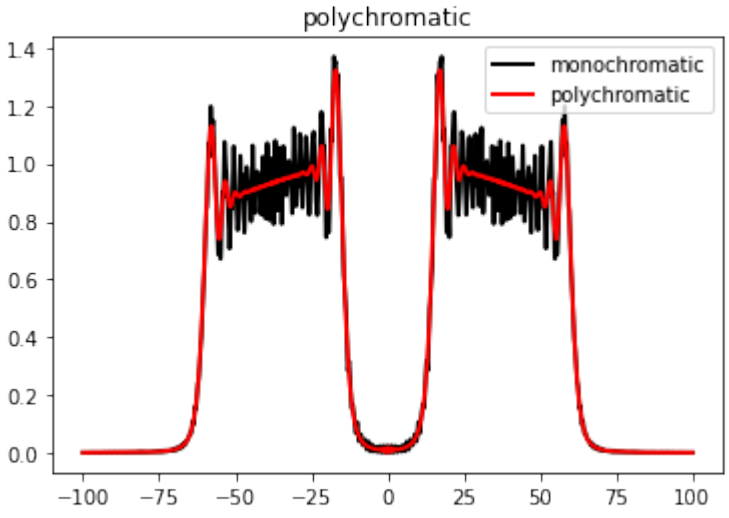
[23]:
```
I_wavelengths, time_proc = execute_multiprocessing(
    __experiment_double_slit_dictionary__,
    dict_Parameters,
    num_processors,
    verbose=False)

print("num_proc= {}, time={:2.2f}s".format(num_processors, time_proc))
```

```
num_proc= 8, time=2.03s
```

[24]:
```
intensity = np.zeros_like(I_wavelengths[0].x)
for i in range(len(wavelengths)):
    intensity = intensity + spectrum_gauss[i] * abs(I_wavelengths[i].u)**2

plt.figure()
plt.plot(x0, abs(I0.u)**2, 'k', lw=2, label='monochromatic')
plt.plot(x0, intensity, 'r', lw=2, label='polychromatic')
plt.title('polychromatic')
plt.legend()
```



8.4.5 Several independent experiments

[25]:
```
def __experiment_double_slit_array__(slit_size):
    x0 = np.linspace(-75 * um, 75 * um, 1024 * 32)
    wavelength = 0.6328 * um
    separation = 50 * um
    z = 100 * um

    t1 = Scalar_mask_X(x0, wavelength)
    t1.slit(x0=0, size=20 * um)
    t1.double_slit(x0=0, size=slit_size, separation=separation)

    f1 = Scalar_source_X(x0, wavelength)
    f1.gauss_beam(x0=0 * um, w0=200 * um, z0=0 * um, A=1, theta=0. * degrees)

    u1 = f1 * t1
    u1.RS(z, new_field=False, verbose=False)

    return u1
```

[26]:
```
slit_sizes = np.linspace(5 * um, 50 * um, 20)
num_processors = num_max_processors

u_s, time_proc = execute_multiprocessing(__experiment_double_slit_array__,
                                         slit_sizes,
                                         1,
                                         verbose=True)
```

```
num_proc: 1, time=2.5196688175201416
```

[27]:
```
slit_sizes = np.linspace(5 * um, 50 * um, 20)
num_processors = num_max_processors

u_s, time_proc = execute_multiprocessing(__experiment_double_slit_array__,
                                         slit_sizes,
                                         num_max_processors,
                                         verbose=True)
```

```
num_proc: 8, time=1.3638935089111328
```

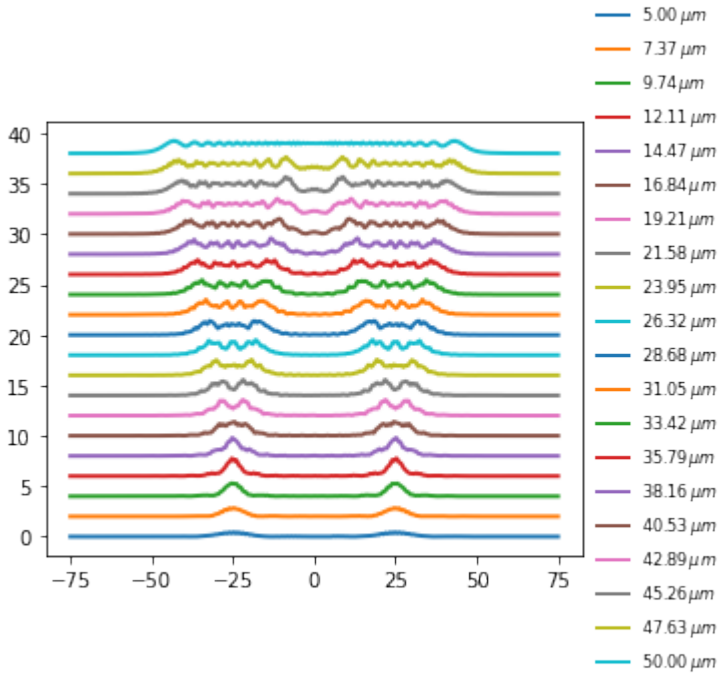
[28]:
```
plt.figure()
ax = plt.subplot(111)
for i, slit_size in enumerate(slit_sizes):
    ax.plot(u_s[i].x,
            abs(u_s[i].u)**2 + 2 * i,
            lw=2,
            label=r"${:2.2f}\,\mu m$".format(slit_size))
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])

# Put a legend to the right of the current axis
```

(continues on next page)

(continued from previous page)

```
ax.legend(fontsize=8,
          frameon=False,
          loc='center left',
          bbox_to_anchor=(1, 0.5))
```



8.4.6 def test_extended_polychromatic_source(self):

[29]:
```
x0s = np.linspace(-50 * um, 50 * um, 21)
wavelengths = np.linspace(.55 * um, .65 * um, 11)
w_central = wavelengths.mean()
spectrum_gauss = gauss_spectrum(wavelengths=wavelengths,
                                w_central=w_central,
                                Dw=0.01,
                                normalize=True)
```

[30]:
```
def __experiment_extended_polychromatic_source__(dict_params):
    wavelength = dict_params['wavelength']
    x0 = dict_params['x0']

    x = np.linspace(-1250 * um, 1250 * um, 1024 * 8)
    periodo = 100 * um
    z = 5 * periodo**2 / (0.6 * um)
    z0 = -50 * mm

    t1 = Scalar_mask_X(x, wavelength)
```

(continues on next page)

(continued from previous page)

```
    t1.ronchi_grating(x0=0 * um, period=periodo, fill_factor=0.5)

    f1 = Scalar_source_X(x, wavelength)
    f1.spherical_wave(A=1, x0=x0, z0=z0,  normalize=True)

    u1 = f1 * t1
    u1.RS(z, new_field=False, verbose=False)

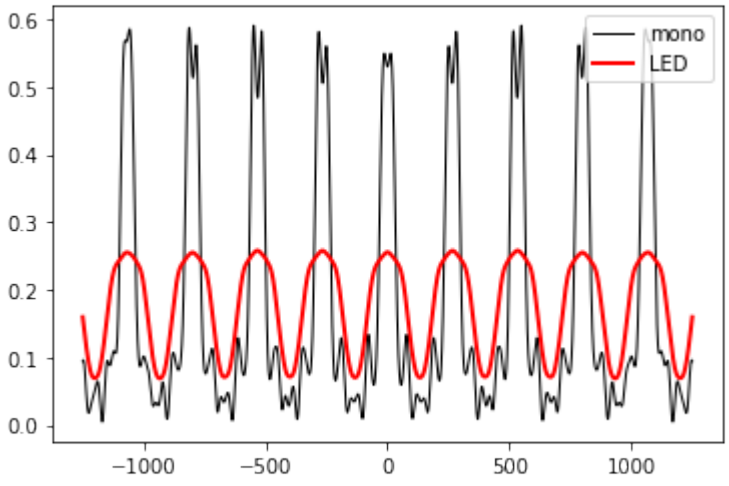
    return u1
```

[31]:
```
u0 = __experiment_extended_polychromatic_source__(
    dict(x0=0, wavelength=w_central))

intensity, u_s, time_proc = extended_polychromatic_source(
    __experiment_extended_polychromatic_source__,
    x0s,
    wavelengths,
    spectrum_gauss,
    num_processors=num_max_processors,
    verbose=True)
```

```
num_proc: 8, time=1.8315293788909912
```

[32]:
```
plt.figure()
plt.plot(u_s[0].x, abs(u0.u)**2, 'k', lw=1, label='mono')
plt.plot(u_s[0].x, intensity, 'r', lw=2, label='LED')
plt.legend()
```



8.4.7 Multiprocessing implemented in XZ module

[33]:
```
x = np.linspace(-350 * um, 350 * um, 1024 * 8)
z = np.linspace(1 * mm, 10 * mm, 1024)
wavelength = 0.6238 * um
period = 40 * um
z_talbot = 2 * period**2 / wavelength
```

[34]:
```
u0 = Scalar_source_X(x, wavelength)
u0.plane_wave(A=1)

t = Scalar_mask_X(x, wavelength)
t.ronchi_grating(x0=0 * um, period=40 * um, fill_factor=0.5)

talbot_effect = Scalar_field_XZ(x, z, wavelength)
talbot_effect.incident_field(u0 * t)
```

[35]:
```
t0 = time.time()
talbot_effect.RS(num_processors=1)
t1 = time.time()
t_proc1 = t1 - t0
print(t_proc1, t_proc1 / len(z))
```

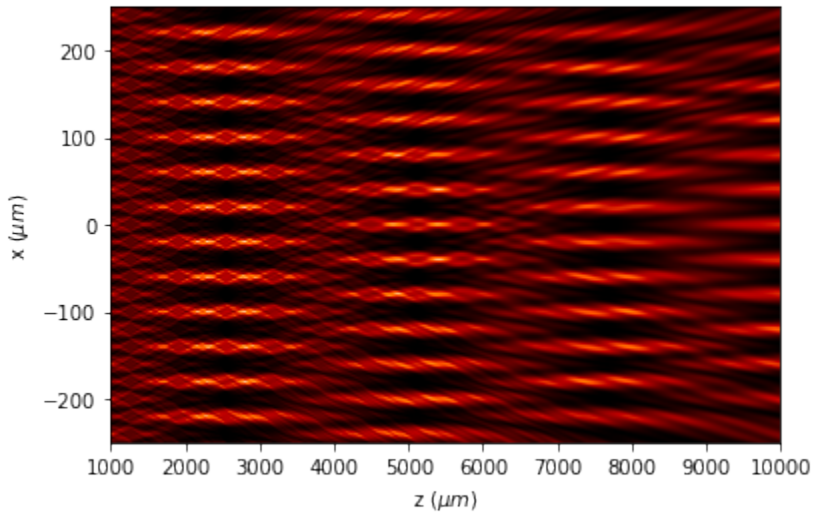
```
22.909979343414307 0.022373026702553034
```

[36]:
```
t0 = time.time()
talbot_effect.RS(num_processors=num_max_processors - 2)
t1 = time.time()
t_procn = t1 - t0

print(t_procn, t_procn / len(z))
```

```
65.31737923622131 0.06378650316037238
```

[37]:
```
talbot_effect.draw()
plt.ylim(-250 * um, 250 * um)
```



[ ]:

8.5 Check power in Rayleigh-Sommerfeld propagation

In this example, we verify that the energy of the beam (here we use a Gauss beam) is constant for propagation. We use the XYZ module and Rayleigh-Sommerfeld algorithm.

[1]:
```
from diffractio import degrees, eps, mm, no_date, np, um, plt
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
from diffractio.scalar_sources_XY import Scalar_source_XY
```

[2]:
```
x0 = np.linspace(-100 * um, 100 * um, 256)
y0 = np.linspace(-100 * um, 100 * um, 256)
z0 = np.linspace(500 * um, 10 * mm, 32)
wavelength = .6328 * um
```

[3]:
```
t1 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
t1.gauss_beam(r0=(0, 0), w0=40 * um, z0=0.0, A=1, theta=0.0, phi=0.0)

uxyz = Scalar_mask_XYZ(x=x0, y=y0, z=z0, wavelength=wavelength)

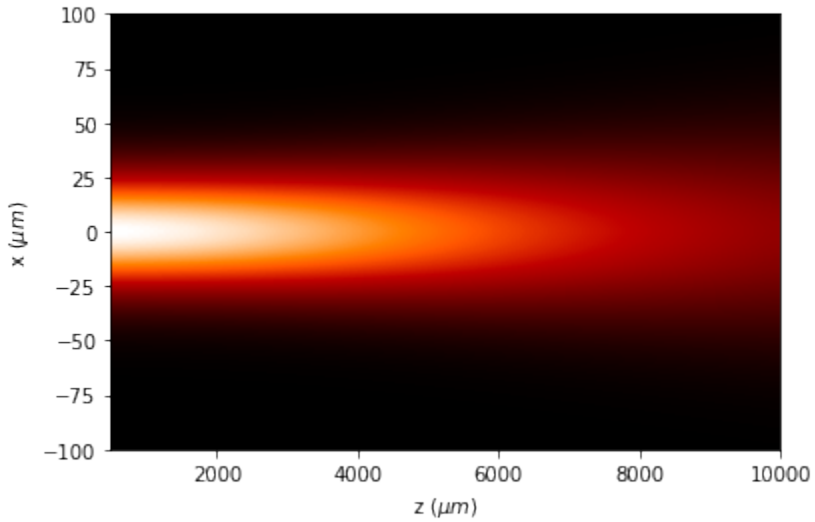
uxyz.incident_field(u0=t1)

uxyz.RS(verbose=True, num_processors=4)
```

```
time in RS= 7.766473054885864. num proc= 4
```

[4]:
```
uxyz.draw_XZ(y0=0)
```

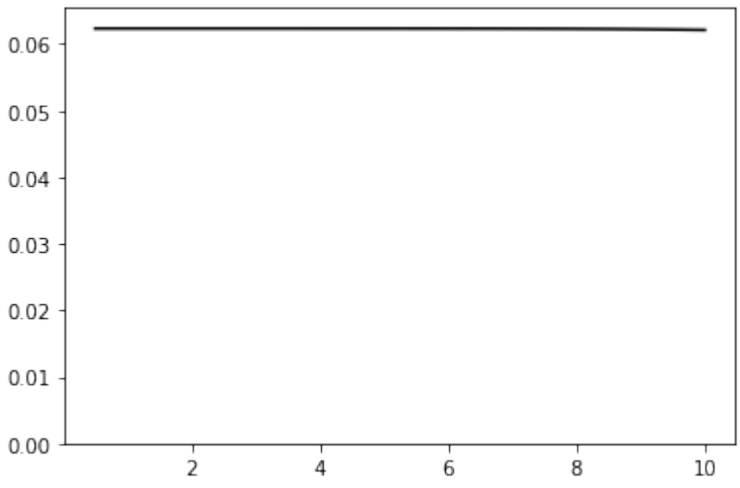
```
<Figure size 432x288 with 0 Axes>
```



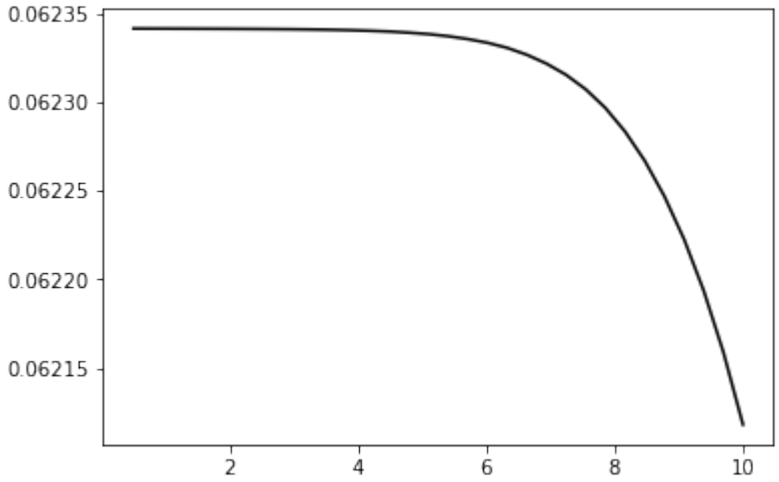
[5]:
```
intensity = np.abs(uxyz.u)**2
```

[6]:
```
total_energy = intensity.mean(axis=0).mean(axis=0)
```

[7]:
```
plt.plot(z0 / mm, total_energy, 'k')
plt.ylim(ymin=0, ymax=1.05 * total_energy.max())
```



[8]:
```
plt.plot(z0 / mm, total_energy, 'k')
```



[ ]:

8.6 Showing phase figures only at positions with high intensity

[1]:
```
from diffractio import um, np, degrees, mm
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
```

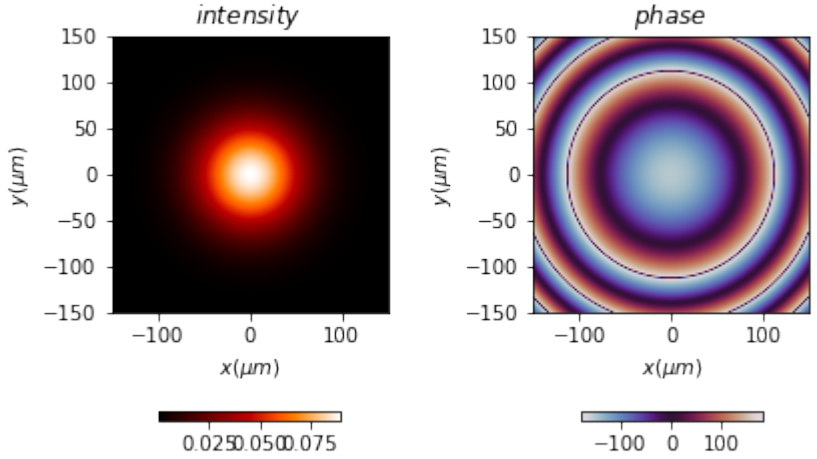
8.6.1 XY

[2]:
```
x = np.linspace(-150 * um, 150 * um, 1024)
y = np.linspace(-150 * um, 150 * um, 1024)
wavelength = 0.6238 * um

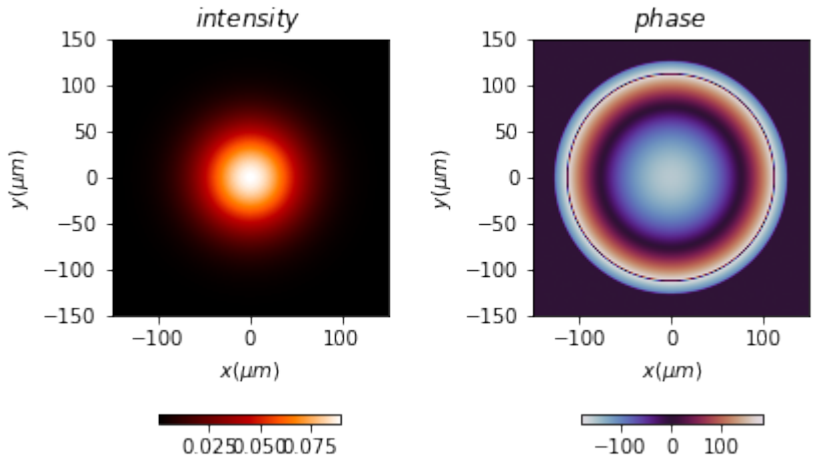
t1 = Scalar_source_XY(x, y, wavelength)
t1.gauss_beam(r0=(0, 0), w0=25 * um, z0=0*mm)
```

[3]:
```
t2 = t1.RS(z=10 * mm)
```

[4]:
```
t2.draw('field', percentage_intensity=0)
```



[5]:
```
t2.draw('field', percentage_intensity=0.01)
```



8.6.2 XZ

[6]:
```
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[7]:
```
length = 50 * um
wavelength = .5 * um
x0 = np.linspace(-length / 2, length / 2, 512)
z0 = np.linspace(0 * um, 500 * um, 256)

x_ini = -250 * um * np.tan(30 * degrees)

# source
```

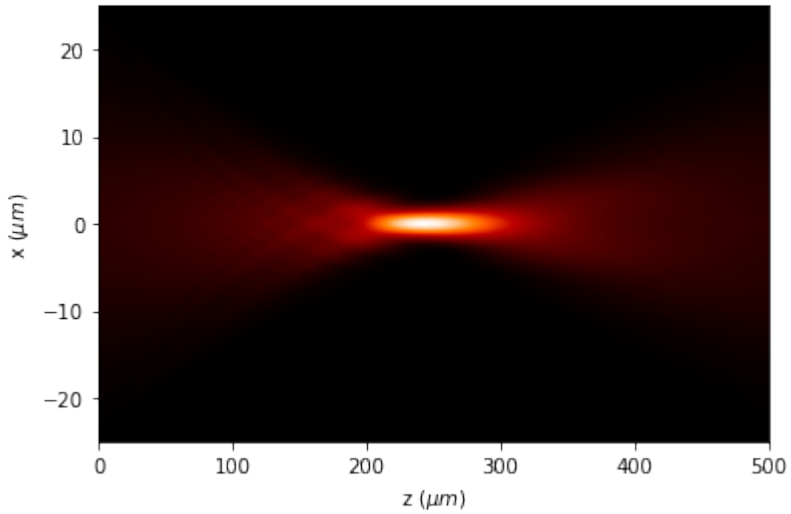
(continues on next page)

(continued from previous page)

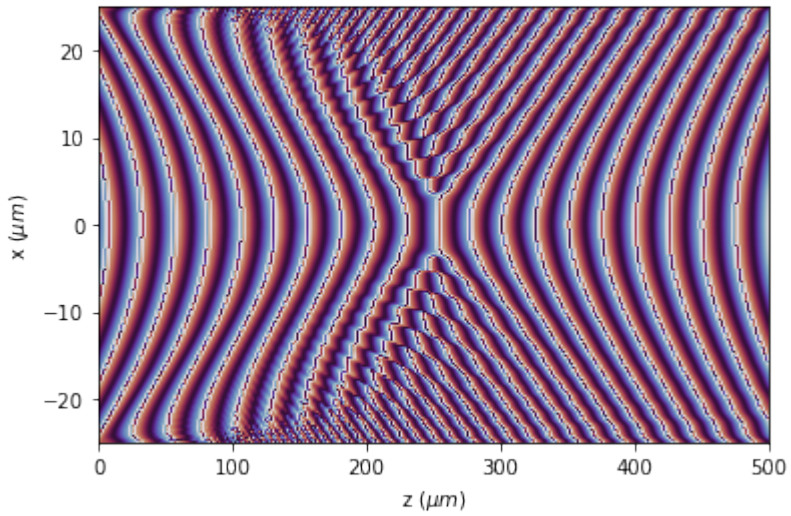
```
f1 = Scalar_source_X(x0, wavelength)
f1.gauss_beam(A=1, x0=0, z0=250 * um, w0=2 * um, theta=0 * degrees)

u1 = Scalar_field_XZ(x=x0, z=z0, wavelength=wavelength)
u1.clear_refractive_index()
u1.incident_field(f1)
u1.WPM(verbose=False)
```

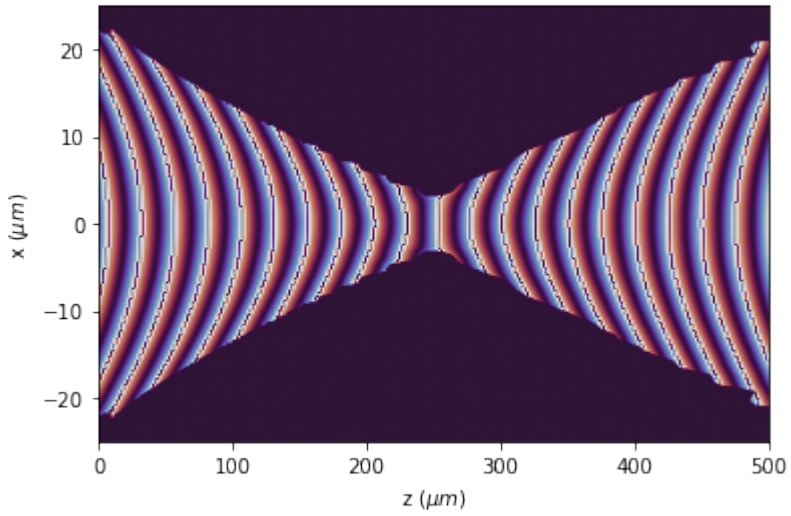
[8]:
```
u1.draw('intensity')
```



[9]:
```
u1.draw('phase', percentage_intensity=0)
```



[10]:
```
u1.draw('phase', percentage_intensity=0.01)
```



8.7 Quality factor

In RS propagation, the algorithm provides information about the quality in the simulation, as described in F. Shen and A. Wang, "Fast-Fourier-transform based numerical integration method for the Rayleigh-Sommerfeld diffraction formula," Appl. Opt., vol. 45, no. 6, pp. 1102–1110, 2006.

[1]:
```
%load_ext autoreload
%autoreload 2
```

8.7.1 XY Scheme

[2]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_masks_XY import Scalar_mask_XY

from diffractio.scalar_fields_XY import get_RS_minimum_z, quality_factor
```

[3]:
```
wavelength = 0.5 * um
num_x = 512
num_y = 512
range_x=1*mm
range_y=1*mm
x0 = np.linspace(-range_x / 2, range_x / 2, num_x)
y0 = np.linspace(-range_y / 2, range_y / 2, num_y)
```

To check the minimum z distance where the quality is good we have the get_RS_minimum_z function.

[4]:
```
z_min = get_RS_minimum_z(range_x,
                         range_y,
```

(continues on next page)

(continued from previous page)

```
                          num_x,
                          num_y,
                          wavelength,
                          n=1,
                          quality=1,
                          verbose=True)
```

```
z min = 3845.47
```

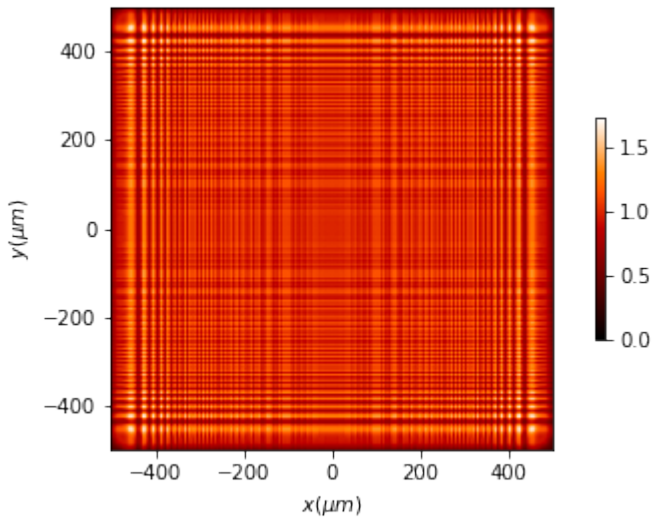
[5]:
```
quality = quality_factor(range_x,
                          range_y,
                          num_x,
                          num_y,
                          z_min,
                          wavelength,
                          n=1,
                          verbose=True)
```

```
Quality factor = 1.05
```

Let us check that the simulation is OK

[6]:
```
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.square(r0=(0,0), size=range_x)
u2 = t0.RS(z=z_min, new_field=True, verbose=True)
u2.draw(has_colorbar='vertical')
plt.clim(vmin=0)
```

```
Good result: factor 1.00
```



Function *get_RS_minimum_z* is also available inside the class:

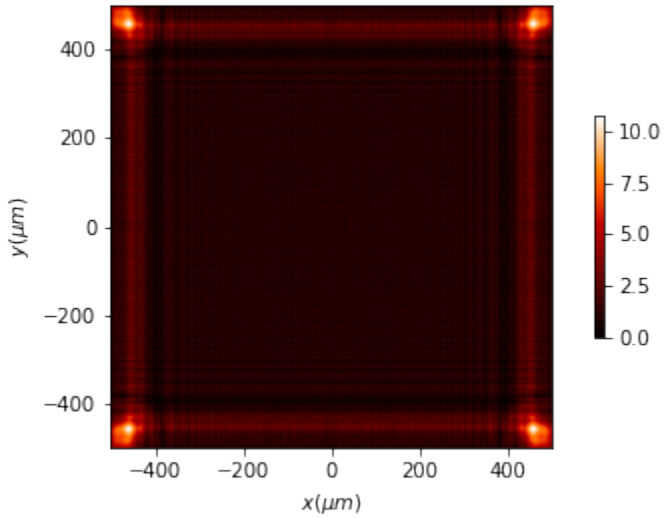
[7]:
```
z_min = t0.get_RS_minimum_z(verbose=True)
```

```
z min = 3.85 mm
```

Now, let us reduce the distance, to see if the simulation si working worse.

[8]:
```
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.square(r0=(0, 0), size=range_x)
u2 = t0.RS(z=0.9*z_min, new_field=True, verbose=True)
u2.draw(has_colorbar='vertical')
plt.clim(vmin=0)
```

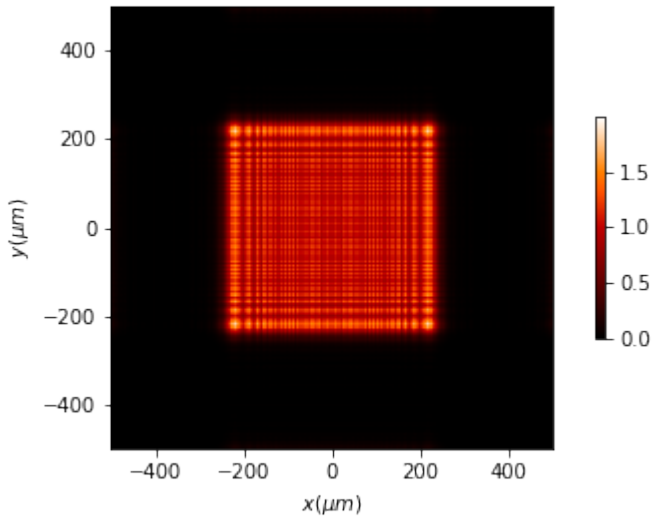
```
- Needs denser sampling: factor 0.90
```



The quality factor is computed considering the total area. When the mask is smaller that the whole area, good results can be obtained even with quality<1.

[9]:
```
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.square(r0=(0, 0), size=range_x/2)
u2 = t0.RS(z=0.75 * z_min, new_field=True, verbose=True)
u2.draw(has_colorbar='vertical')
plt.clim(vmin=0)
```

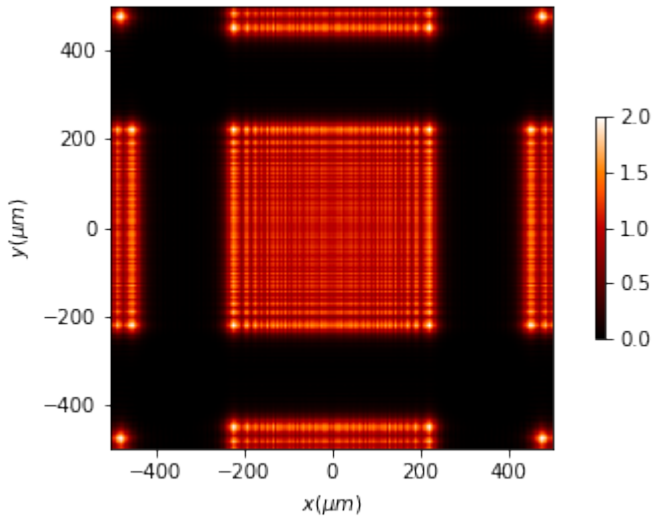
```
- Needs denser sampling: factor 0.76
```



But eventually, it starts to work badly:

[10]:
```
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength, info='t0')
t0.square(r0=(0, 0), size=range_x / 2)
u2 = t0.RS(z=0.66 * z_min, new_field=True, verbose=True)
u2.draw(has_colorbar='vertical')
plt.clim(vmin=0)
```

```
- Needs denser sampling: factor 0.67
```



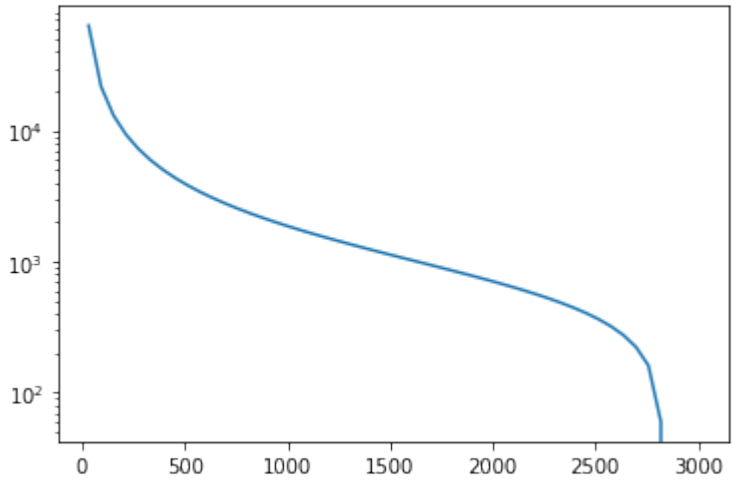
8.7.1.1 Distance in terms of the number of data

[11]:
```
wavelength = 0.5 * um
range_x=1*mm
range_y=1*mm

num_data=np.linspace(32,3000)
zs = np.zeros_like(num_data)
```

[12]:
```
for i, data in enumerate(num_data):
    z_min = get_RS_minimum_z(range_x,
                             range_y,
                             data,
                             data,
                             wavelength,
                             n=1,
                             quality=1,
                             verbose=False)
    zs[i]=z_min
```

[13]:
```
plt.semilogy(num_data, zs)
```



8.7.2 X Scheme

The quality factor is also implemented in the X scheme

[14]:
```
from diffractio import sp, nm, plt, np, mm, degrees, um
from diffractio.scalar_fields_X import Scalar_field_X
from diffractio.scalar_sources_X import Scalar_source_X
from diffractio.scalar_masks_X import Scalar_mask_X

from diffractio.scalar_fields_X import get_RS_minimum_z, quality_factor
```

[15]:
```
wavelength = .5 * um
num_x = 512
range_x = 1 * mm
x0 = np.linspace(-range_x / 2, range_x / 2, num_x)
```

[16]:
```
z_min = get_RS_minimum_z(range_x,
                         num_x,
                         wavelength,
                         n=1,
                         quality=1,
                         verbose=True)
```

```
z min = 3779.77
```

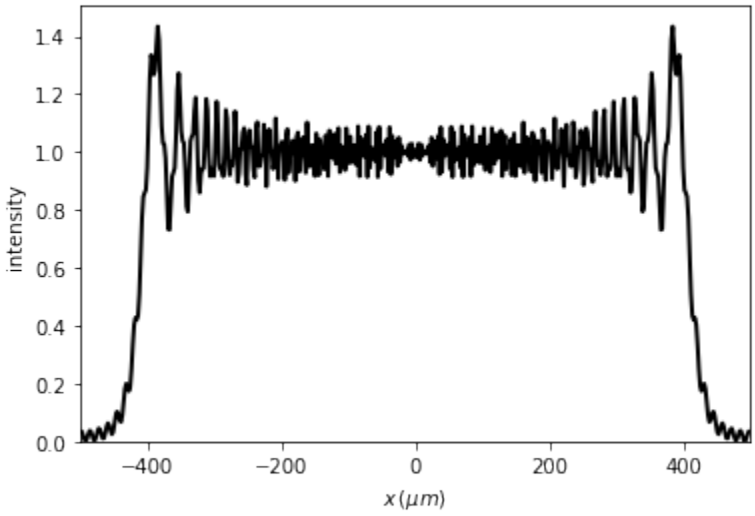
[17]:
```
t0 = Scalar_mask_X(x=x0, wavelength=wavelength, info='t0')
t0.slit(x0=0, size=0.85*mm)
```

[18]:
```
z_min = t0.get_RS_minimum_z()
```

```
z min = 3.78 mm
```

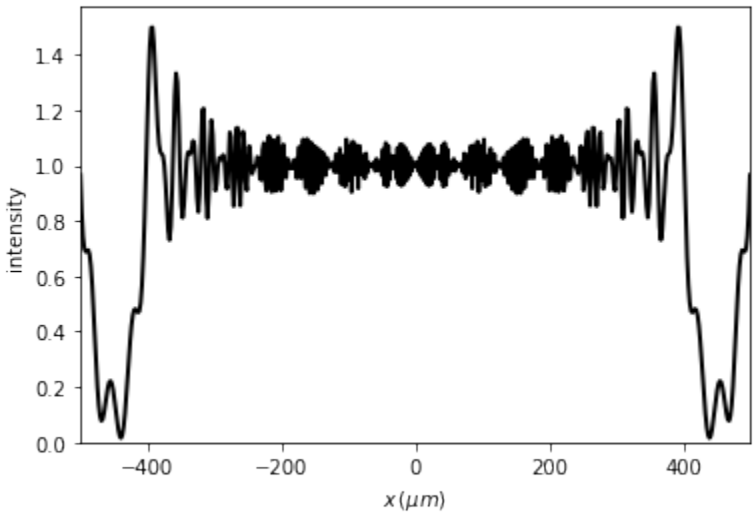
[19]:
```
u2 = t0.RS(z=z_min, new_field=True,  verbose=True)
u2.draw()
```

```
- Needs denser sampling: factor 0.97
```



[20]:
```
u2 = t0.RS(z=0.9*z_min, new_field=True, verbose=True)
u2.draw()
```

```
- Needs denser sampling: factor 0.88
```



8.8 WPM algorithm in X frame for big masks

Propagation algorithms for wave optics usually allows masks with a small size, since the propagation step is around the wavelength. When big masks, with long propagation distances is required, the involved matrices are huge and cannot be properly stored.

For this case, we have developed WPM function in X frame, where only a small quantity of data are stored:

- Field at the end of the simulation.
- Field at the focus, that is, at position z with the maximum concentration of intensity.
- Field at an axis.
- Field at a z axis.
- Global XZ field, with reduced sampling.
- XZ field, in a small region of interest.

For this case, we have implemented WPM in X and XY schemes. The refraction index in the XZ or XYZ space is not stored in a matrix, but a function is required.

[27]:
```
from diffractio import np, plt, sp, um, mm, degrees
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_XZ import Scalar_mask_XZ

from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_sources_X import Scalar_source_X
```

8.8.1 X scheme

[28]:
```
# Parameters

x = np.linspace(-3 * mm, 3 * mm, 1024 * 16)
z = np.linspace(-1 * mm, 21 * mm, 1024 * 8)
wavelength = 10 * um
```

[29]:
```
dx = x[1] - x[0]
dz = z[1] - z[0]

print("x = {:2.5f}, z = {:2.2f}".format(dx, dz))
print("x/wavelength = {:2.5f}, z/wavelength = {:2.2f}".format(
    dx / wavelength, dz / wavelength))
```

```
x = 0.36623, z = 2.69
x/wavelength = 0.03662, z/wavelength = 0.27
```

[30]:
```
u0 = Scalar_source_X(x, wavelength)
u0.plane_wave(A=1, theta=0.0, z0=0.0)
```

[31]:
```
def fn(x, z, wavelength):
    """refraction index definition for a given example.

    Args:
        x (np.array): x positions
        z (np.array): z positions
        wavelength (float): wavelength

    Returns:
        (np.array): XZ array with refraction index
    """

    u = Scalar_mask_XZ(x, z, wavelength)
    u.sphere(r0=(0, 6 * mm), radius=6 * mm, refractive_index=1.32, angle=0)
    return np.squeeze(u.n)
```

[32]:
```
# This would be the standard code, defining huge matrices for refraction index and u␣
↪field.

if False:
    u_3d = Scalar_mask_XZ(x, z, wavelength)
    u_3d.n = fn(x, z, wavelength)
    u_3d.draw_refractive_index(scale='scaled')
    u_3d.incident_field(u0)
    u_3d.WPM(has_edges=False, matrix=False, verbose=True)
    u_3d.draw(logarithm=True, draw_borders=True, scale='scaled')
```

[33]:
```
# Region of interest (ROI)

xout_roi = np.linspace(-500 * um, 500 * um, 256)
zout_roi = np.linspace(15 * mm, 21 * mm, 256)
ROI = (xout_roi, zout_roi)
```

[34]:
```
# Execution

u_final_x, u_out_gv, u_out_roi, u_axis_x, u_axis_z, u_max, z_max = u0.WPM(
    fn,
    z,
    ROI=ROI,
    num_sampling=(1024, 1024),
    x_pos=0. * um,
    z_pos=12 * mm,
    get_u_max=True,
    has_edges=False,
    verbose=True)
```

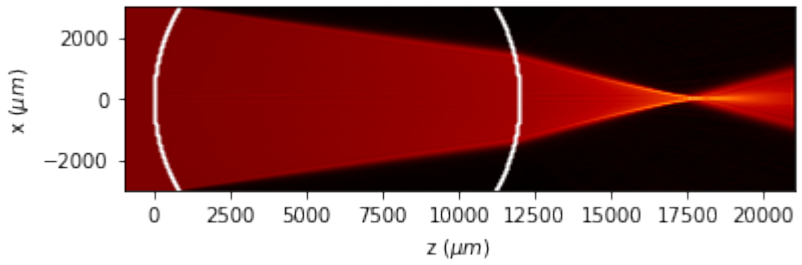
```
Time = 39.93 s, time/loop = 4.874 ms
```

8.8.2 Output data

[35]:
```
# Global region

print(u_out_gv)
u_out_gv.draw(logarithm=1e1, draw_borders=True, scale='scaled')
```

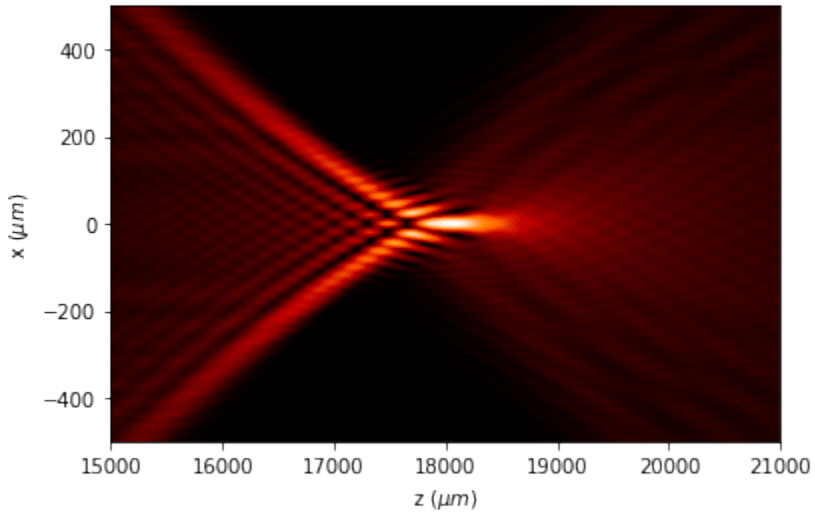
```
Scalar_mask_XZ
 - x:  (1024,),   z:  (1024,),   u:  (1024, 1024)
 - xmin:       -3000.00 um,  xmax:       3000.00 um,  Dx:   5.87 um
 - zmin:       -1000.00 um,  zmax:       21000.00 um,  Dz:   21.51 um
 - Imin:       0.00,     Imax:      139.19
 - phase_min:  -180.00 deg, phase_max: 180.00 deg
 - wavelength: 10.00 um
 - date:       2022-11-20_18_17_26
 - info:       from WPM_no_storage_1D
```



[36]:
```
# Region of interest

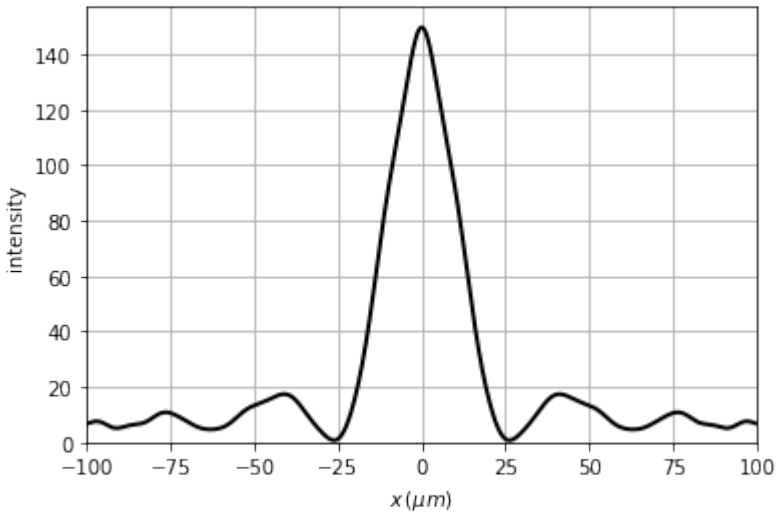
print(u_out_roi)
u_out_roi.draw(logarithm=1e-1)
```

```
Scalar_mask_XZ
 - x:  (256,),   z:  (256,),   u:  (256, 256)
 - xmin:       -500.00 um,  xmax:      500.00 um,  Dx:   3.92 um
 - zmin:       15000.00 um,  zmax:      21000.00 um,  Dz:   23.53 um
 - Imin:       0.00,     Imax:      142.92
 - phase_min:  -179.99 deg, phase_max: 180.00 deg
 - wavelength: 10.00 um
 - date:       2022-11-20_18_17_27
 - info:       from WPM_no_storage_1D
```



[37]:
```
# Field at focus

u_max.draw()
plt.xlim(-10 * wavelength, 10 * wavelength)
plt.ylim(ymin=0)
plt.grid('on')
```



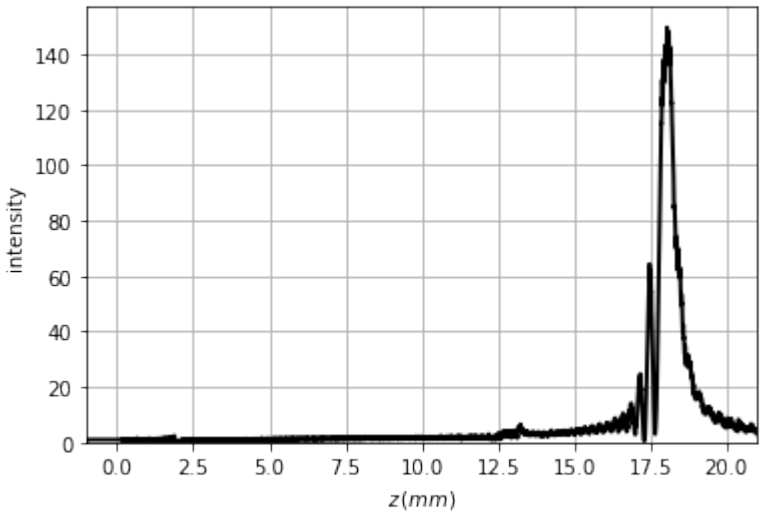
[38]:
```
# Field at a x given axis

print("pos focus = {:2.3f} mm".format(z_max / mm))

u_axis_x.draw(z_scale='mm')

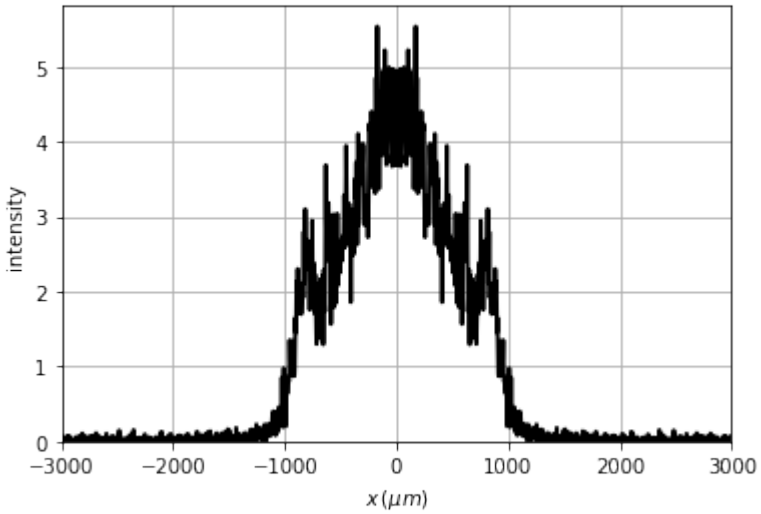
plt.ylim(ymin=0)
plt.xlim(z[0] / mm, z[-1] / mm)
plt.grid('on')
```

```
pos focus = 18.051 mm
```



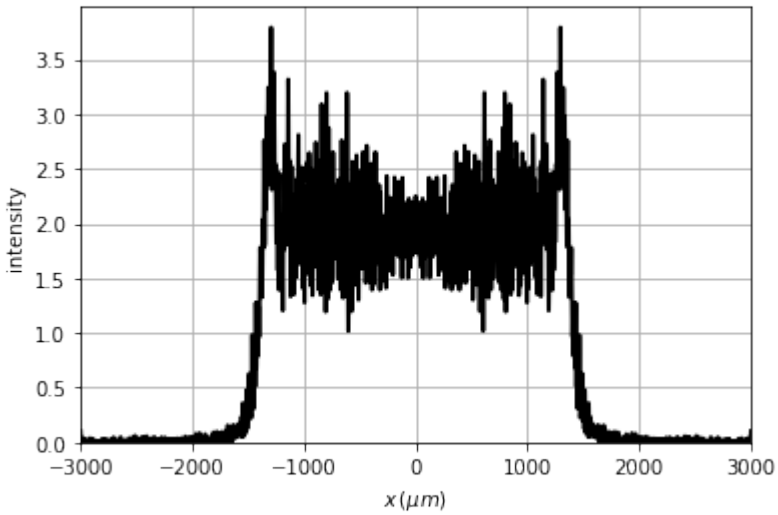
[39]:
```
# Final field

u_final_x.draw()
plt.ylim(ymin=0)
plt.grid('on')
```



[40]:
```
# Field at a z distance

u_axis_z.draw()
plt.grid('on')
```



8.8.3 XZ Scheme

[41]:
```
from diffractio import np, plt, sp, um, mm, degrees
from diffractio.scalar_fields_XYZ import Scalar_field_XYZ

from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
from diffractio.scalar_fields_XY import Scalar_field_XY
from diffractio.scalar_masks_XYZ import Scalar_mask_XYZ
```

[42]:
```
# Parameters

num_z = 256
x = np.linspace(-50 * um, 50 * um, 512)
y = np.linspace(-50 * um, 50 * um, 512)
z = np.linspace(-10, 150 * um, num_z)
wavelength = 2 * um

dx = x[1] - x[0]
dy = y[1] - y[0]
dz = z[1] - z[0]

print("x = {:2.5f}, x = {:2.5f}, z = {:2.2f}".format(dx, dy, dz))
print("x/l = {:2.2f}, y/l = {:2.2f}, z/l = {:2.2f}".format(
    dx / wavelength, dy / wavelength, dz / wavelength))
```

```
x = 0.19569, x = 0.19569, z = 0.63
x/l = 0.10, y/l = 0.10, z/l = 0.31
```

[43]:
```
u0 = Scalar_source_XY(x, y, wavelength)
u0.plane_wave(A=1)
```

[44]:
```
def fn(x, y, z, wavelegth):
    u = Scalar_mask_XYZ(x, y, z, wavelength)
    u.sphere(r0=(0, 0, 50),
             radius=40 * um,
             refractive_index=1.33,
             angles=(0, 0, 0))
    index = u.n
    del u
    return np.squeeze(index)
```

[45]:
```
xout_roi = np.linspace(-20 * um, 20 * um, 128)
yout_roi = np.linspace(-20 * um, 20 * um, 128)
zout_roi = np.linspace(90 * um, 140 * um, 64)
ROI = (xout_roi, yout_roi, zout_roi)

num_sampling = (128, 128, 64)
```

[46]:
```
z_pos = 0.1 * mm
```

[47]:
```
# Execution

u_final_x, u_out_gv, u_out_roi, u_axis_x, u_axis_z, u_max, z_max = u0.WPM(
    fn,
    zs=z,
    ROI=ROI,
```

(continues on next page)

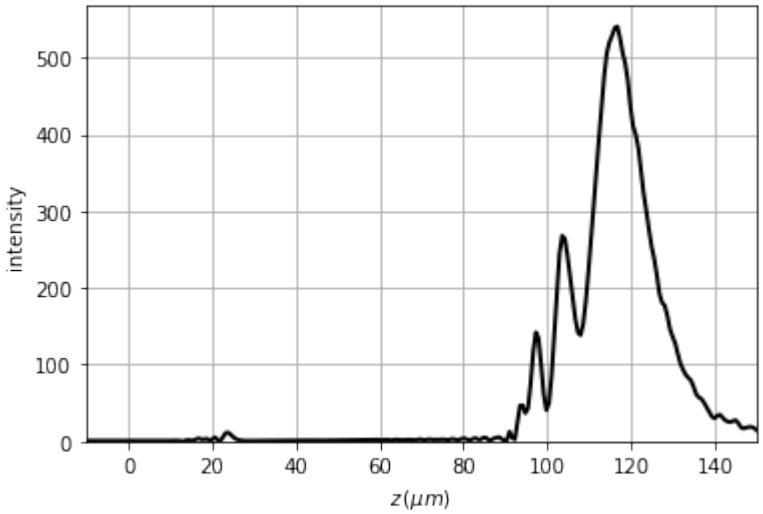
(continued from previous page)

```
    num_sampling=num_sampling,
    r_pos=(0. * um, 0. * um),
    z_pos=z_pos,
    get_u_max=True,
    has_edges=False,
    verbose=True)
```

```
Time = 25.73 s, time/loop = 100.5 ms
```

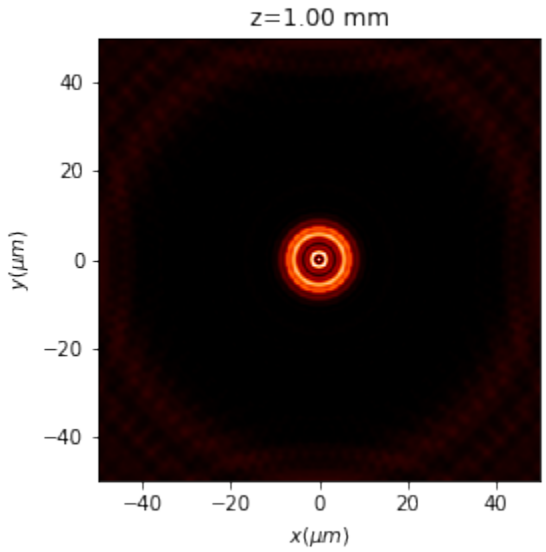
[48]:
```
### Field at axis

u_axis_x.draw()
plt.grid('on')
```

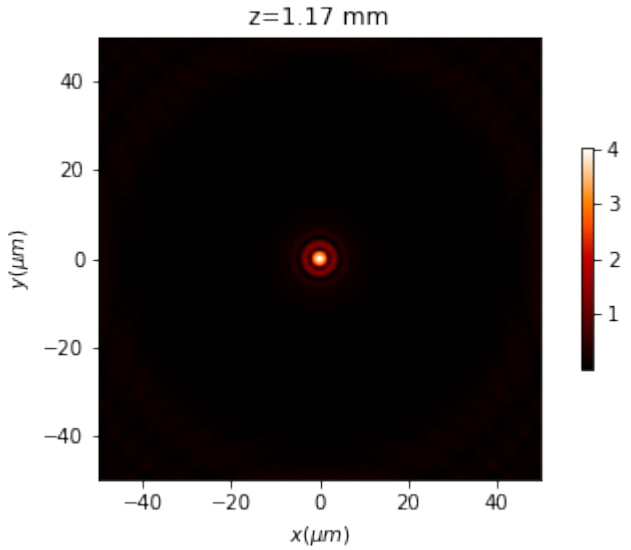


[49]:
```
# Field at z distance

u_axis_z.draw(logarithm=1e-1)
plt.title("z={:2.2f} mm".format(z_pos / 100))
```

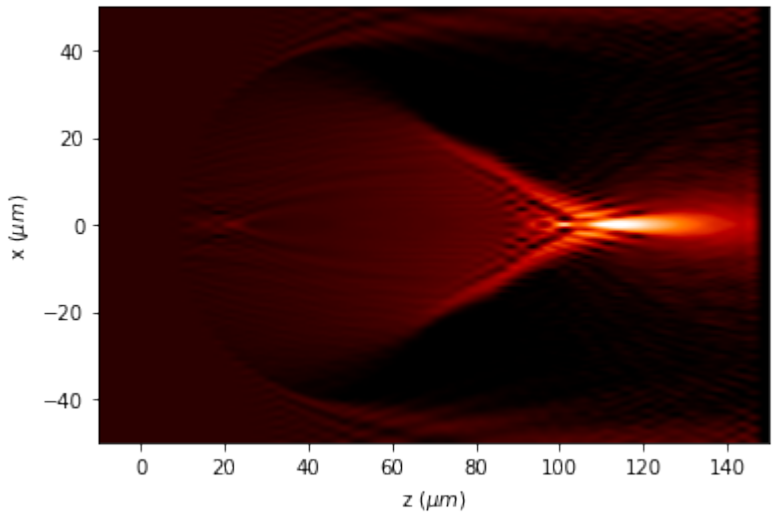


[50]:
```
# Field at focus
u_max.draw(logarithm=1e-1, has_colorbar='vertical')
plt.title("z={:2.2f} mm".format(z_max / 100))
```



[51]:
```
#Global view
uxz_gv = u_out_gv.draw_XZ(y0=0, logarithm=1e0)
```

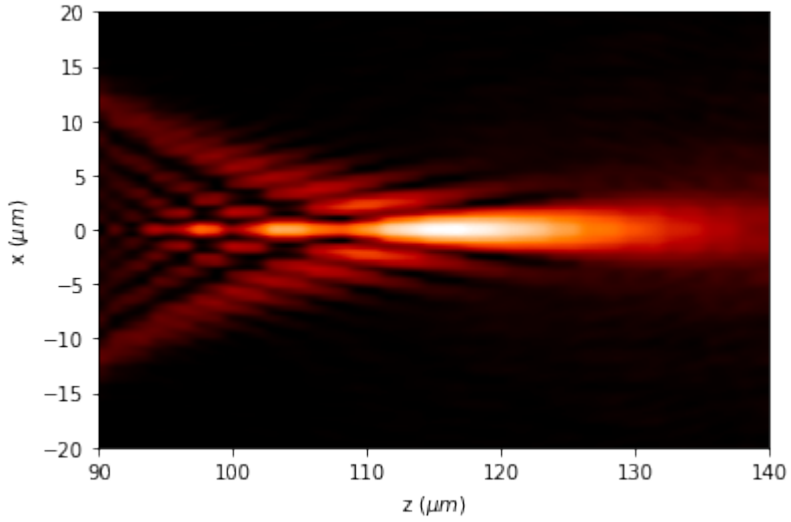
```
<Figure size 432x288 with 0 Axes>
```



[52]:
```
# Region of interest

u_out_roi.draw_XZ(logarithm=1e-1)
```

```
<Figure size 432x288 with 0 Axes>
```



8.9 Has_edges parameter in WPM and BPM

This parameter is used to avoid artifacts when propagatin light.

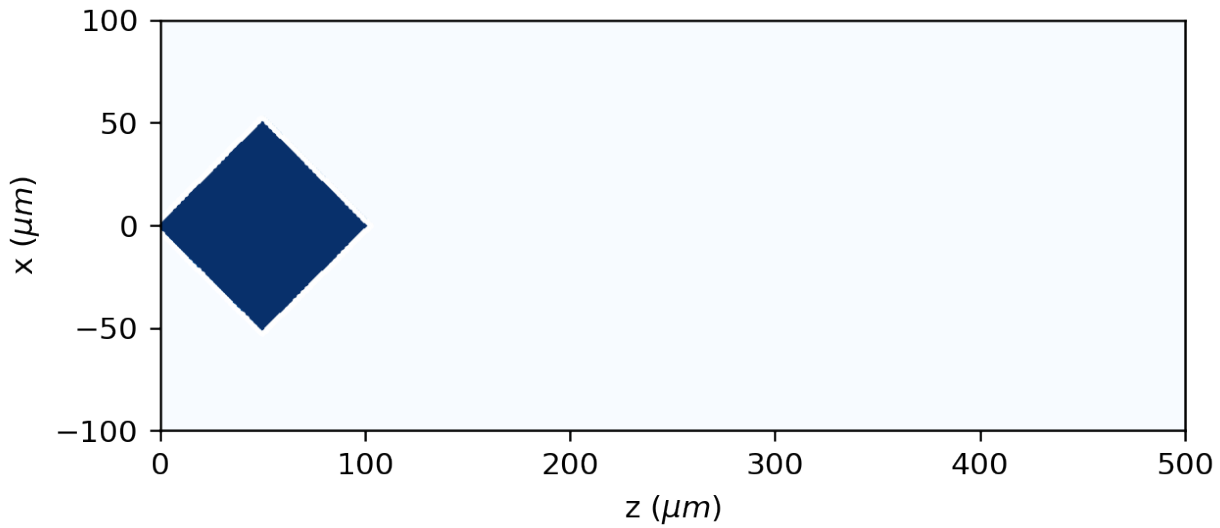
- When has_edges = False, the algoritm is similar to a period grating, as light exiting by the edges enters by the other size.
- When has_edges = True, light cannot enter again. However, when an incident field is a plane wave, it is masked by the x or xy frame acting as a slit - rectangle.
- With an array has_edges=z_edges, then this functioning depends on the z position.

[86]:
```
from diffractio import degrees,  um
from diffractio import np
from diffractio.scalar_fields_XZ import Scalar_field_XZ
from diffractio.scalar_masks_X import Scalar_mask_X
from diffractio.scalar_masks_XZ import Scalar_mask_XZ
from diffractio.scalar_sources_X import Scalar_source_X
```

[87]:
```
x0 = np.linspace(-100 * um, 100 * um, 512)
z0 = np.linspace(0 * um, 500 * um, 512)
wavelength = 5 * um

u0 = Scalar_source_X(x=x0, wavelength=wavelength)
u0.plane_wave(A=1, theta=0 * degrees)
```

[88]:
```
u1 = Scalar_mask_XZ(x=x0, z=z0, wavelength=wavelength)
u1.incident_field(u0)
u1.rectangle(r0=(0 * um, 50 * um),
             size=(75 * um, 75 * um),
             angle=45 * degrees,
             refractive_index=1.5)
u1.draw_refractive_index(scale='scaled')
```



[89]:
```
edges = u1.surface_detection()
```

has_edges = False

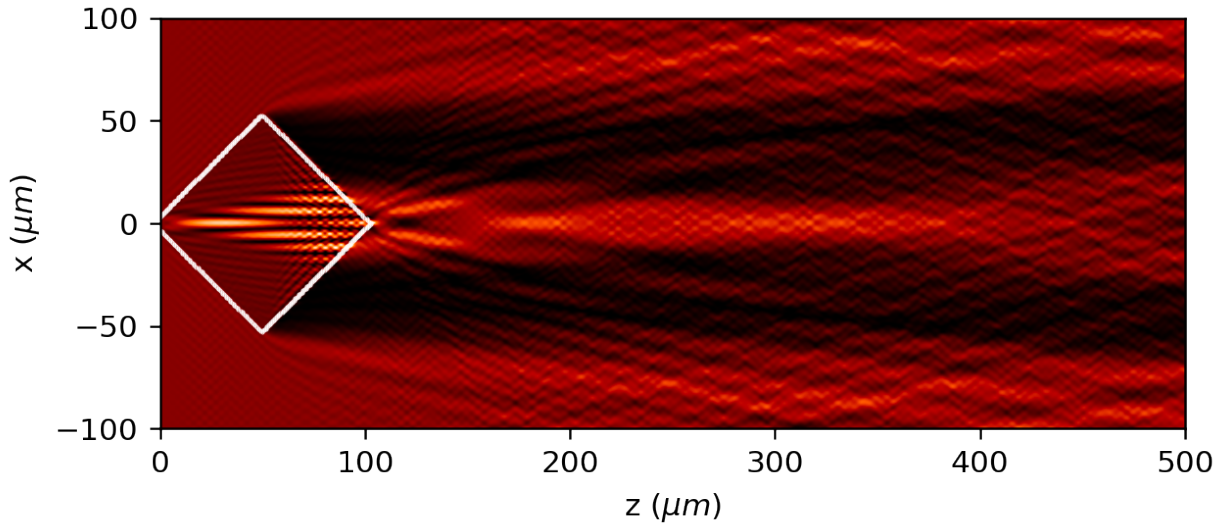
Light enteres again. Ripples in the light distribution are produced

[91]:
```
u1.clear_field()
u1.WPM(verbose=False, has_edges=False)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges,
```

(continues on next page)

(continued from previous page)

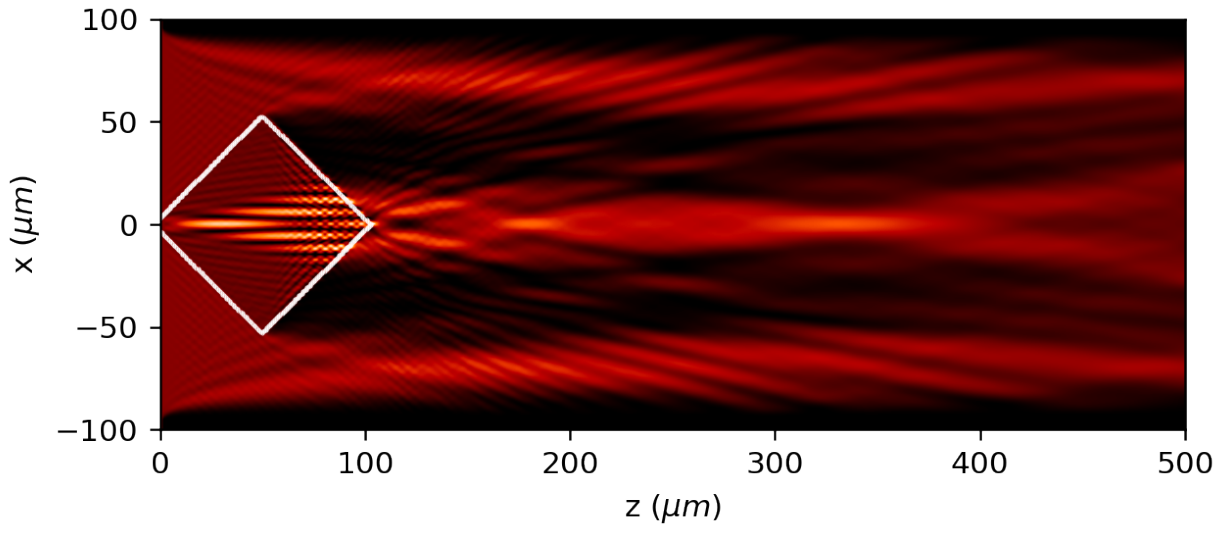
```
        scale='scaled')
```



has_edges = True

Light does not enter again, but the incident field is not a plane wave, but a slit.

[95]:
```
u1.clear_field()
u1.WPM(verbose=False, has_edges=True)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges,
        scale='scaled')
```



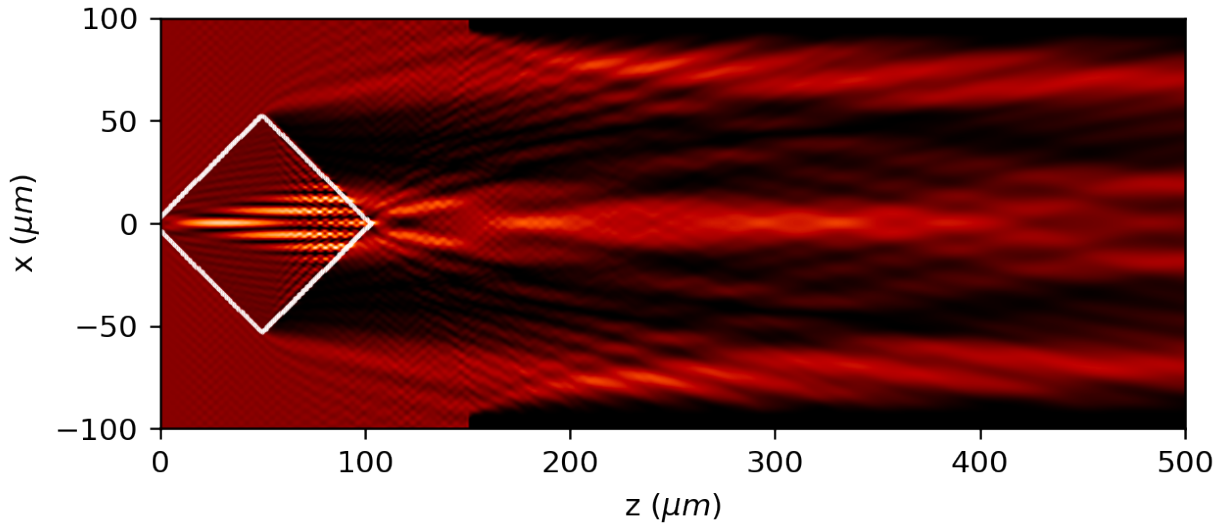
has_edges = np.array

Both effects are avoided. Light is like a plane wave, and lateral light does not enter when diffracted. Ripples are

reduced.

[ ]:
```
z_edges = np.zeros_like(z0)
z_edges[z0>150]= 1
```

[93]:
```
u1.clear_field()
u1.WPM(verbose=False, has_edges=z_edges)
u1.draw(logarithm=True,
        normalize='maximum',
        draw_borders=True,
        edge_matrix=edges,
        scale='scaled')
```



[ ]:
```

```

CHAPTER

NINE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

9.1 Types of Contributions

9.1.1 Report Bugs

Report bugs at https://bitbucket.com/optbrea/diffractio/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

9.1.2 Fix Bugs

Look through the Bitbucket issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

9.1.3 Implement Features

Look through the Bitbucket issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

9.1.4 Write Documentation

Python diffraction and interference could always use more documentation, whether as part of the official Python diffraction and interference docs, in docstrings, or even on the web in blog posts, articles, and such.

9.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://bitbucket.com/optbrea/diffractio/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

9.2 Get Started!

Ready to contribute? Here's how to set up *diffractio* for local development.

1. Fork the *diffractio* repo on Bitbucket.
2. Clone your fork locally:

```
$ git clone git@bitbucket.org:optbrea/diffractio.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv diffractio
$ cd diffractio/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 diffractio tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the Bitbucket website.

9.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/optbrea/diffractio/pull_requests and make sure that the tests pass for all supported Python versions.

9.4 Tips

To run a subset of tests:

```
$ py.test tests.test_diffractio
```

9.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

TEN

CREDITS

10.1 Development Lead

- Luis Miguel Sanchez Brea <optbrea@ucm.es>

  Universidad Complutense de Madrid, Faculty of Physical Sciences, Department of Optics Plaza de las ciencias 1, ES-28040 Madrid (Spain)

10.2 Contributors

- Ángela Soria Garcia
- Jesús del Hoyo Muñoz
- Francisco Javier Torcal-Milla

CHAPTER

ELEVEN

RESULTS OBTAINED WITH DIFFRACTIO

11.1 Scientific articles

- F.J. Torcal-Milla, L.M. Sanchez-Brea, J.A. Gomez-Pedrero "Sector-based Fresnel zone plate with extended depth of focus" Optics & Laser Technology, Vol 154, (2022) 108294 https://doi.org/10.1016/j.optlastec.2022.108294.
- F.J. Torcal-Milla, L.M. Sanchez-Brea "Near field diffraction of steel tape gratings illuminated with finite-size incoherent sources" Optik, Vol 251, 2022, 168326, https://doi.org/10.1016/j.ijleo.2021.168326.
- Tang, Cheng, and Jianping Li. "Quasi-non-diffracting static light sheets generated by holistically optimized pupil masks." Optics Letters 47.7 1737-1740 (2022).
- Vishniakou, Ivan, and Johannes D. Seelig. "Differentiable model-based adaptive optics for two-photon microscopy." Optics Express 29.14 21418-21427 (2021).
- Khasanov, I. S. & Zykova, L. A. "Terahertz ghost imaging and surface plasmon resonance microscopy : quality," Proc. SPIE 11582, Fourth International Conference on Terahertz and Microwave Radiation: Generation, Detection, and Applications, 1158215, (2020).
- Vishniakou, I. & Seelig, J. D. Differentiable model-based adaptive optics with transmitted and reflected light. Opt. Express 28, 26436 (2020). https://www.osapublishing.org/oe/abstract.cfm?uri=oe-28-18-26436
- Vishniakou, I. & Seelig, J. D. Wavefront correction for adaptive optics with reflected light and deep neural networks. Opt. Express 28, 15459–15471 (2020). https://www.osapublishing.org/oe/abstract.cfm?uri=oe-28-10-15459
- Sanchez-Brea, L. M., Torcal-Milla, F. J., del Hoyo, J., Cuadrado, A. & Gomez-Pedrero, J. A. Optimization of angular diffractive lenses with extended depth of focus. J. Opt. 22, 065601 (2020). https://iopscience.iop.org/article/10.1088/2040-8986/ab8614
- Sanchez-Brea, L. M., Torcal-Milla, F. J. & Buencuerpo, J. Far-field diffraction of linear chirped gratings. Opt. Laser Technol. 107, 337–343 (2018). https://doi.org/10.1016/j.optlastec.2018.06.007
- Torcal-Milla, F. J. & Sanchez-Brea, L. M. Single-focus binary Fresnel zone plate. Opt. Laser Technol. 97, 316–320 (2017). http://dx.doi.org/10.1016/j.optlastec.2017.07.008

11.2 Congresses

- F.J. Torcal-Milla, L.M. Sanchez-Brea, J.A. Gomez-Pedrero, A. Cuadrado “Placa zonal del Fresnel binaria sectorial” XI Reunión española de optoelectŕonica (2019).
- L.M. Sanchez-Brea, J.A. Gómez Pedrero, F.J. Torcal-Milla, A. Cuadrado “Análisis difractivo del modelo de ojo”XII Reunión Nacional de Óptica (2018).

CHAPTER

TWELVE

HISTORY

12.1 0.0.0 (2017-01-01)

Pre-alpha state.

- I have been developing and using the module diffractio for at least 5 years for teaching and research. It was in python2 version and not completely clear.

12.2 0.0.1 (2019-02-09)

Pre-alpha state.

- Modules are passed to python3.

12.3 0.0.2 (2019-10-01)

Alpha state

- copyreg removed (previous not worked)
- change fft to scipy: "from scipy.fftpack import fft, ifft, fftshift"

  First release on PyPI in pre-alpha state.

12.4 0.0.5 (2019-10-06)

Alpha state

- Included vector (fields, sources, masks) modules, tests, and tutorial.

12.5 0.0.6 (2019-10-22)

Alpha state

- Finished first version of examples and tutorial.

12.6 0.0.7 and 0.0.8 (2020-05-02)

Alpha state

- convert_mask -> extrude_mask
- Improved documentation
- Implemented PWD
- Reduced size of html

12.7 0.0.9 and 0.0.10 (2020-05-02)

Alpha state

- Improvement to documentation (readthedocs)

12.8 0.0.11 and 0.0.12 (2021-05-09)

Alpha state

- Solved big errata in vector_paraxial_masks
- Solved errata in XY: kernelRS and kernelRSinverse
- Improved documentation, tutorial and examples of vector_paraxial
- Scalar_mask_XY: dots
- Scalar_mask_X: dots
- change save_data in all classes to simplify
- Changes in docs to include WPM
- Execution tests

12.9 0.0.13 (2021-08-11)

Alpha state

- wpm bpm 3d, wpm without storing, xy search focus
- fix bug 2D gratings
- fix bug aspheric X and new aspheric XY

12.10 0.1.0 (2022-10-12)

Beta state

- Fix bugs:
  - radial and azimuthal vector waves
  - Change in criterion of plane waves to Physics (ISO 80000-2:2019 convention): https://en.wikipedia.org/wiki/Spherical_coordinate_system#Coordinate_system_conversions
  - constant_wave -> constant_polarization in vector_sources_XY
  - mask_circle -> pupil
  -
- New vector_fields_XY:
  - vector_paraxial_fields -> vector_fields
- New schemes mainly for representation:
  - Scalar_field_Z
  - vector_X, vector_Z, vector_XZ, vector_XYZ
- New Scalar_mask_XY:
  - angular_aperture
  - edge_series
  - slit_series
  - rings
- New propagation algorithms:
  - WPM and BPM in 3D
  - WPM and BPM in 3D without storing intermediate planes
  - VFFT (vector FFT and inverse FFT)
  - VRS (vector Rayleigh-Sommerfeld propagation, with Ez field)
  - Chirped z-transform (CZT) for scalar fields
  - Chirped z-transform (CZT) for vector fields
- Other:
  - Pupil function in XY and vector XY
  - Remove mask parameter from lenses. This may produce incompatibilities with former code
  - Improving drawings

12.11 0.1.1 (2022-10-16)

- Vector fields are not longer paraxial.
  - The propagation algorithms implemented (VRS, VFFT and VCZT) provide :E_z: field. This allows to analyze longitudinal fields.The modules and classes elliminate changes their name. For example vector_paraxial_fields_X is now vector_fields_X.
- New propagation algorithm Chirped Z-Transform (CZT) is avaliable for X and XY fields.
  - This algorithms produce similar results to RS and VRS fields, but there are significant advantages:
  - The output field is not necessarily the same as the input field. This is important, for example, when analyzing the focus of a lens, since the computing time is dedicated to areas with light.
  - The output field does not necessarily have the same dimensionality of the input field. For example, when the mask is XY, we can have the data only at (x=0, y=0, z) axis.
  - Acceleration in computing and reduction of memory usage.
- New modules for visualization and data analysis are provided.
  - Scalar_field_Z can be used, for example, to analysis of intensity light at axis.
  - Vector_fields_Z, Vector_fields_XZ, and Vector_fields_XYZ have been developed, as VCZT algorithm can provide these data.
- Plane Wave Descomposition algorithm (PWD) is deprecated.
- Some importante bugs have been solved. For example the definition of the spherical coordinates in some sources (which not used standard physics criterion).
- Mask parameters is removed in some XY masks, as lenses, FPZ, etc. The new way to do this is the .pupil() function.
- Smooth_refraction index can be used also for Wave Propagation Method algorithm (WPM).

12.12 0.2.3 (2023-11-21)

- Improvement:
  - XYZ drawings is removed temporally.
  - utils_slicer.py is deprecated.
  -
- Fix bugs:
  - Bug in XYZ video
  - Blazed grating wrong defined.
  - bug in Scalar_field_XY.kernelRSinverse
- New features:
  - XY masks: new polygon, regular_polygon, star, and superformula functions.
  - XY masks: new circular_sector function.
  - XY masks: new lens_cyliindrical function.
  - X, XY, XYZ fields: new conjugate function.

  - WPM function without storing all the data.
  - X fields: inverse_amplitude, inverse_phase, pupil

12.13 0.3.1

- Improvement:
  - new fresnel_equations_kx, fresnel_equations, transmitances_reflectances_kx, transmitances_reflectances functions.

12.14 0.3.1

- Fix bugs:
  - new definition of X mask: fresnel_lens
- Refactoring:
  - refraction_index -> refractive_index ... in all words.
- New features:
  - XZ masks: New refractive_index_from_scalar_mask_XY method.
  - has_edges parameters in WPM and BPM methods can be an array masking in desired z positions.

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

diffractio, ??
diffractio.config, ??
diffractio.scalar_fields_X, ??
diffractio.scalar_fields_XY, ??
diffractio.scalar_fields_XYZ, ??
diffractio.scalar_fields_XZ, ??
diffractio.scalar_fields_Z, ??
diffractio.scalar_masks_X, ??
diffractio.scalar_masks_XYZ, ??
diffractio.scalar_masks_XZ, ??
diffractio.scalar_sources_X, ??
diffractio.scalar_sources_XY, ??
diffractio.utils_common, ??
diffractio.utils_drawing, ??
diffractio.utils_math, ??
diffractio.utils_multiprocessing, ??
diffractio.utils_optics, ??
diffractio.vector_fields_X, ??
diffractio.vector_fields_Z, ??