beampy

Release 1.11

Jonathan Peltier and Marcel Soubkovsky

Dec 31, 2020

CONTENTS

Beampy is a python module based on the Beam Propagation Method[1] used to compute light propagation into a varying refractive index. The light propagation is done by the bpm module. An user interface - done using Qt desginer - allows to control the parameters and display the results.

This project was initiate by Jonathan Peltier and Marcel Soubkovsky during a master university course from the PAIP master of the université de Lorraine, under the directive of Pr. Nicolas Fressengeas.

The bpm core codes are mainly based on a compilation of MatLab codes initialy developed by Régis Grasser during his PhD thesis[2], and later modified at the FEMTO-ST institute of the Université de Franche-Comté and at the LMOPS laboratory[3] of the Université de Lorraine.

[1] K. Okamoto, in Fundamentals of Optical Waveguides, 2nd ed., edited by K. Okamoto (Academic, Burlington, 2006), pp. 329–397.

[2] "Generation et propagation de reseaux periodiques de solitons spatiaux dans un milieu de kerr massif" PhD thesis, université de Franche-Comté 1998.

[3] H. Oukraou et. al., Broadband photonic transport between waveguides by adiabatic elimination Phys. Rev. A, 97 023811 (2018).

CHAPTER

ONE

REFERENCES

CHAPTER

TWO

LINKS

The online documentation can be found at https://beampy.readthedocs.io/.

The source code of the whole project can be found at https://github.com/Python-simulation/Beampy/.

The PyPI repository can be found at https://pypi.org/project/beampy/.

CHAPTER

THREE

INSTALLATION

This package can be download in a python environment using pip install:

```
pip install beampy
```

Or by downloading the github folder and setting beampy as a PYTHONPATH. If so, make sure to download Qt5, matplotlib and numpy by using this command in the docs folder:

```
pip install -r requirements.txt
```

CHAPTER

FOUR

STARTING THE SOFTWARE

To start the Beampy interface, import beampy and start the open_app function:

```
import beampy
beampy.open_app()
```

Or open direclty the user_interface.py file to launch the interface. Or even open the bpm.py to have a non-based interface version.

For more informations, see the `detailed user guide` describing Beampy and explaining the choices made (not updated and not finished).

CHAPTER

FIVE

OVERVIEW

The three modules making Beampy, the interface explanations, results and examples are accessible bellow:

5.1 Interface

This page will explain the interface functioning. For now it is just a collection of screenshots of the interface.

5.2 Examples

5.2.1 Main examples

beampy.examples code.

List of examples from the beampy.examples module.

beampy.examples.**benchmark_kerr**()
Kerr benchmark by looking at the critical power at which a beam become a soliton. Several test were done with this function and for now, it seems that the critical power find by the simulations is about 85% of the theorical value. Meaning a 15% error. Further tests are needed to understand the observed differences.

beampy.examples.**free_propag**(*dist_x=0.1*, *length_x=1000*, *length_z=10000*, *no=1*, *lo=1*)
Show the free propagation of a beam and compare Beampy results with the theorical values.

Parameters

- **dist_x** (*float*) – Step over x (µm).
- **length_x** (*float*) – Size of the compute window over x (µm).
- **length_z** (*float*) – Size of the compute window over z (µm).
- **no** (*float*) – Refractive index of the cladding
- **lo** (*float*) – Wavelength of the beam in vaccum (µm).

beampy.examples.**gaussian_beam**(*fwhm=10*)
Display a Gaussian beam at the given fwhm.

Parameters **fwhm** (*float*) – Full width at half maximum (for intensity not amplitude) (µm).

beampy.examples.**guides_x**()
Display a Gaussian guide, two super-Gaussian guides and a flat-top guide to illustrate the width definition.

beampy.examples.**guides_z**()
Display an array of guides and the curved guides system.

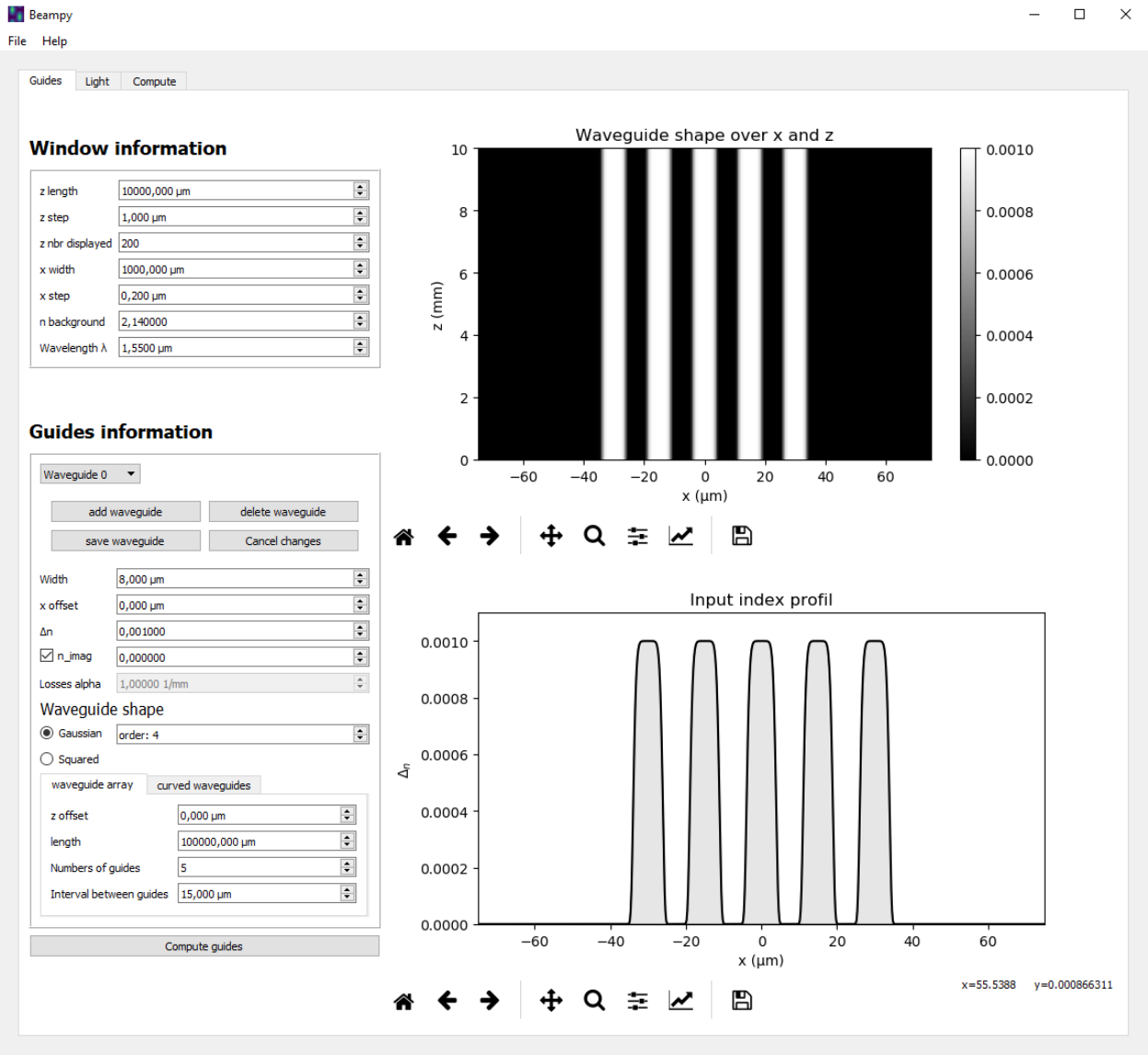

Fig. 1: Array of guides settings

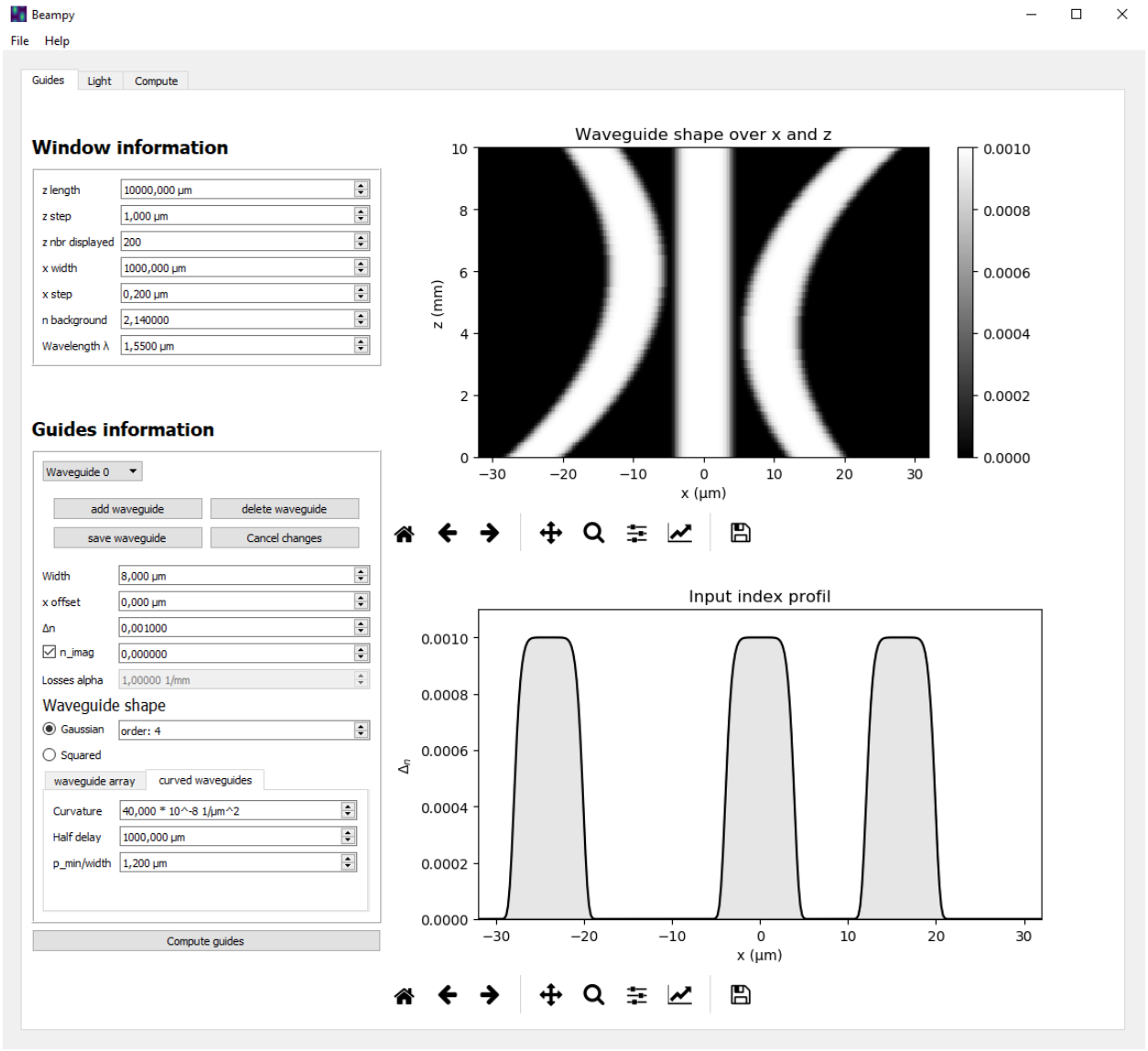

Fig. 2: Curved guides settings (STIRAP)

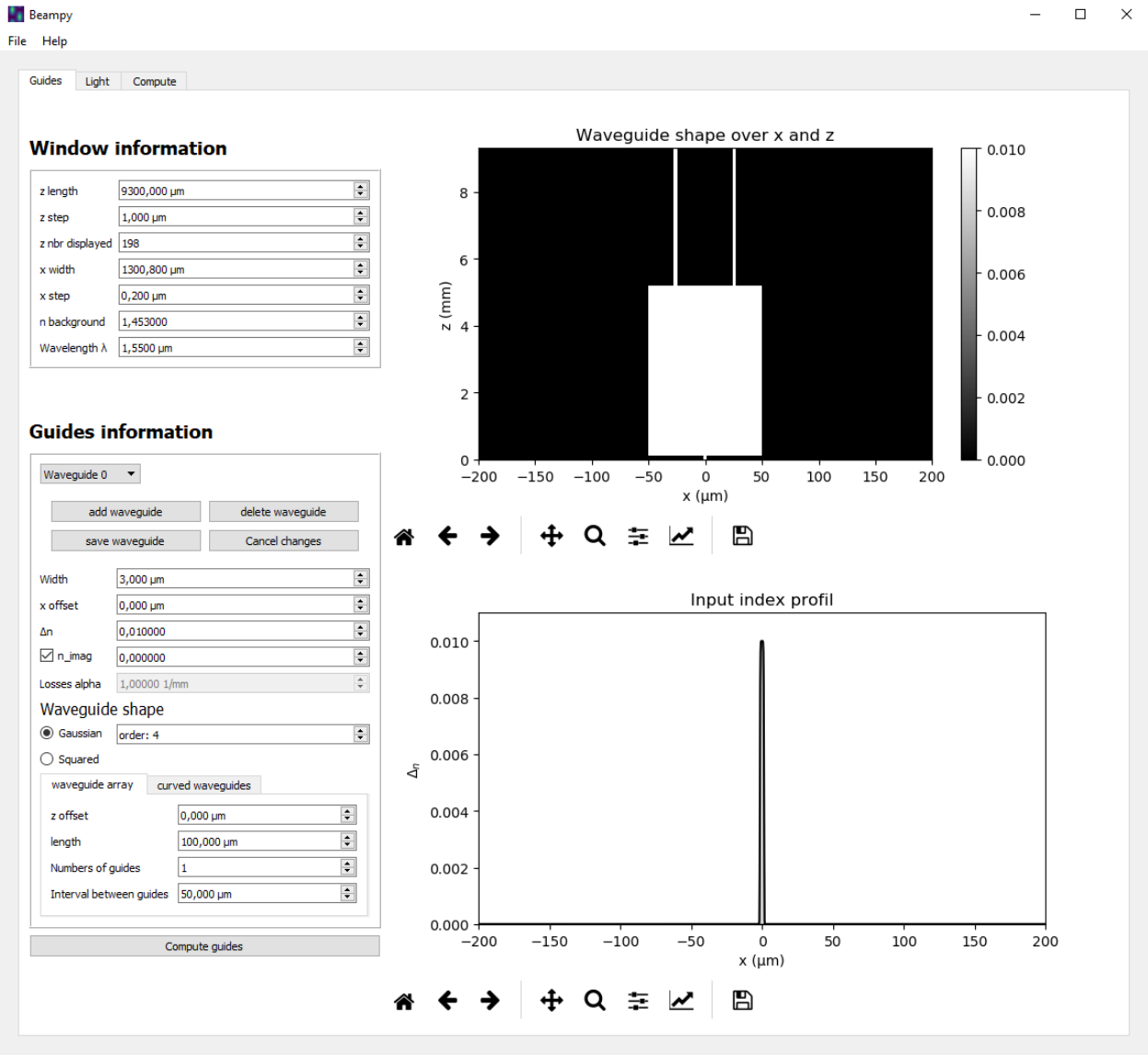

Fig. 3: Multimodal beam splitter made with the waveguide creation interface

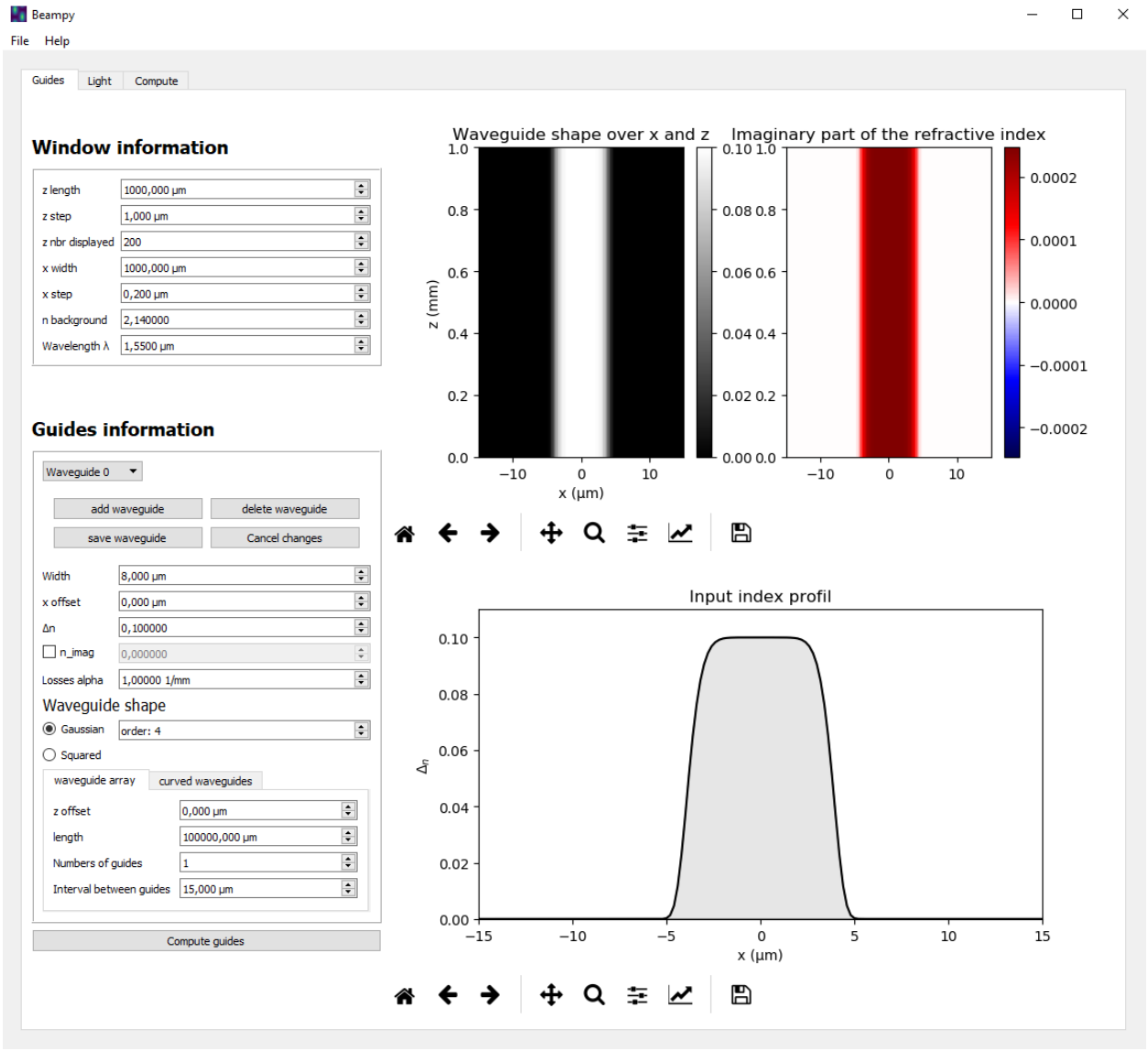

Fig. 4: Adding losses by specifying the imaginary part of the refractive index or by the losses factor alpha

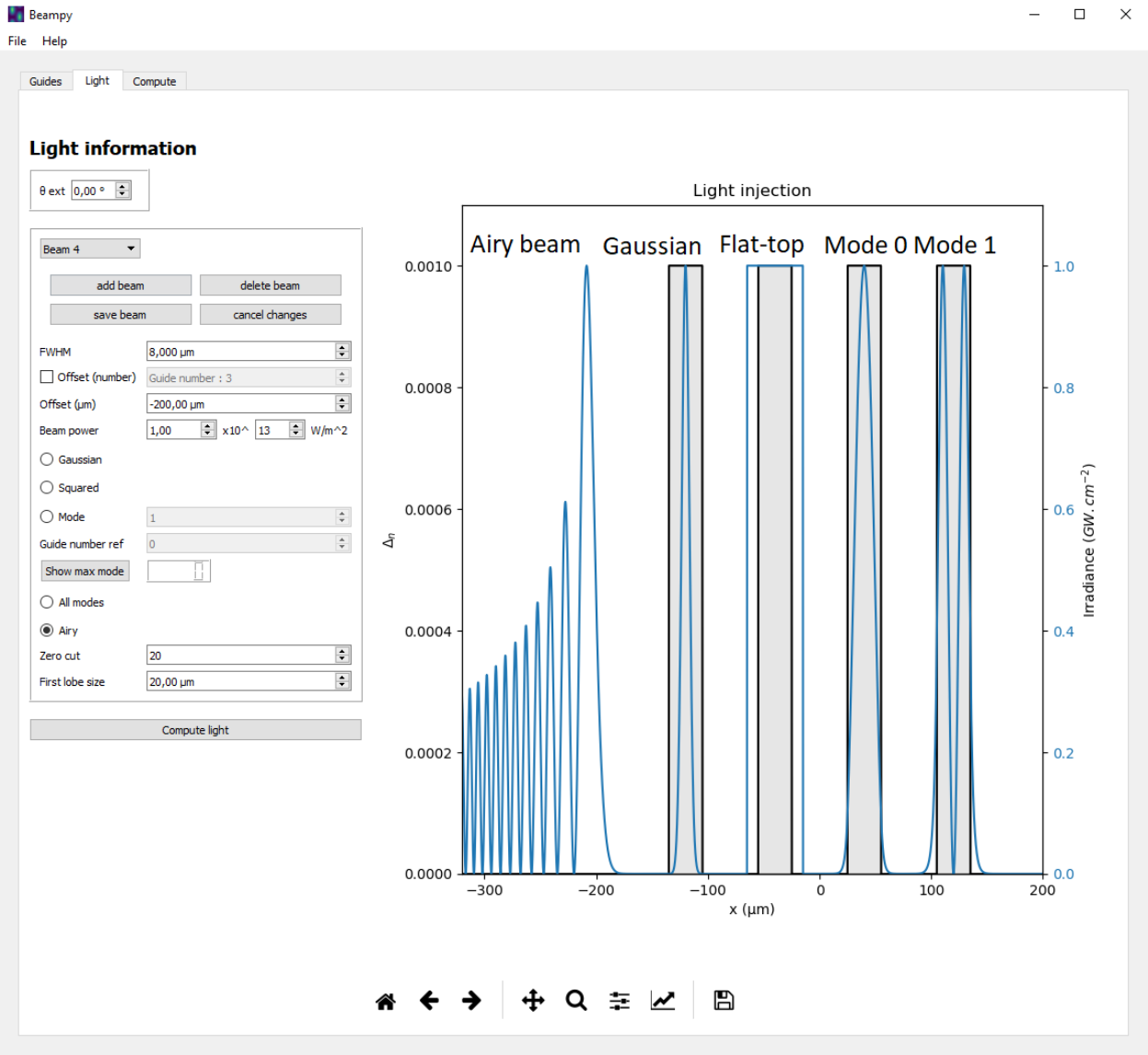

Fig. 5: Different Beams settings

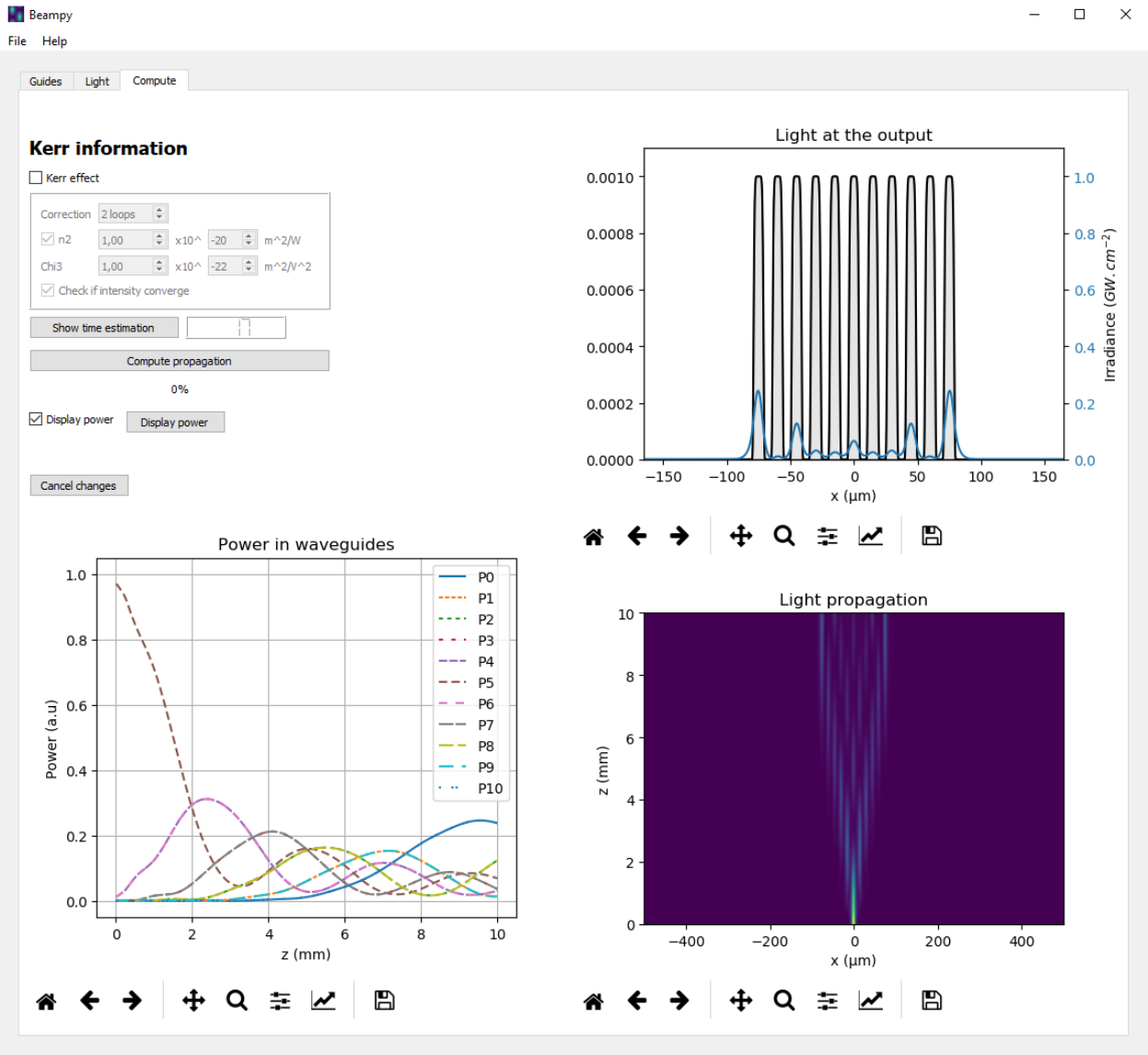

Fig. 6: Array of guides results

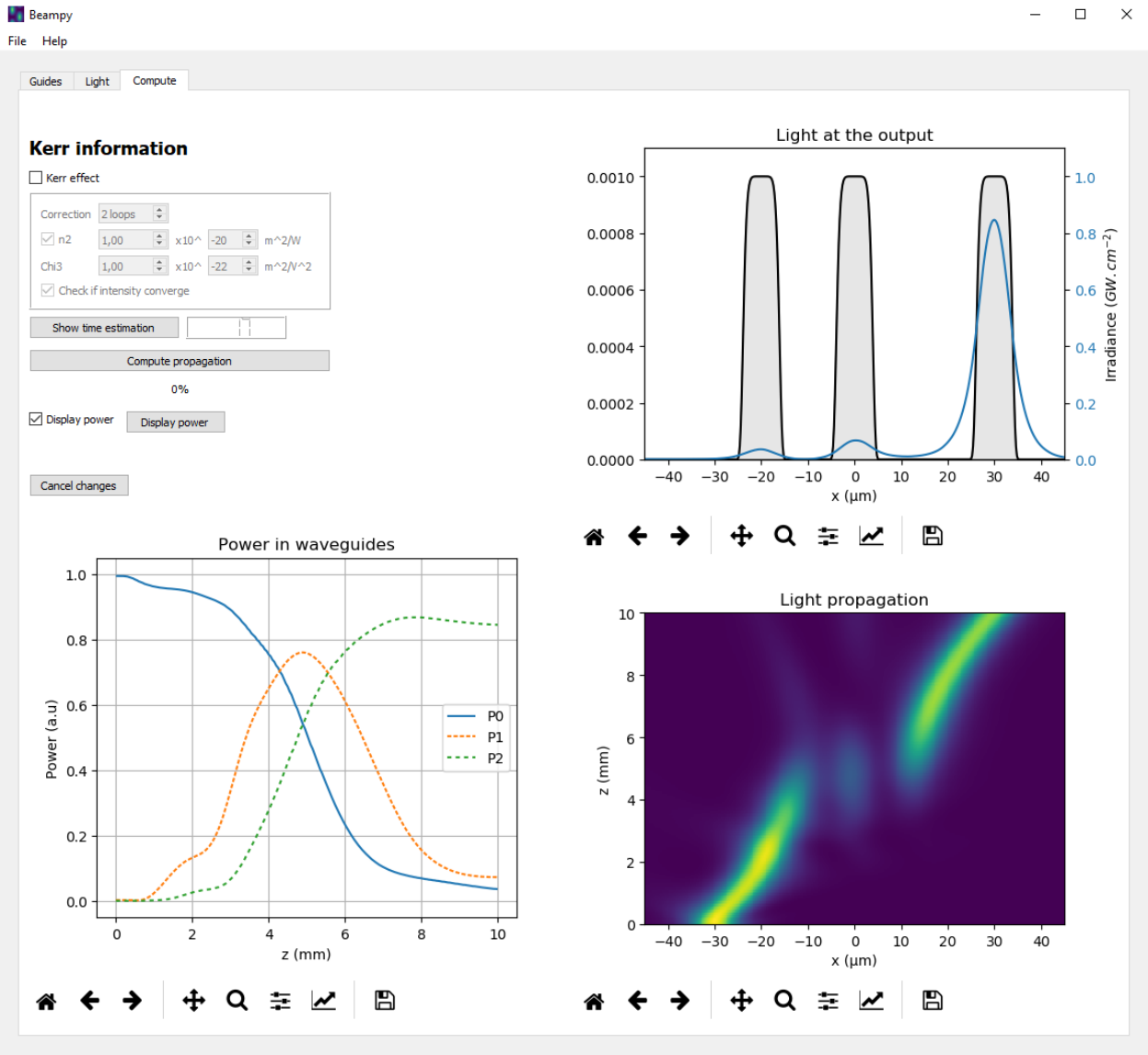

Fig. 7: Curved guides results

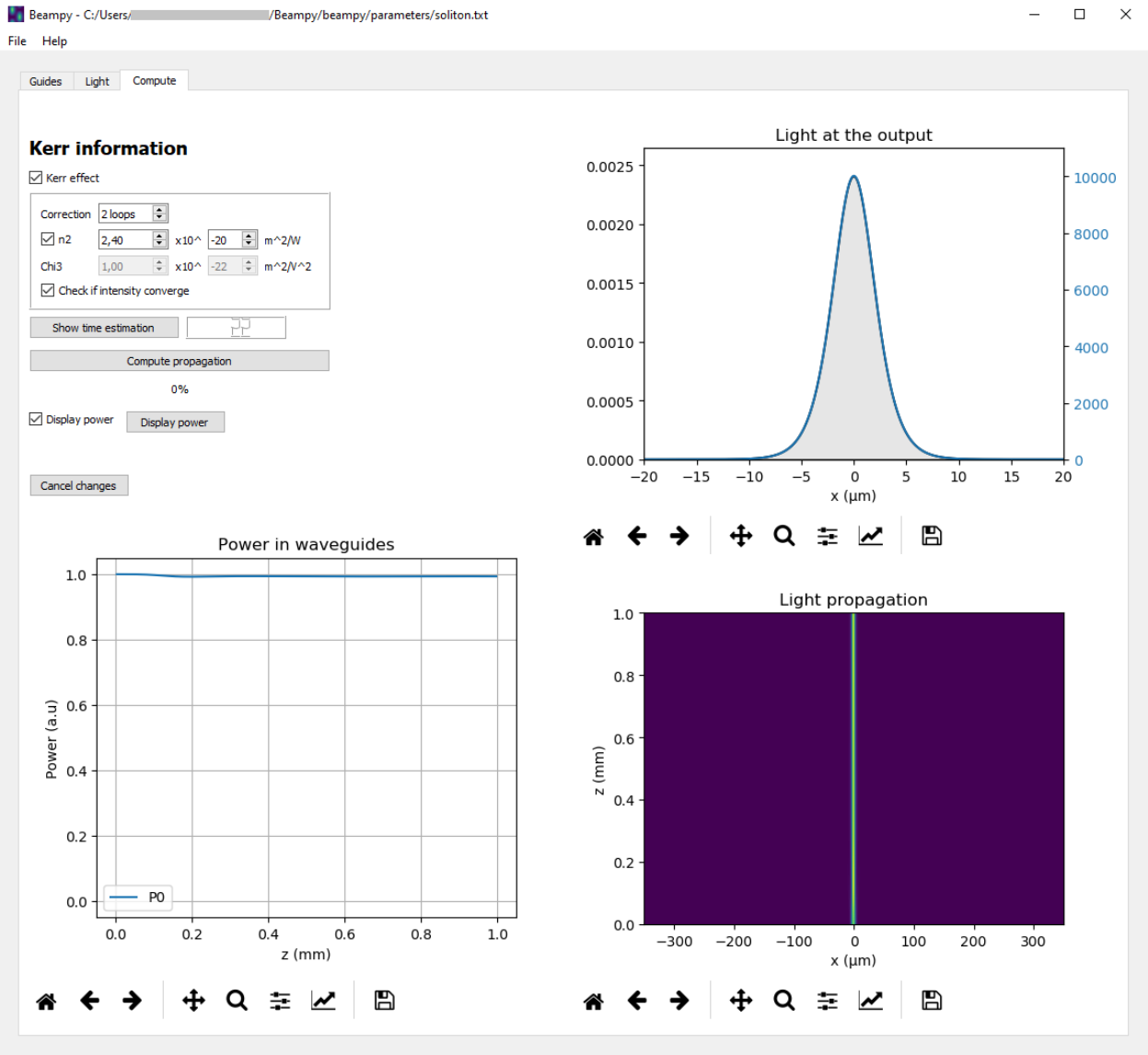

Fig. 8: Soliton propagation creating its own waveguide

beampy.examples.**multimodal_splitter**()
Multimodal splitter 1x2. A single mode beam is split into two single mode waveguide by the use of an intermediate multimodal waveguide.

beampy.examples.**show_grid**()
Show the computation grid and the displayed grid

beampy.examples.**stability**()
Show the possible BPM approximations for implementing a refractive index variation

beampy.examples.**test_kerr**()
More test than example. Show different approximations for the BPM implementation of the Kerr effect.

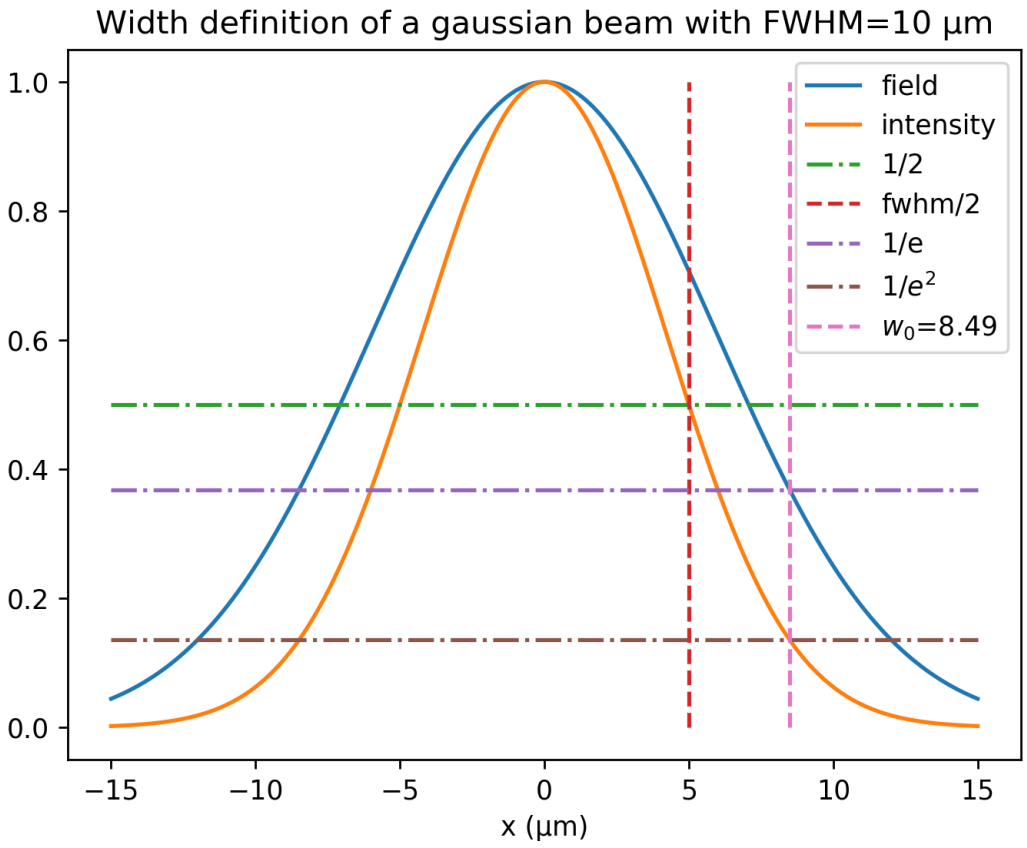

Fig. 9: Gaussian beam width definition

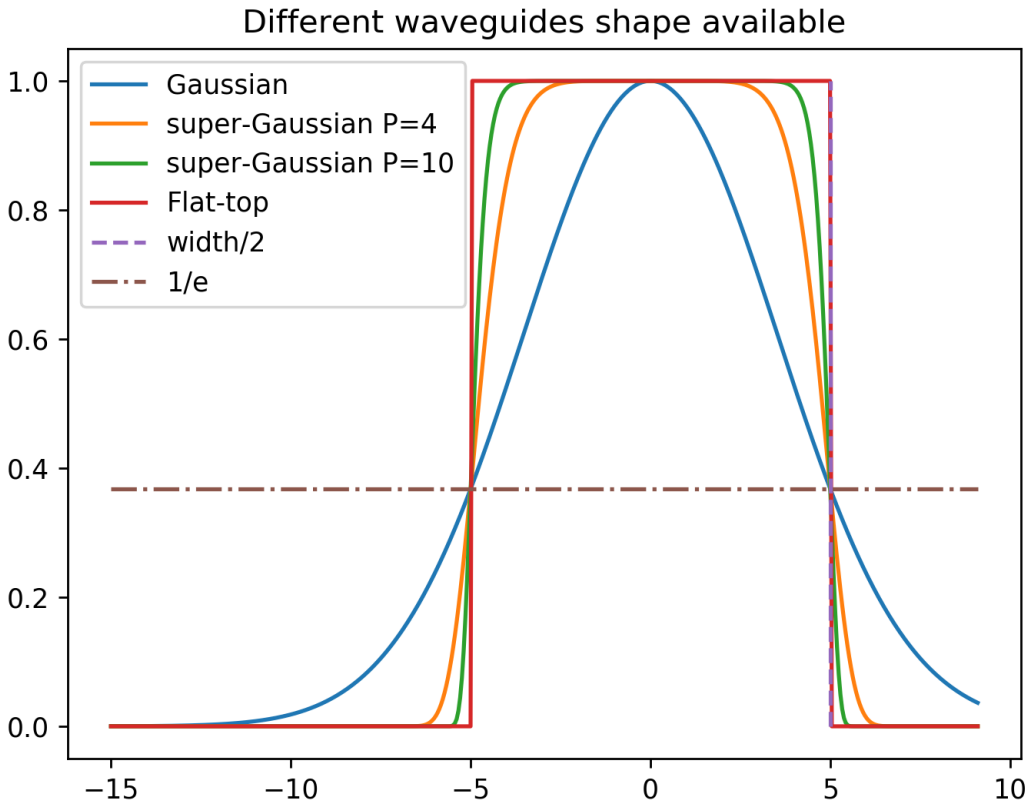

Fig. 10: Waveguide profil definition

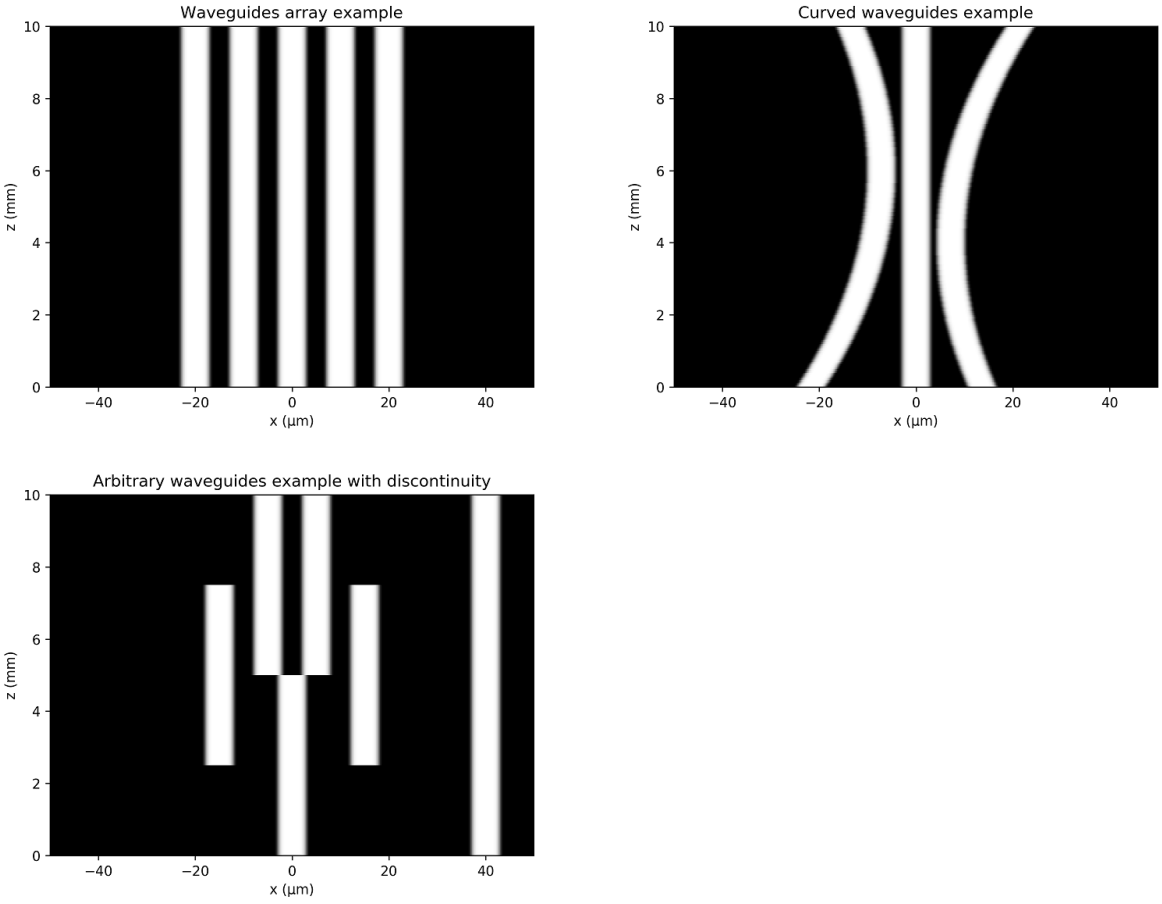

Fig. 11: Array of waveguides array, curved waveguides and arbitrary waveguides

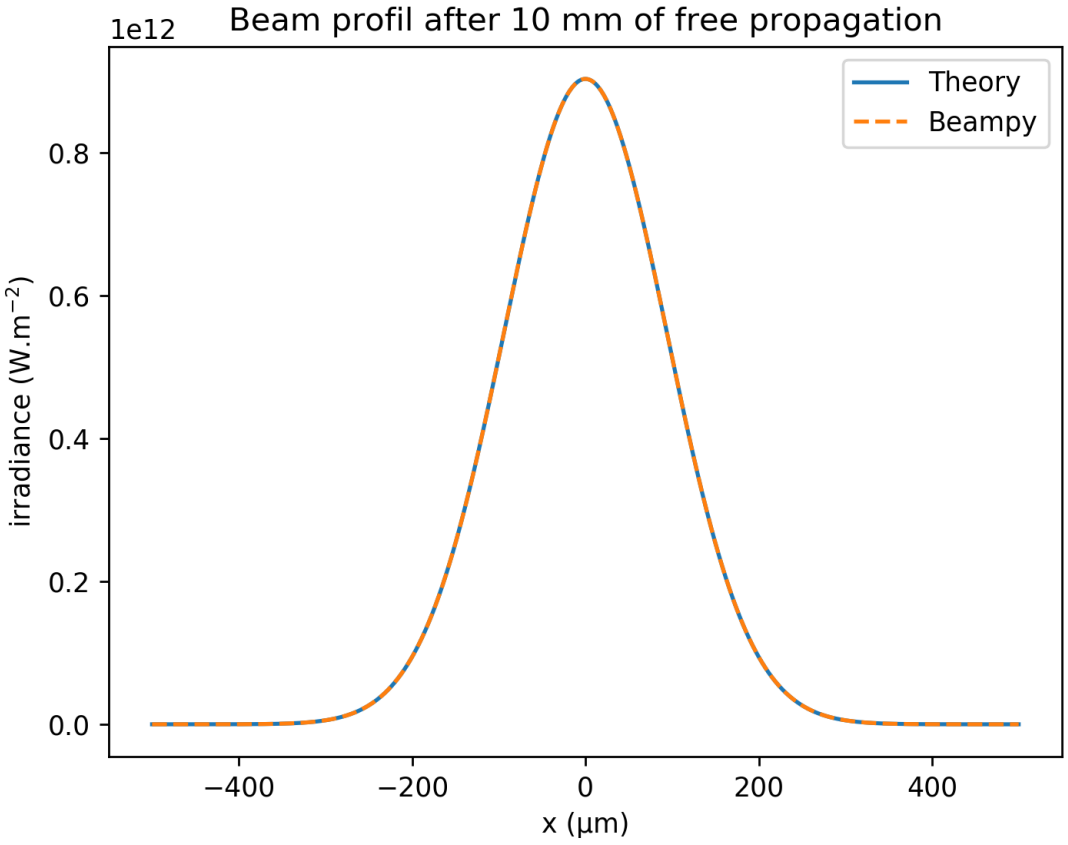

Fig. 12: Free propagation

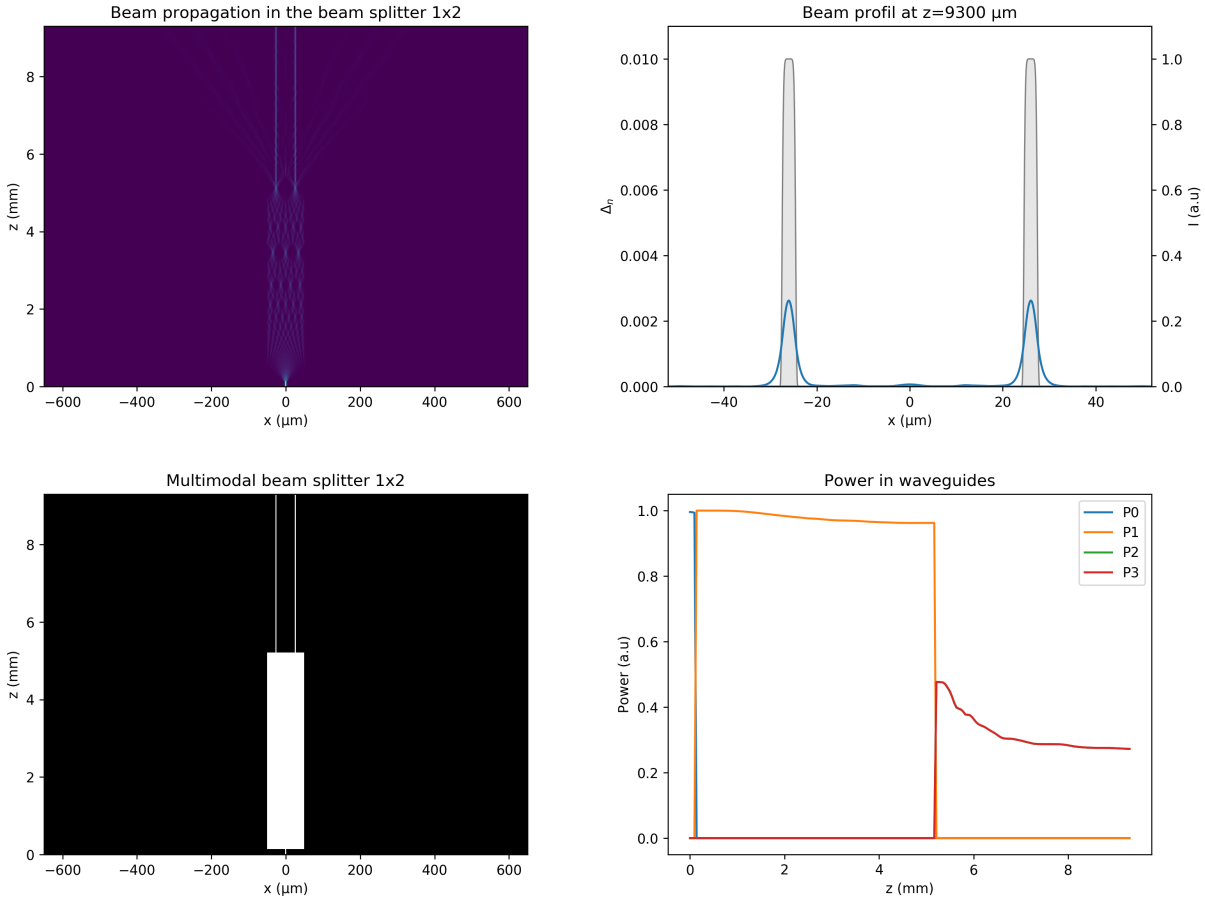

Fig. 13: Multimodal beam splitter 1x2

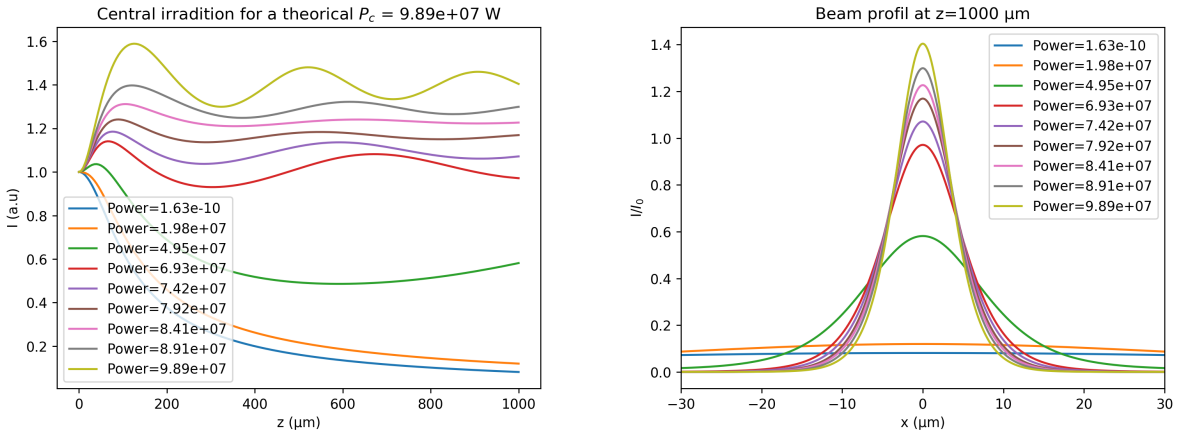

Fig. 14: Benchmark attempt on the critical power needed to create a soliton in a Kerr medium. The found power is 15% smaller than the theorical value

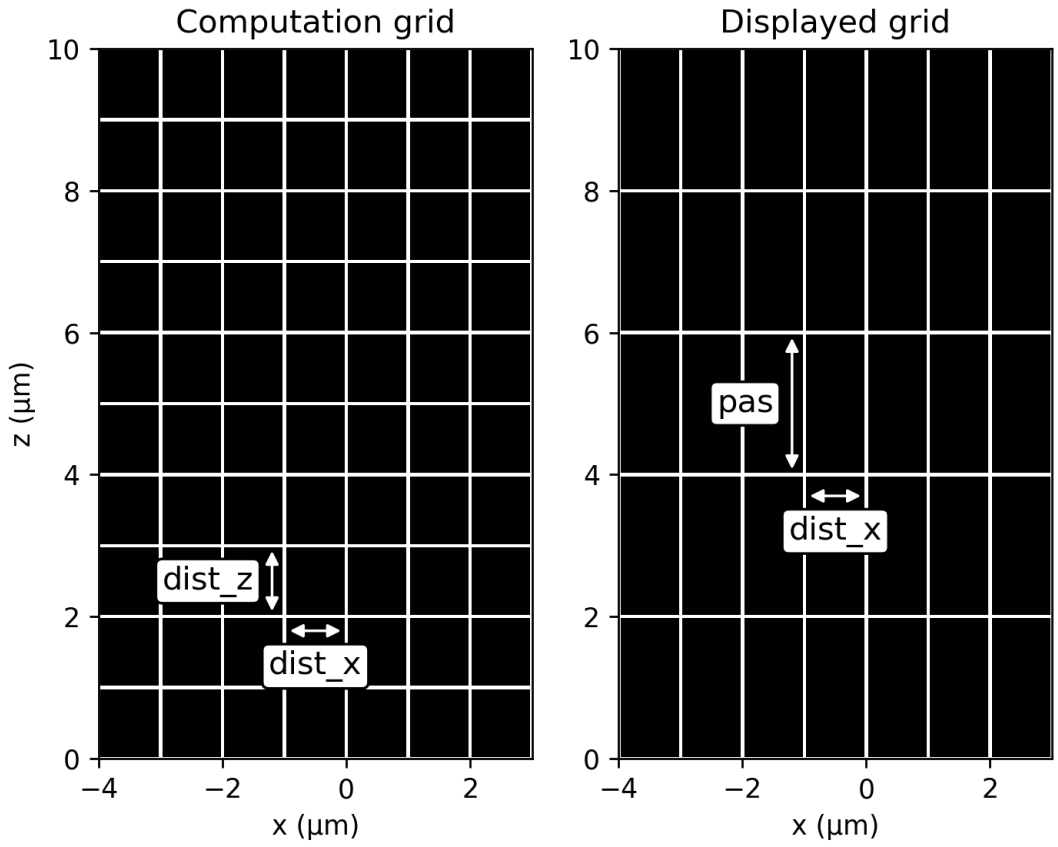

Fig. 15: Grid definition

5.2.2 Other examples & results

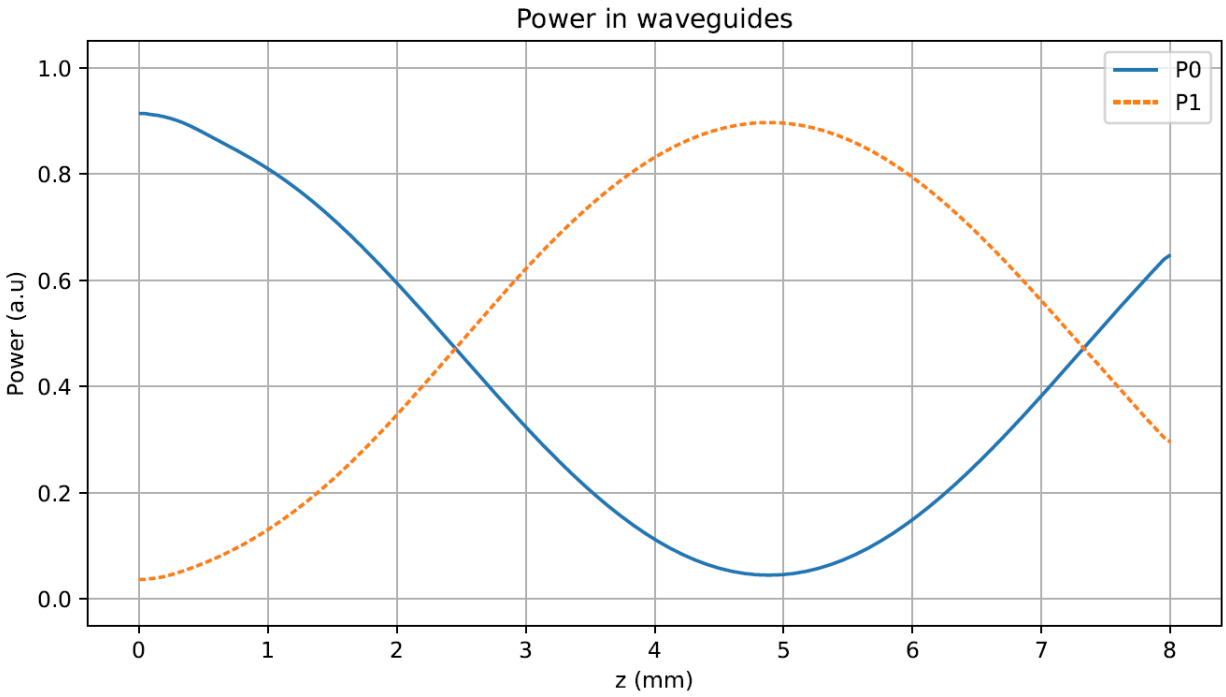

Fig. 16: Power in two waveguides

5.3 Beampy modules

List of all the modules used in Beampy as of Dec 31, 2020 for the version 1.11

Source codes

5.3.1 beampy.bpm

beampy.bpm code.

The bpm module contain the Bpm class used to simulate the light propagation - within low refractive index variation and small angle (paraxial approximation) - using the Beam Propagation Method.

This module was done by Jonathan Peltier during a master university course from the PAIP master of the université de Lorraine, under the directive of Pr. Nicolas Fressengeas.

The bpm codes are mainly based on a compilation of MatLab codes initialy developed by Régis Grasser during his PhD thesis[2], and later modified at the FEMTO-ST institute of the Université de Franche-Comté and at the LMOPS laboratory [3] of the Université de Lorraine.

[1] K. Okamoto, in Fundamentals of Optical Waveguides, 2nd ed., edited by K. Okamoto (Academic, Burlington, 2006), pp. 329–397.

[2] “Generation et propagation de reseaux periodiques de solitons spatiaux dans un milieu de kerr massif” PhD thesis, université de Franche-Comté 1998.

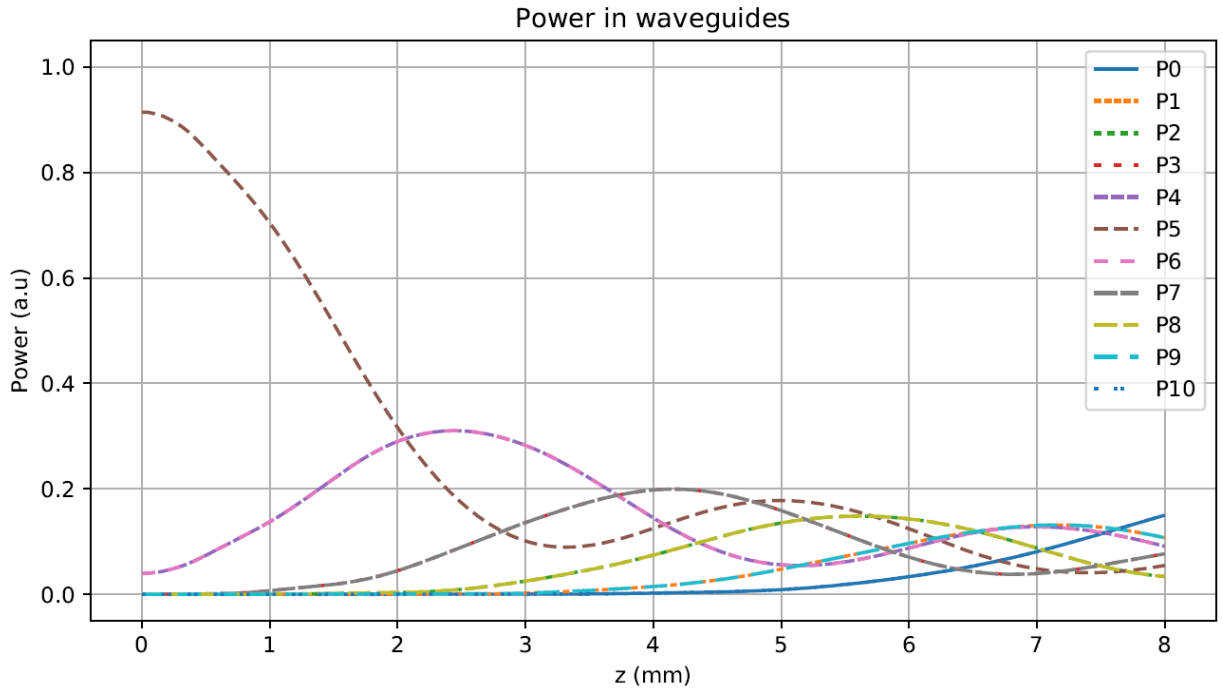

Fig. 17: Power over z in each waveguide of the array. The power distribution in a waveguide array follow the Bessel function.

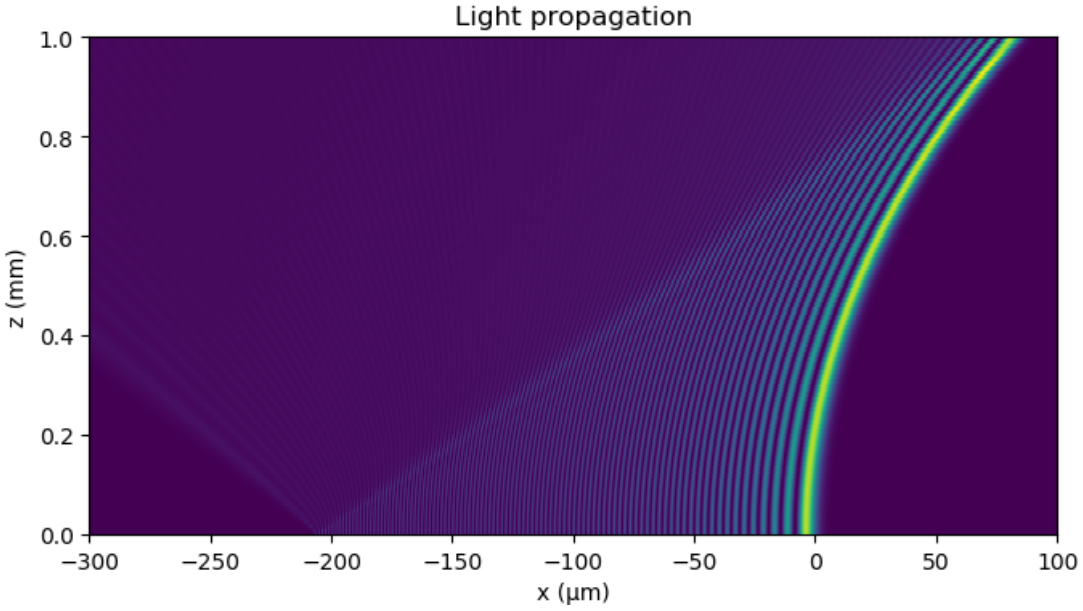

Fig. 18: Airy beam propagation

[3] H. Oukraou et. al., Broadband photonic transport between waveguides by adiabatic elimination Phys. Rev. A, 97 023811 (2018).

class beampy.bpm.**Bpm**(*no*, *lo*, *length_z*, *dist_z*, *nbr_z_disp*, *length_x*, *dist_x*)
Bases: object

The Bpm class is used to simulate light propagation - within small refractive index variation guides and small angle of propagation (paraxial) - using the Beam Propagation Method.

Parameters

- **no** (*float*) – Refractive index of the cladding.
- **lo** (*float*) – Wavelength of the beam in vaccum (µm).
- **length_z** (*float*) – Size of the compute window over z (µm).
- **dist_z** (*float*) – Step over z (µm)
- **nbr_z_disp** (*int*) – Number of points to display over z.
- **length_x** (*float*) – Size of the compute window over x (µm).
- **dist_x** (*float*) – Step over x (µm)

airy_light(*lobe_size*, *airy_zero*, *offset_light=0*)
Create an Airy beam using scipy.special.airy(x).

Parameters

- **lobe_size** (*float*) – Size of the first lobe (µm).
- **airy_zero** (*int*) – Cut the beam at the asked zero of the Airy function. n lobes will be displayed.
- **offset_light** (*float, optional*) – Light offset from center in µm. 0 by default.

Returns

- **field** (*array*) – Amplitude values over x (µm).
- **airy_zero** (*int*) – Number of lobes. Corrected if higher than the window size.

Notes

This methods uses the following variables defined in *Bpm*: nbr_x, length_x, dist_x, x.

all_modes(*width*, *delta_no*, *offset_light=0*)
Compute all modes allowed by the guide and sum them into one field.

Parameters

- **width** (*float*) – Waveguide width (µm) at 1/e^2 intensity.
- **delta_no** (*float*) – Difference of refractive index between the core and the cladding.
- **offset_light** (*float, optional*) – Light offset from center in µm. 0 by default.

Returns

- **field** (*array*) – Sum of all possibles fields in the guide.
- **h** (*array, float*) – Transverse propagation constant over x in µm of all modes.
- **gamma** (*array, float*) – Extinction coefficient over z in µm of all modes.
- **beta** (*array, float*) – Longitudinal constant of propagation over z in µm of all modes.

Notes

This methods uses the following variables defined in *Bpm*: nbr_x and the :meth`mode_light` method.

bpm_compute(*dn*, *n2=None*, *chi3=None*, *kerr_loop=1*, *variance_check=False*)
Compute BPM principle : free_propag over dz/2, index modulation, free_propag over dz/2.

Parameters

- **n2** (*float, optional*) – Nonlinear refractive index responsable for the optical Kerr effect in m^2/W. None by default.
- **chi3** (*float, optional*) – Value of the third term of the electric susceptibility tensor in m^2/V^2. None by default.
- **kerr** (*bool, optional*) – Activate the kerr effect. False by default.
- **kerr_loop** (*int, optional*) – Number of corrective loops for the Kerr effect. 1 by default.
- **variance_check** (*bool*) – Check if the kerr effect converge fast enought. False by default.

Returns **current_power** – Power after the propagation over dz.

Return type array

Notes

This method uses the *Bpm* class variables: nbr_lost, i, field, dist_z, dn, nl_mat, phase_mat, epnc, *kerr_effect()*.

This method change the values of: field, dn, nl_mat, current_power.

check_modes(*width*, *delta_no*)
Return the last possible mode number. Could be merged with *all_modes()* but would increase the needed time to compute just to display a number.

Parameters

- **width** (*float*) – Waveguide width (µm) at 1/e^2 intensity.
- **delta_no** (*float*) – Difference of refractive index between the core and the cladding.

Returns **m** – Number of the last possible mode for a squared guide.

Return type int

Notes

This methods uses the :meth`mode_light` method defined in *Bpm*.

create_curved_guides(*shape*, *width*, *delta_no*, *curve*, *half_delay*, *distance_factor*, *offset_guide=0*)
Create two curved guides and one linear guide on the center (STIRAP).

The central positions over x and z are defined as follow:

Left guide: $x_0 - p_{min} - curve(z - length\_z/2 - half\_delay)^2$

Right guide: $x_0 + p_{min} + curve(z - length\_z/2 + half\_delay)^2$

Central guide: x_0

Parameters

- **shape** (*method*) – `square()` or `gauss()`
- **width** (*float*) – Waveguide width (µm) at 1/e^2 intensity.
- **delta_no** (*float*) – Difference of refractive index between the core and the cladding. Can contain the losses throught the imaginary part.
- **curve** (*float*) – curvature factor in $10^{-8}m^{-2}$.
- **half_delay** (*float*) – Half distance over z in µm bewteen the two external guides where they are the closest. In other words, the distance from the center and the minimum of one of the curved guides over z.
- **distance_factor** (*float*) – Factor between the guide width and the minimal distance between the two guides =p_min/width. If distance_factor=1, the curved guides will touch the central guide (p_min=width).
- **offset_guide** (*float, optional*) – Waveguide offset from the center (µm). 0 by default.

Returns

- **peaks** (*array*) – Central position of each guide as peaks[guide,z].
- **dn** (*array*) – Difference of refractive index as dn[z,x]. Can contain the losses throught the imaginary part.

Notes

This methods uses the following variables defined in *Bpm*: length_z, nbr_z, nbr_x, x, dist_x.

create_guides(*shape*, *delta_no*, *nbr_p*, *p*, *offset_guide=0*, *z=0*)

Create an array of guides over x using peaks positions and for a given shape.

Parameters

- **shape** (*method*) – *squared_guide()*, *gauss_guide()* or any lambda function that takes one argument and return the relative refractive index for the input position.
- **delta_no** (*float*) – Difference of refractive index between the core and the cladding. Can contain the losses throught the imaginary part.
- **nbr_p** (*int*) – Number of guides.
- **p** (*float*) – Distance between two guides center (µm).
- **offset_guide** (*float, optional*) – Guide offset from the center (µm). 0 by default.
- **z** (*list*) – list [start, end] defining the waveguide length. Default length= windows length.

Returns

- **peaks** (*array-like*) – Central position of each guide [guide,z].
- **dn** (*array-like*) – Difference of refractive index [z,x]. Can contain the losses throught the imaginary part.

Notes

This methods uses the following variables defined in *Bpm*: nbr_z, nbr_x, x, dist_x.

create_x_z()
Create the x, z array and ajust the resolution variables.

Returns

- **length_z** (*float*) – Corrected value due to nbr_z being an int.
- **nbr_z** (*int*) – Number of points computed over z.
- **nbr_z_disp** (*int*) – Corrected value due to pas being an int.
- **length_x** (*float*) – Corrected value due to nbr_x being an int.
- **nbr_x** (*int*) – Number of point over x (µm).
- **x** (*array*) – x values between [-length_x/2, length_x/2-dist_x] center on 0.

Notes

This method creates the following variables within the class *Bpm*:

- pas : Interval of computed points between each displayed points.

gauss_guide(*width*, *gauss_pow=1*)
A lambda function than return a centered super-Gaussian shape.

$e^{-(x/w)^{2P}}$

The waist is defined as width/2 and correspond to the 1/e relative value.

See `example_guides_x()` for more details.

Parameters

- **width** (*float*) – Waveguide width (µm) at 1/e^2 intensity.
- **gauss_pow** (*int, optional*) – Index of the super-gaussian guide with 1 being a regural gaussian guide and 4 being the conventionnal super-gaussian guide used to describe realistic waveguides. See on en.wikipedia.org/wiki/Gaussian_function #Higher-order_Gaussian_or_super-Gaussian_function. 1 by Default.

gauss_light(*fwhm*, *offset_light=0*)
Create a gaussien beam in amplitude.

$E = e^{-((x-x_0)/w)^{2P}}$

The waist is defined as fwhm/sqrt(2*log(2)) and correspond to the 1/e field value and $1/e^2$ intensity value.

Parameters

- **fwhm** (*float*) – Full width at half maximum (for intensity not amplitude) (µm).
- **offset_light** (*float, optional*) – Light offset from center in µm. 0 by default.

Returns **field** – Amplitude values over x in µm.

Return type array

Notes

This methods uses the x and dist_x variables defined in *Bpm*.

guide_position(*peaks*, *guide*, *size*)
Return the left and right position index over x of a given guide for each z.

Parameters

- **peaks** (*array-like*) – Central position of each guide [guide,z].
- **guide** (*int*) – Number of the guide.
- **size** (*float*) – Width (µm).

Returns

- **x_beg** (*array*) – Left indices position of the selected guide over z.
- **x_end** (*array*) – Right indices position of the selected guide over z.

Notes

This methods uses the following variables defined in *Bpm*: nbr_z, x, length_x.

init_field(*field*, *theta_ext*, *irrad*)
Initialize phase, field and power variables.

Parameters

- **field** (*array, array-like*) – Amplitude values for each beams over x (µm) [beam,E] or E
- **theta_ext** (*float*) – Exterior inclinaison angle (°).
- **irrad** (*array, array-like*) – Irradiance for each beam (W/m^2).

Returns **progress_pow** – Intensity values over x (µm).

Return type array

Notes

This method creates the following variables within the class *Bpm*:

- epnc: Convertion factor used to set unit of the field and irradiance.
- phase_mat: Free propagation in Fourier space over dz/2.
- current_power: Intensity for z=0.
- field: Field value with the unit.

This methods uses the following variables defined in *Bpm*: no, x, dist_x, nbr_x, nbr_z_disp.

kerr_effect(*dn*, *n2=None*, *chi3=None*, *kerr_loop=1*, *variance_check=False*, *field_start=None*, *dn_start=None*, *phase_mat=None*)
Kerr effect: refractive index modulation by the light intensity. See: https://optiwave.com/optibpm-manuals/bpm-non-linear-bpm-algorithm/

Parameters

- **dn** (*array*) – Difference of refractive index as dn[z,x]. Can contain the losses throught the imaginary part.

- **n2** (*float, optional*) – Nonlinear refractive index responsable for the optical Kerr effect in m^2/W. None by default.
- **chi3** (*float, optional*) – Value of the third term of the electric susceptibility tensor in m^2/V^2. None by default.
- **kerr_loop** (*int, optional*) – Number of corrective loops for the Kerr effect. 1 by default.
- **variance_check** (*bool, optional*) – Check if the kerr effect converge fast enought. False by default.
- **field_start** (*array, optional*) – Field without kerr effect. If None were given, take the *main_compute()* field.
- **dn_start** (*array, optional*) – Refractive index without kerr effect. If None were given, take the *main_compute()* dn.
- **phase_mat** (*array, optional*) – Free propagation in Fourier space over dz/2. If None were given, take the *main_compute()* phase_mat.

Returns

- **dn** (*array*) – Refractive index with kerr effect.
- **nl_mat** (*array*) – refractive index modulation with kerr effect.
- **field_x** (*array*) – Field with the kerr effect at the self.i step.
- **cur_pow** (*array*) – Beam power with the kerr effect after the dz propagation.

Notes

This methods uses the following variables defined in *Bpm*: i, epnc, no, ko, dist_z and the *variance()* method.

main_compute(*dn*, *n2=None*, *chi3=None*, *kerr_loop=1*, *variance_check=False*, *disp_progress=True*)
main method used to compute propagation.

Parameters

- **n2** (*float, optional*) – Nonlinear refractive index responsable for the optical Kerr effect in m^2/W. None by default.
- **chi3** (*float, optional*) – Value of the third term of the electric susceptibility tensor in m^2/V^2. None by default.
- **kerr** (*bool, optional*) – Activate the kerr effect. False by default.
- **kerr_loop** (*int, optional*) – Number of corrective loop for the Kerr effect. 1 by default.
- **variance_check** (*bool, optional*) – Check if the kerr effect converge fast enought. False by default.
- **alpha** (*float, optional*) – Absorption per µm. 0 by default
- **lost_beg** (*array-like, optional*) – Left indices position of the selected waveguide over z. None by default.
- **lost_end** (*array-like, optional*) – Right indices position of the selected waveguide over z. None by default.

Returns **progress_pow** – Intensity values (W/m^2) over x (µm) and z (µm).

Return type array

Notes

This method creates the following variables within the class *Bpm*: nl_mat: Refractive index modulation.

This method uses the *Bpm* class variables: phase_mat, field, i, nbr_z, pas, current_power, dist_z, length_z, nbr_lost, dn, nl_mat, epnc and uses the *bpm_compute()*, *kerr_effect()*.

This method change the values of the *Bpm* class variables: field and if kerr, dn and nl_mat.

mode_determ(*width*, *delta_no*, *mode*)

Solve the transcendental equation tan=sqrt that give the modes allowed in a squared guide.

Parameters

- **width** (*float*) – Waveguide width (µm) at 1/e^2 intensity.
- **delta_no** (*float*) – Difference of refractive index between the core and the cladding.
- **mode** (*int*) – Number of the searched mode.

Returns

- **h_m** (*float*) – Transverse propagation constant over x (µm).
- **gamma_m** (*float*) – Extinction coefficient over x (µm).
- **beta_m** (*float*) – Longitudinal constant of propagation over z (µm).

Raises **ValueError** – if no mode exists.

Notes

This methods uses the following variables defined in *Bpm*: lo, no, ko.

mode_light(*width*, *delta_no*, *mode*, *offset_light=0*)

Create light based on propagated mode inside a squared guide.

Parameters

- **width** (*float*) – Waveguide width (µm) at 1/e^2 intensity.
- **delta_no** (*float*) – Difference of refractive index between the core and the cladding.
- **mode** (*int*) – Number of the searched mode.
- **offset_light** (*float, optional*) – Light offset from center (µm). 0 by default.

Returns

- **field** (*array*) – Amplitude values over x (µm).
- **h_m** (*float*) – Transverse propagation constant over x (µm).
- **gamma_m** (*float*) – Extinction coefficient over x (µm).
- **beta_m** (*float*) – Longitudinal constant of propagation over z (µm).

Notes

This methods uses the following variables defined in *Bpm*: nbr_x, x and the :meth`mode_determ` method.

power_guide(*x_beg*, *x_end*)
return the power over z in a given interval by integrating the beam irradiance.

Parameters

- **x_beg** (*array*) – Left indices position over z for a selected guide.
- **x_end** (*array*) – Right indices position over z for a selected guide.

Returns **P** – Normalized power in the guide over z.

Return type array

Notes

This methods uses the following variables defined in *Bpm*: nbr_z_disp, progress_pow, pas.

squared_guide(*width*)
A lambda function than returns a centered rectangular shape.

return 1 if $x >= -width/2$ and $x <= width/2$ else return 0.

Parameters **width** (*float*) – Waveguide width.

squared_light(*fwhm*, *offset_light=0*)
Create a flat-top beam (squared).

Parameters

- **fwhm** (*float*) – Beam width in µm.
- **offset_light** (*float, optional*) – Light offset from center in µm. 0 by default.

Returns **field** – Amplitude values over x in µm.

Return type array

Notes

This methods uses the following variables defined in *Bpm*: nbr_x, x.

variance(*initial*, *final*)
This function alerts the user when the kerr effect don't converge fast enough. Raise a ValueError when the power standard deviation exceed 10^{-7}.

Parameters

- **initial** (*array*) – Power of the kerr effect looped n-1 time.
- **final** (*array*) – Power of the kerr effect looped n time.

Raises **ValueError** – when the power standard deviation exceed 10^{-7}.

beampy.bpm.**example_bpm**()
Version allowing to compute BPM without the user interface. Used for quick test.

5.3.2 beampy.interface

beampy.interface code.

class beampy.interface.**Ui_MainWindow**
Bases: object

retranslateUi(*MainWindow*)

setupUi(*MainWindow*)

5.3.3 beampy.user_interface

beampy.user_interface code.

The user_interface module is the main file of the beampy module. It contain the UserInterface class used to computed and displayed the bpm results onto the interface.

This module was at first developed by Marcel Soubkovsky for the implementation of the array of guides, of one gaussian beam and of the plotting methods. Then, continued by Jonathan Peltier.

class beampy.user_interface.**UserInterface**
Bases: PyQt5.QtWidgets.QMainWindow, *beampy.interface.Ui_MainWindow*

This class connect the *Bpm* class from the *bpm* module with the *setupUi()* method from the *interface* module. The interface seems to close itself when openning from spyder 3.5. In that case, open the interface in an external console.

addmpl(*tab='guide'*, *pow_index=0*)
Add the selected plots on the waveguide, light or compute window.

Parameters

- **tab** (*str, optional*) – 'guide' or 'light'. 'guide' by default.
- **pow_index** (*int, optional*) – Add the first waveguide and light step if 0 or the last step if -1. Also display the propagation over (x,z) and guide power if -1 is choosen. 0 by default.

calculate_guide()
Initialized the *Bpm* class and creates the guides.

Notes

Creats many variables, including :

- peaks : Central position of each waveguides [waveguide,z].
- dn : Difference of reefractive index [z,x].

This method calls the following methods from *Bpm*:

create_x_z(), *squared_guide()*, *gauss_guide()*, *create_guides()*, *create_curved_guides()*.

calculate_light()
Create the choosen beams.

Notes

Creates the progress_pow variable.

This method calls the following methods from the *Bpm* class:

gauss_light(), *squared_light()*, *all_modes()*, *mode_light()*, *airy_light()*, *init_field()*.

calculate_propagation()
Calculate the propagation based on the input light and guides shapes.

Notes

Creates the progress_pow variable.

Calls the following methods from *Bpm*: `losses_position()`, meth`.main_compute`.

check_modes_display()
Display on the interface the last mode that can propagated into a squared waveguide.

connect_buttons()
Connect the interface buttons to their corresponding functions:

on_click_guide(), `on_click_curved()`, *on_click_light()*, *on_click_compute()*, *on_click_create_light()*, *on_click_delete_light()*, *save_light()*, *get_guide()*, *get_light()*, *get_compute()*, *show_estimate_time()*, *check_modes_display()*.

create_menu()
Create a menu to open, save a file, or exit the app.

Notes

This method connect the following methods and function to the menu buttons:

open_file_name(), *save_quick()*, *save_file_name()*, *open_doc()*.

display_power()
Display the power in each waveguide.

find_guide_number(*guide_number*)
Return the waveguide group number for a given waveguide number.

Parameters **guide_number** (*int*) – Number of a waveguide

Returns **guide_group_number** – Number of the waveguide group

Return type int

get_compute()
Set the saved values of the compute variables onto the interface.

get_guide()
Set the saved values of the waveguide variables onto the interface.

get_light()
Set the saved values of the light variables onto the interface.

main_compute(*dn*, *n2=None*, *chi3=None*, *kerr_loop=1*, *variance_check=False*, *disp_progress=True*)
main method used to compute propagation.

Parameters

- **n2** (*float, optional*) – Nonlinear refractive index responsable for the optical Kerr effect in m^2/W. None by default.
- **chi3** (*float, optional*) – Value of the third term of the electric susceptibility tensor in m^2/V^2. None by default.
- **kerr** (*bool, optional*) – Activate the kerr effect. False by default.
- **kerr_loop** (*int, optional*) – Number of corrective loop for the Kerr effect. 1 by default.
- **variance_check** (*bool, optional*) – Check if the kerr effect converge fast enought. False by default.
- **alpha** (*float, optional*) – Absorption per µm. 0 by default
- **lost_beg** (*array-like, optional*) – Left indices position of the selected waveguide over z. None by default.
- **lost_end** (*array-like, optional*) – Right indices position of the selected waveguide over z. None by default.

Returns **progress_pow** – Intensity values (W/m^2) over x (µm) and z (µm).

Return type array

Notes

This method creates the following variables within the class `Bpm`: nl_mat: Refractive index modulation.

This method uses the `Bpm` class variables: phase_mat, field, i, nbr_z, pas, current_power, dist_z, length_z, nbr_lost, dn, nl_mat, epnc and uses the `bpm_compute()`, `kerr_effect()`.

This method change the values of the `Bpm` class variables: field and if kerr, dn and nl_mat.

on_click_compute()
Compute the propagation using the waveguides and bemas informations.

on_click_create_guide()
Create a new waveguide with the displayed variables.

on_click_create_light()
Create a new beam with the displayed variables.

on_click_delete_guide()
Delete the current displayed waveguide and displayed the next one if exist else the previous one.

on_click_delete_light()
Delete the current displayed beam and displayed the next one.

on_click_guide()
Create and displayed the waguides.

on_click_light()
Create the light and display it with the input profil waveguides.

open_file(*filename*)
Set guides, beams and computes variables from a choosen file.

Parameters **filename** (*str*) – Name of the file.

open_file_name()
Open a dialog window to select the file to open, and call *open_file()* to open the file.

Source: https://pythonspot.com/pyqt5-file-dialog/

Notes

This method has a try/except implemented to check if the openned file contains all the variables.

rmmpl(*tab*, *pow_index=0*)
Remove the selected plots

Parameters

- **tab** (*str*) – 'guide' or 'light'.
- **pow_index** (*int, optional*) – Remove the first light step if 0 or the last step if -1. 0 by default.

save_compute()
Save the interface variables into the compute variables.

save_file(*filename*)
Save guides, beams and computes variables into a choosen file.

Parameters **filename** (*str*) – Name of the file.

save_file_name()
Open a dialog window to select the saved file name and call *save_file()* to save the file.

save_guide(*guide_selec=False*)
Save the interface variables into the guides variables.

save_light(*beam_selec=False*)
Save the interface variables into the lights variables.

Parameters **beam_selec** (*int, bool, optional*) – Number of the beam to save into the variables. False by default to get the currently displayed beam.

save_quick()
Check if a file is already selected and if so, save into it. Else, call the *save_file_name()* to ask a filename.

show_estimate_time()
Display - on the interface - the estimate time needed to compute the propagation, based on linearized experimental values. The estimation takes into account the looses, Kerr, and control parameters.

beampy.user_interface.**open_app**()
Function used to open the app. Can be called directly from beampy.

beampy.user_interface.**open_doc**()
Function that open the local html documentation - describing the beampy modules - if exist, or the online version otherwise.

5.4 Source codes

5.4.1 beampy.bpm

```
"""
The bpm module contain the Bpm class used to simulate the light propagation -
within low refractive index variation
and small angle (paraxial approximation) -
using the Beam Propagation Method.

This module was done by Jonathan Peltier during a master
university course from the PAIP master of the université de Lorraine,
under the directive of Pr. Nicolas Fressengeas.

The bpm codes are mainly based on a compilation of MatLab codes initialy
developed by Régis Grasser during his PhD thesis[2],
and later modified at the FEMTO-ST institute of the Université de
Franche-Comté and at the LMOPS laboratory [3] of the
Université de Lorraine.

[1] K. Okamoto, in Fundamentals of Optical Waveguides,
2nd ed., edited by K. Okamoto (Academic, Burlington, 2006), pp. 329-397.

[2] "Generation et propagation de reseaux periodiques de solitons spatiaux
dans un milieu de kerr massif" PhD thesis, université de Franche-Comté 1998.

[3] H. Oukraou et. al., Broadband photonic transport between waveguides by
adiabatic elimination Phys. Rev. A, 97 023811 (2018).
"""

from math import pi, ceil, radians, sqrt, log, sin, cos, acos, asin, exp
import time
from scipy import special
from numpy.fft import fft, ifft, fftshift
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

import numba


@numba.vectorize([numba.float64(numba.complex128),
                  numba.float32(numba.complex64)])
def abs2(x):
    """Square modulus of x. Fastest way possible for a numpy array."""
    return x.real**2 + x.imag**2


class Bpm():
    """
    The Bpm class is used to simulate light propagation -
    within small refractive index variation guides
    and small angle of propagation (paraxial) -
    using the Beam Propagation Method.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    no : float
        Refractive index of the cladding.
    lo : float
        Wavelength of the beam in vaccum (µm).
    length_z : float
        Size of the compute window over z (µm).
    dist_z : float
        Step over z (µm)
    nbr_z_disp : int
        Number of points to display over z.
    length_x : float
        Size of the compute window over x (µm).
    dist_x : float
        Step over x (µm)
    """

    def __init__(self, no, lo,
                 length_z, dist_z, nbr_z_disp,
                 length_x, dist_x):
        """
        The Bpm class is used to simulate light propagation -
        within small refractive index variation guides
        and small angle of propagation (paraxial) -
        using the Beam Propagation Method.

        Parameters
        ----------
        no : float
            Refractive index of the cladding
        lo : float
            Wavelength of the beam in vaccum (µm).
        length_z : float
            Size of the compute window over z (µm).
        dist_z : float
            Step over z (µm).
        nbr_z_disp : int
            Number of points to display over z.
        length_x : float
            Size of the compute window over x (µm).
        dist_x : float
            Step over x (µm).

        Notes
        -----
        This method creates the following variables within the class
        :class:`Bpm`:

        - All input variables.
        - ko: the free space vector (1/µm).
        """
        self.no = no
        self.lo = lo
        self.ko = 2*pi / self.lo  # linear wave vector in free space (1/µm)
        self.length_z = length_z
        self.dist_z = dist_z
```

(continues on next page)

(continued from previous page)

```
        self.nbr_z_disp = nbr_z_disp
        self.dist_x = dist_x
        self.length_x = length_x

    def create_x_z(self):
        """
        Create the x, z array and ajust the resolution variables.

        Returns
        -------
        length_z : float
            Corrected value due to nbr_z being an int.
        nbr_z : int
            Number of points computed over z.
        nbr_z_disp : int
            Corrected value due to pas being an int.
        length_x : float
            Corrected value due to nbr_x being an int.
        nbr_x : int
            Number of point over x (µm).
        x : array
            x values between [-length_x/2, length_x/2-dist_x] center on 0.

        Notes
        -----
        This method creates the following variables within the class
        :class:`Bpm`:

        - pas : Interval of computed points between each displayed points.
        """
        assert self.nbr_z_disp > 0

        self.nbr_z = ceil(self.length_z / self.dist_z)
        self.length_z = self.nbr_z * self.dist_z
        self.pas = ceil(self.length_z / (self.nbr_z_disp * self.dist_z))
        self.nbr_z_disp = ceil(self.length_z / (self.pas * self.dist_z))
        self.nbr_z_disp += 1  # add 1 for the initial field
        self.nbr_z += 1  # add 1 for the initial field
        self.nbr_x = ceil(self.length_x / self.dist_x)  # nbr points over x

        # check if even number
        if self.nbr_x % 2 != 0:
            self.nbr_x += 1

        # check if multiple of 8: speeds up execution
        # (was also needed for a obsolete feature)
        for _ in range(3):
            if self.nbr_x % 8 != 0:
                self.nbr_x += 2
            else:
                break

        self.length_x = self.nbr_x * self.dist_x
        self.x = np.linspace(-self.length_x/2,
                             self.length_x/2 - self.dist_x,
                             self.nbr_x)
```

(continues on next page)

(continued from previous page)

```
        return [self.length_z, self.nbr_z, self.nbr_z_disp-1,
                self.length_x, self.nbr_x, self.x]

    # Guides #

    def squared_guide(self, width):
        """
        A lambda function than returns a centered rectangular shape.

        return 1 if :math:`x >= -width/2` and :math:`x <= width/2`
        else return 0.

        Parameters
        ----------
        width : float
            Waveguide width.
        """
        return lambda x: (x >= -width/2) & (x <= width/2)

    def gauss_guide(self, width, gauss_pow=1):
        """
        A lambda function than return a centered super-Gaussian shape.

        :math:`e^{-(x/w)^{2P}}`

        The waist is defined as width/2 and correspond to the 1/e
        relative value.

        See :func:`.example_guides_x` for more details.

        Parameters
        ----------
        width : float
            Waveguide width (µm) at 1/e^2 intensity.
        gauss_pow : int, optional
            Index of the super-gaussian guide with 1 being a regural gaussian
            guide and 4 being the conventionnal super-gaussian guide used to
            describe realistic waveguides.
            See on en.wikipedia.org/wiki/Gaussian_function
            #Higher-order_Gaussian_or_super-Gaussian_function.
            1 by Default.
        """
        if width == 0:
            return lambda x: 0
        w = width / 2  # width is diameter and w is radius
        return lambda x: np.exp(-(x / w)**(2*gauss_pow))

    def create_guides(self, shape, delta_no, nbr_p, p, offset_guide=0, z=0):
        """
        Create an array of guides over x using peaks positions and for a given
        shape.

        Parameters
        ----------
        shape : method
            :meth:`squared_guide`, :meth:`gauss_guide` or any lambda function
            that takes one argument and return the relative refractive index
```

(continues on next page)

(continued from previous page)

```
        for the input position.
    delta_no : float
        Difference of refractive index between the core and the cladding.
        Can contain the losses throught the imaginary part.
    nbr_p : int
        Number of guides.
    p : float
        Distance between two guides center (µm).
    offset_guide : float, optional
        Guide offset from the center (µm). 0 by default.
    z : list
        list [start, end] defining the waveguide length. Default length=
        windows length.

    Returns
    -------
    peaks : array-like
        Central position of each guide [guide,z].
    dn : array-like
        Difference of refractive index [z,x]. Can contain the losses
        throught the imaginary part.

    Notes
    -----
    This methods uses the following variables defined in :class:`Bpm`:
    nbr_z, nbr_x, x, dist_x.
    """
    peaks = np.array([[None]*self.nbr_z]*nbr_p)
    dn = np.zeros((self.nbr_z, self.nbr_x))
    dn_z = np.zeros(self.nbr_x)

    if nbr_p == 0:
        return [np.array([[None]*self.nbr_z]), dn]
    peaks_z = (p*np.linspace(-nbr_p/2, nbr_p/2-1, nbr_p)
               + p/2
               + offset_guide)
    dn_fix = shape(self.x)  # guide shape center on 0

    # Sum each identical guide with an offset defined by peaks_z
    for i in range(nbr_p):
        dn_z += np.roll(dn_fix, int(round(peaks_z[i] / self.dist_x)))

    if z == 0:
        start = 0
        end = self.nbr_z
    else:
        # assert z[0] >= 0 and z[1] <= self.length_z and z[0] <= z[1]
        if z[0] > z[1]:
            print("Warning, the waveguide beginning occurs after the end.",
                  z[0], "should be <=", z[1])
        if z[1] > self.length_z:
            z[1] = self.length_z

        start = int(z[0]/self.dist_z)
        end = int(z[1]/self.dist_z)

    dn[start:end] = dn_z
```

(continues on next page)

(continued from previous page)

```
        for i in range(start, end):
            peaks[:, i] = peaks_z

        dn = dn*delta_no  # give a value to the shape
        return [peaks, dn]

    def create_curved_guides(self, shape, width, delta_no, curve, half_delay,
                             distance_factor, offset_guide=0):
        """
        Create two curved guides and one linear guide on the center (STIRAP).

        The central positions over x and z are defined as follow:

        Left guide: :math:`x_0-p_{min}-curve(z-length\_z/2-half\_delay)^2`

        Right guide: :math:`x_0+p_{min}+curve(z-length\_z/2+half\_delay)^2`

        Central guide: :math:`x_0`

        Parameters
        ----------
        shape : method
            :meth:`square` or :meth:`gauss`
        width : float
            Waveguide width (µm) at 1/e^2 intensity.
        delta_no : float
            Difference of refractive index between the core and the cladding.
            Can contain the losses throught the imaginary part.
        curve : float
            curvature factor in :math:`10^{-8} µm^{-2}`.
        half_delay : float
            Half distance over z in µm bewteen the two external guides where
            they are the closest.
            In other words, the distance from the center and the minimum of one
            of the curved guides over z.
        distance_factor : float
            Factor between the guide width and the minimal distance between the
            two guides =p_min/width.
            If distance_factor=1, the curved guides will touch the central
            guide (p_min=width).
        offset_guide : float, optional
            Waveguide offset from the center (µm). 0 by default.

        Returns
        -------
        peaks : array
            Central position of each guide as peaks[guide,z].
        dn : array
            Difference of refractive index as dn[z,x]. Can contain the losses
            throught the imaginary part.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        length_z, nbr_z, nbr_x, x, dist_x.
        """
        # all points over z
```

(continues on next page)

(continued from previous page)

```
        z = np.linspace(0, self.length_z, self.nbr_z)

        # left curved guide
        sa = (- offset_guide
              + curve*(z - self.length_z/2 - half_delay)**2
              + width*distance_factor)

        # right curved guide
        sb = (offset_guide
              + curve*(z - self.length_z/2 + half_delay)**2
              + width*distance_factor)

        peaks = np.array([-sa,
                          np.array([offset_guide] * self.nbr_z),
                          sb])

        dn = np.zeros((self.nbr_z, self.nbr_x))
        dn_fix = shape(self.x)    # guide shape center on 0

        for i in range(self.nbr_z):
            dn[i, :] = np.roll(dn_fix, int(round(-sa[i] / self.dist_x))) \
                + np.roll(dn_fix, int(round(offset_guide / self.dist_x))) \
                + np.roll(dn_fix, int(round(sb[i] / self.dist_x)))

        dn = dn * delta_no  # give a value to the shape
        return [peaks, dn]

    # Light #

    def gauss_light(self, fwhm, offset_light=0):
        """
        Create a gaussien beam in amplitude.

        :math:`E = e^{-((x-x_0)/w)^{2P}}`

        The waist is defined as fwhm/sqrt(2*log(2)) and correspond to the 1/e
        field value and 1/:math:`e^2` intensity value.

        Parameters
        ----------
        fwhm : float
            Full width at half maximum (for intensity not amplitude) (µm).
        offset_light : float, optional
            Light offset from center in µm. 0 by default.

        Returns
        -------
        field : array
            Amplitude values over x in µm.

        Notes
        -----
        This methods uses the x and dist_x variables defined in :class:`Bpm`.
        """
        spot_size = fwhm / sqrt(2 * log(2))  # such as I=1/e^2 in intensity
        if spot_size != 0:
            field = np.exp(-(self.x / spot_size)**2)
```

(continues on next page)

(continued from previous page)

```
            field = np.roll(field, int(round(offset_light / self.dist_x)))
        else:
            field = 0 * self.x  # Avoid division by zero error
        return field

    def squared_light(self, fwhm, offset_light=0):
        """
        Create a flat-top beam (squared).

        Parameters
        ----------
        fwhm : float
            Beam width in µm.
        offset_light : float, optional
            Light offset from center in µm. 0 by default.

        Returns
        -------
        field : array
            Amplitude values over x in µm.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        nbr_x, x.
        """
        field = np.zeros(self.nbr_x)

        for j in range(self.nbr_x):
            if self.x[j] >= -fwhm/2 and self.x[j] <= fwhm/2:
                field[j] = 1
            else:
                field[j] = 0

        field = np.roll(field, int(round(offset_light / self.dist_x)))
        return field

    def mode_determ(self, width, delta_no, mode):
        """
        Solve the transcendental equation tan=sqrt that give the modes
        allowed in a squared guide.

        Parameters
        ----------
        width : float
            Waveguide width (µm) at 1/e^2 intensity.
        delta_no : float
            Difference of refractive index between the core and the cladding.
        mode : int
            Number of the searched mode.

        Returns
        -------
        h_m : float
            Transverse propagation constant over x (µm).
        gamma_m : float
            Extinction coefficient over x (µm).
```

(continues on next page)

(continued from previous page)

```
        beta_m : float
            Longitudinal constant of propagation over z (µm).

        Raises
        ------
        ValueError
            if no mode exists.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        lo, no, ko.
        """
        width = float(width)

        if width == 0:
            raise ValueError("no mode " + str(mode) + " existing")

        delta_no = float(delta_no.real)
        lim = self.lo/(2 * width * (self.no + delta_no)) - 1e-12
        theta_c = acos(self.no / (self.no + delta_no))  # Critical angle
        solu = np.linspace(
            mode*lim + 0.000001,
            (mode + 1) * lim,
            round(1 + (lim - 0.000001)/0.000001))
        lhs = np.tan(
            pi * width * (self.no + delta_no) / self.lo * solu
            - mode*pi/2)
        rhs = np.sqrt(
            0j  # to avoid sqrt error when complexe
            + (sin(theta_c) / solu)**2
            - 1)
        result = rhs - lhs  # 0 if left == right
        minimum = abs(result).min()  # return min value : where two equations~=
        i_min = int(np.where(abs(result) == minimum)[0])  # min value index

        if i_min == 0:
            raise ValueError("no mode " + str(mode) + " existing")

        sin_theta_m = solu[i_min]
        theta_m = asin(sin_theta_m)  # angle at which the mode propagate

        beta_m = self.ko * (self.no + delta_no) * cos(theta_m)
        h_m = sqrt((self.ko * (self.no + delta_no))**2 - beta_m**2)
        gamma_m = (self.no * self.ko
                   * np.sqrt((cos(theta_m) / cos(theta_c))**2 - 1))

        return [h_m, gamma_m, beta_m]

    def mode_light(self, width, delta_no, mode, offset_light=0):
        """
        Create light based on propagated mode inside a squared guide.

        Parameters
        ----------
        width : float
            Waveguide width (µm) at 1/e^2 intensity.
```

(continues on next page)

(continued from previous page)

```
        delta_no : float
            Difference of refractive index between the core and the cladding.
        mode : int
            Number of the searched mode.
        offset_light : float, optional
            Light offset from center (µm). 0 by default.

        Returns
        -------
        field : array
            Amplitude values over x (µm).
        h_m : float
            Transverse propagation constant over x (µm).
        gamma_m : float
            Extinction coefficient over x (µm).
        beta_m : float
            Longitudinal constant of propagation over z (µm).

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        nbr_x, x and the :meth`mode_determ` method.
        """
        field = np.zeros(self.nbr_x)

        [h_m, gamma_m, beta_m] = self.mode_determ(width, delta_no, mode)

        if mode % 2 == 0:  # if even mode

            b_b = cos(h_m * width / 2)  # Continuity value where x=width/2

            for j in range(self.nbr_x):  # Compute light based on h,gamma,beta

                if abs(self.x[j]) <= width/2:  # in core
                    field[j] = cos(h_m * self.x[j])

                else:  # in cladding
                    field[j] = b_b * exp(-gamma_m * (
                        abs(self.x[j])
                        - width/2))
        else:  # if odd mode

            c_c = sin(h_m * width / 2)  # Continuity value where x=width/2

            for j in range(self.nbr_x):  # Compute light based on h,gamma,beta

                if abs(self.x[j]) <= width/2:  # in core
                    field[j] = sin(h_m * self.x[j])

                elif self.x[j] >= width/2:  # Right cladding
                    field[j] = c_c * exp(-gamma_m * (
                        self.x[j]
                        - width/2))

                else:  # Left cladding
                    field[j] = -c_c * exp(gamma_m * (
                        self.x[j]
```

(continues on next page)

(continued from previous page)

```
                        + width/2))

        field = np.roll(field, int(round(offset_light / self.dist_x)))

        return [field, h_m, gamma_m, beta_m]

    def all_modes(self, width, delta_no, offset_light=0):
        """
        Compute all modes allowed by the guide and sum them into one field.

        Parameters
        ----------
        width : float
            Waveguide width (µm) at 1/e^2 intensity.
        delta_no : float
            Difference of refractive index between the core and the cladding.
        offset_light : float, optional
            Light offset from center in µm. 0 by default.

        Returns
        -------
        field : array
            Sum of all possibles fields in the guide.
        h : array, float
            Transverse propagation constant over x in µm of all modes.
        gamma : array, float
            Extinction coefficient over z in µm of all modes.
        beta : array, float
            Longitudinal constant of propagation over z in µm of all modes.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        nbr_x and the :meth`mode_light` method.
        """
        i = 0
        field = np.zeros(self.nbr_x)
        h = np.array([])
        gamma = np.array([])
        beta = np.array([])

        while True:
            try:
                [field_i, h_m, gamma_m, beta_m] = self.mode_light(
                    width, delta_no, i, offset_light)
                field = field + field_i
                h = np.append(h, h_m)
                gamma = np.append(gamma, gamma_m)
                beta = np.append(beta, beta_m)
                i += 1
            except ValueError:
                break

        return [field, h, gamma, beta]

    def check_modes(self, width, delta_no):
        """
```

(continues on next page)

(continued from previous page)

```
        Return the last possible mode number.
        Could be merged with :meth:`all_modes` but would increase the needed
        time to compute just to display a number.

        Parameters
        ----------
        width : float
            Waveguide width (µm) at 1/e^2 intensity.
        delta_no : float
            Difference of refractive index between the core and the cladding.

        Returns
        -------
        m : int
            Number of the last possible mode for a squared guide.

        Notes
        -----
        This methods uses the :meth`mode_light` method defined in :class:`Bpm`.
        """
        i = 0

        while True:
            try:
                self.mode_light(width, delta_no, i)
                i += 1
            except ValueError:
                print("This guide can propagate up to the modes", i-1)
                return i-1

    def airy_light(self, lobe_size, airy_zero, offset_light=0):
        """
        Create an Airy beam using scipy.special.airy(x).

        Parameters
        ----------
        lobe_size : float
            Size of the first lobe (µm).
        airy_zero : int
            Cut the beam at the asked zero of the Airy function. n lobes will
            be displayed.
        offset_light : float, optional
            Light offset from center in µm. 0 by default.

        Returns
        -------
        field : array
            Amplitude values over x (µm).
        airy_zero : int
            Number of lobes. Corrected if higher than the window size.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        nbr_x, length_x, dist_x, x.
        """
        if lobe_size == 0 or airy_zero == 0:
```

(continues on next page)

(continued from previous page)

```
            return [np.zeros(self.nbr_x), 0]

        lobe_size = -abs(lobe_size)

        # Position of the first zero and the asked one
        zero_pos = special.ai_zeros(airy_zero)[0]
        first_zero = zero_pos[0]
        last_zero = zero_pos[-1]

        # Positions/size of the wanted beam
        left = last_zero * lobe_size / first_zero
        right = 10 * lobe_size / first_zero  # Airy=1e-10 at x=10

        # Reduce the last zero number to stay in the window
        if -left > self.length_x:
            left = zero_pos * lobe_size / first_zero  # All possibles left posi
            airy_zero = np.where(-left <= self.length_x)[0]  # Higher index

            if airy_zero.size == 0:  # interface don't allow this situation
                print("The first lobe is bigger than the windows size")
                return [np.zeros(self.nbr_x), 0]

            else:  # take the higher lobe possible
                airy_zero = int(airy_zero[-1])

            last_zero = zero_pos[airy_zero]  # Value of the last lobe
            airy_zero += 1  # +1 to return the zero number

            left = last_zero * lobe_size / first_zero  # Corrected left positio

        # Number of points in the Airy window to match the full window
        nbr_point = int(round(abs((left - right) / self.dist_x)))

        # Airy window size
        x_airy = np.linspace(last_zero, 10, nbr_point)

        # Positions of the Airy and full window center
        center_airy = int(np.where(x_airy >= 0)[0][0])
        center = int(np.where(self.x >= 0)[0][0])

        # Airy field
        field = np.array(special.airy(x_airy)[0])

        # add last empty field to reach the windows size
        if self.x.size > field.size:
            field = np.append(field, np.zeros((self.x.size-field.size)))

        else:
            field.resize(self.x.size)  # Cut if exceed windows size

        # Recenter on 0
        field = np.roll(field, int(round(center - center_airy)))

        field = np.roll(field, int(round(offset_light / self.dist_x)))
        field /= np.max(field)  # Normalized

        return [field, airy_zero]
```

(continues on next page)

(continued from previous page)

```
    def init_field(self, field, theta_ext, irrad):
        """
        Initialize phase, field and power variables.

        Parameters
        ----------
        field : array, array-like
            Amplitude values for each beams over x (µm) [beam,E] or E
        theta_ext : float
            Exterior inclinaison angle (°).
        irrad : array, array-like
            Irradiance for each beam (:math:`W/m^2`).

        Returns
        -------
        progress_pow : array
            Intensity values over x (µm).

        Notes
        -----
        This method creates the following variables within the class
        :class:`Bpm`:

        - epnc: Convertion factor used to set unit of the field and irradiance.
        - phase_mat: Free propagation in Fourier space over dz/2.
        - current_power: Intensity for z=0.
        - field: Field value with the unit.


        This methods uses the following variables defined in :class:`Bpm`:
        no, x, dist_x, nbr_x, nbr_z_disp.
        """
        assert theta_ext <= 28  # paraxial approximation limitation
        self.field = field.astype(complex)
        # see en.wiki: Gaussian_beam#Mathematical_form for intensity definition
        eta = 376.730313668  # Impedance of free space mu_0*c
        self.epnc = self.no / (2*eta)  # used to converte E into I
        #  unit(epnc)= W/V^2

        try:  # if multiple beams or one beam as [beam]
            _ = self.field.shape[1]  # Raise a IndexError if not
            nbr_light = self.field.shape[0]  # [beam1(x),beam2,beam3] -> 3
#            Eo = sqrt(irrad[i] / self.epnc)  # Peak value of the field (V/m).
            for i in range(nbr_light):
                self.field[i] *= sqrt(irrad[i] / self.epnc)

            self.field = np.sum(self.field, axis=0)  # merge all beam into one

        except IndexError:  # if only one beam and not in form [beam]
            self.field *= sqrt(irrad / self.epnc)

#        https://support.lumerical.com/hc/en-us/articles/
#        360034382894-Understanding-injection-angles-in-broadband-simulations
        theta = asin(sin(radians(theta_ext)) / self.no)  # angle in the guide
        ph = self.no * self.ko * sin(theta) * self.x  # k projection over x
        self.field *= np.exp(1j * ph)  # Initial phase due to angle
```

(continues on next page)

(continued from previous page)

```
        nu_max = 1 / (2*self.dist_x)  # max frequency due to sampling
        # Spacial frequencies over x (1/µm)
        nu = np.linspace(-nu_max,
                         nu_max * (1 - 2/self.nbr_x),
                         self.nbr_x)
        intermed = self.no * cos(theta) / self.lo
        # Linear propagation phase
        fr = 2 * pi * nu**2 / (intermed + np.sqrt(
            intermed**2
            - nu**2
            + 0j))

        # Free space matrix
        self.phase_mat = fftshift(np.exp(-1j * self.dist_z / 2 * fr))

        # Initial irradiance
        self.current_power = self.epnc * abs2(self.field)

        self.progress_pow = np.zeros([self.nbr_z_disp, self.nbr_x])
        self.progress_pow[0, :] = np.array([self.current_power])

        return [self.progress_pow]

    def guide_position(self, peaks, guide, size):
        """
        Return the left and right position index over x of a given guide
        for each z.

        Parameters
        ----------
        peaks : array-like
            Central position of each guide [guide,z].
        guide : int
            Number of the guide.
        size : float
            Width (µm).

        Returns
        -------
        x_beg : array
            Left indices position of the selected guide over z.
        x_end : array
            Right indices position of the selected guide over z.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        nbr_z, x, length_x.
        """
        x_beg = np.array([None]*self.nbr_z)
        x_end = np.array([None]*self.nbr_z)

        if peaks.shape[0] != 0:

            for j in range(self.nbr_z):
                if peaks[guide, j] is None:
```

(continues on next page)

(continued from previous page)

```
                continue
            pos_beg = (peaks[guide, j] - size/2)  # Left position

            # If the position is out of boundery, change interval to
            # (-length_x/2, length_x)
            if pos_beg < self.x[0] or pos_beg > self.x[-1]:
                pos_beg = pos_beg % self.length_x

            # If the pos_beg is between length_x/2 and length_x then change
            # interval to (-length_x/2, length_x/2)
            if pos_beg >= self.x[-1]:
                pos_beg -= self.length_x

            # Search the closest index value for this position
            x_beg[j] = np.where(self.x >= pos_beg)[0][0]

            pos_end = (peaks[guide, j] + size/2)

            if pos_end < self.x[0] or pos_end > self.x[-1]:
                pos_end = pos_end % self.length_x

            if pos_end >= self.x[-1]:
                pos_end -= self.length_x

            x_end[j] = np.where(self.x >= pos_end)[0][0]
    return [x_beg, x_end]

def power_guide(self, x_beg, x_end):
    """
    return the power over z in a given interval by integrating the beam
    irradiance.

    Parameters
    ----------
    x_beg : array
        Left indices position over z for a selected guide.
    x_end : array
        Right indices position over z for a selected guide.

    Returns
    -------
    P : array
        Normalized power in the guide over z.

    Notes
    -----
    This methods uses the following variables defined in :class:`Bpm`:
    nbr_z_disp, progress_pow, pas.
    """
    P = np.zeros(self.nbr_z_disp)
    # explaination: power[0] is input so take dn[0] but dn[0] is for propag
    # from 0 to 1 unit so next power power[1] is also dn[0]
    for i in range(self.nbr_z_disp):
        if i == 0:
            index = 0
        elif i == self.nbr_z_disp-1:
            # -1 for beginning at 0 and -1 for final useless value
```

(continues on next page)

(continued from previous page)

```
                index = len(x_beg)-2
            else:
                index = i*self.pas-1

            if x_beg[index] is None or x_end[index] is None:
                continue

            if x_beg[index] <= x_end[index]:
                P[i] = np.trapz(
                    self.progress_pow[i, x_beg[index]:x_end[index]],
                    dx=self.dist_x*1e-6)

            else:  # Take into account guides that crosses the window edges
                P[i] = np.trapz(
                    self.progress_pow[i, x_beg[index]:],
                    dx=self.dist_x*1e-6)
                P[i] += np.trapz(
                    self.progress_pow[i, :x_end[index]],
                    dx=self.dist_x*1e-6)

        P /= np.trapz(self.progress_pow[0], dx=self.dist_x*1e-6)
        return P  # f not normalized, unit: (W/m)

    def kerr_effect(self, dn, n2=None, chi3=None, kerr_loop=1,
                    variance_check=False, field_start=None,
                    dn_start=None, phase_mat=None):
        """
        Kerr effect: refractive index modulation by the light intensity.
        See: https://optiwave.com/optibpm-manuals/bpm-non-linear-bpm-algorithm/

        Parameters
        ----------
        dn : array
            Difference of refractive index as dn[z,x]. Can contain the losses
            throught the imaginary part.
        n2 : float, optional
            Nonlinear refractive index responsable for the optical Kerr effect
            in m^2/W. None by default.
        chi3 : float, optional
            Value of the third term of the electric susceptibility tensor
            in m^2/V^2. None by default.
        kerr_loop : int, optional
            Number of corrective loops for the Kerr effect. 1 by default.
        variance_check : bool, optional
            Check if the kerr effect converge fast enought. False by default.
        field_start : array, optional
            Field without kerr effect.
            If None were given, take the :meth:`main_compute` field.
        dn_start : array, optional
            Refractive index without kerr effect.
            If None were given, take the :meth:`main_compute` dn.
        phase_mat: array, optional
            Free propagation in Fourier space over dz/2.
            If None were given, take the :meth:`main_compute` phase_mat.

        Returns
        -------
```

(continues on next page)

(continued from previous page)

```
        dn : array
            Refractive index with kerr effect.
        nl_mat : array
            refractive index modulation with kerr effect.
        field_x : array
            Field with the kerr effect at the self.i step.
        cur_pow : array
            Beam power with the kerr effect after the dz propagation.

        Notes
        -----
        This methods uses the following variables defined in :class:`Bpm`:
        i, epnc, no, ko, dist_z and the :meth:`variance` method.
        """
        assert n2 is None or chi3 is None
        # assert n2 is not None or chi3 is not None
        # Set the default value if none were given
        dn_start = dn[self.i, :] if dn_start is None else dn_start
        nl_mat = self.ko * self.dist_z * dn_start
        field_start = self.field if field_start is None else field_start
        phase_mat = self.phase_mat if phase_mat is None else phase_mat

        # Influence of the index modulation on the field
        field_x = field_start * np.exp(1j * nl_mat)

        # Linear propagation over dz/2
        field_x = ifft(phase_mat * fft(field_x))
        cur_pow = self.epnc * abs2(field_x)

        for _ in range(kerr_loop):
            prev_pow = cur_pow

            # influence of the beam intensity on the index modulation
            if n2 is not None:  # dn = dn1+dn2*I with I unit: W/m^2
                dn_kerr = dn_start + n2*prev_pow

            elif chi3 is not None:  # dn = dn1+ 3chi3/8*no*|E|^2
                dn_kerr = dn_start + (3*chi3)/(8*self.no)*(prev_pow/self.epnc)
            else:
                dn_kerr = dn_start  # identical to no kerr effect but slower

            nl_mat = self.ko * self.dist_z * dn_kerr

            # influence of the index modulation on the field
            field_x = field_start * np.exp(1j * nl_mat)

            # Linear propagation over dz/2
            field_x = ifft(phase_mat * fft(field_x))

            # power at pos z
            cur_pow = self.epnc * abs2(field_x)

        if variance_check:
            try:
                self.variance(prev_pow, cur_pow)  # Check if converge
            except ValueError as ex:
                print(ex)
```

(continues on next page)

(continued from previous page)

```
                print("for the step i=", self.i)

            if np.max(dn_kerr) > self.no/10:
                print("Careful: index variation too high:",
                      "\t%.2f > %f/10" % (np.max(dn_kerr), self.no), sep="\n")

        return [dn_kerr, nl_mat, field_x, cur_pow]

    def variance(self, initial, final):
        """
        This function alerts the user when the kerr effect don't converge fast
        enough.
        Raise a ValueError when the power standard deviation exceed
        :math:`10^{-7}`.

        Parameters
        ----------
        initial : array
            Power of the kerr effect looped n-1 time.
        final : array
            Power of the kerr effect looped n time.

        Raises
        ------
        ValueError
            when the power standard deviation exceed :math:`10^{-7}`.
        """
        finish_sum = np.sum(final)
        nl_control_amp = 1/finish_sum * np.sqrt(np.abs(
            np.sum(np.subtract(final, initial)**2)
            - np.sum(np.subtract(final, initial))**2))

        if nl_control_amp > 1e-7:
            message = "Warning: solution doesn't converge " + \
                "for a deviation of " + str(nl_control_amp)
            raise ValueError(message)

    def bpm_compute(self, dn, n2=None, chi3=None, kerr_loop=1,
                    variance_check=False):
        """
        Compute BPM principle : free_propag over dz/2, index modulation,
        free_propag over dz/2.

        Parameters
        ----------
        n2 : float, optional
            Nonlinear refractive index responsable for the optical Kerr effect
            in m^2/W. None by default.
        chi3 : float, optional
            Value of the third term of the electric susceptibility tensor
            in m^2/V^2. None by default.
        kerr : bool, optional
            Activate the kerr effect. False by default.
        kerr_loop : int, optional
            Number of corrective loops for the Kerr effect. 1 by default.
        variance_check : bool
            Check if the kerr effect converge fast enought. False by default.
```

(continues on next page)

(continued from previous page)

```
        Returns
        -------
        current_power : array
            Power after the propagation over dz.

        Notes
        -----
        This method uses the :class:`Bpm` class variables:
        nbr_lost, i, field, dist_z, dn, nl_mat, phase_mat, epnc,
        :meth:`kerr_effect`.

        This method change the values of:
        field, dn, nl_mat, current_power.
        """
        # Linear propagation over dz/2
        self.field = ifft(self.phase_mat * fft(self.field))

        if n2 is not None or chi3 is not None:
            [dn[self.i, :], self.nl_mat[self.i, :],
             self.field, self.current_power] = self.kerr_effect(
                 dn, n2=n2, chi3=chi3, kerr_loop=kerr_loop,
                 variance_check=variance_check)
        else:
            # Influence of the index modulation on the field (contain losses)
            self.field *= np.exp(1j * self.nl_mat[self.i, :])

            # Linear propagation over dz/2
            self.field = ifft(self.phase_mat * fft(self.field))

            # power at z
            self.current_power = self.epnc * abs2(self.field)

        # useless but act as a reminder for what the the method does
        return self.current_power

    def main_compute(self, dn, n2=None, chi3=None, kerr_loop=1,
                     variance_check=False, disp_progress=True):
        """
        main method used to compute propagation.

        Parameters
        ----------
        n2 : float, optional
            Nonlinear refractive index responsable for the optical Kerr effect
            in m^2/W. None by default.
        chi3 : float, optional
            Value of the third term of the electric susceptibility tensor
            in m^2/V^2. None by default.
        kerr : bool, optional
            Activate the kerr effect. False by default.
        kerr_loop : int, optional
            Number of corrective loop for the Kerr effect. 1 by default.
        variance_check : bool, optional
            Check if the kerr effect converge fast enought. False by default.
        alpha : float, optional
            Absorption per µm. 0 by default
```

(continues on next page)

(continued from previous page)

```
        lost_beg : array-like, optional
            Left indices position of the selected waveguide over z.
            None by default.
        lost_end : array-like, optional
            Right indices position of the selected waveguide over z.
            None by default.

        Returns
        -------
        progress_pow : array
            Intensity values (:math:`W/m^2`) over x (µm) and z (µm).

        Notes
        -----

        This method creates the following variables within the class
        :class:`Bpm`:
        nl_mat: Refractive index modulation.

        This method uses the :class:`Bpm` class variables:
        phase_mat, field, i, nbr_z, pas, current_power, dist_z, length_z,
        nbr_lost, dn, nl_mat, epnc and uses the :meth:`bpm_compute`,
        :meth:`kerr_effect`.

        This method change the values of the :class:`Bpm` class variables:
        field and if kerr, dn and nl_mat.
        """
        # Refractive index modulation
        self.nl_mat = self.ko * self.dist_z * dn

        index = 0
        self.i = 0
        #  from i=0 to i=final-1 because don't use last dn
        for i in range(self.nbr_z-1):
            self.i = i
            # Compute non-linear and linear propagation for every z
            self.bpm_compute(dn, n2=n2, chi3=chi3, kerr_loop=kerr_loop,
                             variance_check=variance_check)

            # Display condition: if i+1 is a multiple of pas: i+1 % pas = 0
            # = False, so must use if not to have True
            # last condition to have last point if not a multiple of pas
            if not (self.i + 1) % self.pas or self.i+1 == self.nbr_z-1:
                index += 1
                self.progress_pow[index, :] = np.array([self.current_power])

                if disp_progress:
                    current = (self.i+1)*self.dist_z/1e3
                    print(current, "/", self.length_z/1e3, 'mm')

        return [self.progress_pow]


def example_bpm():
    """
    Version allowing to compute BPM without the user interface.
    Used for quick test.
```

(continues on next page)

(continued from previous page)

```
    """
    lo = 1.5
    width = 6
    no = 2.14
    # losses = 0.8/1e3
    # no_imag = 1.9e-4
    # no_imag = losses/(2*pi/lo)
    delta_no = 0.001  # + 1j*no_imag
    length_z = 2000
    dist_z = 1
    nbr_z_disp = 200
    dist_x = 0.1
    length_x = 500

    bpm = Bpm(no, lo,
              length_z, dist_z, nbr_z_disp,
              length_x, dist_x)

    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()

#    shape = bpm.squared_guide(width)
    shape = bpm.gauss_guide(width, 4)

    nbr_p = 3
    p = 13
    offset_guide = 0

    [peaks, dn] = bpm.create_guides(
        shape, delta_no, nbr_p, p, offset_guide=offset_guide)
#    curve = 40 * 1E-8
#    half_delay = 1000
#    distance_factor = 1.2
#    [peaks, dn] = bpm.create_curved_guides(shape, width, delta_no,
#                                           curve, half_delay,
#                                           distance_factor,
#                                           offset_guide=offset_guide)

    z_disp = np.linspace(0, length_z/1000, nbr_z_disp+1)
    xv, zv = np.meshgrid(x, z_disp)
    dn_disp = np.linspace(0, nbr_z-1, nbr_z_disp+1, dtype=int)

#    plt.figure()
#    for i in range(nbr_z_disp+1):
#        plt.plot(x,dn[i,:])

    plt.figure()
    plt.title("Refractive index over (x,z)")
    plt.xlabel('x (µm)')
    plt.ylabel('z (mm)')
    plt.pcolormesh(xv,
                   zv,
                   dn[dn_disp].real,
                   cmap='gray')

    # ax1 = plt.subplot()
    # ax1.set_title("Losses map")
    # ax1.set_xlabel('x (µm)')
```

(continues on next page)

(continued from previous page)

```
    # ax1.set_ylabel('z (mm)')
    # ax1.pcolormesh(xv, zv, dn[dn_disp].imag)

    fwhm = 8
    offset_light = peaks[0, 0]  # If guide exists
    offset_light = 0  # Else

    nbr_light = 1
    field = np.array([np.zeros(nbr_x)] * nbr_light)
    for i in range(nbr_light):
        field_i = bpm.gauss_light(fwhm, offset_light=offset_light)

    #    field_i = bpm.squared_light(fwhm, offset_light=offset_light)

#            [field_i, h, gamma, beta] = bpm.all_modes(
#                width, delta_no
#                offset_light=offset_light)

#        mode = 0
#            [field_i, h, gamma, beta] = bpm.mode_light(
#                width, delta_no,
#                mode, offset_light=offset_light)

        field[i] = field_i

    irrad = [1E13]*nbr_light
    theta_ext = 0
    [progress_pow] = bpm.init_field(field, theta_ext, irrad)

    def _show_plot(pow_index):
        plt.figure()
        ax1 = plt.subplot(111)
        if pow_index == 0:
            ax1.set_title("Light injection into a guide")
        else:
            ax1.set_title("Light at the end of guides")
        ax1.set_xlabel('x (µm)')
        ax2 = ax1.twinx()
        for tl in ax1.get_yticklabels():
            tl.set_color('k')
        for tl in ax2.get_yticklabels():
            tl.set_color('#1f77b4')
        ax1.set_ylabel(r'$\Delta_n$')
        ax2.set_ylabel('Irradiance ($GW.cm^{-2}$)')

        if nbr_p != 0 and p != 0:
            ax1.set_xlim(-nbr_p*p, nbr_p*p)
            verts = [(x[0], 0),
                     *zip(x, dn[pow_index, :].real),
                     (x[-1], 0)]
            poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
            ax1.add_patch(poly)
        ax1.set_ylim(0,
                     max(dn[0, :].real)*1.1 + 1E-20
                     )
        if max(progress_pow[0]) != 0:
            ax2.set_ylim(0,  1.1e-13*max(progress_pow[0]))
```

(continues on next page)

(continued from previous page)

```
        ax1.plot(x, dn[pow_index].real, 'k')
        ax2.plot(x, 1e-13*progress_pow[pow_index], '#1f77b4')
        plt.show()

    pow_index = 0
    print("May need to close the graph to continue.")
    _show_plot(pow_index)

    kerr_loop = 5
    variance_check = False
#    n2 = 1e-16
#    chi3 = 10 * 1E-22
    n2 = None
    chi3 = None

    kerr = n2 is not None or chi3 is not None

    estimation = round(
        8.8 / 5e7 * nbr_z * nbr_x  # without kerr
        * (1 + 0.72*float(kerr)*(kerr_loop))  # with kerr
        + 3.8e-8*nbr_z*nbr_x*float(variance_check),  # control
        1)

    print("Time estimate:", str(estimation))

    debut = time.process_time()
    [progress_pow] = bpm.main_compute(
        dn,
        n2=n2, chi3=chi3, kerr_loop=kerr_loop,
        variance_check=variance_check, disp_progress=False)
    fin = time.process_time()
    print('time elapsed:', fin-debut)

    plt.figure()
    ax1 = plt.subplot(111)
    ax1.set_title("Light propagation into guides")
    ax1.set_xlabel('x (µm)')
    ax1.set_ylabel('z (mm)')
    if nbr_p != 0 and p != 0:
        ax1.set_xlim(-nbr_p*p, nbr_p*p)
    ax1.pcolormesh(xv, zv, 1e-13*progress_pow)

    pow_index = -2
    print("May need to close the graphs to continue.")
    _show_plot(pow_index)
    print("Finished")


if __name__ == "__main__":
    print("version without the user interface, note that user_interface.py")
    print("calls the Bpm class and performs the same calculations")

    choice = input("Start ?: ")

    if not choice != "yes":
        example_bpm()
```

5.4.2 beampy.user_interface

```
"""
The user_interface module is the main file of the beampy module.
It contain the UserInterface class used to computed and displayed the bpm
results onto the interface.

This module was at first developed by Marcel Soubkovsky for the implementation
of the array of guides, of one gaussian beam and of the plotting methods.
Then, continued by Jonathan Peltier.
"""
import sys
import os
import webbrowser
import time
from math import pi
import numpy as np

from PyQt5.QtWidgets import (QApplication, QMainWindow, QFileDialog)
from PyQt5.QtCore import (pyqtSlot, Qt, QSize)
from PyQt5.QtGui import QIcon

from matplotlib.figure import Figure
from matplotlib.backends.backend_qt4agg import (
    FigureCanvasQTAgg as FigureCanvas,
    NavigationToolbar2QT as NavigationToolbar)
from matplotlib.patches import Polygon

# interface.py generated from interface.ui made using Qt designer
from beampy.interface import Ui_MainWindow
from beampy.bpm import Bpm  # Bpm class with all the BPM methods

# ! To translate from .ui to .py -> pyuic5 -x interface.ui -o interface.py

# Check out if doubts on the interface:
# http://blog.rcnelson.com/building-a-matplotlib-gui-with-qt-designer-part-1/


class UserInterface(QMainWindow, Ui_MainWindow):
    """
    This class connect the :class:`.Bpm` class from the :mod:`.bpm` module
    with the :meth:`.setupUi` method from the :mod:`.interface` module.
    The interface seems to close itself when openning from spyder 3.5.
    In that case, open the interface in an external console.
    """
    def __init__(self):
        QMainWindow.__init__(self)
        self.setupUi(self)  # Linked to Ui_MainWindow through interface.py

        # Prepare variables before assigning values to them
        self.width = []
        self.offset_guide = []
        self.guide_length = []
        self.offset_guide_z = []
        self.delta_no = []
        self.no_imag = []
        self.alpha = []
```

(continues on next page)

(continued from previous page)

```
        self.loss_check = []
        self.full_index = []
        self.shape_gauss_check = []
        self.gauss_pow = np.array([], dtype=int)
        self.shape_squared_check = []
        self.nbr_p = np.array([], dtype=int)
        self.p = []
        self.curve = []
        self.half_delay = []
        self.distance_factor = []
        self.tab_index = []
        self.previous_guide = 0

        self.fwhm = []
        self.offset_light = []
        self.irrad = []
        self.irrad_significand = []
        self.irrad_exponent = []  # Warning: if exponent is set to int, will
        # cause a error when using 10**exponent because outside int limit and
        # goes in the negative values
        self.offset_check = []
        self.gaussian_check = []
        self.square_check = []
        self.mode_check = []
        self.all_modes_check = []
        self.mode = np.array([], dtype=int)
        self.mode_guide_ref = np.array([], dtype=int)
        self.offset_light_peak = np.array([], dtype=int)
        self.airy_check = []
        self.airy_zero = np.array([], dtype=int)
        self.lobe_size = []
        self.previous_beam = 0

        self.on_click_create_guide()  # Initialized variables with buttons
        self.on_click_create_light()  # Initialized variables with buttons

        self.calculate_guide()  # Compute waveguides
        self.calculate_light()  # Compute light

        self.addmpl('guide')  # Display waveguides
        self.addmpl('light')  # Display light

        self.show_estimate_time()

        # Initialized compute graphics
        self.canvas_propag = FigureCanvas(Figure())
        self.plot_compute.addWidget(self.canvas_propag)
        self.toolbar_propag = FigureCanvas(Figure())
        self.plot_compute.addWidget(self.toolbar_propag)
        self.toolbar_propag.close()
        self.canvas_pow = FigureCanvas(Figure())
        self.verticalLayout_compute_plot.addWidget(self.canvas_pow)
        self.toolbar_pow = FigureCanvas(Figure())
        self.verticalLayout_compute_plot.addWidget(self.toolbar_pow)
        self.canvas_end = FigureCanvas(Figure())
        self.plot_compute.addWidget(self.canvas_end)
        self.toolbar_end = FigureCanvas(Figure())
```

(continues on next page)

(continued from previous page)

```
        self.plot_compute.addWidget(self.toolbar_end)

        self.filename = None

        self.connect_buttons()
        self.create_menu()

        QApplication.setOverrideCursor(Qt.ArrowCursor)
        QApplication.restoreOverrideCursor()

    def connect_buttons(self):
        """
        Connect the interface buttons to their corresponding functions:

        :meth:`.on_click_guide`, :meth:`.on_click_curved`,
        :meth:`.on_click_light`, :meth:`.on_click_compute`,
        :meth:`.on_click_create_light`, :meth:`.on_click_delete_light`,
        :meth:`.save_light`, :meth:`.get_guide`,
        :meth:`.get_light`, :meth:`.get_compute`,
        :meth:`.show_estimate_time`, :meth:`.check_modes_display`.
        """
        self.calculateButton_guide.clicked.connect(self.on_click_guide)
        self.calculateButton_light.clicked.connect(self.on_click_light)
        self.calculateButton_compute.clicked.connect(self.on_click_compute)
        self.pushButton_create_beam.clicked.connect(self.on_click_create_light)
        self.pushButton_delete_beam.clicked.connect(self.on_click_delete_light)
        self.pushButton_save_beam.clicked.connect(self.save_light)
        self.pushButton_create_guide.clicked.connect(
            self.on_click_create_guide)
        self.pushButton_delete_guide.clicked.connect(
            self.on_click_delete_guide)
        self.pushButton_save_guide.clicked.connect(self.save_guide)
        self.pushButton_cancel_guide.clicked.connect(self.get_guide)
        self.pushButton_cancel_light.clicked.connect(self.get_light)
        self.pushButton_cancel_compute.clicked.connect(self.get_compute)
        self.pushButton_estimate_time.clicked.connect(self.show_estimate_time)
        self.pushButton_mode_number.clicked.connect(self.check_modes_display)
        self.pushButton_power.clicked.connect(self.display_power)

    def create_menu(self):
        """
        Create a menu to open, save a file, or exit the app.

        Notes
        -----
        This method connect the following methods and function to the
        menu buttons:

        :meth:`.open_file_name`, :meth:`.save_quick`, :meth:`.save_file_name`,
        :func:`.open_doc`.
        """
        folder = __file__  # Module name
        # Replaces characters only when called from outer files
        folder = folder.replace("\\", "/")
        folder = folder.split("/")
        folder = folder[:-1]  # Remove the file name
        folder2 = str()
```

(continues on next page)

(continued from previous page)

```
        for line in folder:
            folder2 = folder2+"/"+line

        folder = folder2[1:]+"/"

        icon = QIcon()
        icon.addFile(folder+'icons/beampy-logo.png', QSize(256, 256))
        self.setWindowIcon(icon)

        menubar = self.menuBar()

        file = menubar.addMenu('File')

        action = file.addAction('Open')
        action.triggered.connect(self.open_file_name)
        action.setShortcut('Ctrl+O')
        icon = QIcon()
        icon.addFile(folder+'icons/document-open.png', QSize(22, 22))
        action.setIcon(icon)

        action = file.addAction('Save')
        action.triggered.connect(self.save_quick)
        action.setShortcut('Ctrl+S')
        icon = QIcon()
        icon.addFile(folder+'icons/document-save.png', QSize(22, 22))
        action.setIcon(icon)

        action = file.addAction('Save as')
        action.triggered.connect(self.save_file_name)
        action.setShortcut('Ctrl+Shift+S')
        icon = QIcon()
        icon.addFile(folder+'icons/document-save-as.png', QSize(22, 22))
        action.setIcon(icon)

        action = file.addAction('Exit')  # Clean exit for spyder
        action.setShortcut('Ctrl+Q')
        action.triggered.connect(QApplication.quit)
        icon = QIcon()
        icon.addFile(folder+'icons/application-exit.png', QSize(22, 22))
        action.setIcon(icon)

        file = menubar.addMenu('Help')

        action = file.addAction('Documentation')
        action.triggered.connect(open_doc)
        icon = QIcon()
        icon.addFile(folder+'icons/help-about.png', QSize(22, 22))
        action.setIcon(icon)

    def calculate_guide(self):
        """
        Initialized the :class:`.Bpm` class and creates the guides.

        Notes
        -----
        Creats many variables, including :
```

(continues on next page)

(continued from previous page)

```
        - peaks : Central position of each waveguides [waveguide,z].
        - dn : Difference of reefractive index [z,x].

        This method calls the following methods from :class:`.Bpm`:

        :meth:`.create_x_z`, :meth:`.squared_guide`, :meth:`.gauss_guide`,
        :meth:`.create_guides`, :meth:`.create_curved_guides`.
        """
        print("Waveguides creation")
        t_guide_start = time.process_time()

        self.save_guide()  # Get all waveguide values

        # Create the Bpm class (overwrite existing one)
        self.bpm = Bpm(self.no, self.lo,
                       self.length_z, self.dist_z, self.nbr_z_disp,
                       self.length_x, self.dist_x)

        # windows variables
        [self.length_z, self.nbr_z, self.nbr_z_disp,
         self.length_x, self.nbr_x, self.x] = self.bpm.create_x_z()

        self.doubleSpinBox_length_x.setValue(self.length_x)
        self.doubleSpinBox_length_z.setValue(self.length_z)
        self.spinBox_nbr_z_disp.setValue(self.nbr_z_disp)

        if (self.nbr_x * self.nbr_z_disp) > (5000 * 1000):
            print("Error: if you want to have more points,")
            print("change the condition nbr_x * nbr_z_disp in calculate_guide")
            raise RuntimeError("Too many points:", self.nbr_x*self.nbr_z_disp)

        if (self.nbr_x * self.nbr_z) > (10000 * 15000) or (
                self.nbr_z > 40000 or self.nbr_x > 40000):
            print("Error: if you want to have more points,")
            print("change the condition nbr_x * nbr_z in calculate_guide")
            raise RuntimeError("Too many points:", self.nbr_x*self.nbr_z)

        nbr_guide = self.comboBox_guide.count()  # Total waveguide count

        self.peaks = np.zeros((0, self.nbr_z))
        self.dn = np.zeros((self.nbr_z, self.nbr_x))

        for i in range(nbr_guide):
            # Waveguide shape choice
            if self.shape_squared_check[i]:  # Squared waveguides
                shape = self.bpm.squared_guide(self.width[i])
            elif self.shape_gauss_check[i]:  # Gaussian waveguides
                shape = self.bpm.gauss_guide(self.width[i],
                                             gauss_pow=self.gauss_pow[i])

            # Topology waveguides choice
            if self.tab_index[i] == 0:  # array of waveguides
                length_max = self.length_x - self.dist_x

                if self.nbr_p[i]*self.p[i] > length_max:
                    # give info about possible good values
```

(continues on next page)

(continued from previous page)

```
                print("nbr_p*p > length_x: ")
                print(self.nbr_p[i]*self.p[i], ">", length_max)

                print("p_max=", round(length_max/self.nbr_p[i], 3))

                if int(length_max / self.p[i]) == length_max / self.p[i]:
                    print("nbr_p_max=", int(length_max / self.p[i])-1)
                else:
                    print("nbr_p_max=", int(length_max / self.p[i]))

                self.nbr_p[i] = 0

            [peaks, dn] = self.bpm.create_guides(
                shape, self.full_index[i],
                self.nbr_p[i], self.p[i],
                offset_guide=self.offset_guide[i],
                z=[self.offset_guide_z[i],
                   self.offset_guide_z[i]+self.guide_length[i]])

        elif self.tab_index[i] == 1:  # curved waveguides
            curve = self.curve[i] * 1E-8  # curvature factor
            [peaks, dn] = self.bpm.create_curved_guides(
                shape, self.width[i], self.full_index[i],
                curve, self.half_delay[i], self.distance_factor[i],
                offset_guide=self.offset_guide[i])

        if self.delta_no[i] > self.no/10:
            print("Careful: index variation too high for waveguide %i" % i,
                  "\t%f > %f/10" % (self.delta_no[i], self.no), sep="\n")

        self.peaks = np.append(self.peaks, peaks, 0)
        self.dn = np.add(self.dn, dn)
        self.dn_first = np.array(self.dn[0])

        # Display Waveguides
        self.z_disp = np.linspace(0,
                                  self.length_z/1000,
                                  self.nbr_z_disp+1)
        # Note that z_disp can be wrong by 0.4% if the space between the
        # displayed points is not a multiple of length_z

        self.xv, self.zv = np.meshgrid(self.x, self.z_disp)
        self.dn_disp = np.linspace(0,
                                   self.nbr_z-1,
                                   self.nbr_z_disp+1, dtype=int)

    self.pushButton_power.setDisabled(True)
    # only display available settings
    if (self.nbr_p.sum() == 0 or self.p.sum() == 0
            or self.guide_length.sum() == 0):
        # If one of the waveguides has width=0, can chose invisible
        # waveguide has beam reference and display power=0. Not a big deal
        self.spinBox_offset_light_peak.setDisabled(True)
        self.spinBox_guide_nbr_ref.setDisabled(True)
        self.spinBox_guide_nbr_ref.setMaximum(0)
        self.doubleSpinBox_offset_light.setEnabled(True)
        self.checkBox_offset_light.setChecked(False)
```

(continues on next page)

(continued from previous page)

```
            self.checkBox_offset_light.setDisabled(True)
            self.offset_check *= 0
            self.checkBox_power.setDisabled(True)

        else:
            self.checkBox_offset_light.setEnabled(True)
            self.spinBox_offset_light_peak.setMaximum(self.peaks.shape[0]-1)
            self.spinBox_guide_nbr_ref.setMaximum(self.peaks.shape[0]-1)
            self.checkBox_power.setEnabled(True)

#         Define new min/max for light and looses, based on selected guides
        self.doubleSpinBox_lobe_size.setMaximum(self.length_x-self.dist_x)
        # If want to use min/max for offset: multiple beams will have issue if
        # the windows size change (min/max will only be for the displayed beam)
#         self.doubleSpinBox_offset_light.setMinimum(self.x[0])
#         self.doubleSpinBox_offset_light.setMaximum(self.x[-1])

        self.calculate_guide_done = True
        t_guide_end = time.process_time()
        print('Waveguides creation time: ', t_guide_end-t_guide_start)

    def calculate_light(self):
        """
        Create the choosen beams.

        Notes
        -----
        Creates the progress_pow variable.

        This method calls the following methods from the :class:`.Bpm` class:

        :meth:`.gauss_light`, :meth:`.squared_light`, :meth:`.all_modes`,
        :meth:`.mode_light`, :meth:`.airy_light`, :meth:`.init_field`.
        """
        print("Beams creation")
        t_light_start = time.process_time()

        self.save_light()  # Get all light variables
        # must have same wavelength and angle or must compute for each
        # different wavelength or angle
        nbr_light = self.comboBox_light.count()  # Number of beams
        field = np.zeros((nbr_light, self.nbr_x))

        if 1 in self.all_modes_check:  # Display only once the max mode
            self.check_modes_display()

        for i in range(nbr_light):

            # Check if offset relative to waveguide number or else in µm
            if self.offset_check[i] and self.peaks.shape[0] != 0:

                # Reduce light # if > guides #
                peaks_i = self.offset_light_peak[i]
                peaks_max = self.spinBox_offset_light_peak.maximum()

                if peaks_i > peaks_max:
                    print("beam", i, "has a non-existing waveguide position")
```

(continues on next page)

(continued from previous page)

```
                print("Change position from", peaks_i, "to", peaks_max)
                self.offset_light_peak[i] = peaks_max

            temp = np.array(self.peaks[self.offset_light_peak[i]])
            temp = temp[temp != np.array(None)]

            if len(temp) != 0:
                offset_light = temp[0]
            else:
                print("There is no waveguide n°",
                      self.offset_light_peak[i],
                      "The beam n°", i, "is disabled")
                continue

        else:
            offset_light = self.offset_light[i]

        guide_index = self.find_guide_number(self.mode_guide_ref[i])

        if guide_index is None and (self.mode_check[i]
                                    or self.all_modes_check[i]):
            print("skipping the beam", i, "because the waveguide",
                  self.mode_guide_ref[i], "doesn't exist")
            continue

        #  Compute lights
        if self.gaussian_check[i]:
            field_i = self.bpm.gauss_light(
                self.fwhm[i], offset_light=offset_light)

        elif self.square_check[i]:
            field_i = self.bpm.squared_light(
                self.fwhm[i], offset_light=offset_light)

        elif self.all_modes_check[i]:

            field_i = self.bpm.all_modes(
                self.width[guide_index], self.delta_no[guide_index],
                offset_light=offset_light)[0]

        elif self.mode_check[i]:

            try:
                field_i = self.bpm.mode_light(
                    self.width[guide_index], self.delta_no[guide_index],
                    self.mode[i], offset_light=offset_light)[0]

            except ValueError as ex:  # Say that no mode exist
                print(ex, "for the beam", i)
                continue  # Go to the next field

        elif self.airy_check[i]:
            [field_i, last_zero_pos] = self.bpm.airy_light(
                self.lobe_size[i], self.airy_zero[i],
                offset_light=offset_light)
            self.spinBox_airy_zero.setValue(last_zero_pos)  # Corrected val
```

(continues on next page)

(continued from previous page)

```
            field[i] = field_i

        [self.progress_pow] = self.bpm.init_field(
            field, self.theta_ext, self.irrad)

#         self.pushButton_power.setDisabled(True)

        t_light_end = time.process_time()
        print('Beams creation time: ', t_light_end-t_light_start)

    def calculate_propagation(self):
        """
        Calculate the propagation based on the input light and guides shapes.

        Notes
        -----
        Creates the progress_pow variable.

        Calls the following methods from :class:`.Bpm`:
        :meth:`.losses_position`, meth`.main_compute`.
        """
        print("Propagation computation")
        t_compute_start = time.process_time()
        self.calculate_guide_done = False

        self.show_estimate_time()  # do also save_compute() needed here

        if self.kerr_check:
            if self.n2_check:
                n2 = self.n2
                chi3 = None
            else:
                n2 = None
                chi3 = self.chi3
        else:
            n2 = None
            chi3 = None

        [self.progress_pow] = self.main_compute(
            self.dn, n2=n2, chi3=chi3, kerr_loop=self.kerr_loop,
            variance_check=self.variance_check, disp_progress=False)

        self.progressBar_compute.setValue(0)

        if (self.nbr_p.sum() != 0 and self.p.sum() != 0
                and self.guide_length.sum() != 0):
            self.pushButton_power.setEnabled(True)

        t_compute_end = time.process_time()
        print('Computation time: ', t_compute_end-t_compute_start)

    def main_compute(self, dn, n2=None, chi3=None, kerr_loop=1,
                     variance_check=False, disp_progress=True):
        """
        main method used to compute propagation.

        Parameters
```

(continues on next page)

(continued from previous page)

```
        ----------
        n2 : float, optional
            Nonlinear refractive index responsable for the optical Kerr effect
            in m^2/W. None by default.
        chi3 : float, optional
            Value of the third term of the electric susceptibility tensor
            in m^2/V^2. None by default.
        kerr : bool, optional
            Activate the kerr effect. False by default.
        kerr_loop : int, optional
            Number of corrective loop for the Kerr effect. 1 by default.
        variance_check : bool, optional
            Check if the kerr effect converge fast enought. False by default.
        alpha : float, optional
            Absorption per µm. 0 by default
        lost_beg : array-like, optional
            Left indices position of the selected waveguide over z.
            None by default.
        lost_end : array-like, optional
            Right indices position of the selected waveguide over z.
            None by default.

        Returns
        -------
        progress_pow : array
            Intensity values (:math:`W/m^2`) over x (µm) and z (µm).

        Notes
        -----

        This method creates the following variables within the class
        :class:`Bpm`:
        nl_mat: Refractive index modulation.

        This method uses the :class:`Bpm` class variables:
        phase_mat, field, i, nbr_z, pas, current_power, dist_z, length_z,
        nbr_lost, dn, nl_mat, epnc and uses the :meth:`bpm_compute`,
        :meth:`kerr_effect`.

        This method change the values of the :class:`Bpm` class variables:
        field and if kerr, dn and nl_mat.
        """
        # Refractive index modulation
        self.bpm.nl_mat = self.bpm.ko * self.bpm.dist_z * dn

        index = 0
        self.bpm.i = 0
        #  from i=0 to i=final-1 because don't use last dn
        for i in range(self.bpm.nbr_z-1):
            self.bpm.i = i
            # Compute non-linear and linear propagation for every z
            self.bpm.bpm_compute(dn, n2=n2, chi3=chi3, kerr_loop=kerr_loop,
                                 variance_check=variance_check)

            # Display condition: if i+1 is a multiple of pas: i+1 % pas = 0
            # = False, so must use if not to have True
            # last condition to have last point if not a multiple of pas
```

(continues on next page)

(continued from previous page)

```
            if (not (self.bpm.i + 1) % self.bpm.pas
                    or self.bpm.i+1 == self.bpm.nbr_z-1):
                index += 1
                self.bpm.progress_pow[index, :] = np.array(
                        [self.bpm.current_power])

                current = (self.bpm.i+1)*self.bpm.dist_z/1e3
                final = self.bpm.length_z/1e3
                self.progressBar_compute.setValue(current/final*100)

                if disp_progress:
                    print(current, "/", final, 'mm')

        return [self.bpm.progress_pow]

    def show_estimate_time(self):
        """
        Display - on the interface - the estimate time needed to compute the
        propagation, based on linearized experimental values.
        The estimation takes into account the looses, Kerr, and control
        parameters.
        """
        self.save_compute()
        estimation = round(
            16.6/5e7*self.nbr_z*self.nbr_x  # usual propagation
            * (1 + 0.62*self.kerr_check*(self.kerr_loop))  # with kerr
            + 3.8e-8*self.nbr_z*self.nbr_x*self.kerr_check*self.variance_check,
            1)

        self.estimate_time_display.display(estimation)

    def find_guide_number(self, guide_number):
        """
        Return the waveguide group number for a given waveguide number.


        Parameters
        ----------
        guide_number : int
            Number of a waveguide

        Returns
        -------
        guide_group_number : int
            Number of the waveguide group
        """
        nbr_guide = np.zeros(len(self.nbr_p))

        for i in range(len(self.nbr_p)):

            if self.tab_index[i] == 0:
                nbr_guide[i] = self.nbr_p[i]

            elif self.tab_index[i] == 1:
                nbr_guide[i] = 3

        cumul = np.cumsum(nbr_guide)
```

(continues on next page)

(continued from previous page)

```
        guide_list_pos = np.where(cumul >= (guide_number+1))[0]

        if len(guide_list_pos) != 0:
            guide_pos = guide_list_pos[0]
            return guide_pos
        else:
            return None

    def check_modes_display(self):
        """
        Display on the interface the last mode that can propagated into a
        squared waveguide.
        """
        guide_index = self.find_guide_number(
            self.spinBox_guide_nbr_ref.value())
        if guide_index is None:
            print("Error: trying to check the mode of a unexisting waveguide")
        else:
            mode_max = self.bpm.check_modes(
                self.width[guide_index], self.delta_no[guide_index])
            self.mode_number.display(mode_max)

    def addmpl(self, tab='guide', pow_index=0):
        """
        Add the selected plots on the waveguide, light or compute window.

        Parameters
        ----------
        tab : str, optional
            'guide' or 'light'.
            'guide' by default.
        pow_index : int, optional
            Add the first waveguide and light step if 0 or the last step if -1.
            Also display the propagation over (x,z) and guide power if -1 is
            choosen.
            0 by default.
        """
        pow_index_guide = pow_index

        if pow_index < 0:
            pow_index_guide -= 1  # Display the -2 waveguide for the -1 beam

        x_min = self.x[0]
        x_max = self.x[-1]

        temp = self.peaks.reshape(self.peaks.shape[0]*self.peaks.shape[1])
        temp = temp[temp != np.array(None)]
        if len(temp) == 0:
            temp = np.array([self.x[0], self.x[-1]])

        if (0 in self.tab_index  # If array of guides
                and self.nbr_p.sum() != 0 and self.p.sum() != 0  # If exists
                and temp.min() >= self.x[0]  # If in the windows
                and temp.max() <= self.x[-1]):
            x_min = np.min(self.offset_guide - self.nbr_p*self.p)
            x_max = np.max(self.offset_guide + self.nbr_p*self.p)
            no_array = False
```

(continues on next page)

(continued from previous page)

```
        else:
            no_array = True

        if (1 in self.tab_index  # If curved guides
                and temp.min() >= self.x[0]  # If guides in the windows
                and temp.max() <= self.x[-1]):
            x_min_bis = temp.min() - self.width.max()
            x_max_bis = temp.max() + self.width.max()

            if x_min_bis < x_min or no_array:
                x_min = x_min_bis

            if x_max_bis > x_max or no_array:
                x_max = x_max_bis

        if tab == 'guide':
            fig = Figure()
            # BUG: if click on menu, change size until overlap or reduce to
            # oblivion. set_tight_layout(true) is the cause.
#             fig.set_tight_layout(True)  # Prevent axes to be cut when resizing
            if np.sum(self.full_index.imag) == 0:
                ax1 = fig.add_subplot(111)
            else:

                ax1 = fig.add_subplot(121)
                ax2 = fig.add_subplot(122)
                ax2.set_title("Imaginary part of the refractive index")
                ax2.set_xlim(x_min, x_max)
            ax1.set_title("Waveguide shape over x and z")
            ax1.set_xlabel('x (µm)')
            ax1.set_ylabel('z (mm)')

            ax1.set_xlim(x_min, x_max)

            # note that a colormesh pixel is based on 4 points
            graph = ax1.pcolormesh(self.xv,
                                   self.zv,
                                   self.dn[self.dn_disp].real,
                                   cmap='gray')
            fig.colorbar(graph, ax=ax1)

            if np.sum(self.full_index.imag) != 0:
                z_max = abs(self.dn[self.dn_disp].imag).max()
                graph2 = ax2.pcolormesh(self.xv,
                                        self.zv,
                                        self.dn[self.dn_disp].imag,
                                        cmap='seismic',
                                        vmin=-z_max, vmax=z_max)
                fig.colorbar(graph2, ax=ax2)

            self.canvas_guide_xz = FigureCanvas(fig)
            self.plot_guide.addWidget(self.canvas_guide_xz)

            self.toolbar_guide_xz = NavigationToolbar(self.canvas_guide_xz,
                                                      self.canvas_guide_xz,
                                                      coordinates=True)
            self.plot_guide.addWidget(self.toolbar_guide_xz)
```

(continues on next page)

(continued from previous page)

```
            fig = Figure()
#             fig.set_tight_layout(True)
            ax1 = fig.add_subplot(111)
            ax1.set_title("Input index profil")
            ax1.set_xlabel('x (µm)')
            ax1.set_ylabel(r'$\Delta_n$')

            if self.nbr_p.sum() != 0:
                verts = [(self.x[0], 0),
                         *zip(self.x, self.dn[pow_index_guide, :].real),
                         (self.x[-1], 0)]
                poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
                ax1.add_patch(poly)

            ax1.set_xlim(x_min, x_max)

            if max(self.dn[0, :].real) > max(self.dn[-1, :].real):
                ax1.set_ylim(0,
                             max(self.dn[0, :].real)*1.1 + 1E-20)
            else:
                ax1.set_ylim(0,
                             max(self.dn[-1, :].real)*1.1 + 1E-20)

            ax1.plot(self.x, self.dn[pow_index_guide].real, 'k')

            self.canvas_guide_x = FigureCanvas(fig)
            self.plot_guide.addWidget(self.canvas_guide_x)

            self.toolbar_guide_x = NavigationToolbar(self.canvas_guide_x,
                                                     self.canvas_guide_x,
                                                     coordinates=True)
            self.plot_guide.addWidget(self.toolbar_guide_x)

        elif tab == 'light':
            fig = Figure()
#             fig.set_tight_layout(True)
            ax1 = fig.add_subplot(111)
            ax1.set_title("Light injection")
            ax1.set_xlabel('x (µm)')
            ax2 = ax1.twinx()

            for tl in ax1.get_yticklabels():
                tl.set_color('k')

            for tl in ax2.get_yticklabels():
                tl.set_color('#1f77b4')

            ax1.set_ylabel(r'$\Delta_n$')
            ax2.set_ylabel('Irradiance ($GW.cm^{-2}$)')

            if self.nbr_p.sum() != 0:
                if pow_index_guide == 0:
                    verts = [(self.x[0], 0),
                             *zip(self.x, self.dn_first.real),
                             (self.x[-1], 0)]
                else:
                    verts = [(self.x[0], 0),
```

(continues on next page)

(continued from previous page)

```
                            *zip(self.x, self.dn[pow_index_guide, :].real),
                            (self.x[-1], 0)]
                poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
                ax1.add_patch(poly)

            ax1.set_xlim(x_min, x_max)

            if pow_index_guide == 0:
                ax1.set_ylim(
                        0,  max(1.1*self.dn_first.real) + 1E-20)
            else:
                ax1.set_ylim(
                        0,  max(1.1*self.dn[pow_index_guide, :].real) + 1E-20)

            if max(self.progress_pow[0]) != 0:
                ax2.set_ylim(0, 1.1e-13*max(self.progress_pow[0]))

            if pow_index_guide == 0:
                ax1.plot(self.x, self.dn_first.real, 'k')
            else:
                ax1.plot(self.x, self.dn[pow_index_guide].real, 'k')
            ax2.plot(self.x, 1e-13*self.progress_pow[pow_index], '#1f77b4')

            # Display light at the beginning of guides
            if pow_index == 0:
                self.canvas_light = FigureCanvas(fig)
                self.plot_light.addWidget(self.canvas_light)

                self.toolbar_light = NavigationToolbar(self.canvas_light,
                                                       self.canvas_light,
                                                       coordinates=True)
                self.plot_light.addWidget(self.toolbar_light)

            if pow_index < 0:
                ax1.set_title("Light at the output")
                self.canvas_end = FigureCanvas(fig)
                self.plot_compute.addWidget(self.canvas_end)

                self.toolbar_end = NavigationToolbar(self.canvas_end,
                                                     self.canvas_end,
                                                     coordinates=True)
                self.plot_compute.addWidget(self.toolbar_end)

                # Display light propagation into guides
                fig = Figure()
#                fig.set_tight_layout(True)
                ax1 = fig.add_subplot(111)
                ax1.set_title("Light propagation")
                ax1.set_xlabel('x (µm)')
                ax1.set_ylabel('z (mm)')

                ax2.set_xlim(x_min, x_max)

                graph = ax1.pcolormesh(self.xv,
                                       self.zv,
                                       1e-13*self.progress_pow)
#                fig.colorbar(graph, ax=ax1)
```

(continues on next page)

(continued from previous page)

```
                self.canvas_propag = FigureCanvas(fig)
                self.plot_compute.addWidget(self.canvas_propag)

                self.toolbar_propag = NavigationToolbar(self.canvas_propag,
                                                        self.canvas_propag,
                                                        coordinates=True)
                self.plot_compute.addWidget(self.toolbar_propag)

                self.display_power()

    def display_power(self):
        """Display the power in each waveguide."""
        if (not self.checkBox_power.isChecked()
                or self.nbr_p.sum() == 0 or self.p.sum() == 0):
            return None

        t_power_start = time.process_time()
        fig = Figure()
#                   fig.set_tight_layout(True)
        ax1 = fig.add_subplot(111)
        ax1.set_title("Power in waveguides")
        ax1.set_ylim(-0.05, 1.05)
        ax1.set_xlabel('z (mm)')
        ax1.set_ylabel('Power (a.u)')

        dot = [1, 2]
        sp1 = [0, 5]
        line = [1, 3, 6]
        sp2 = [1, 2, 4]
        dashes = [0]*len(dot)*len(sp1)*len(line)*len(sp2)

        for i, val1 in enumerate(dot):
            for j, val2 in enumerate(sp1):
                for k, val3 in enumerate(line):
                    for l, val4 in enumerate(sp2):
                        dashes[i*len(sp1)*len(line)*len(sp2)
                               + j*len(line)*len(sp2)
                               + k*len(sp2)+l
                               ] = (val1, val2, val3, val4)
        dashes.insert(0, (1, 0, 1, 0))

        x_beg = np.array([[None]*self.nbr_z]*self.peaks.shape[0])
        x_end = np.array([[None]*self.nbr_z]*self.peaks.shape[0])
        P = np.zeros((self.peaks.shape[0], self.nbr_z_disp+1))

        num_gd = 0
        for i, n in enumerate(self.nbr_p):
            if self.tab_index[i] == 0:  # Array of waveguide
                for _ in range(n):
                    [x_beg[num_gd, :],
                     x_end[num_gd, :]] = self.bpm.guide_position(
                         self.peaks, num_gd, self.p[i])
                    num_gd += 1
                if n == 0:  # needed if no waveguide
                    num_gd += 1

            elif self.tab_index[i] == 1:  # Curved waveguide
```

(continues on next page)

(continued from previous page)

```
                # Choose precision at the end for right waveguide
                # and choose safety when guides overlapse
                if self.peaks[num_gd+2, -1] <= self.x[-1]:
                    # accurate at end but may overlap before
                    p0 = (self.peaks[num_gd+2, -1]
                          - self.peaks[num_gd+1, -1])
                else:
                    # accurate in middle but miss evanescente part
                    p0 = self.width[i] * self.distance_factor[i]

                for j in range(3):
                    [x_beg[num_gd, :],
                     x_end[num_gd, :]] = self.bpm.guide_position(
                          self.peaks, num_gd, p0)
                    num_gd += 1

        for i in range(self.peaks.shape[0]):
            P[i, :] = self.bpm.power_guide(x_beg[i, :],
                                           x_end[i, :])
            # plot each power with a different style
            ax1.plot(self.z_disp, P[i, :],
                     dashes=dashes[i % len(dashes)],
                     label='P'+str(i))

        self.canvas_pow = FigureCanvas(fig)
        self.verticalLayout_compute_plot.addWidget(self.canvas_pow)

        self.toolbar_pow = NavigationToolbar(self.canvas_pow,
                                             self.canvas_pow,
                                             coordinates=True)
        self.verticalLayout_compute_plot.addWidget(self.toolbar_pow)
        if self.peaks.shape[0] > 10:
            ax1.legend(loc="upper right")  # Fast if many plot
        else:
            ax1.legend()  # Best if not too many plot
        ax1.grid()
        t_power_end = time.process_time()
        print('Power time: ', t_power_end-t_power_start)

    def rmmpl(self, tab, pow_index=0):
        """
        Remove the selected plots

        Parameters
        ----------
        tab : str
            'guide' or 'light'.
        pow_index : int, optional
            Remove the first light step if 0 or the last step if -1.
            0 by default.
        """
        if tab == 'guide':
            self.plot_guide.removeWidget(self.canvas_guide_xz)
            self.plot_guide.removeWidget(self.canvas_guide_x)
            self.canvas_guide_xz.close()
            self.canvas_guide_x.close()
```

(continues on next page)

(continued from previous page)

```
            self.plot_guide.removeWidget(self.toolbar_guide_xz)
            self.plot_guide.removeWidget(self.toolbar_guide_x)
            self.toolbar_guide_xz.close()
            self.toolbar_guide_x.close()

        elif tab == 'light':
            if pow_index == 0:
                self.plot_light.removeWidget(self.canvas_light)
                self.canvas_light.close()
                self.plot_light.removeWidget(self.toolbar_light)
                self.toolbar_light.close()

            if pow_index < 0:
                self.plot_compute.removeWidget(self.canvas_propag)
                self.canvas_propag.close()
                self.plot_compute.removeWidget(self.toolbar_propag)
                self.toolbar_propag.close()

                self.plot_compute.removeWidget(self.canvas_end)
                self.canvas_end.close()
                self.plot_compute.removeWidget(self.toolbar_end)
                self.toolbar_end.close()

                self.verticalLayout_compute_plot.removeWidget(self.canvas_pow)
                self.canvas_pow.close()
                self.verticalLayout_compute_plot.removeWidget(self.toolbar_pow)
                self.toolbar_pow.close()

    def save_guide(self, guide_selec=False):
        """
        Save the interface variables into the guides variables.
        """
        # if more than one waveguide and if no waveguide selected manually
        if str(guide_selec) == 'False':
            guide_selec = int(self.comboBox_guide.currentIndex())  # Choice

        self.length_z = self.doubleSpinBox_length_z.value()
        self.dist_z = self.doubleSpinBox_dist_z.value()
        self.nbr_z_disp = self.spinBox_nbr_z_disp.value()
        self.length_x = self.doubleSpinBox_length_x.value()
        self.dist_x = self.doubleSpinBox_dist_x.value()
        self.no = self.doubleSpinBox_n.value()
        self.lo = self.doubleSpinBox_lo.value()

        self.width[guide_selec] = self.doubleSpinBox_width.value()
        self.offset_guide[
            guide_selec] = self.doubleSpinBox_offset_guide.value()
        self.guide_length[
            guide_selec] = self.doubleSpinBox_guide_length.value()
        self.offset_guide_z[
            guide_selec] = self.doubleSpinBox_offset_guide_z.value()
        self.delta_no[guide_selec] = self.doubleSpinBox_dn.value()
        self.no_imag[guide_selec] = self.doubleSpinBox_n_imag.value()
        self.alpha[guide_selec] = self.doubleSpinBox_lost.value()
        self.loss_check[guide_selec] = self.checkBox_n_imag.isChecked()
        delta_no = self.delta_no[guide_selec]
        no_imag = self.no_imag[guide_selec]
```

(continues on next page)

(continued from previous page)

```
        alpha = self.alpha[guide_selec]/1000  # unit conversion mm-1 -> µm-1
        no_imag2 = alpha / (2*pi/self.lo)
        loss_check = self.loss_check[guide_selec]
        self.full_index[guide_selec] = (delta_no + 1j*loss_check*no_imag
                                        + 1j*(1-loss_check)*no_imag2)

        self.shape_gauss_check[guide_selec] = float(
            self.radioButton_gaussian.isChecked())
        self.gauss_pow[guide_selec] = int(self.spinBox_gauss_pow.value())
        self.shape_squared_check[guide_selec] = float(
            self.radioButton_squared.isChecked())
        self.nbr_p[guide_selec] = self.spinBox_nb_p.value()
        self.p[guide_selec] = self.doubleSpinBox_p.value()
        self.curve[guide_selec] = self.doubleSpinBox_curve.value()
        self.half_delay[guide_selec] = self.doubleSpinBox_half_delay.value()
        self.distance_factor[
            guide_selec] = self.doubleSpinBox_distance_factor.value()

        self.tab_index[
            guide_selec] = self.tabWidget_morphology_guide.currentIndex()
#         print("Guide variables saved")

    def get_guide(self):
        """
        Set the saved values of the waveguide variables onto the interface.
        """
        self.doubleSpinBox_length_z.setValue(self.length_z)
        self.doubleSpinBox_dist_z.setValue(self.dist_z)
        self.spinBox_nbr_z_disp.setValue(self.nbr_z_disp)
        self.doubleSpinBox_length_x.setValue(self.length_x)
        self.doubleSpinBox_dist_x.setValue(self.dist_x)
        self.doubleSpinBox_n.setValue(self.no)
        self.doubleSpinBox_lo.setValue(self.lo)

        guide_selec = int(self.comboBox_guide.currentIndex())  # choice

        if self.previous_guide != guide_selec:
            self.save_guide(self.previous_guide)

        if self.comboBox_guide.count() >= 1:  # if more than one beams
            guide_selec = int(self.comboBox_guide.currentIndex())  # choice
        else:  # Not supposed to happen
            raise ValueError("Can't have no waveguide variables")

        self.doubleSpinBox_width.setValue(self.width[guide_selec])
        self.doubleSpinBox_offset_guide.setValue(
            self.offset_guide[guide_selec])
        self.doubleSpinBox_guide_length.setValue(
            self.guide_length[guide_selec])
        self.doubleSpinBox_offset_guide_z.setValue(
            self.offset_guide_z[guide_selec])
        self.doubleSpinBox_dn.setValue(self.delta_no[guide_selec])
        self.doubleSpinBox_n_imag.setValue(self.no_imag[guide_selec])
        self.doubleSpinBox_lost.setValue(self.alpha[guide_selec])
        self.checkBox_n_imag.setChecked(self.loss_check[guide_selec])
        self.doubleSpinBox_n_imag.setEnabled(self.loss_check[guide_selec])
        self.doubleSpinBox_lost.setDisabled(self.loss_check[guide_selec])
```

(continues on next page)

(continued from previous page)

```
        self.radioButton_gaussian.setChecked(
            self.shape_gauss_check[guide_selec])
        self.spinBox_gauss_pow.setValue(self.gauss_pow[guide_selec])
        self.radioButton_squared.setChecked(
            self.shape_squared_check[guide_selec])
        self.spinBox_nb_p.setValue(self.nbr_p[guide_selec])
        self.doubleSpinBox_p.setValue(self.p[guide_selec])
        self.doubleSpinBox_curve.setValue(self.curve[guide_selec])
        self.doubleSpinBox_half_delay.setValue(self.half_delay[guide_selec])
        self.doubleSpinBox_distance_factor.setValue(
            self.distance_factor[guide_selec])
        self.tabWidget_morphology_guide.setCurrentIndex(
            self.tab_index[guide_selec])
        self.spinBox_gauss_pow.setEnabled(self.shape_gauss_check[guide_selec])

        self.previous_guide = guide_selec  # Save the n° of current waveguide

    def save_light(self, beam_selec=False):
        """
        Save the interface variables into the lights variables.

        Parameters
        ----------
        beam_selec: int, bool, optional
            Number of the beam to save into the variables.
            False by default to get the currently displayed beam.
        """
        self.theta_ext = self.doubleSpinBox_theta_ext.value()

        # if more than one beams and if no beams selected manualy
        if str(beam_selec) == 'False':
            beam_selec = int(self.comboBox_light.currentIndex())  # Choice

        self.fwhm[beam_selec] = self.doubleSpinBox_fwhm.value()
        self.offset_light[beam_selec] = self.doubleSpinBox_offset_light.value()
        self.irrad_significand[beam_selec] = (
                self.doubleSpinBox_irrad_significand.value())
        self.irrad_exponent[beam_selec] = (
                self.spinBox_irrad_exponent.value())
        self.irrad[beam_selec] = (self.doubleSpinBox_irrad_significand.value()
                                  * 10**self.spinBox_irrad_exponent.value())
        self.mode[beam_selec] = self.spinBox_mode.value()
        self.mode_guide_ref[beam_selec] = self.spinBox_guide_nbr_ref.value()
        self.offset_check[beam_selec] = (
            self.checkBox_offset_light.isChecked())
        self.offset_light_peak[beam_selec] = (
            self.spinBox_offset_light_peak.value())
        self.gaussian_check[beam_selec] = (
            self.radioButton_gaussian_light.isChecked())
        self.square_check[beam_selec] = (
            self.radioButton_squared_light.isChecked())
        self.mode_check[beam_selec] = self.radioButton_mode.isChecked()
        self.all_modes_check[beam_selec] = (
            self.radioButton_all_modes.isChecked())
        self.airy_check[beam_selec] = (
            self.radioButton_airy.isChecked())
```

(continues on next page)

(continued from previous page)

```
        self.airy_zero[beam_selec] = self.spinBox_airy_zero.value()
        self.lobe_size[beam_selec] = self.doubleSpinBox_lobe_size.value()

    def get_light(self):
        """
        Set the saved values of the light variables onto the interface.
        """
        beam_selec = int(self.comboBox_light.currentIndex())  # choice

        if self.previous_beam != beam_selec:
            self.save_light(self.previous_beam)

        self.doubleSpinBox_theta_ext.setValue(self.theta_ext)

        if self.comboBox_light.count() >= 1:  # if more than one beams
            beam_selec = int(self.comboBox_light.currentIndex())  # choice
        else:  # Not supposed to happen
            raise ValueError("Can't have no beam variables")

        self.doubleSpinBox_fwhm.setValue(self.fwhm[beam_selec])
        self.doubleSpinBox_offset_light.setValue(
            self.offset_light[beam_selec])
        self.doubleSpinBox_irrad_significand.setValue(
                self.irrad_significand[beam_selec])
        self.spinBox_irrad_exponent.setValue(
                self.irrad_exponent[beam_selec])
        self.spinBox_mode.setValue(self.mode[beam_selec])
        self.spinBox_guide_nbr_ref.setValue(self.mode_guide_ref[beam_selec])
        self.checkBox_offset_light.setChecked(self.offset_check[beam_selec])
        self.spinBox_offset_light_peak.setValue(
            self.offset_light_peak[beam_selec])
        self.radioButton_gaussian_light.setChecked(
            self.gaussian_check[beam_selec])
        self.radioButton_squared_light.setChecked(
            self.square_check[beam_selec])
        self.radioButton_mode.setChecked(self.mode_check[beam_selec])
        self.radioButton_all_modes.setChecked(self.all_modes_check[beam_selec])
        self.radioButton_airy.setChecked(self.airy_check[beam_selec])
        self.spinBox_airy_zero.setValue(self.airy_zero[beam_selec])
        self.doubleSpinBox_lobe_size.setValue(self.lobe_size[beam_selec])

        self.spinBox_airy_zero.setEnabled(self.airy_check[beam_selec])
        self.doubleSpinBox_lobe_size.setEnabled(self.airy_check[beam_selec])
        self.spinBox_mode.setEnabled(self.mode_check[beam_selec])
        self.spinBox_guide_nbr_ref.setEnabled(self.mode_check[beam_selec])

        self.previous_beam = beam_selec  # Save the n° of the current beam

    def save_compute(self):
        """
        Save the interface variables into the compute variables.
        """
        self.kerr_check = float(self.checkBox_kerr.isChecked())
        self.kerr_loop = self.spinBox_kerr_loop.value()
        self.n2_significand = self.doubleSpinBox_n2_significand.value()
        self.n2_exponent = self.spinBox_n2_exponent.value()
        self.n2 = self.n2_significand * 10**self.n2_exponent
```

(continues on next page)

(continued from previous page)

```
        self.chi3_significand = self.doubleSpinBox_chi3_significand.value()
        self.chi3_exponent = self.spinBox_chi3_exponent.value()
        self.chi3 = self.n2_significand * 10**self.chi3_exponent
        self.variance_check = float(self.checkBox_variance.isChecked())
        self.power_check = float(self.checkBox_power.isChecked())
        self.n2_check = float(self.checkBox_n2.isChecked())

    def get_compute(self):
        """
        Set the saved values of the compute variables onto the interface.
        """
        self.checkBox_kerr.setChecked(self.kerr_check)
        self.spinBox_kerr_loop.setValue(self.kerr_loop)
        self.checkBox_n2.setChecked(self.n2_check)
        self.doubleSpinBox_n2_significand.setValue(self.n2_significand)
        self.spinBox_n2_exponent.setValue(self.n2_exponent)
        self.doubleSpinBox_chi3_significand.setValue(self.chi3_significand)
        self.spinBox_chi3_exponent.setValue(self.chi3_exponent)
        self.frame_kerr.setEnabled(self.kerr_check)
        self.checkBox_variance.setChecked(self.variance_check)
        self.checkBox_power.setChecked(self.power_check)

    @pyqtSlot()
    def on_click_guide(self):
        """
        Create and displayed the waguides.
        """
        QApplication.setOverrideCursor(Qt.WaitCursor)
        self.rmmpl('guide')
        self.rmmpl('light')
        self.calculate_guide()
        self.calculate_light()
        self.addmpl('guide')
        self.addmpl('light')
        QApplication.restoreOverrideCursor()
        self.show_estimate_time()

    @pyqtSlot()
    def on_click_light(self):
        """
        Create the light and display it with the input profil waveguides.
        """
        QApplication.setOverrideCursor(Qt.WaitCursor)
        self.rmmpl(tab='light')
        self.calculate_light()
        self.addmpl(tab='light')
        QApplication.restoreOverrideCursor()
        self.show_estimate_time()

    @pyqtSlot()
    def on_click_compute(self):
        """
        Compute the propagation using the waveguides and bemas informations.
        """
        QApplication.setOverrideCursor(Qt.WaitCursor)
        self.show_estimate_time()
```

(continues on next page)

(continued from previous page)

```
        if np.max(self.progress_pow[0]) != 0:

            self.rmmpl(tab='light', pow_index=-1)

            if not self.calculate_guide_done:
                self.rmmpl('guide')
                self.calculate_guide()
                self.addmpl('guide')
                self.rmmpl(tab='light')
                self.calculate_light()
                self.addmpl(tab='light')

            self.calculate_propagation()
            self.addmpl(tab='light', pow_index=-1)

        else:
            print("no light to compute")

        QApplication.restoreOverrideCursor()

    @pyqtSlot()
    def on_click_create_guide(self):
        """Create a new waveguide with the displayed variables.
        """
        width = self.doubleSpinBox_width.value()
        offset_guide = self.doubleSpinBox_offset_guide.value()
        guide_length = self.doubleSpinBox_guide_length.value()
        offset_guide_z = self.doubleSpinBox_offset_guide_z.value()
        delta_no = self.doubleSpinBox_dn.value()
        no_imag = self.doubleSpinBox_n_imag.value()
        alpha = self.doubleSpinBox_lost.value()
        loss_check = self.checkBox_n_imag.isChecked()
        self.lo = self.doubleSpinBox_lo.value()
        no_imag2 = (alpha/1000) / (2*pi/self.lo)
        full_index = (delta_no + 1j*loss_check*no_imag
                      + 1j*(1-loss_check)*no_imag2)
        shape_gauss_check = self.radioButton_gaussian.isChecked()
        gauss_pow = int(self.spinBox_gauss_pow.value())
        shape_squared_check = self.radioButton_squared.isChecked()
        nbr_p = self.spinBox_nb_p.value()
        p = self.doubleSpinBox_p.value()
        curve = self.doubleSpinBox_curve.value()
        half_delay = self.doubleSpinBox_half_delay.value()
        distance_factor = self.doubleSpinBox_distance_factor.value()
        tab_index = self.tabWidget_morphology_guide.currentIndex()

        self.width = np.append(self.width, width)
        self.offset_guide = np.append(self.offset_guide, offset_guide)
        self.guide_length = np.append(self.guide_length, guide_length)
        self.offset_guide_z = np.append(self.offset_guide_z, offset_guide_z)
        self.delta_no = np.append(self.delta_no, delta_no)
        self.no_imag = np.append(self.no_imag, no_imag)
        self.alpha = np.append(self.alpha, alpha)
        self.loss_check = np.append(self.loss_check, loss_check)
        self.full_index = np.append(self.full_index, full_index)
        self.shape_gauss_check = np.append(self.shape_gauss_check,
                                           shape_gauss_check)
```

(continues on next page)

(continued from previous page)

```
        self.gauss_pow = np.append(self.gauss_pow, gauss_pow)
        self.shape_squared_check = np.append(self.shape_squared_check,
                                             shape_squared_check)
        self.nbr_p = np.append(self.nbr_p, nbr_p)
        self.p = np.append(self.p, p)
        self.curve = np.append(self.curve, curve)
        self.half_delay = np.append(self.half_delay, half_delay)
        self.distance_factor = np.append(self.distance_factor, distance_factor)
        self.tab_index = np.append(self.tab_index, tab_index)

        nbr_guide = self.comboBox_guide.count()  # how many item left
        self.comboBox_guide.addItem("Waveguide "+str(nbr_guide))
        self.comboBox_guide.setCurrentIndex(nbr_guide)  # show new index
        self.previous_guide = nbr_guide  # save new waveguide n°

    @pyqtSlot()
    def on_click_create_light(self):
        """Create a new beam with the displayed variables.
        """
        fwhm = self.doubleSpinBox_fwhm.value()
        offset_light = self.doubleSpinBox_offset_light.value()
        irrad_significand = self.doubleSpinBox_irrad_significand.value()
        irrad_exponent = self.spinBox_irrad_exponent.value()
        irrad = irrad_significand * 10**irrad_exponent
        offset_check = self.checkBox_offset_light.isChecked()
        gaussian_check = self.radioButton_gaussian_light.isChecked()
        square_check = self.radioButton_squared_light.isChecked()
        mode_check = self.radioButton_mode.isChecked()
        all_modes_check = self.radioButton_all_modes.isChecked()
        mode = self.spinBox_mode.value()
        mode_guide_ref = self.spinBox_guide_nbr_ref.value()
        offset_light_peak = self.spinBox_offset_light_peak.value()
        airy_check = self.radioButton_airy.isChecked()
        airy_zero = self.spinBox_airy_zero.value()
        lobe_size = self.doubleSpinBox_lobe_size.value()

        self.fwhm = np.append(self.fwhm, fwhm)
        self.offset_light = np.append(self.offset_light, offset_light)
        self.irrad_significand = np.append(self.irrad_significand,
                                           irrad_significand)
        self.irrad_exponent = np.append(self.irrad_exponent, irrad_exponent)
        self.irrad = np.append(self.irrad, irrad)
        self.mode = np.append(self.mode, mode)
        self.mode_guide_ref = np.append(self.mode_guide_ref, mode_guide_ref)
        self.offset_check = np.append(self.offset_check, offset_check)
        self.offset_light_peak = np.append(self.offset_light_peak,
                                           offset_light_peak)
        self.gaussian_check = np.append(self.gaussian_check, gaussian_check)
        self.square_check = np.append(self.square_check, square_check)
        self.mode_check = np.append(self.mode_check, mode_check)
        self.all_modes_check = np.append(self.all_modes_check, all_modes_check)
        self.airy_check = np.append(self.airy_check, airy_check)
        self.airy_zero = np.append(self.airy_zero, airy_zero)
        self.lobe_size = np.append(self.lobe_size, lobe_size)

        nbr_light = self.comboBox_light.count()  # how many item left
        self.comboBox_light.addItem("Beam "+str(nbr_light))  # add new index
```

(continues on next page)

(continued from previous page)

```
        self.comboBox_light.setCurrentIndex(nbr_light)  # show new index
        self.previous_beam = nbr_light  # Change the current selected beam

    @pyqtSlot()
    def on_click_delete_guide(self):
        """
        Delete the current displayed waveguide and displayed the next one if
        exist else the previous one.
        """
        nbr_guide = self.comboBox_guide.count()

        if nbr_guide > 1:  # Can't delete if remains only 1 waveguide
            guide_selec = int(self.comboBox_guide.currentIndex())  # choice
            self.width = np.delete(self.width, guide_selec)
            self.offset_guide = np.delete(self.offset_guide, guide_selec)
            self.guide_length = np.delete(self.guide_length, guide_selec)
            self.offset_guide_z = np.delete(self.offset_guide_z, guide_selec)
            self.delta_no = np.delete(self.delta_no, guide_selec)
            self.no_imag = np.delete(self.no_imag, guide_selec)
            self.alpha = np.delete(self.alpha, guide_selec)
            self.loss_check = np.delete(self.loss_check, guide_selec)
            self.full_index = np.delete(self.full_index, guide_selec)

            self.shape_gauss_check = np.delete(self.shape_gauss_check,
                                               guide_selec)
            self.gauss_pow = np.delete(self.gauss_pow, guide_selec)
            self.shape_squared_check = np.delete(self.shape_squared_check,
                                                 guide_selec)
            self.nbr_p = np.delete(self.nbr_p, guide_selec)
            self.p = np.delete(self.p, guide_selec)
            self.curve = np.delete(self.curve, guide_selec)
            self.half_delay = np.delete(self.half_delay, guide_selec)
            self.distance_factor = np.delete(self.distance_factor, guide_selec)

            nbr_guide -= 1

            self.comboBox_guide.clear()  # remove all beams number

            for i in range(nbr_guide):  # Add waveguides n°
                self.comboBox_guide.addItem("Waveguide "+str(i))

            # set same waveguide n° if not the last else reduce the index by 1
            if guide_selec == nbr_guide and guide_selec != 0:
                guide_selec -= 1

            self.comboBox_guide.setCurrentIndex(guide_selec)
            self.previous_guide = guide_selec  # Change the selected waveguide
            self.get_guide()  # Display previous or next waveguide values

    @pyqtSlot()
    def on_click_delete_light(self):
        """
        Delete the current displayed beam and displayed the next one.
        """
        nbr_light = self.comboBox_light.count()

        if nbr_light > 1:  # Can't delete if remains only 1 beam
```

(continues on next page)

(continued from previous page)

```
            beam_selec = int(self.comboBox_light.currentIndex())  # choice
            self.fwhm = np.delete(self.fwhm, beam_selec)
            self.offset_light = np.delete(self.offset_light, beam_selec)
            self.irrad_significand = np.delete(self.irrad_significand,
                                               beam_selec)
            self.irrad_exponent = np.delete(self.irrad_exponent, beam_selec)
            self.irrad = np.delete(self.irrad, beam_selec)
            self.mode = np.delete(self.mode, beam_selec)
            self.mode_guide_ref = np.delete(self.mode_guide_ref, beam_selec)
            self.offset_check = np.delete(self.offset_check, beam_selec)
            self.offset_light_peak = np.delete(self.offset_light_peak,
                                               beam_selec)
            self.gaussian_check = np.delete(self.gaussian_check, beam_selec)
            self.square_check = np.delete(self.square_check, beam_selec)
            self.mode_check = np.delete(self.mode_check, beam_selec)
            self.all_modes_check = np.delete(self.all_modes_check,
                                             beam_selec)
            self.airy_check = np.delete(self.airy_check, beam_selec)
            self.airy_zero = np.delete(self.airy_zero, beam_selec)
            self.lobe_size = np.delete(self.lobe_size, beam_selec)

            nbr_light -= 1

            self.comboBox_light.clear()  # remove all beams number

            for i in range(nbr_light):  # create again with new number
                self.comboBox_light.addItem("Beam "+str(i))

            # set same beam index if not the last else reduce the index by 1
            if beam_selec == nbr_light and beam_selec != 0:
                beam_selec -= 1

            self.comboBox_light.setCurrentIndex(beam_selec)
            self.previous_beam = beam_selec  # Change the current selected beam
            self.get_light()  # Display values of the previous or next beam

    @pyqtSlot()
    def open_file_name(self):
        """
        Open a dialog window to select the file to open, and call
        :meth:`open_file` to open the file.

        Source: https://pythonspot.com/pyqt5-file-dialog/

        Notes
        -----
        This method has a try/except implemented to check if the openned file
        contains all the variables.
        """

        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        filename, _ = QFileDialog.getOpenFileName(self,
                                                  "Import data",
                                                  "",
                                                  "Text Files (*.txt)",
                                                  options=options)
```

(continues on next page)

(continued from previous page)

```
        if filename:
            try:
                self.open_file(filename)
                self.filename = filename  # Next save will overwrite this file
                self.setWindowTitle("Beampy - "+filename)
            except KeyError as ex:
                print("missing variable", ex, "in the file.")
                print("Add it manually to remove this error.")
#             except Exception as ex:  # This execption was removed to see the
                # # error location. The program won't crash thanks to try
#                 print("Unknown error when openning the file:", filename)
#                 print("Error:", ex)

    def open_file(self, filename):
        """
        Set guides, beams and computes variables from a choosen file.

        Parameters
        ----------
        filename : str
            Name of the file.
        """
#         https://www.tutorialspoint.com/
#         How-to-create-a-Python-dictionary-from-text-file
        dico = {}
        f = open(filename, 'r')
        for line in f:
            (variables, *val) = line.split()  # Assume: variable_name values
#             print(variables, val)
            dico[str(variables)] = val
        f.close()

        # Waveguide variables
        self.length_z = float(dico['length_z'][0])
        self.dist_z = float(dico['dist_z'][0])
        self.nbr_z_disp = int(dico['nbr_z_disp'][0])
        self.length_x = float(dico['length_x'][0])
        self.dist_x = float(dico['dist_x'][0])
        self.no = float(dico['no'][0])
        self.lo = float(dico['lo'][0])

        self.width = np.array(dico['width'], dtype=float)
        self.offset_guide = np.array(dico['offset_guide'], dtype=float)
        self.guide_length = np.array(dico['guide_length'], dtype=float)
        self.offset_guide_z = np.array(dico['offset_guide_z'], dtype=float)
        self.delta_no = np.array(dico['delta_no'], dtype=float)
        self.no_imag = np.array(dico['no_imag'], dtype=float)
        self.alpha = np.array(dico['alpha'], dtype=float)
        self.loss_check = np.array(dico['loss_check'], dtype=float)
        no_imag2 = (self.alpha/1000) / (2*pi/self.lo)
        self.full_index = np.array(
                self.delta_no + 1j*self.loss_check*self.no_imag
                + 1j*(1-self.loss_check)*no_imag2)
        self.shape_gauss_check = np.array(
                dico['shape_gauss_check'], dtype=float)
        self.gauss_pow = np.array(dico['gauss_pow'], dtype=int)
        self.shape_squared_check = np.array(
```

(continues on next page)

(continued from previous page)

```
            dico['shape_squared_check'], dtype=float)
        self.nbr_p = np.array(dico['nbr_p'], dtype=int)
        self.p = np.array(dico['p'], dtype=float)
        self.curve = np.array(dico['curve'], dtype=float)
        self.half_delay = np.array(dico['half_delay'], dtype=float)
        self.distance_factor = np.array(dico['distance_factor'], dtype=float)
        self.tab_index = np.array(dico['tab_index'], dtype=float)

        # Light variables
        self.theta_ext = float(dico['theta_ext'][0])

        self.fwhm = np.array(dico['fwhm'], dtype=float)
        self.offset_light = np.array(dico['offset_light'], dtype=float)
        self.irrad_significand = np.array(
            dico['irrad_significand'], dtype=float)
        self.irrad_exponent = np.array(dico['irrad_exponent'], dtype=float)
        self.irrad = self.irrad_significand*10**self.irrad_exponent
        self.offset_check = np.array(dico['offset_check'], dtype=float)
        self.gaussian_check = np.array(dico['gaussian_check'], dtype=float)
        self.square_check = np.array(dico['square_check'], dtype=float)
        self.mode_check = np.array(dico['mode_check'], dtype=float)
        self.all_modes_check = np.array(dico['all_modes_check'], dtype=float)
        self.mode = np.array(dico['mode'], dtype=int)
        self.mode_guide_ref = np.array(dico['mode_guide_ref'], dtype=int)
        self.offset_light_peak = np.array(
            dico['offset_light_peak'], dtype=int)
        self.spinBox_offset_light_peak.setMaximum(99)
        self.airy_check = np.array(dico['airy_check'], dtype=float)
        self.airy_zero = np.array(dico['airy_zero'], dtype=int)
        self.lobe_size = np.array(dico['lobe_size'], dtype=float)

        # Compute variables
        self.kerr_check = float(dico['kerr_check'][0])
        self.kerr_loop = int(dico['kerr_loop'][0])
        self.n2_check = float(dico['n2_check'][0])
        self.n2_significand = float(dico['n2_significand'][0])
        self.n2_exponent = float(dico['n2_exponent'][0])
        self.chi3_significand = float(dico['chi3_significand'][0])
        self.chi3_exponent = float(dico['chi3_exponent'][0])
        self.variance_check = float(dico['variance_check'][0])
        self.power_check = float(dico['power_check'][0])

        nbr_guide = len(self.width)
        self.comboBox_guide.clear()  # Remove all guides number

        nbr_light = len(self.fwhm)
        self.comboBox_light.clear()  # Remove all beams number

        for i in range(nbr_guide):  # Add waveguides n°
            self.comboBox_guide.addItem("Waveguide "+str(i))

        for i in range(nbr_light):  # Add beams n°
            self.comboBox_light.addItem("Beam "+str(i))

        self.previous_guide = 0  # Will show the first waveguide
        self.previous_beam = 0  # Will show the first beam
```

(continues on next page)

(continued from previous page)

```
        self.get_guide()  # Set waveguide values
        self.get_light()  # Set lights values
        self.get_compute()  # Set compute values

        self.on_click_guide()

        print("file openned")

    @pyqtSlot()
    def save_quick(self):
        """
        Check if a file is already selected and if so, save into it.
        Else, call the :meth:`save_file_name` to ask a filename.
        """
        if self.filename is None:
            self.save_file_name()
        else:
            self.save_file(self.filename)

    def save_file_name(self):
        """
        Open a dialog window to select the saved file name and call
        :meth:`save_file` to save the file.
        """
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        filename, _ = QFileDialog.getSaveFileName(self,
                                                  "Save data",
                                                  "",
                                                  "Text Files (*.txt)",
                                                  options=options)
        if filename:
            if filename[-4:] != '.txt':
                filename = filename + '.txt'
            self.filename = filename
            self.save_file(filename)
            self.setWindowTitle("Beampy - "+filename)

    def save_file(self, filename):
        """
        Save guides, beams and computes variables into a choosen file.

        Parameters
        ----------
        filename : str
            Name of the file.
        """
        self.save_guide()
        self.save_light()
        self.save_compute()

        f = open(filename, "w")

        # Waveguide variables
        f.write('length_z ' + str(self.length_z) + '\n')
        f.write('dist_z ' + str(self.dist_z) + '\n')
        f.write('nbr_z_disp ' + str(self.nbr_z_disp) + '\n')
```

(continues on next page)

(continued from previous page)

```
        f.write('length_x ' + str(self.length_x) + '\n')
        f.write('dist_x ' + str(self.dist_x) + '\n')
        f.write('no ' + str(self.no) + '\n')
        f.write('lo ' + str(self.lo) + '\n')

        f.write('width ' + str(self.width).replace("[", "").replace("]", "")
                + '\n')
        f.write('offset_guide ' + str(
            self.offset_guide).replace("[", "").replace("]", "")
                + '\n')
        f.write('guide_length ' + str(
            self.guide_length).replace("[", "").replace("]", "")
                + '\n')
        f.write('offset_guide_z ' + str(
            self.offset_guide_z).replace("[", "").replace("]", "")
                + '\n')
        f.write('delta_no ' + str(
            self.delta_no).replace("[", "").replace("]", "")
                + '\n')
        f.write('no_imag ' + str(
            self.no_imag).replace("[", "").replace("]", "")
                + '\n')
        f.write('alpha ' + str(
            self.alpha).replace("[", "").replace("]", "")
                + '\n')
        f.write('loss_check ' + str(
            self.loss_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('shape_gauss_check ' + str(
            self.shape_gauss_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('gauss_pow ' + str(
            self.gauss_pow).replace("[", "").replace("]", "")
                + '\n')
        f.write('shape_squared_check ' + str(
            self.shape_squared_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('nbr_p ' + str(self.nbr_p).replace("[", "").replace("]", "")
                + '\n')
        f.write('p ' + str(self.p).replace("[", "").replace("]", "")
                + '\n')
        f.write('curve ' + str(self.curve).replace("[", "").replace("]", "")
                + '\n')
        f.write('half_delay ' + str(
            self.half_delay).replace("[", "").replace("]", "")
                + '\n')
        f.write('distance_factor ' + str(
            self.distance_factor).replace("[", "").replace("]", "")
                + '\n')
        f.write('tab_index ' + str(
            self.tab_index).replace("[", "").replace("]", "")
                + '\n')

        # light variables
        f.write('theta_ext ' + str(self.theta_ext) + '\n')

        f.write('fwhm '
```

(continues on next page)

(continued from previous page)

```
                + str(self.fwhm).replace("[", "").replace("]", "")
                + '\n')
        f.write('offset_light '
                + str(self.offset_light).replace("[", "").replace("]", "")
                + '\n')
        f.write('irrad_significand '
                + str(self.irrad_significand).replace("[", "").replace("]", "")
                + '\n')
        f.write('irrad_exponent '
                + str(self.irrad_exponent).replace("[", "").replace("]", "")
                + '\n')
        f.write('offset_check '
                + str(self.offset_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('gaussian_check '
                + str(self.gaussian_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('square_check '
                + str(self.square_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('mode_check '
                + str(self.mode_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('all_modes_check '
                + str(self.all_modes_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('mode '
                + str(self.mode).replace("[", "").replace("]", "")
                + '\n')
        f.write('mode_guide_ref '
                + str(self.mode_guide_ref).replace("[", "").replace("]", "")
                + '\n')
        f.write('offset_light_peak '
                + str(self.offset_light_peak).replace("[", "").replace("]", "")
                + '\n')
        f.write('airy_check '
                + str(self.airy_check).replace("[", "").replace("]", "")
                + '\n')
        f.write('airy_zero '
                + str(self.airy_zero).replace("[", "").replace("]", "")
                + '\n')
        f.write('lobe_size '
                + str(self.lobe_size).replace("[", "").replace("]", "")
                + '\n')

        # compute variables
        f.write('kerr_check ' + str(self.kerr_check) + '\n')
        f.write('kerr_loop ' + str(self.kerr_loop) + '\n')
        f.write('n2_check ' + str(self.n2_check) + '\n')
        f.write('n2_significand ' + str(self.n2_significand) + '\n')
        f.write('n2_exponent ' + str(self.n2_exponent) + '\n')
        f.write('chi3_significand ' + str(self.chi3_significand) + '\n')
        f.write('chi3_exponent ' + str(self.chi3_exponent) + '\n')
        f.write('variance_check ' + str(self.variance_check) + '\n')
        f.write('power_check ' + str(self.power_check) + '\n')
        f.close()
        print("file saved")
```

(continues on next page)

(continued from previous page)

```
def open_doc():
    """
    Function that open the local html documentation - describing the beampy
    modules - if exist, or the online version otherwise.
    """
    file = __file__  # Module name
    # Replaces characters only when called from outer files
    file = file.replace("\\", "/")
    file = file.split("/")
    file = file[:-2]  # Remove the folder and file name

    file2 = str()
    for line in file:
        file2 = file2+"/"+line

    file = file2[1:]+"/docs/html/index.html"
    exists = os.path.isfile(file)

    if exists:
        webbrowser.open(file, new=2)  # Open file in a new tab (new=2)
    else:
        print("The documentation can't be found localy in:", file)
        file = "https://beampy.readthedocs.io"
        print("Openning the online version at:", file)
        webbrowser.open(file, new=2)  # Open file in a new tab (new=2)


def open_app():
    """
    Function used to open the app.
    Can be called directly from beampy.
    """
    app = QApplication(sys.argv)  # Define the app
    myapp = UserInterface()  # Run the app
    myapp.show()  # Show the form
    app.exec_()  # Execute the app in a loop


if __name__ == "__main__":
    open_app()
```

5.4.3 beampy.interface

```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'interface.ui'
#
# Created by: PyQt5 UI code generator 5.15.1
#
# WARNING: Any manual changes made to this file will be lost when pyuic5 is
# run again.  Do not edit this file unless you know what you are doing.
```

(continues on next page)

(continued from previous page)

```
from PyQt5 import QtCore, QtGui, QtWidgets


class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(1144, 1019)
        MainWindow.setMinimumSize(QtCore.QSize(800, 935))
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.gridLayout = QtWidgets.QGridLayout(self.centralwidget)
        self.gridLayout.setObjectName("gridLayout")
        self.frame_file_data = QtWidgets.QFrame(self.centralwidget)
        self.frame_file_data.setFrameShape(QtWidgets.QFrame.NoFrame)
        self.frame_file_data.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_file_data.setObjectName("frame_file_data")
        self.verticalLayout = QtWidgets.QVBoxLayout(self.frame_file_data)
        self.verticalLayout.setObjectName("verticalLayout")
        self.tabWidget_main = QtWidgets.QTabWidget(self.frame_file_data)
        self.tabWidget_main.setObjectName("tabWidget_main")
        self.tabWidget_guide = QtWidgets.QWidget()
        self.tabWidget_guide.setObjectName("tabWidget_guide")
        self.horizontalLayout_2 = QtWidgets.QHBoxLayout(self.tabWidget_guide)
        self.horizontalLayout_2.setObjectName("horizontalLayout_2")
        self.verticalLayout_guide = QtWidgets.QVBoxLayout()
        self.verticalLayout_guide.setObjectName("verticalLayout_guide")
        self.label_window_parameters = QtWidgets.QLabel(self.tabWidget_guide)
        self.label_window_parameters.setMinimumSize(QtCore.QSize(0, 62))
        self.label_window_parameters.setMaximumSize(QtCore.QSize(16777215, 30))
        self.label_window_parameters.setObjectName("label_window_parameters")
        self.verticalLayout_guide.addWidget(self.label_window_parameters)
        self.frame_window = QtWidgets.QFrame(self.tabWidget_guide)
        self.frame_window.setMinimumSize(QtCore.QSize(200, 0))
        self.frame_window.setMaximumSize(QtCore.QSize(350, 16777215))
        self.frame_window.setFrameShape(QtWidgets.QFrame.Box)
        self.frame_window.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_window.setObjectName("frame_window")
        self.formLayout_10 = QtWidgets.QFormLayout(self.frame_window)
        self.formLayout_10.setObjectName("formLayout_10")
        self.label_length = QtWidgets.QLabel(self.frame_window)
        self.label_length.setAccessibleDescription("")
        self.label_length.setOpenExternalLinks(False)
        self.label_length.setObjectName("label_length")
        self.formLayout_10.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.label_
↪length)
        self.doubleSpinBox_length_z = QtWidgets.QDoubleSpinBox(self.frame_window)
        self.doubleSpinBox_length_z.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_length_z.setDecimals(3)
        self.doubleSpinBox_length_z.setMinimum(0.001)
        self.doubleSpinBox_length_z.setMaximum(100000.0)
        self.doubleSpinBox_length_z.setProperty("value", 10000.0)
        self.doubleSpinBox_length_z.setObjectName("doubleSpinBox_length_z")
        self.formLayout_10.setWidget(0, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_length_z)
        self.label_dist_z = QtWidgets.QLabel(self.frame_window)
        self.label_dist_z.setObjectName("label_dist_z")
```

(continues on next page)

(continued from previous page)

```
        self.formLayout_10.setWidget(1, QtWidgets.QFormLayout.LabelRole, self.label_
↪dist_z)
        self.doubleSpinBox_dist_z = QtWidgets.QDoubleSpinBox(self.frame_window)
        self.doubleSpinBox_dist_z.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_dist_z.setDecimals(3)
        self.doubleSpinBox_dist_z.setMinimum(0.001)
        self.doubleSpinBox_dist_z.setMaximum(100000.0)
        self.doubleSpinBox_dist_z.setProperty("value", 1.0)
        self.doubleSpinBox_dist_z.setObjectName("doubleSpinBox_dist_z")
        self.formLayout_10.setWidget(1, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_dist_z)
        self.label_nbr_z_disp = QtWidgets.QLabel(self.frame_window)
        self.label_nbr_z_disp.setObjectName("label_nbr_z_disp")
        self.formLayout_10.setWidget(2, QtWidgets.QFormLayout.LabelRole, self.label_
↪nbr_z_disp)
        self.spinBox_nbr_z_disp = QtWidgets.QSpinBox(self.frame_window)
        self.spinBox_nbr_z_disp.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_nbr_z_disp.setMinimum(1)
        self.spinBox_nbr_z_disp.setMaximum(10000)
        self.spinBox_nbr_z_disp.setProperty("value", 200)
        self.spinBox_nbr_z_disp.setDisplayIntegerBase(10)
        self.spinBox_nbr_z_disp.setObjectName("spinBox_nbr_z_disp")
        self.formLayout_10.setWidget(2, QtWidgets.QFormLayout.FieldRole, self.spinBox_
↪nbr_z_disp)
        self.label_length_x = QtWidgets.QLabel(self.frame_window)
        self.label_length_x.setObjectName("label_length_x")
        self.formLayout_10.setWidget(3, QtWidgets.QFormLayout.LabelRole, self.label_
↪length_x)
        self.doubleSpinBox_length_x = QtWidgets.QDoubleSpinBox(self.frame_window)
        self.doubleSpinBox_length_x.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_length_x.setDecimals(3)
        self.doubleSpinBox_length_x.setMinimum(0.001)
        self.doubleSpinBox_length_x.setMaximum(10000.0)
        self.doubleSpinBox_length_x.setProperty("value", 1000.0)
        self.doubleSpinBox_length_x.setObjectName("doubleSpinBox_length_x")
        self.formLayout_10.setWidget(3, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_length_x)
        self.label_dist_x = QtWidgets.QLabel(self.frame_window)
        self.label_dist_x.setObjectName("label_dist_x")
        self.formLayout_10.setWidget(4, QtWidgets.QFormLayout.LabelRole, self.label_
↪dist_x)
        self.doubleSpinBox_dist_x = QtWidgets.QDoubleSpinBox(self.frame_window)
        self.doubleSpinBox_dist_x.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_dist_x.setDecimals(3)
        self.doubleSpinBox_dist_x.setMinimum(0.001)
        self.doubleSpinBox_dist_x.setMaximum(100.0)
        self.doubleSpinBox_dist_x.setProperty("value", 0.2)
        self.doubleSpinBox_dist_x.setObjectName("doubleSpinBox_dist_x")
        self.formLayout_10.setWidget(4, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_dist_x)
        self.label_n = QtWidgets.QLabel(self.frame_window)
        self.label_n.setObjectName("label_n")
        self.formLayout_10.setWidget(5, QtWidgets.QFormLayout.LabelRole, self.label_n)
```

(continues on next page)

(continued from previous page)

```
        self.doubleSpinBox_n = QtWidgets.QDoubleSpinBox(self.frame_window)
        self.doubleSpinBox_n.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_n.setDecimals(6)
        self.doubleSpinBox_n.setMinimum(1.0)
        self.doubleSpinBox_n.setMaximum(1000.0)
        self.doubleSpinBox_n.setSingleStep(0.1)
        self.doubleSpinBox_n.setProperty("value", 2.14)
        self.doubleSpinBox_n.setObjectName("doubleSpinBox_n")
        self.formLayout_10.setWidget(5, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_n)
        self.doubleSpinBox_lo = QtWidgets.QDoubleSpinBox(self.frame_window)
        self.doubleSpinBox_lo.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_lo.setDecimals(4)
        self.doubleSpinBox_lo.setMinimum(0.01)
        self.doubleSpinBox_lo.setMaximum(100.0)
        self.doubleSpinBox_lo.setProperty("value", 1.55)
        self.doubleSpinBox_lo.setObjectName("doubleSpinBox_lo")
        self.formLayout_10.setWidget(6, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_lo)
        self.label_lo = QtWidgets.QLabel(self.frame_window)
        self.label_lo.setObjectName("label_lo")
        self.formLayout_10.setWidget(6, QtWidgets.QFormLayout.LabelRole, self.label_
↪lo)
        self.verticalLayout_guide.addWidget(self.frame_window)
        spacerItem = QtWidgets.QSpacerItem(20, 10, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Maximum)
        self.verticalLayout_guide.addItem(spacerItem)
        self.label_guides_information = QtWidgets.QLabel(self.tabWidget_guide)
        self.label_guides_information.setMinimumSize(QtCore.QSize(190, 62))
        self.label_guides_information.setMaximumSize(QtCore.QSize(16777215, 30))
        self.label_guides_information.setObjectName("label_guides_information")
        self.verticalLayout_guide.addWidget(self.label_guides_information)
        self.frame_guides = QtWidgets.QFrame(self.tabWidget_guide)
        self.frame_guides.setMinimumSize(QtCore.QSize(270, 0))
        self.frame_guides.setMaximumSize(QtCore.QSize(350, 16777215))
        self.frame_guides.setFrameShape(QtWidgets.QFrame.Box)
        self.frame_guides.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_guides.setObjectName("frame_guides")
        self.formLayout_6 = QtWidgets.QFormLayout(self.frame_guides)
        self.formLayout_6.setObjectName("formLayout_6")
        self.comboBox_guide = QtWidgets.QComboBox(self.frame_guides)
        self.comboBox_guide.setMaximumSize(QtCore.QSize(100, 16777215))
        self.comboBox_guide.setObjectName("comboBox_guide")
        self.formLayout_6.setWidget(0, QtWidgets.QFormLayout.SpanningRole, self.
↪comboBox_guide)
        self.Qframe_guide_create = QtWidgets.QFrame(self.frame_guides)
        self.Qframe_guide_create.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.Qframe_guide_create.setFrameShadow(QtWidgets.QFrame.Raised)
        self.Qframe_guide_create.setObjectName("Qframe_guide_create")
        self.gridLayout_2 = QtWidgets.QGridLayout(self.Qframe_guide_create)
        self.gridLayout_2.setObjectName("gridLayout_2")
        self.pushButton_save_guide = QtWidgets.QPushButton(self.Qframe_guide_create)
        self.pushButton_save_guide.setObjectName("pushButton_save_guide")
        self.gridLayout_2.addWidget(self.pushButton_save_guide, 1, 0, 1, 1)
        self.pushButton_delete_guide = QtWidgets.QPushButton(self.Qframe_guide_create)
```

(continues on next page)

(continued from previous page)

```
        self.pushButton_delete_guide.setObjectName("pushButton_delete_guide")
        self.gridLayout_2.addWidget(self.pushButton_delete_guide, 0, 1, 1, 1)
        self.pushButton_create_guide = QtWidgets.QPushButton(self.Qframe_guide_create)
        self.pushButton_create_guide.setObjectName("pushButton_create_guide")
        self.gridLayout_2.addWidget(self.pushButton_create_guide, 0, 0, 1, 1)
        self.pushButton_cancel_guide = QtWidgets.QPushButton(self.Qframe_guide_create)
        self.pushButton_cancel_guide.setStatusTip("")
        self.pushButton_cancel_guide.setObjectName("pushButton_cancel_guide")
        self.gridLayout_2.addWidget(self.pushButton_cancel_guide, 1, 1, 1, 1)
        self.formLayout_6.setWidget(1, QtWidgets.QFormLayout.SpanningRole, self.
↪Qframe_guide_create)
        self.label_width = QtWidgets.QLabel(self.frame_guides)
        self.label_width.setMinimumSize(QtCore.QSize(40, 0))
        self.label_width.setObjectName("label_width")
        self.formLayout_6.setWidget(2, QtWidgets.QFormLayout.LabelRole, self.label_
↪width)
        self.doubleSpinBox_width = QtWidgets.QDoubleSpinBox(self.frame_guides)
        self.doubleSpinBox_width.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_width.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_width.setDecimals(3)
        self.doubleSpinBox_width.setMinimum(0.0)
        self.doubleSpinBox_width.setMaximum(10000.0)
        self.doubleSpinBox_width.setProperty("value", 8.0)
        self.doubleSpinBox_width.setObjectName("doubleSpinBox_width")
        self.formLayout_6.setWidget(2, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_width)
        self.label_offset_guide = QtWidgets.QLabel(self.frame_guides)
        self.label_offset_guide.setMinimumSize(QtCore.QSize(40, 0))
        self.label_offset_guide.setObjectName("label_offset_guide")
        self.formLayout_6.setWidget(3, QtWidgets.QFormLayout.LabelRole, self.label_
↪offset_guide)
        self.doubleSpinBox_offset_guide = QtWidgets.QDoubleSpinBox(self.frame_guides)
        self.doubleSpinBox_offset_guide.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_offset_guide.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_offset_guide.setDecimals(3)
        self.doubleSpinBox_offset_guide.setMinimum(-5000.0)
        self.doubleSpinBox_offset_guide.setMaximum(5000.0)
        self.doubleSpinBox_offset_guide.setObjectName("doubleSpinBox_offset_guide")
        self.formLayout_6.setWidget(3, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_offset_guide)
        self.label_dn = QtWidgets.QLabel(self.frame_guides)
        self.label_dn.setMinimumSize(QtCore.QSize(40, 0))
        self.label_dn.setObjectName("label_dn")
        self.formLayout_6.setWidget(4, QtWidgets.QFormLayout.LabelRole, self.label_dn)
        self.doubleSpinBox_dn = QtWidgets.QDoubleSpinBox(self.frame_guides)
        self.doubleSpinBox_dn.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_dn.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_dn.setDecimals(6)
        self.doubleSpinBox_dn.setMaximum(1000.0)
        self.doubleSpinBox_dn.setProperty("value", 0.001)
        self.doubleSpinBox_dn.setObjectName("doubleSpinBox_dn")
        self.formLayout_6.setWidget(4, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_dn)
        self.checkBox_n_imag = QtWidgets.QCheckBox(self.frame_guides)
```

(continues on next page)

(continued from previous page)

```
        self.checkBox_n_imag.setMinimumSize(QtCore.QSize(60, 0))
        self.checkBox_n_imag.setChecked(True)
        self.checkBox_n_imag.setObjectName("checkBox_n_imag")
        self.formLayout_6.setWidget(5, QtWidgets.QFormLayout.LabelRole, self.checkBox_
↪n_imag)
        self.doubleSpinBox_n_imag = QtWidgets.QDoubleSpinBox(self.frame_guides)
        self.doubleSpinBox_n_imag.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_n_imag.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_n_imag.setDecimals(6)
        self.doubleSpinBox_n_imag.setMaximum(20.0)
        self.doubleSpinBox_n_imag.setObjectName("doubleSpinBox_n_imag")
        self.formLayout_6.setWidget(5, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_n_imag)
        self.label_lost = QtWidgets.QLabel(self.frame_guides)
        self.label_lost.setMinimumSize(QtCore.QSize(60, 0))
        self.label_lost.setObjectName("label_lost")
        self.formLayout_6.setWidget(6, QtWidgets.QFormLayout.LabelRole, self.label_
↪lost)
        self.doubleSpinBox_lost = QtWidgets.QDoubleSpinBox(self.frame_guides)
        self.doubleSpinBox_lost.setEnabled(False)
        self.doubleSpinBox_lost.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_lost.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_lost.setDecimals(5)
        self.doubleSpinBox_lost.setMaximum(10000.0)
        self.doubleSpinBox_lost.setProperty("value", 1.0)
        self.doubleSpinBox_lost.setObjectName("doubleSpinBox_lost")
        self.formLayout_6.setWidget(6, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_lost)
        self.label = QtWidgets.QLabel(self.frame_guides)
        self.label.setMinimumSize(QtCore.QSize(130, 0))
        self.label.setLineWidth(1)
        self.label.setObjectName("label")
        self.formLayout_6.setWidget(8, QtWidgets.QFormLayout.SpanningRole, self.label)
        self.radioButton_gaussian = QtWidgets.QRadioButton(self.frame_guides)
        self.radioButton_gaussian.setMinimumSize(QtCore.QSize(70, 0))
        self.radioButton_gaussian.setChecked(True)
        self.radioButton_gaussian.setObjectName("radioButton_gaussian")
        self.formLayout_6.setWidget(9, QtWidgets.QFormLayout.LabelRole, self.
↪radioButton_gaussian)
        self.spinBox_gauss_pow = QtWidgets.QSpinBox(self.frame_guides)
        self.spinBox_gauss_pow.setMinimumSize(QtCore.QSize(70, 0))
        self.spinBox_gauss_pow.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_gauss_pow.setSuffix("")
        self.spinBox_gauss_pow.setMinimum(1)
        self.spinBox_gauss_pow.setMaximum(10)
        self.spinBox_gauss_pow.setSingleStep(1)
        self.spinBox_gauss_pow.setProperty("value", 4)
        self.spinBox_gauss_pow.setObjectName("spinBox_gauss_pow")
        self.formLayout_6.setWidget(9, QtWidgets.QFormLayout.FieldRole, self.spinBox_
↪gauss_pow)
        self.radioButton_squared = QtWidgets.QRadioButton(self.frame_guides)
        self.radioButton_squared.setMinimumSize(QtCore.QSize(70, 0))
        self.radioButton_squared.setObjectName("radioButton_squared")
        self.formLayout_6.setWidget(10, QtWidgets.QFormLayout.LabelRole, self.
↪radioButton_squared)
```

(continues on next page)

(continued from previous page)

```
        self.tabWidget_morphology_guide = QtWidgets.QTabWidget(self.frame_guides)
        self.tabWidget_morphology_guide.setMinimumSize(QtCore.QSize(250, 0))
        self.tabWidget_morphology_guide.setToolTip("")
        self.tabWidget_morphology_guide.setObjectName("tabWidget_morphology_guide")
        self.tab_array = QtWidgets.QWidget()
        self.tab_array.setObjectName("tab_array")
        self.formLayout = QtWidgets.QFormLayout(self.tab_array)
        self.formLayout.setObjectName("formLayout")
        self.label_nb_p = QtWidgets.QLabel(self.tab_array)
        self.label_nb_p.setMinimumSize(QtCore.QSize(90, 0))
        self.label_nb_p.setObjectName("label_nb_p")
        self.formLayout.setWidget(2, QtWidgets.QFormLayout.LabelRole, self.label_nb_p)
        self.spinBox_nb_p = QtWidgets.QSpinBox(self.tab_array)
        self.spinBox_nb_p.setMinimumSize(QtCore.QSize(40, 0))
        self.spinBox_nb_p.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_nb_p.setMaximum(1000)
        self.spinBox_nb_p.setProperty("value", 11)
        self.spinBox_nb_p.setObjectName("spinBox_nb_p")
        self.formLayout.setWidget(2, QtWidgets.QFormLayout.FieldRole, self.spinBox_nb_
↪p)
        self.label_p = QtWidgets.QLabel(self.tab_array)
        self.label_p.setMinimumSize(QtCore.QSize(120, 0))
        self.label_p.setObjectName("label_p")
        self.formLayout.setWidget(3, QtWidgets.QFormLayout.LabelRole, self.label_p)
        self.doubleSpinBox_p = QtWidgets.QDoubleSpinBox(self.tab_array)
        self.doubleSpinBox_p.setMinimumSize(QtCore.QSize(80, 0))
        self.doubleSpinBox_p.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_p.setDecimals(3)
        self.doubleSpinBox_p.setMaximum(10000.0)
        self.doubleSpinBox_p.setProperty("value", 15.0)
        self.doubleSpinBox_p.setObjectName("doubleSpinBox_p")
        self.formLayout.setWidget(3, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_p)
        self.doubleSpinBox_offset_guide_z = QtWidgets.QDoubleSpinBox(self.tab_array)
        self.doubleSpinBox_offset_guide_z.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_offset_guide_z.setCorrectionMode(QtWidgets.
↪QAbstractSpinBox.CorrectToNearestValue)
        self.doubleSpinBox_offset_guide_z.setDecimals(3)
        self.doubleSpinBox_offset_guide_z.setMaximum(100000.0)
        self.doubleSpinBox_offset_guide_z.setObjectName("doubleSpinBox_offset_guide_z
↪")
        self.formLayout.setWidget(0, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_offset_guide_z)
        self.label_offset_guide_z = QtWidgets.QLabel(self.tab_array)
        self.label_offset_guide_z.setMinimumSize(QtCore.QSize(40, 0))
        self.label_offset_guide_z.setObjectName("label_offset_guide_z")
        self.formLayout.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.label_
↪offset_guide_z)
        self.doubleSpinBox_guide_length = QtWidgets.QDoubleSpinBox(self.tab_array)
        self.doubleSpinBox_guide_length.setMinimumSize(QtCore.QSize(100, 0))
        self.doubleSpinBox_guide_length.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_guide_length.setDecimals(3)
        self.doubleSpinBox_guide_length.setMaximum(100000.0)
        self.doubleSpinBox_guide_length.setProperty("value", 100000.0)
```

(continues on next page)

(continued from previous page)

```
        self.doubleSpinBox_guide_length.setObjectName("doubleSpinBox_guide_length")
        self.formLayout.setWidget(1, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_guide_length)
        self.label_guide_length = QtWidgets.QLabel(self.tab_array)
        self.label_guide_length.setMinimumSize(QtCore.QSize(40, 0))
        self.label_guide_length.setObjectName("label_guide_length")
        self.formLayout.setWidget(1, QtWidgets.QFormLayout.LabelRole, self.label_
↪guide_length)
        self.tabWidget_morphology_guide.addTab(self.tab_array, "")
        self.tab_curved = QtWidgets.QWidget()
        self.tab_curved.setObjectName("tab_curved")
        self.formLayout_2 = QtWidgets.QFormLayout(self.tab_curved)
        self.formLayout_2.setObjectName("formLayout_2")
        self.label_curve = QtWidgets.QLabel(self.tab_curved)
        self.label_curve.setMinimumSize(QtCore.QSize(60, 0))
        self.label_curve.setObjectName("label_curve")
        self.formLayout_2.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.label_
↪curve)
        self.doubleSpinBox_curve = QtWidgets.QDoubleSpinBox(self.tab_curved)
        self.doubleSpinBox_curve.setMinimumSize(QtCore.QSize(140, 0))
        self.doubleSpinBox_curve.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_curve.setDecimals(3)
        self.doubleSpinBox_curve.setMinimum(-100000000.0)
        self.doubleSpinBox_curve.setMaximum(100000000.0)
        self.doubleSpinBox_curve.setProperty("value", 40.0)
        self.doubleSpinBox_curve.setObjectName("doubleSpinBox_curve")
        self.formLayout_2.setWidget(0, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_curve)
        self.label_half_delay = QtWidgets.QLabel(self.tab_curved)
        self.label_half_delay.setMinimumSize(QtCore.QSize(60, 0))
        self.label_half_delay.setObjectName("label_half_delay")
        self.formLayout_2.setWidget(1, QtWidgets.QFormLayout.LabelRole, self.label_
↪half_delay)
        self.doubleSpinBox_half_delay = QtWidgets.QDoubleSpinBox(self.tab_curved)
        self.doubleSpinBox_half_delay.setMinimumSize(QtCore.QSize(90, 0))
        self.doubleSpinBox_half_delay.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_half_delay.setDecimals(3)
        self.doubleSpinBox_half_delay.setMaximum(100000.0)
        self.doubleSpinBox_half_delay.setProperty("value", 1000.0)
        self.doubleSpinBox_half_delay.setObjectName("doubleSpinBox_half_delay")
        self.formLayout_2.setWidget(1, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_half_delay)
        self.label_distance_factor = QtWidgets.QLabel(self.tab_curved)
        self.label_distance_factor.setMinimumSize(QtCore.QSize(60, 0))
        self.label_distance_factor.setObjectName("label_distance_factor")
        self.formLayout_2.setWidget(2, QtWidgets.QFormLayout.LabelRole, self.label_
↪distance_factor)
        self.doubleSpinBox_distance_factor = QtWidgets.QDoubleSpinBox(self.tab_curved)
        self.doubleSpinBox_distance_factor.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_distance_factor.setCorrectionMode(QtWidgets.
↪QAbstractSpinBox.CorrectToNearestValue)
        self.doubleSpinBox_distance_factor.setDecimals(3)
        self.doubleSpinBox_distance_factor.setMaximum(10000.0)
        self.doubleSpinBox_distance_factor.setProperty("value", 1.2)
        self.doubleSpinBox_distance_factor.setObjectName("doubleSpinBox_distance_
↪factor")
```

(continues on next page)

(continued from previous page)

```
        self.formLayout_2.setWidget(2, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_distance_factor)
        self.tabWidget_morphology_guide.addTab(self.tab_curved, "")
        self.formLayout_6.setWidget(12, QtWidgets.QFormLayout.SpanningRole, self.
↪tabWidget_morphology_guide)
        self.verticalLayout_guide.addWidget(self.frame_guides)
        self.calculateButton_guide = QtWidgets.QPushButton(self.tabWidget_guide)
        self.calculateButton_guide.setMaximumSize(QtCore.QSize(350, 16777215))
        self.calculateButton_guide.setCursor(QtGui.QCursor(QtCore.Qt.
↪PointingHandCursor))
        self.calculateButton_guide.setObjectName("calculateButton_guide")
        self.verticalLayout_guide.addWidget(self.calculateButton_guide)
        spacerItem1 = QtWidgets.QSpacerItem(20, 10, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Maximum)
        self.verticalLayout_guide.addItem(spacerItem1)
        spacerItem2 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Expanding)
        self.verticalLayout_guide.addItem(spacerItem2)
        self.horizontalLayout_2.addLayout(self.verticalLayout_guide)
        self.plot_guide = QtWidgets.QVBoxLayout()
        self.plot_guide.setObjectName("plot_guide")
        self.horizontalLayout_2.addLayout(self.plot_guide)
        self.tabWidget_main.addTab(self.tabWidget_guide, "")
        self.tabWidget_light = QtWidgets.QWidget()
        self.tabWidget_light.setObjectName("tabWidget_light")
        self.horizontalLayout_4 = QtWidgets.QHBoxLayout(self.tabWidget_light)
        self.horizontalLayout_4.setObjectName("horizontalLayout_4")
        self.verticalLayout_light = QtWidgets.QVBoxLayout()
        self.verticalLayout_light.setObjectName("verticalLayout_light")
        self.label_light_informations = QtWidgets.QLabel(self.tabWidget_light)
        self.label_light_informations.setMinimumSize(QtCore.QSize(0, 62))
        self.label_light_informations.setMaximumSize(QtCore.QSize(16777215, 30))
        self.label_light_informations.setObjectName("label_light_informations")
        self.verticalLayout_light.addWidget(self.label_light_informations)
        self.frame_light = QtWidgets.QFrame(self.tabWidget_light)
        self.frame_light.setMinimumSize(QtCore.QSize(120, 0))
        self.frame_light.setMaximumSize(QtCore.QSize(120, 16777215))
        self.frame_light.setFrameShape(QtWidgets.QFrame.Box)
        self.frame_light.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_light.setObjectName("frame_light")
        self.formLayout_lo_theta = QtWidgets.QFormLayout(self.frame_light)
        self.formLayout_lo_theta.setObjectName("formLayout_lo_theta")
        self.label_theta_ext = QtWidgets.QLabel(self.frame_light)
        self.label_theta_ext.setObjectName("label_theta_ext")
        self.formLayout_lo_theta.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.
↪label_theta_ext)
        self.doubleSpinBox_theta_ext = QtWidgets.QDoubleSpinBox(self.frame_light)
        self.doubleSpinBox_theta_ext.setMinimumSize(QtCore.QSize(60, 0))
        self.doubleSpinBox_theta_ext.setMaximumSize(QtCore.QSize(60, 16777215))
        self.doubleSpinBox_theta_ext.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_theta_ext.setMinimum(-28.0)
        self.doubleSpinBox_theta_ext.setMaximum(28.0)
        self.doubleSpinBox_theta_ext.setObjectName("doubleSpinBox_theta_ext")
        self.formLayout_lo_theta.setWidget(0, QtWidgets.QFormLayout.FieldRole, self.
↪doubleSpinBox_theta_ext)
        self.verticalLayout_light.addWidget(self.frame_light)
```

(continues on next page)

(continued from previous page)

```
        spacerItem3 = QtWidgets.QSpacerItem(20, 10, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Maximum)
        self.verticalLayout_light.addItem(spacerItem3)
        self.frame_beam = QtWidgets.QFrame(self.tabWidget_light)
        self.frame_beam.setMinimumSize(QtCore.QSize(250, 0))
        self.frame_beam.setMaximumSize(QtCore.QSize(350, 16777215))
        self.frame_beam.setFrameShape(QtWidgets.QFrame.Box)
        self.frame_beam.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_beam.setObjectName("frame_beam")
        self.gridLayout_5 = QtWidgets.QGridLayout(self.frame_beam)
        self.gridLayout_5.setObjectName("gridLayout_5")
        self.label_offset_light = QtWidgets.QLabel(self.frame_beam)
        self.label_offset_light.setEnabled(True)
        self.label_offset_light.setMaximumSize(QtCore.QSize(60, 16777215))
        self.label_offset_light.setObjectName("label_offset_light")
        self.gridLayout_5.addWidget(self.label_offset_light, 4, 0, 1, 1)
        self.label_intensity = QtWidgets.QLabel(self.frame_beam)
        self.label_intensity.setMaximumSize(QtCore.QSize(60, 16777215))
        self.label_intensity.setObjectName("label_intensity")
        self.gridLayout_5.addWidget(self.label_intensity, 5, 0, 1, 1)
        self.radioButton_mode = QtWidgets.QRadioButton(self.frame_beam)
        self.radioButton_mode.setMaximumSize(QtCore.QSize(70, 16777215))
        self.radioButton_mode.setObjectName("radioButton_mode")
        self.gridLayout_5.addWidget(self.radioButton_mode, 11, 0, 1, 1)
        self.radioButton_gaussian_light = QtWidgets.QRadioButton(self.frame_beam)
        self.radioButton_gaussian_light.setEnabled(True)
        self.radioButton_gaussian_light.setMaximumSize(QtCore.QSize(70, 16777215))
        self.radioButton_gaussian_light.setChecked(True)
        self.radioButton_gaussian_light.setObjectName("radioButton_gaussian_light")
        self.gridLayout_5.addWidget(self.radioButton_gaussian_light, 7, 0, 1, 2)
        self.radioButton_squared_light = QtWidgets.QRadioButton(self.frame_beam)
        self.radioButton_squared_light.setMaximumSize(QtCore.QSize(70, 16777215))
        self.radioButton_squared_light.setChecked(False)
        self.radioButton_squared_light.setObjectName("radioButton_squared_light")
        self.gridLayout_5.addWidget(self.radioButton_squared_light, 9, 0, 1, 2)
        spacerItem4 = QtWidgets.QSpacerItem(20, 5, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Minimum)
        self.gridLayout_5.addItem(spacerItem4, 6, 0, 1, 1)
        spacerItem5 = QtWidgets.QSpacerItem(20, 5, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Minimum)
        self.gridLayout_5.addItem(spacerItem5, 8, 0, 1, 1)
        spacerItem6 = QtWidgets.QSpacerItem(20, 5, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Minimum)
        self.gridLayout_5.addItem(spacerItem6, 10, 0, 1, 1)
        self.label_guide_nbr_ref = QtWidgets.QLabel(self.frame_beam)
        self.label_guide_nbr_ref.setMaximumSize(QtCore.QSize(90, 16777215))
        self.label_guide_nbr_ref.setObjectName("label_guide_nbr_ref")
        self.gridLayout_5.addWidget(self.label_guide_nbr_ref, 12, 0, 1, 1)
        self.label_fwhm = QtWidgets.QLabel(self.frame_beam)
        self.label_fwhm.setMinimumSize(QtCore.QSize(30, 0))
        self.label_fwhm.setMaximumSize(QtCore.QSize(40, 16777215))
        self.label_fwhm.setObjectName("label_fwhm")
        self.gridLayout_5.addWidget(self.label_fwhm, 2, 0, 1, 1)
        self.doubleSpinBox_fwhm = QtWidgets.QDoubleSpinBox(self.frame_beam)
        self.doubleSpinBox_fwhm.setMinimumSize(QtCore.QSize(70, 0))
        self.doubleSpinBox_fwhm.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
```

(continues on next page)

(continued from previous page)

```
        self.doubleSpinBox_fwhm.setDecimals(3)
        self.doubleSpinBox_fwhm.setMaximum(10000.0)
        self.doubleSpinBox_fwhm.setProperty("value", 8.0)
        self.doubleSpinBox_fwhm.setObjectName("doubleSpinBox_fwhm")
        self.gridLayout_5.addWidget(self.doubleSpinBox_fwhm, 2, 1, 1, 5)
        self.comboBox_light = QtWidgets.QComboBox(self.frame_beam)
        self.comboBox_light.setMaximumSize(QtCore.QSize(100, 16777215))
        self.comboBox_light.setToolTipDuration(-1)
        self.comboBox_light.setObjectName("comboBox_light")
        self.gridLayout_5.addWidget(self.comboBox_light, 0, 0, 1, 1)
        self.checkBox_offset_light = QtWidgets.QCheckBox(self.frame_beam)
        self.checkBox_offset_light.setMaximumSize(QtCore.QSize(100, 16777215))
        self.checkBox_offset_light.setChecked(True)
        self.checkBox_offset_light.setObjectName("checkBox_offset_light")
        self.gridLayout_5.addWidget(self.checkBox_offset_light, 3, 0, 1, 1)
        self.label_lobe_size = QtWidgets.QLabel(self.frame_beam)
        self.label_lobe_size.setMaximumSize(QtCore.QSize(70, 16777215))
        self.label_lobe_size.setObjectName("label_lobe_size")
        self.gridLayout_5.addWidget(self.label_lobe_size, 19, 0, 1, 1)
        self.radioButton_airy = QtWidgets.QRadioButton(self.frame_beam)
        self.radioButton_airy.setMaximumSize(QtCore.QSize(50, 16777215))
        self.radioButton_airy.setObjectName("radioButton_airy")
        self.gridLayout_5.addWidget(self.radioButton_airy, 17, 0, 1, 1)
        self.label_zero_cut = QtWidgets.QLabel(self.frame_beam)
        self.label_zero_cut.setMaximumSize(QtCore.QSize(50, 16777215))
        self.label_zero_cut.setObjectName("label_zero_cut")
        self.gridLayout_5.addWidget(self.label_zero_cut, 18, 0, 1, 1)
        self.mode_number = QtWidgets.QLCDNumber(self.frame_beam)
        self.mode_number.setMaximumSize(QtCore.QSize(100, 16777215))
        self.mode_number.setObjectName("mode_number")
        self.gridLayout_5.addWidget(self.mode_number, 13, 1, 1, 1)
        self.pushButton_mode_number = QtWidgets.QPushButton(self.frame_beam)
        self.pushButton_mode_number.setMaximumSize(QtCore.QSize(90, 16777215))
        self.pushButton_mode_number.setObjectName("pushButton_mode_number")
        self.gridLayout_5.addWidget(self.pushButton_mode_number, 13, 0, 1, 1)
        self.radioButton_all_modes = QtWidgets.QRadioButton(self.frame_beam)
        self.radioButton_all_modes.setMaximumSize(QtCore.QSize(70, 16777215))
        self.radioButton_all_modes.setObjectName("radioButton_all_modes")
        self.gridLayout_5.addWidget(self.radioButton_all_modes, 15, 0, 1, 1)
        spacerItem7 = QtWidgets.QSpacerItem(20, 5, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Minimum)
        self.gridLayout_5.addItem(spacerItem7, 16, 0, 1, 1)
        spacerItem8 = QtWidgets.QSpacerItem(20, 5, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Minimum)
        self.gridLayout_5.addItem(spacerItem8, 14, 0, 1, 1)
        self.spinBox_offset_light_peak = QtWidgets.QSpinBox(self.frame_beam)
        self.spinBox_offset_light_peak.setMinimumSize(QtCore.QSize(105, 0))
        self.spinBox_offset_light_peak.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_offset_light_peak.setSuffix("")
        self.spinBox_offset_light_peak.setMaximum(10000)
        self.spinBox_offset_light_peak.setProperty("value", 5)
        self.spinBox_offset_light_peak.setObjectName("spinBox_offset_light_peak")
        self.gridLayout_5.addWidget(self.spinBox_offset_light_peak, 3, 1, 1, 5)
        self.doubleSpinBox_offset_light = QtWidgets.QDoubleSpinBox(self.frame_beam)
        self.doubleSpinBox_offset_light.setEnabled(False)
        self.doubleSpinBox_offset_light.setMinimumSize(QtCore.QSize(60, 0))
```

(continues on next page)

(continued from previous page)

```
        self.doubleSpinBox_offset_light.setMinimum(-10000.0)
        self.doubleSpinBox_offset_light.setMaximum(10000.0)
        self.doubleSpinBox_offset_light.setObjectName("doubleSpinBox_offset_light")
        self.gridLayout_5.addWidget(self.doubleSpinBox_offset_light, 4, 1, 1, 5)
        self.label_3 = QtWidgets.QLabel(self.frame_beam)
        self.label_3.setMinimumSize(QtCore.QSize(40, 0))
        self.label_3.setObjectName("label_3")
        self.gridLayout_5.addWidget(self.label_3, 5, 5, 1, 1)
        self.spinBox_irrad_exponent = QtWidgets.QSpinBox(self.frame_beam)
        self.spinBox_irrad_exponent.setMinimumSize(QtCore.QSize(50, 0))
        self.spinBox_irrad_exponent.setProperty("value", 13)
        self.spinBox_irrad_exponent.setObjectName("spinBox_irrad_exponent")
        self.gridLayout_5.addWidget(self.spinBox_irrad_exponent, 5, 4, 1, 1)
        self.label_2 = QtWidgets.QLabel(self.frame_beam)
        self.label_2.setMinimumSize(QtCore.QSize(25, 0))
        self.label_2.setObjectName("label_2")
        self.gridLayout_5.addWidget(self.label_2, 5, 3, 1, 1)
        self.spinBox_mode = QtWidgets.QSpinBox(self.frame_beam)
        self.spinBox_mode.setEnabled(False)
        self.spinBox_mode.setMinimumSize(QtCore.QSize(30, 0))
        self.spinBox_mode.setAutoFillBackground(False)
        self.spinBox_mode.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_mode.setMaximum(50)
        self.spinBox_mode.setObjectName("spinBox_mode")
        self.gridLayout_5.addWidget(self.spinBox_mode, 11, 1, 1, 5)
        self.spinBox_guide_nbr_ref = QtWidgets.QSpinBox(self.frame_beam)
        self.spinBox_guide_nbr_ref.setEnabled(False)
        self.spinBox_guide_nbr_ref.setMinimumSize(QtCore.QSize(30, 0))
        self.spinBox_guide_nbr_ref.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_guide_nbr_ref.setObjectName("spinBox_guide_nbr_ref")
        self.gridLayout_5.addWidget(self.spinBox_guide_nbr_ref, 12, 1, 1, 5)
        self.spinBox_airy_zero = QtWidgets.QSpinBox(self.frame_beam)
        self.spinBox_airy_zero.setEnabled(False)
        self.spinBox_airy_zero.setMinimumSize(QtCore.QSize(40, 0))
        self.spinBox_airy_zero.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_airy_zero.setMinimum(1)
        self.spinBox_airy_zero.setMaximum(10000)
        self.spinBox_airy_zero.setProperty("value", 10)
        self.spinBox_airy_zero.setObjectName("spinBox_airy_zero")
        self.gridLayout_5.addWidget(self.spinBox_airy_zero, 18, 1, 1, 5)
        self.doubleSpinBox_irrad_significand = QtWidgets.QDoubleSpinBox(self.frame_
↪beam)
        self.doubleSpinBox_irrad_significand.setMinimumSize(QtCore.QSize(50, 0))
        self.doubleSpinBox_irrad_significand.setCorrectionMode(QtWidgets.
↪QAbstractSpinBox.CorrectToNearestValue)
        self.doubleSpinBox_irrad_significand.setSuffix("")
        self.doubleSpinBox_irrad_significand.setDecimals(2)
        self.doubleSpinBox_irrad_significand.setMaximum(9.99)
        self.doubleSpinBox_irrad_significand.setProperty("value", 1.0)
        self.doubleSpinBox_irrad_significand.setObjectName("doubleSpinBox_irrad_
↪significand")
        self.gridLayout_5.addWidget(self.doubleSpinBox_irrad_significand, 5, 1, 1, 2)
        self.doubleSpinBox_lobe_size = QtWidgets.QDoubleSpinBox(self.frame_beam)
        self.doubleSpinBox_lobe_size.setEnabled(False)
```

(continues on next page)

(continued from previous page)

```
        self.doubleSpinBox_lobe_size.setMinimumSize(QtCore.QSize(60, 0))
        self.doubleSpinBox_lobe_size.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.doubleSpinBox_lobe_size.setMaximum(10000.0)
        self.doubleSpinBox_lobe_size.setProperty("value", 8.0)
        self.doubleSpinBox_lobe_size.setObjectName("doubleSpinBox_lobe_size")
        self.gridLayout_5.addWidget(self.doubleSpinBox_lobe_size, 19, 1, 1, 5)
        self.Qframe_beam_create = QtWidgets.QFrame(self.frame_beam)
        self.Qframe_beam_create.setFrameShape(QtWidgets.QFrame.NoFrame)
        self.Qframe_beam_create.setFrameShadow(QtWidgets.QFrame.Raised)
        self.Qframe_beam_create.setObjectName("Qframe_beam_create")
        self.gridLayout_creat_delete_beam = QtWidgets.QGridLayout(self.Qframe_beam_
↪create)
        self.gridLayout_creat_delete_beam.setObjectName("gridLayout_creat_delete_beam
↪")
        self.pushButton_delete_beam = QtWidgets.QPushButton(self.Qframe_beam_create)
        self.pushButton_delete_beam.setObjectName("pushButton_delete_beam")
        self.gridLayout_creat_delete_beam.addWidget(self.pushButton_delete_beam, 2, 1,
↪ 1, 1)
        self.pushButton_create_beam = QtWidgets.QPushButton(self.Qframe_beam_create)
        self.pushButton_create_beam.setObjectName("pushButton_create_beam")
        self.gridLayout_creat_delete_beam.addWidget(self.pushButton_create_beam, 2, 0,
↪ 1, 1)
        self.pushButton_save_beam = QtWidgets.QPushButton(self.Qframe_beam_create)
        self.pushButton_save_beam.setObjectName("pushButton_save_beam")
        self.gridLayout_creat_delete_beam.addWidget(self.pushButton_save_beam, 3, 0,␣
↪1, 1)
        self.pushButton_cancel_light = QtWidgets.QPushButton(self.Qframe_beam_create)
        self.pushButton_cancel_light.setObjectName("pushButton_cancel_light")
        self.gridLayout_creat_delete_beam.addWidget(self.pushButton_cancel_light, 3,␣
↪1, 1, 1)
        self.gridLayout_5.addWidget(self.Qframe_beam_create, 1, 0, 1, 6)
        self.verticalLayout_light.addWidget(self.frame_beam)
        spacerItem9 = QtWidgets.QSpacerItem(20, 10, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Maximum)
        self.verticalLayout_light.addItem(spacerItem9)
        self.calculateButton_light = QtWidgets.QPushButton(self.tabWidget_light)
        self.calculateButton_light.setMinimumSize(QtCore.QSize(100, 0))
        self.calculateButton_light.setMaximumSize(QtCore.QSize(350, 16777215))
        self.calculateButton_light.setCursor(QtGui.QCursor(QtCore.Qt.
↪PointingHandCursor))
        self.calculateButton_light.setObjectName("calculateButton_light")
        self.verticalLayout_light.addWidget(self.calculateButton_light)
        spacerItem10 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Expanding)
        self.verticalLayout_light.addItem(spacerItem10)
        self.horizontalLayout_4.addLayout(self.verticalLayout_light)
        self.plot_light = QtWidgets.QVBoxLayout()
        self.plot_light.setObjectName("plot_light")
        self.horizontalLayout_4.addLayout(self.plot_light)
        self.tabWidget_main.addTab(self.tabWidget_light, "")
        self.tabWidget_compute = QtWidgets.QWidget()
        self.tabWidget_compute.setObjectName("tabWidget_compute")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.tabWidget_compute)
        self.horizontalLayout.setObjectName("horizontalLayout")
        self.verticalLayout_compute_plot = QtWidgets.QVBoxLayout()
        self.verticalLayout_compute_plot.setObjectName("verticalLayout_compute_plot")
```

(continues on next page)

(continued from previous page)

```
        self.verticalLayout_compute = QtWidgets.QVBoxLayout()
        self.verticalLayout_compute.setObjectName("verticalLayout_compute")
        self.label_kerr_informations = QtWidgets.QLabel(self.tabWidget_compute)
        self.label_kerr_informations.setMinimumSize(QtCore.QSize(0, 62))
        self.label_kerr_informations.setMaximumSize(QtCore.QSize(16777215, 30))
        self.label_kerr_informations.setMouseTracking(True)
        self.label_kerr_informations.setObjectName("label_kerr_informations")
        self.verticalLayout_compute.addWidget(self.label_kerr_informations)
        self.checkBox_kerr = QtWidgets.QCheckBox(self.tabWidget_compute)
        self.checkBox_kerr.setEnabled(True)
        self.checkBox_kerr.setChecked(False)
        self.checkBox_kerr.setObjectName("checkBox_kerr")
        self.verticalLayout_compute.addWidget(self.checkBox_kerr)
        self.frame_kerr = QtWidgets.QFrame(self.tabWidget_compute)
        self.frame_kerr.setEnabled(False)
        self.frame_kerr.setMinimumSize(QtCore.QSize(250, 0))
        self.frame_kerr.setMaximumSize(QtCore.QSize(300, 16777215))
        self.frame_kerr.setFrameShape(QtWidgets.QFrame.Box)
        self.frame_kerr.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_kerr.setObjectName("frame_kerr")
        self.gridLayout_3 = QtWidgets.QGridLayout(self.frame_kerr)
        self.gridLayout_3.setObjectName("gridLayout_3")
        self.doubleSpinBox_n2_significand = QtWidgets.QDoubleSpinBox(self.frame_kerr)
        self.doubleSpinBox_n2_significand.setMinimumSize(QtCore.QSize(50, 0))
        self.doubleSpinBox_n2_significand.setMaximumSize(QtCore.QSize(70, 16777215))
        self.doubleSpinBox_n2_significand.setCorrectionMode(QtWidgets.
↪QAbstractSpinBox.CorrectToNearestValue)
        self.doubleSpinBox_n2_significand.setSuffix("")
        self.doubleSpinBox_n2_significand.setMaximum(9.99)
        self.doubleSpinBox_n2_significand.setProperty("value", 1.0)
        self.doubleSpinBox_n2_significand.setObjectName("doubleSpinBox_n2_significand
↪")
        self.gridLayout_3.addWidget(self.doubleSpinBox_n2_significand, 1, 1, 1, 1)
        self.spinBox_kerr_loop = QtWidgets.QSpinBox(self.frame_kerr)
        self.spinBox_kerr_loop.setMinimumSize(QtCore.QSize(70, 0))
        self.spinBox_kerr_loop.setMaximumSize(QtCore.QSize(70, 16777215))
        self.spinBox_kerr_loop.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_kerr_loop.setMinimum(1)
        self.spinBox_kerr_loop.setMaximum(100)
        self.spinBox_kerr_loop.setProperty("value", 2)
        self.spinBox_kerr_loop.setObjectName("spinBox_kerr_loop")
        self.gridLayout_3.addWidget(self.spinBox_kerr_loop, 0, 1, 1, 1)
        self.label_chi3 = QtWidgets.QLabel(self.frame_kerr)
        self.label_chi3.setMaximumSize(QtCore.QSize(50, 16777215))
        self.label_chi3.setObjectName("label_chi3")
        self.gridLayout_3.addWidget(self.label_chi3, 3, 0, 1, 1)
        self.doubleSpinBox_chi3_significand = QtWidgets.QDoubleSpinBox(self.frame_
↪kerr)
        self.doubleSpinBox_chi3_significand.setEnabled(False)
        self.doubleSpinBox_chi3_significand.setMinimumSize(QtCore.QSize(50, 0))
        self.doubleSpinBox_chi3_significand.setMaximumSize(QtCore.QSize(70, 16777215))
        self.doubleSpinBox_chi3_significand.setCorrectionMode(QtWidgets.
↪QAbstractSpinBox.CorrectToNearestValue)
        self.doubleSpinBox_chi3_significand.setSuffix("")
        self.doubleSpinBox_chi3_significand.setMaximum(9.99)
        self.doubleSpinBox_chi3_significand.setProperty("value", 1.0)
```

(continues on next page)

(continued from previous page)

```
        self.doubleSpinBox_chi3_significand.setObjectName("doubleSpinBox_chi3_
↪significand")
        self.gridLayout_3.addWidget(self.doubleSpinBox_chi3_significand, 3, 1, 1, 1)
        self.label_correction_loop = QtWidgets.QLabel(self.frame_kerr)
        self.label_correction_loop.setMaximumSize(QtCore.QSize(50, 16777215))
        self.label_correction_loop.setObjectName("label_correction_loop")
        self.gridLayout_3.addWidget(self.label_correction_loop, 0, 0, 1, 1)
        self.checkBox_n2 = QtWidgets.QCheckBox(self.frame_kerr)
        self.checkBox_n2.setMaximumSize(QtCore.QSize(50, 16777215))
        self.checkBox_n2.setChecked(True)
        self.checkBox_n2.setObjectName("checkBox_n2")
        self.gridLayout_3.addWidget(self.checkBox_n2, 1, 0, 1, 1)
        self.spinBox_n2_exponent = QtWidgets.QSpinBox(self.frame_kerr)
        self.spinBox_n2_exponent.setMinimumSize(QtCore.QSize(50, 0))
        self.spinBox_n2_exponent.setMaximumSize(QtCore.QSize(50, 16777215))
        self.spinBox_n2_exponent.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_n2_exponent.setSuffix("")
        self.spinBox_n2_exponent.setMinimum(-50)
        self.spinBox_n2_exponent.setMaximum(50)
        self.spinBox_n2_exponent.setProperty("value", -20)
        self.spinBox_n2_exponent.setObjectName("spinBox_n2_exponent")
        self.gridLayout_3.addWidget(self.spinBox_n2_exponent, 1, 3, 1, 1)
        self.label_n2_unit = QtWidgets.QLabel(self.frame_kerr)
        self.label_n2_unit.setMinimumSize(QtCore.QSize(40, 0))
        self.label_n2_unit.setMaximumSize(QtCore.QSize(40, 16777215))
        self.label_n2_unit.setObjectName("label_n2_unit")
        self.gridLayout_3.addWidget(self.label_n2_unit, 1, 4, 1, 1)
        self.spinBox_chi3_exponent = QtWidgets.QSpinBox(self.frame_kerr)
        self.spinBox_chi3_exponent.setEnabled(False)
        self.spinBox_chi3_exponent.setMinimumSize(QtCore.QSize(50, 0))
        self.spinBox_chi3_exponent.setMaximumSize(QtCore.QSize(50, 16777215))
        self.spinBox_chi3_exponent.setCorrectionMode(QtWidgets.QAbstractSpinBox.
↪CorrectToNearestValue)
        self.spinBox_chi3_exponent.setSuffix("")
        self.spinBox_chi3_exponent.setMinimum(-50)
        self.spinBox_chi3_exponent.setMaximum(50)
        self.spinBox_chi3_exponent.setProperty("value", -22)
        self.spinBox_chi3_exponent.setObjectName("spinBox_chi3_exponent")
        self.gridLayout_3.addWidget(self.spinBox_chi3_exponent, 3, 3, 1, 1)
        self.label_n2_exponent = QtWidgets.QLabel(self.frame_kerr)
        self.label_n2_exponent.setMinimumSize(QtCore.QSize(25, 0))
        self.label_n2_exponent.setMaximumSize(QtCore.QSize(25, 16777215))
        self.label_n2_exponent.setObjectName("label_n2_exponent")
        self.gridLayout_3.addWidget(self.label_n2_exponent, 1, 2, 1, 1)
        self.label_chi3_unit = QtWidgets.QLabel(self.frame_kerr)
        self.label_chi3_unit.setMinimumSize(QtCore.QSize(50, 0))
        self.label_chi3_unit.setMaximumSize(QtCore.QSize(50, 16777215))
        self.label_chi3_unit.setObjectName("label_chi3_unit")
        self.gridLayout_3.addWidget(self.label_chi3_unit, 3, 4, 1, 1)
        self.label_chi3_exponent = QtWidgets.QLabel(self.frame_kerr)
        self.label_chi3_exponent.setMinimumSize(QtCore.QSize(25, 0))
        self.label_chi3_exponent.setMaximumSize(QtCore.QSize(25, 16777215))
        self.label_chi3_exponent.setObjectName("label_chi3_exponent")
        self.gridLayout_3.addWidget(self.label_chi3_exponent, 3, 2, 1, 1)
        self.checkBox_variance = QtWidgets.QCheckBox(self.frame_kerr)
        self.checkBox_variance.setChecked(True)
```

(continues on next page)

(continued from previous page)

```
        self.checkBox_variance.setObjectName("checkBox_variance")
        self.gridLayout_3.addWidget(self.checkBox_variance, 4, 0, 1, 5)
        self.verticalLayout_compute.addWidget(self.frame_kerr)
        self.formLayout_estimate = QtWidgets.QFormLayout()
        self.formLayout_estimate.setObjectName("formLayout_estimate")
        self.estimate_time_display = QtWidgets.QLCDNumber(self.tabWidget_compute)
        self.estimate_time_display.setEnabled(True)
        self.estimate_time_display.setMinimumSize(QtCore.QSize(100, 0))
        self.estimate_time_display.setMaximumSize(QtCore.QSize(100, 16777215))
        self.estimate_time_display.setDigitCount(3)
        self.estimate_time_display.setSegmentStyle(QtWidgets.QLCDNumber.Filled)
        self.estimate_time_display.setObjectName("estimate_time_display")
        self.formLayout_estimate.setWidget(0, QtWidgets.QFormLayout.FieldRole, self.
↪estimate_time_display)
        self.pushButton_estimate_time = QtWidgets.QPushButton(self.tabWidget_compute)
        self.pushButton_estimate_time.setMinimumSize(QtCore.QSize(150, 0))
        self.pushButton_estimate_time.setMaximumSize(QtCore.QSize(150, 16777215))
        self.pushButton_estimate_time.setObjectName("pushButton_estimate_time")
        self.formLayout_estimate.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.
↪pushButton_estimate_time)
        self.verticalLayout_compute.addLayout(self.formLayout_estimate)
        spacerItem11 = QtWidgets.QSpacerItem(20, 10, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Maximum)
        self.verticalLayout_compute.addItem(spacerItem11)
        self.calculateButton_compute = QtWidgets.QPushButton(self.tabWidget_compute)
        self.calculateButton_compute.setMinimumSize(QtCore.QSize(250, 0))
        self.calculateButton_compute.setMaximumSize(QtCore.QSize(300, 16777215))
        self.calculateButton_compute.setCursor(QtGui.QCursor(QtCore.Qt.
↪PointingHandCursor))
        self.calculateButton_compute.setObjectName("calculateButton_compute")
        self.verticalLayout_compute.addWidget(self.calculateButton_compute)
        self.progressBar_compute = QtWidgets.QProgressBar(self.tabWidget_compute)
        self.progressBar_compute.setEnabled(True)
        self.progressBar_compute.setMinimumSize(QtCore.QSize(250, 0))
        self.progressBar_compute.setMaximumSize(QtCore.QSize(300, 16777215))
        self.progressBar_compute.setContextMenuPolicy(QtCore.Qt.ActionsContextMenu)
        self.progressBar_compute.setStyleSheet("background-color : white; border : 1px
↪")
        self.progressBar_compute.setProperty("value", 0)
        self.progressBar_compute.setAlignment(QtCore.Qt.AlignCenter)
        self.progressBar_compute.setTextVisible(True)
        self.progressBar_compute.setOrientation(QtCore.Qt.Horizontal)
        self.progressBar_compute.setObjectName("progressBar_compute")
        self.verticalLayout_compute.addWidget(self.progressBar_compute)
        spacerItem12 = QtWidgets.QSpacerItem(20, 10, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Maximum)
        self.verticalLayout_compute.addItem(spacerItem12)
        self.formLayout_power = QtWidgets.QFormLayout()
        self.formLayout_power.setObjectName("formLayout_power")
        self.checkBox_power = QtWidgets.QCheckBox(self.tabWidget_compute)
        self.checkBox_power.setMinimumSize(QtCore.QSize(90, 0))
        self.checkBox_power.setMaximumSize(QtCore.QSize(100, 16777215))
        self.checkBox_power.setObjectName("checkBox_power")
        self.formLayout_power.setWidget(0, QtWidgets.QFormLayout.LabelRole, self.
↪checkBox_power)
        self.pushButton_power = QtWidgets.QPushButton(self.tabWidget_compute)
        self.pushButton_power.setEnabled(False)
```

(continues on next page)

(continued from previous page)

```
        self.pushButton_power.setMinimumSize(QtCore.QSize(80, 0))
        self.pushButton_power.setMaximumSize(QtCore.QSize(100, 16777215))
        self.pushButton_power.setObjectName("pushButton_power")
        self.formLayout_power.setWidget(0, QtWidgets.QFormLayout.FieldRole, self.
↪pushButton_power)
        self.verticalLayout_compute.addLayout(self.formLayout_power)
        spacerItem13 = QtWidgets.QSpacerItem(20, 40, QtWidgets.QSizePolicy.Minimum,␣
↪QtWidgets.QSizePolicy.Expanding)
        self.verticalLayout_compute.addItem(spacerItem13)
        self.verticalLayout_compute_plot.addLayout(self.verticalLayout_compute)
        self.pushButton_cancel_compute = QtWidgets.QPushButton(self.tabWidget_compute)
        self.pushButton_cancel_compute.setMaximumSize(QtCore.QSize(100, 16777215))
        self.pushButton_cancel_compute.setObjectName("pushButton_cancel_compute")
        self.verticalLayout_compute_plot.addWidget(self.pushButton_cancel_compute)
        self.horizontalLayout.addLayout(self.verticalLayout_compute_plot)
        self.plot_compute = QtWidgets.QVBoxLayout()
        self.plot_compute.setObjectName("plot_compute")
        self.horizontalLayout.addLayout(self.plot_compute)
        self.tabWidget_main.addTab(self.tabWidget_compute, "")
        self.verticalLayout.addWidget(self.tabWidget_main)
        self.gridLayout.addWidget(self.frame_file_data, 0, 1, 1, 1)
        MainWindow.setCentralWidget(self.centralwidget)
        self.label_length.setBuddy(self.frame_light)

        self.retranslateUi(MainWindow)
        self.tabWidget_main.setCurrentIndex(0)
        self.tabWidget_morphology_guide.setCurrentIndex(0)
        self.checkBox_offset_light.toggled['bool'].connect(self.doubleSpinBox_offset_
↪light.setDisabled)
        self.checkBox_kerr.clicked['bool'].connect(self.frame_kerr.setEnabled)
        self.checkBox_offset_light.toggled['bool'].connect(self.spinBox_offset_light_
↪peak.setEnabled)
        self.checkBox_kerr.clicked['bool'].connect(self.checkBox_variance.setEnabled)
        self.comboBox_light.activated['QString'].connect(self.pushButton_cancel_light.
↪click)
        self.comboBox_guide.activated['QString'].connect(self.pushButton_cancel_guide.
↪click)
        self.radioButton_airy.toggled['bool'].connect(self.doubleSpinBox_lobe_size.
↪setEnabled)
        self.radioButton_airy.toggled['bool'].connect(self.spinBox_airy_zero.
↪setEnabled)
        self.radioButton_mode.toggled['bool'].connect(self.spinBox_mode.setEnabled)
        self.radioButton_gaussian.toggled['bool'].connect(self.spinBox_gauss_pow.
↪setEnabled)
        self.radioButton_mode.toggled['bool'].connect(self.spinBox_guide_nbr_ref.
↪setEnabled)
        self.checkBox_n_imag.toggled['bool'].connect(self.doubleSpinBox_n_imag.
↪setEnabled)
        self.checkBox_n_imag.toggled['bool'].connect(self.doubleSpinBox_lost.
↪setDisabled)
        self.checkBox_n2.toggled['bool'].connect(self.doubleSpinBox_chi3_significand.
↪setDisabled)
        self.checkBox_n2.toggled['bool'].connect(self.doubleSpinBox_n2_significand.
↪setEnabled)
        self.checkBox_n2.toggled['bool'].connect(self.spinBox_chi3_exponent.
↪setDisabled)
        self.checkBox_n2.toggled['bool'].connect(self.spinBox_n2_exponent.setEnabled)
```

(continues on next page)

(continued from previous page)

```
        self.radioButton_mode.toggled['bool'].connect(self.doubleSpinBox_fwhm.
↪setDisabled)
        self.radioButton_all_modes.toggled['bool'].connect(self.doubleSpinBox_fwhm.
↪setDisabled)
        self.radioButton_all_modes.toggled['bool'].connect(self.spinBox_guide_nbr_ref.
↪setEnabled)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)
        MainWindow.setTabOrder(self.tabWidget_main, self.doubleSpinBox_length_z)
        MainWindow.setTabOrder(self.doubleSpinBox_length_z, self.doubleSpinBox_dist_z)
        MainWindow.setTabOrder(self.doubleSpinBox_dist_z, self.spinBox_nbr_z_disp)
        MainWindow.setTabOrder(self.spinBox_nbr_z_disp, self.doubleSpinBox_length_x)
        MainWindow.setTabOrder(self.doubleSpinBox_length_x, self.doubleSpinBox_dist_x)
        MainWindow.setTabOrder(self.doubleSpinBox_dist_x, self.doubleSpinBox_n)
        MainWindow.setTabOrder(self.doubleSpinBox_n, self.doubleSpinBox_lo)
        MainWindow.setTabOrder(self.doubleSpinBox_lo, self.comboBox_guide)
        MainWindow.setTabOrder(self.comboBox_guide, self.pushButton_create_guide)
        MainWindow.setTabOrder(self.pushButton_create_guide, self.pushButton_delete_
↪guide)
        MainWindow.setTabOrder(self.pushButton_delete_guide, self.pushButton_save_
↪guide)
        MainWindow.setTabOrder(self.pushButton_save_guide, self.pushButton_cancel_
↪guide)
        MainWindow.setTabOrder(self.pushButton_cancel_guide, self.doubleSpinBox_width)
        MainWindow.setTabOrder(self.doubleSpinBox_width, self.doubleSpinBox_offset_
↪guide)
        MainWindow.setTabOrder(self.doubleSpinBox_offset_guide, self.doubleSpinBox_dn)
        MainWindow.setTabOrder(self.doubleSpinBox_dn, self.checkBox_n_imag)
        MainWindow.setTabOrder(self.checkBox_n_imag, self.doubleSpinBox_n_imag)
        MainWindow.setTabOrder(self.doubleSpinBox_n_imag, self.doubleSpinBox_lost)
        MainWindow.setTabOrder(self.doubleSpinBox_lost, self.radioButton_gaussian)
        MainWindow.setTabOrder(self.radioButton_gaussian, self.spinBox_gauss_pow)
        MainWindow.setTabOrder(self.spinBox_gauss_pow, self.radioButton_squared)
        MainWindow.setTabOrder(self.radioButton_squared, self.tabWidget_morphology_
↪guide)
        MainWindow.setTabOrder(self.tabWidget_morphology_guide, self.doubleSpinBox_
↪offset_guide_z)
        MainWindow.setTabOrder(self.doubleSpinBox_offset_guide_z, self.doubleSpinBox_
↪guide_length)
        MainWindow.setTabOrder(self.doubleSpinBox_guide_length, self.spinBox_nb_p)
        MainWindow.setTabOrder(self.spinBox_nb_p, self.doubleSpinBox_p)
        MainWindow.setTabOrder(self.doubleSpinBox_p, self.calculateButton_guide)
        MainWindow.setTabOrder(self.calculateButton_guide, self.doubleSpinBox_curve)
        MainWindow.setTabOrder(self.doubleSpinBox_curve, self.doubleSpinBox_half_
↪delay)
        MainWindow.setTabOrder(self.doubleSpinBox_half_delay, self.doubleSpinBox_
↪distance_factor)
        MainWindow.setTabOrder(self.doubleSpinBox_distance_factor, self.doubleSpinBox_
↪theta_ext)
        MainWindow.setTabOrder(self.doubleSpinBox_theta_ext, self.comboBox_light)
        MainWindow.setTabOrder(self.comboBox_light, self.pushButton_create_beam)
        MainWindow.setTabOrder(self.pushButton_create_beam, self.pushButton_delete_
↪beam)
        MainWindow.setTabOrder(self.pushButton_delete_beam, self.pushButton_save_beam)
        MainWindow.setTabOrder(self.pushButton_save_beam, self.pushButton_cancel_
↪light)
        MainWindow.setTabOrder(self.pushButton_cancel_light, self.checkBox_offset_
↪light)
```

(continues on next page)

(continued from previous page)

```
        MainWindow.setTabOrder(self.checkBox_offset_light, self.radioButton_gaussian_
↪light)
        MainWindow.setTabOrder(self.radioButton_gaussian_light, self.radioButton_
↪squared_light)
        MainWindow.setTabOrder(self.radioButton_squared_light, self.radioButton_mode)
        MainWindow.setTabOrder(self.radioButton_mode, self.radioButton_all_modes)
        MainWindow.setTabOrder(self.radioButton_all_modes, self.radioButton_airy)
        MainWindow.setTabOrder(self.radioButton_airy, self.calculateButton_light)
        MainWindow.setTabOrder(self.calculateButton_light, self.checkBox_kerr)
        MainWindow.setTabOrder(self.checkBox_kerr, self.spinBox_kerr_loop)
        MainWindow.setTabOrder(self.spinBox_kerr_loop, self.checkBox_n2)
        MainWindow.setTabOrder(self.checkBox_n2, self.doubleSpinBox_n2_significand)
        MainWindow.setTabOrder(self.doubleSpinBox_n2_significand, self.spinBox_n2_
↪exponent)
        MainWindow.setTabOrder(self.spinBox_n2_exponent, self.doubleSpinBox_chi3_
↪significand)
        MainWindow.setTabOrder(self.doubleSpinBox_chi3_significand, self.spinBox_chi3_
↪exponent)
        MainWindow.setTabOrder(self.spinBox_chi3_exponent, self.checkBox_variance)
        MainWindow.setTabOrder(self.checkBox_variance, self.pushButton_estimate_time)
        MainWindow.setTabOrder(self.pushButton_estimate_time, self.pushButton_cancel_
↪compute)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "Beampy"))
        self.label_window_parameters.setText(_translate("MainWindow", "<html><head/>
↪<body><p><span style=\" font-size:14pt; font-weight:600;\">Window information</span>
↪</p></body></html>"))
        self.label_length.setToolTip(_translate("MainWindow", "Size of the computed␣
↪window over z"))
        self.label_length.setText(_translate("MainWindow", "z length"))
        self.doubleSpinBox_length_z.setToolTip(_translate("MainWindow", "Size of the␣
↪computed window over z"))
        self.doubleSpinBox_length_z.setSuffix(_translate("MainWindow", " µm"))
        self.label_dist_z.setToolTip(_translate("MainWindow", "Step over z in µm"))
        self.label_dist_z.setText(_translate("MainWindow", "z step"))
        self.doubleSpinBox_dist_z.setToolTip(_translate("MainWindow", "Step over z in␣
↪µm"))
        self.doubleSpinBox_dist_z.setSuffix(_translate("MainWindow", " µm"))
        self.label_nbr_z_disp.setToolTip(_translate("MainWindow", "Number of points␣
↪to display over z"))
        self.label_nbr_z_disp.setText(_translate("MainWindow", "z nbr displayed"))
        self.spinBox_nbr_z_disp.setToolTip(_translate("MainWindow", "Number of points␣
↪to display over z"))
        self.label_length_x.setToolTip(_translate("MainWindow", "Size of the compute␣
↪window over x in µm"))
        self.label_length_x.setText(_translate("MainWindow", "x width"))
        self.doubleSpinBox_length_x.setToolTip(_translate("MainWindow", "Size of the␣
↪compute window over x in µm"))
        self.doubleSpinBox_length_x.setSuffix(_translate("MainWindow", " µm"))
        self.label_dist_x.setToolTip(_translate("MainWindow", "Step over x in µm"))
        self.label_dist_x.setText(_translate("MainWindow", "x step"))
        self.doubleSpinBox_dist_x.setToolTip(_translate("MainWindow", "Step over x in␣
↪µm"))
        self.doubleSpinBox_dist_x.setSuffix(_translate("MainWindow", " µm"))
        self.label_n.setToolTip(_translate("MainWindow", "Refractive index of the␣
↪background \"cladding\""))
```

(continues on next page)

(continued from previous page)

```
        self.label_n.setText(_translate("MainWindow", "n background"))
        self.doubleSpinBox_n.setToolTip(_translate("MainWindow", "Refractive index of␣
↪the cladding"))
        self.doubleSpinBox_lo.setToolTip(_translate("MainWindow", " Wavelengh in µm"))
        self.doubleSpinBox_lo.setSuffix(_translate("MainWindow", " µm"))
        self.label_lo.setToolTip(_translate("MainWindow", " Wavelengh in µm"))
        self.label_lo.setText(_translate("MainWindow", "Wavelength "))
        self.label_guides_information.setText(_translate("MainWindow", "<html><head/>
↪<body><p><span style=\" font-size:14pt; font-weight:600;\">Guides information</span>
↪</p></body></html>"))
        self.comboBox_guide.setToolTip(_translate("MainWindow", "Select the waveguide
↪"))
        self.pushButton_save_guide.setText(_translate("MainWindow", "save waveguide"))
        self.pushButton_delete_guide.setToolTip(_translate("MainWindow", "Delete the␣
↪current displayed waveguide"))
        self.pushButton_delete_guide.setText(_translate("MainWindow", "delete␣
↪waveguide"))
        self.pushButton_create_guide.setToolTip(_translate("MainWindow", "Add a new␣
↪waveguide with current displayed values"))
        self.pushButton_create_guide.setText(_translate("MainWindow", "add waveguide
↪"))
        self.pushButton_cancel_guide.setToolTip(_translate("MainWindow", "cancel␣
↪guide changes"))
        self.pushButton_cancel_guide.setText(_translate("MainWindow", "Cancel changes
↪"))
        self.label_width.setToolTip(_translate("MainWindow", "Guide width define as␣
↪the diameter in µm at 1/e"))
        self.label_width.setText(_translate("MainWindow", "Width"))
        self.doubleSpinBox_width.setToolTip(_translate("MainWindow", "Guide width␣
↪define as the diameter in µm at 1/e"))
        self.doubleSpinBox_width.setSuffix(_translate("MainWindow", " m"))
        self.label_offset_guide.setToolTip(_translate("MainWindow", "Guide offset in␣
↪the x direction in µm"))
        self.label_offset_guide.setText(_translate("MainWindow", "x offset"))
        self.doubleSpinBox_offset_guide.setToolTip(_translate("MainWindow", "Guide␣
↪offset in the x direction in µm"))
        self.doubleSpinBox_offset_guide.setSuffix(_translate("MainWindow", " µm"))
        self.label_dn.setToolTip(_translate("MainWindow", "Difference of refractive␣
↪index between the core and the cladding"))
        self.label_dn.setText(_translate("MainWindow", "n"))
        self.doubleSpinBox_dn.setToolTip(_translate("MainWindow", "Difference of␣
↪refractive index between the core and the cladding"))
        self.checkBox_n_imag.setToolTip(_translate("MainWindow", "Imaginary part of␣
↪the refractive index. Translate the waveguide losses"))
        self.checkBox_n_imag.setText(_translate("MainWindow", "n_imag"))
        self.doubleSpinBox_n_imag.setToolTip(_translate("MainWindow", "Imaginary part␣
↪of the refractive index. Translate the waveguide losses"))
        self.label_lost.setToolTip(_translate("MainWindow", "Amplitude looses over z␣
↪in 1/mm. alpha=ko*n_imag. Field losses: exp(-alpha * dist_z)"))
        self.label_lost.setText(_translate("MainWindow", "Losses alpha"))
        self.doubleSpinBox_lost.setToolTip(_translate("MainWindow", "Amplitude looses␣
↪over z in 1/mm. alpha=ko*n_imag. Field losses: exp(-alpha * dist_z)"))
        self.doubleSpinBox_lost.setSuffix(_translate("MainWindow", " 1/mm"))
        self.label.setText(_translate("MainWindow", "<span style=\" font-size:12pt;\">
↪Waveguide shape"))
        self.radioButton_gaussian.setToolTip(_translate("MainWindow", "Choose super-
↪gaussian guides"))
```

(continues on next page)

(continued from previous page)

```
        self.radioButton_gaussian.setText(_translate("MainWindow", "Gaussian"))
        self.spinBox_gauss_pow.setToolTip(_translate("MainWindow", "Order of the␣
↪super-Gaussian beam, 1 is gaussian, 4 is the usual super-Gaussian"))
        self.spinBox_gauss_pow.setPrefix(_translate("MainWindow", "order: "))
        self.radioButton_squared.setToolTip(_translate("MainWindow", "Choose squared␣
↪guides"))
        self.radioButton_squared.setText(_translate("MainWindow", "Squared"))
        self.label_nb_p.setToolTip(_translate("MainWindow", "Number of guides"))
        self.label_nb_p.setText(_translate("MainWindow", "Numbers of guides"))
        self.spinBox_nb_p.setToolTip(_translate("MainWindow", "Number of guides"))
        self.label_p.setToolTip(_translate("MainWindow", "Distance between two guides␣
↪center in µm"))
        self.label_p.setText(_translate("MainWindow", "Interval between guides"))
        self.doubleSpinBox_p.setToolTip(_translate("MainWindow", "Distance between␣
↪two guides center in µm"))
        self.doubleSpinBox_p.setSuffix(_translate("MainWindow", " µm"))
        self.doubleSpinBox_offset_guide_z.setToolTip(_translate("MainWindow", "Guide␣
↪offset in the z direction in µm"))
        self.doubleSpinBox_offset_guide_z.setSuffix(_translate("MainWindow", " µm"))
        self.label_offset_guide_z.setToolTip(_translate("MainWindow", "Guide offset␣
↪in the z direction in µm"))
        self.label_offset_guide_z.setText(_translate("MainWindow", "z offset"))
        self.doubleSpinBox_guide_length.setToolTip(_translate("MainWindow", "Length␣
↪of the waveguide"))
        self.doubleSpinBox_guide_length.setSuffix(_translate("MainWindow", " µm"))
        self.label_guide_length.setText(_translate("MainWindow", "length"))
        self.tabWidget_morphology_guide.setTabText(self.tabWidget_morphology_guide.
↪indexOf(self.tab_array), _translate("MainWindow", "waveguide array"))
        self.label_curve.setToolTip(_translate("MainWindow", "curvature factor in 10^-
↪8 1/µm^2"))
        self.label_curve.setText(_translate("MainWindow", "Curvature"))
        self.doubleSpinBox_curve.setToolTip(_translate("MainWindow", "curvature␣
↪factor in 10^-8 1/µm^2"))
        self.doubleSpinBox_curve.setSuffix(_translate("MainWindow", " * 10^-8 1/µm^2
↪"))
        self.label_half_delay.setToolTip(_translate("MainWindow", "Half distance over␣
↪z in µm bewteen the two external guides where they are the closest"))
        self.label_half_delay.setText(_translate("MainWindow", "Half delay"))
        self.doubleSpinBox_half_delay.setToolTip(_translate("MainWindow", "Half␣
↪distance over z in µm bewteen the two external guides where they are the closest"))
        self.doubleSpinBox_half_delay.setSuffix(_translate("MainWindow", " µm"))
        self.label_distance_factor.setToolTip(_translate("MainWindow", "Distance␣
↪factor: p_min/width"))
        self.label_distance_factor.setText(_translate("MainWindow", "p_min/width"))
        self.doubleSpinBox_distance_factor.setToolTip(_translate("MainWindow",
↪"Distance factor: p_min/width"))
        self.doubleSpinBox_distance_factor.setSuffix(_translate("MainWindow", " µm"))
        self.tabWidget_morphology_guide.setTabText(self.tabWidget_morphology_guide.
↪indexOf(self.tab_curved), _translate("MainWindow", "curved waveguides"))
        self.calculateButton_guide.setToolTip(_translate("MainWindow", "Compute an␣
↪array of guides"))
        self.calculateButton_guide.setText(_translate("MainWindow", "Compute guides"))
        self.tabWidget_main.setTabText(self.tabWidget_main.indexOf(self.tabWidget_
↪guide), _translate("MainWindow", "Guides"))
        self.label_light_informations.setText(_translate("MainWindow", "<html><head/>
↪<body><p><span style=\" font-size:14pt; font-weight:600;\">Light information</span>
↪</p></body></html>"))
```

(continues on next page)

(continued from previous page)

```
        self.label_theta_ext.setToolTip(_translate("MainWindow", "Number of the last␣
↪mode that can propagate in the guide"))
        self.label_theta_ext.setText(_translate("MainWindow", " ext"))
        self.doubleSpinBox_theta_ext.setToolTip(_translate("MainWindow", "Exterior␣
↪inclinaison angle in ° define from a interface between air and the cladding"))
        self.doubleSpinBox_theta_ext.setSuffix(_translate("MainWindow", " °"))
        self.label_offset_light.setToolTip(_translate("MainWindow", "Light offset␣
↪from center"))
        self.label_offset_light.setText(_translate("MainWindow", "Offset (µm)"))
        self.label_intensity.setToolTip(_translate("MainWindow", "Beam power"))
        self.label_intensity.setText(_translate("MainWindow", "Beam power"))
        self.radioButton_mode.setToolTip(_translate("MainWindow", "select the n mode␣
↪based on a squared guide"))
        self.radioButton_mode.setText(_translate("MainWindow", "Mode"))
        self.radioButton_gaussian_light.setToolTip(_translate("MainWindow", "Select a␣
↪gaussian beam"))
        self.radioButton_gaussian_light.setText(_translate("MainWindow", "Gaussian"))
        self.radioButton_squared_light.setToolTip(_translate("MainWindow", "Select a␣
↪squared beam"))
        self.radioButton_squared_light.setText(_translate("MainWindow", "Squared"))
        self.label_guide_nbr_ref.setToolTip(_translate("MainWindow", "Waveguide␣
↪number to use its width in the mode calculation"))
        self.label_guide_nbr_ref.setText(_translate("MainWindow", "Guide number ref"))
        self.label_fwhm.setToolTip(_translate("MainWindow", "Full width at half␣
↪maximum (for intensity not amplitude)"))
        self.label_fwhm.setText(_translate("MainWindow", "FWHM"))
        self.doubleSpinBox_fwhm.setToolTip(_translate("MainWindow", "Full width at␣
↪half maximum (for intensity not amplitude)"))
        self.doubleSpinBox_fwhm.setSuffix(_translate("MainWindow", " m"))
        self.comboBox_light.setToolTip(_translate("MainWindow", "Select the beam"))
        self.checkBox_offset_light.setToolTip(_translate("MainWindow", "Choose if␣
↪offset is define by the guides positions or by the distance from center"))
        self.checkBox_offset_light.setText(_translate("MainWindow", "Offset (number)
↪"))
        self.label_lobe_size.setText(_translate("MainWindow", "First lobe size"))
        self.radioButton_airy.setToolTip(_translate("MainWindow", "Select a Airy beam
↪"))
        self.radioButton_airy.setText(_translate("MainWindow", "Airy"))
        self.label_zero_cut.setToolTip(_translate("MainWindow", "Choose to cut at the␣
↪n zero of the Airy function"))
        self.label_zero_cut.setText(_translate("MainWindow", "Zero cut"))
        self.mode_number.setToolTip(_translate("MainWindow", "Number of the last mode␣
↪that can propagate in the guide"))
        self.pushButton_mode_number.setToolTip(_translate("MainWindow", "Displayed␣
↪the number of the last mode that can propagate in the rectangular guide"))
        self.pushButton_mode_number.setText(_translate("MainWindow", "Show max mode"))
        self.radioButton_all_modes.setToolTip(_translate("MainWindow", "Select all␣
↪possible modes for a squared guide"))
        self.radioButton_all_modes.setText(_translate("MainWindow", "All modes"))
        self.spinBox_offset_light_peak.setToolTip(_translate("MainWindow", "Offset␣
↪define by the guide position"))
        self.spinBox_offset_light_peak.setPrefix(_translate("MainWindow", "Guide␣
↪number : "))
        self.doubleSpinBox_offset_light.setToolTip(_translate("MainWindow", "Light␣
↪offset from center in µm"))
        self.doubleSpinBox_offset_light.setSuffix(_translate("MainWindow", " µm"))
        self.label_3.setText(_translate("MainWindow", "W/m^2"))
```

(continues on next page)

(continued from previous page)

```
        self.label_2.setText(_translate("MainWindow", "x10^"))
        self.spinBox_mode.setToolTip(_translate("MainWindow", "The n mode based on a␣
↪squared guide"))
        self.spinBox_guide_nbr_ref.setToolTip(_translate("MainWindow", "Waveguide␣
↪number to use its width in the mode calculation"))
        self.spinBox_airy_zero.setToolTip(_translate("MainWindow", "Choose to cut at␣
↪the n zero of the Airy function"))
        self.doubleSpinBox_irrad_significand.setToolTip(_translate("MainWindow",
↪"Beam power |E|^2 in GW/cm^2"))
        self.doubleSpinBox_lobe_size.setSuffix(_translate("MainWindow", " µm"))
        self.pushButton_delete_beam.setToolTip(_translate("MainWindow", "Delete the␣
↪current displayed beam"))
        self.pushButton_delete_beam.setText(_translate("MainWindow", "delete beam"))
        self.pushButton_create_beam.setToolTip(_translate("MainWindow", "Add a new␣
↪beam with current displayed values"))
        self.pushButton_create_beam.setText(_translate("MainWindow", "add beam"))
        self.pushButton_save_beam.setToolTip(_translate("MainWindow", "Save current␣
↪displayed values"))
        self.pushButton_save_beam.setText(_translate("MainWindow", "save beam"))
        self.pushButton_cancel_light.setToolTip(_translate("MainWindow", "Return to␣
↪previously saved values"))
        self.pushButton_cancel_light.setText(_translate("MainWindow", "cancel changes
↪"))
        self.calculateButton_light.setToolTip(_translate("MainWindow", "Compute the␣
↪beams"))
        self.calculateButton_light.setText(_translate("MainWindow", "Compute light"))
        self.tabWidget_main.setTabText(self.tabWidget_main.indexOf(self.tabWidget_
↪light), _translate("MainWindow", "Light"))
        self.label_kerr_informations.setText(_translate("MainWindow", "<html><head/>
↪<body><p><span style=\" font-size:14pt; font-weight:600;\">Kerr information</span></
↪p></body></html>"))
        self.checkBox_kerr.setToolTip(_translate("MainWindow", "Activate the non-
↪linear effect: Kerr effect"))
        self.checkBox_kerr.setText(_translate("MainWindow", "Kerr effect"))
        self.doubleSpinBox_n2_significand.setToolTip(_translate("MainWindow",
↪"Nonlinear refractive index define as n2=3*chi3/(4*n_0^2*_0*c))"))
        self.spinBox_kerr_loop.setToolTip(_translate("MainWindow", "Number of␣
↪corrective loop for the Kerr effect"))
        self.spinBox_kerr_loop.setSuffix(_translate("MainWindow", " loops"))
        self.label_chi3.setToolTip(_translate("MainWindow", "Induce a change in␣
↪refractive index as delta_n_NL = 3*chi3/(8*n_0)*|E^2|"))
        self.label_chi3.setText(_translate("MainWindow", "Chi3"))
        self.doubleSpinBox_chi3_significand.setToolTip(_translate("MainWindow", "The␣
↪X(3) Electric susceptibility of the material"))
        self.label_correction_loop.setText(_translate("MainWindow", "Correction"))
        self.checkBox_n2.setToolTip(_translate("MainWindow", "Indice a change in␣
↪refractive index proportional to the beam intensity as delta_n_NL=n_2*I"))
        self.checkBox_n2.setText(_translate("MainWindow", "n2"))
        self.label_n2_unit.setText(_translate("MainWindow", "m^2/W"))
        self.label_n2_exponent.setText(_translate("MainWindow", "x10^"))
        self.label_chi3_unit.setText(_translate("MainWindow", "m^2/V^2"))
        self.label_chi3_exponent.setText(_translate("MainWindow", "x10^"))
        self.checkBox_variance.setToolTip(_translate("MainWindow", "Use this control␣
↪to make sure the intensities converge"))
        self.checkBox_variance.setText(_translate("MainWindow", "Check if intensity␣
↪converge"))
        self.estimate_time_display.setToolTip(_translate("MainWindow", "Displayed the␣
↪estimation time needed to do the propagation"))
```

(continues on next page)

(continued from previous page)

```
        self.estimate_time_display.setStatusTip(_translate("MainWindow", "Show the␣
↪estimate computation time"))
        self.pushButton_estimate_time.setToolTip(_translate("MainWindow", "When␣
↪pressed, displayed the estimation time needed to do the propagation"))
        self.pushButton_estimate_time.setText(_translate("MainWindow", "Show time␣
↪estimation"))
        self.calculateButton_compute.setToolTip(_translate("MainWindow", "Compute␣
↪beam propagation"))
        self.calculateButton_compute.setText(_translate("MainWindow", "Compute␣
↪propagation"))
        self.progressBar_compute.setToolTip(_translate("MainWindow", "Display the␣
↪progression calculation"))
        self.checkBox_power.setToolTip(_translate("MainWindow", "Display the power in␣
↪each guide as power=integral(I(x))"))
        self.checkBox_power.setText(_translate("MainWindow", "Display power"))
        self.pushButton_power.setText(_translate("MainWindow", "Display power"))
        self.pushButton_cancel_compute.setText(_translate("MainWindow", "Cancel␣
↪changes"))
        self.tabWidget_main.setTabText(self.tabWidget_main.indexOf(self.tabWidget_
↪compute), _translate("MainWindow", "Compute"))


if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

5.4.4 beampy.examples

```
from math import pi, ceil, radians, sqrt, log, sin, cos, acos, asin, exp

import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
import numpy as np
from numpy.fft import fft, ifft, fftshift

from beampy.bpm import Bpm, abs2


def gaussian_beam(fwhm=10):
    """Display a Gaussian beam at the given fwhm.

    Parameters
    ----------
    fwhm : float
        Full width at half maximum (for intensity not amplitude) (µm).
    """
    w = fwhm / np.sqrt(2 * np.log(2))
    bpm = Bpm(1, 1, 1, 1, 1, 1, 1)
    bpm.x = np.linspace(-1.5*fwhm, 1.5*fwhm, 500)
    x = bpm.x
```

(continues on next page)

(continued from previous page)

```
    plt.figure("Beam")
    plt.title("Width definition of a gaussian beam with FWHM=%s µm" % fwhm)
    plt.xlabel("x (µm)")
    plt.plot(x, bpm.gauss_light(fwhm, 0), label='field')
    plt.plot(x, (bpm.gauss_light(fwhm, 0))**2, label='intensity')
    plt.plot(x, [1/2]*x.size, '-.', label='1/2')
    plt.plot([fwhm/2]*x.size, np.linspace(0, 1, x.size),
             '--', label='fwhm/2')
    plt.plot(x, [np.exp(-1)]*x.size, '-.', label='1/e')
    plt.plot(x, [np.exp(-2)]*x.size, '-.', label='1/$e^2$')
    plt.plot([w]*x.size, np.linspace(0, 1, x.size),
             '--', label='$w_0$=%.2f' % w)
    plt.legend()
    plt.show()
#    plt.savefig("def_gauss_beam.png", bbox="tight", dpi=250)


def guides_x():
    """Display a Gaussian guide, two super-Gaussian guides and a flat-top guide
    to illustrate the width definition."""
    width = 10
    bpm = Bpm(1, 1, 1, 1, 1, 1, 1)
    bpm.x = np.linspace(-15, 9.1, 500)
    x = bpm.x
    plt.figure("guide_x")
    plt.title("Different waveguides shape available")
    plt.plot(x, bpm.gauss_guide(width, 1)(x), label='Gaussian')
    plt.plot(x, bpm.gauss_guide(width, 4)(x), label='super-Gaussian P=4')
    plt.plot(x, bpm.gauss_guide(width, 10)(x), label='super-Gaussian P=10')
    plt.plot(x, bpm.squared_guide(width)(x), label='Flat-top')
    plt.plot([width/2]*x.size, np.linspace(0, 1, x.size), '--',
             label='width/2')
    plt.plot(x, [np.exp(-1)]*x.size, '-.', label='1/e')
    plt.legend()
    plt.show()
#    plt.savefig("def_gauss_guide.png", bbox="tight", dpi=250)


def guides_z():
    """Display an array of guides and the curved guides system."""
    width = 6
    bpm = Bpm(2, 1.55, 10000, 1, 200, 100, 0.1)
    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()
    shape = bpm.gauss_guide(width, 4)
    [peaks, dn] = bpm.create_guides(shape, 0.01, 5, 10)
    z_disp = np.linspace(0, length_z/1000, nbr_z_disp+1)
    xv, zv = np.meshgrid(x, z_disp)
    dn_disp = np.linspace(0, nbr_z-1, nbr_z_disp+1, dtype=int)
    plt.figure("Waveguide array")
    plt.title("Waveguides array example")
    plt.xlabel("x (µm)")
    plt.ylabel("z (mm)")
    plt.pcolormesh(xv, zv, dn[dn_disp], cmap='gray')
    plt.show()
#    plt.savefig("waveguide_array.png", bbox="tight", dpi=250)

    [peaks, dn] = bpm.create_curved_guides(
```

(continues on next page)

(continued from previous page)

```
            shape, width, 0.01, 40*1e-8, 1000, 1.2)
    plt.figure("Curved wavewaveguides")
    plt.title("Curved waveguides example")
    plt.xlabel("x (µm)")
    plt.ylabel("z (mm)")
    plt.pcolormesh(xv, zv, dn[dn_disp], cmap='gray')
    plt.show()
#    plt.savefig("waveguide_curved.png", bbox="tight", dpi=250)

    shape = bpm.gauss_guide(width, 4)
    dn = bpm.create_guides(shape, 0.01, 1, 10, z=[0, length_z/2])[1]
    dn2 = bpm.create_guides(shape, 0.01, 2, 10,
                            z=[length_z/2, length_z])[1]
    dn3 = bpm.create_guides(shape, 0.01, 2, 30,
                            z=[length_z/4, 3*length_z/4])[1]
    dn4 = bpm.create_guides(shape, 0.01, 1, 0, offset_guide=40)[1]
    dn = np.add(dn, dn2)
    dn = np.add(dn, dn3)
    dn = np.add(dn, dn4)
    z_disp = np.linspace(0, length_z/1000, nbr_z_disp+1)
    xv, zv = np.meshgrid(x, z_disp)
    dn_disp = np.linspace(0, nbr_z-1, nbr_z_disp+1, dtype=int)
    plt.figure("Arbitrary waveguides example")
    plt.title("Arbitrary waveguides example with discontinuity")
    plt.xlabel("x (µm)")
    plt.ylabel("z (mm)")
    plt.pcolormesh(xv, zv, dn[dn_disp], cmap='gray')
    plt.show()
#    plt.savefig("waveguide_arbitrary.png", bbox="tight", dpi=250)


def free_propag(dist_x=0.1, length_x=1000, length_z=10000, no=1, lo=1):
    """Show the free propagation of a beam and compare Beampy results with
    the theorical values.

    Parameters
    ----------
    dist_x : float
        Step over x (µm).
    length_x : float
        Size of the compute window over x (µm).
    length_z : float
        Size of the compute window over z (µm).
    no : float
        Refractive index of the cladding
    lo : float
        Wavelength of the beam in vaccum (µm).
    """

    dist_z = length_z
    nbr_z_disp = 1

    bpm = Bpm(no, lo,
              length_z, dist_z, nbr_z_disp,
              length_x, dist_x)

    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()
```

(continues on next page)

(continued from previous page)

```
    dn = np.zeros((nbr_z, nbr_x))

    fwhm = 20

    field = bpm.gauss_light(fwhm)
    irrad = 1E13
    theta_ext = 0

    [progress_pow] = bpm.init_field(field, theta_ext, irrad)

    [progress_pow] = bpm.main_compute(dn)

    intensity = progress_pow[-1]

    index = np.where(intensity >= (np.max(intensity)/2))[0][0]
    fwhm_final = abs(2 * x[index])

    w0_final = fwhm_final / np.sqrt(2 * np.log(2))
    print("Beampy: width w = %f µm" % w0_final)

    w0 = fwhm / np.sqrt(2 * np.log(2))
    z0 = np.pi * no * w0**2 / lo
    w = w0 * np.sqrt(1 + (length_z / z0)**2)
    print("Theory: width w = %f µm" % w)

    diff = abs(w - w0_final)/w*100
    print("Relative difference: %f" % diff, "%")

    if diff > 1:
        print("Check the dist_x or length_x parameters")

    irrad_theo = irrad*w0/w  # in 2D: irrad*(w0/w)**2

    print("Beampy: irradiance I =", np.max(intensity))
    print("Theory: irradiance I =", irrad_theo)

    diff = abs(np.max(intensity) - irrad_theo)/irrad_theo*100
    print("Relative difference: %f" % diff, "%")

    if diff > 1:
        print("Check the dist_x or length_x parameters")

    fwhm_theo = w * np.sqrt(2 * np.log(2))

    profil_theo = bpm.gauss_light(fwhm_theo)
    intensity_theo = irrad_theo * abs2(profil_theo)

    plt.figure()
    plt.title("Beam profil after %.0f mm of free propagation" % (length_z/1e3))
    plt.xlabel("x (µm)")
    plt.ylabel(r"irradiance (W.m$^{-2}$)")
    plt.plot(x, intensity_theo, "-", label="Theory")
    plt.plot(x, intensity, "--", label="Beampy")
    plt.legend()
    plt.show()
#    plt.savefig("free_propagation.png", bbox="tight", dpi=250)
```

(continues on next page)

(continued from previous page)

```
def multimodal_splitter():
    """Multimodal splitter 1x2. A single mode beam is split into two single
    mode waveguide by the use of an intermediate multimodal waveguide."""
    width1 = 3
    length1 = 100
    p1 = 4*width1

    width2 = 100
    length2 = 5110
    p2 = 4*width2

    width3 = width1
    length3 = 4090
    p3 = 2*26

    p = [p1, p2, p3]
    nbr_p = [1, 1, 2]

    no = 1.453
    wavelength = 1.55
    delta_no = 0.01
    length_z = length1+length2+length3
    dist_z = 1
    nbr_z_disp = 200
    dist_x = 0.1
    length_x = 1300

    bpm = Bpm(no, wavelength,
              length_z, dist_z, nbr_z_disp,
              length_x, dist_x)

    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()

    shape1 = bpm.gauss_guide(width1, 4)
    [peaks, dn] = bpm.create_guides(shape1, delta_no, 1, p1, z=[0, length1])

    shape2 = bpm.squared_guide(width2)
    [peaks2, dn2] = bpm.create_guides(
            shape2, delta_no, 1, p2, z=[length1, length1+length2])

    shape3 = bpm.gauss_guide(width3, 4)
    [peaks3, dn3] = bpm.create_guides(
            shape3, delta_no, 2, p3, z=[length1+length2,
                                        length1+length2+length3])
    dn = np.add(dn, dn2)
    dn = np.add(dn, dn3)
    peaks = np.append(peaks, peaks2, 0)
    peaks = np.append(peaks, peaks3, 0)

    z_disp = np.linspace(0, length_z/1000, nbr_z_disp+1)
    xv, zv = np.meshgrid(x, z_disp)
    dn_disp = np.linspace(0, nbr_z-1, nbr_z_disp+1, dtype=int)
    plt.figure("Multimodal beam splitter 1x2")
    plt.title("Multimodal beam splitter 1x2")
    plt.xlabel("x (µm)")
```

(continues on next page)

(continued from previous page)

```
    plt.ylabel("z (mm)")
    plt.pcolormesh(xv, zv, dn[dn_disp], cmap='gray')
    plt.show()
#    plt.savefig("splitter_shape.png", bbox="tight", dpi=250)

    assert bpm.check_modes(width1, delta_no) == 0
    field = bpm.all_modes(width1, delta_no)[0]

    [progress_pow] = bpm.init_field(field, 0, 1)

    [progress_pow] = bpm.main_compute(dn)

    plt.figure("beam profil at the end")
    ax1 = plt.subplot(111)
    plt.title("Beam profil at z=%.0f µm" % length_z)
    plt.xlabel("x (µm)")
    plt.xlim(-p3, p3)
    ax1.set_ylabel(r'$\Delta_n$')
    ax2 = ax1.twinx()
    ax2.set_ylabel(r"I (a.u)")
    ax1.plot(x, dn[-1])
    verts = [(x[0], 0),
             *zip(x, dn[-2, :].real),
             (x[-1], 0)]
    poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
    ax1.set_ylim(0, max(dn[0, :].real)*1.1 + 1E-20)
    ax1.add_patch(poly)
    ax2.plot(x, progress_pow[-1]/np.max(progress_pow[0]))
    ax2.set_ylim(0,  1.1)
    plt.show()
#    plt.savefig("splitter_beam_end.png", bbox="tight", dpi=250)

    plt.figure("Propagation in the multimodal beam splitter 1x2")
    plt.title("Beam propagation in the beam splitter 1x2")
    plt.xlabel("x (µm)")
    plt.ylabel("z (mm)")
    plt.pcolormesh(xv, zv, progress_pow)
    plt.show()
#    plt.savefig("splitter_propagation.png", bbox="tight", dpi=250)

    plt.figure("Power in waveguides")
    ax1 = plt.subplot(111)
    ax1.set_title("Power in waveguides")
    ax1.set_ylim(-0.05, 1.05)
    ax1.set_xlabel('z (mm)')
    ax1.set_ylabel('Power (a.u)')

    x_beg = np.array([[None]*nbr_z]*peaks.shape[0])
    x_end = np.array([[None]*nbr_z]*peaks.shape[0])
    P = np.zeros((peaks.shape[0], nbr_z_disp+1))

    num_gd = 0
    for i, n in enumerate(nbr_p):
        for _ in range(n):
            [x_beg[num_gd, :],
             x_end[num_gd, :]] = bpm.guide_position(peaks, num_gd, p[i])
            num_gd += 1
```

(continues on next page)

(continued from previous page)

```
        if n == 0:  # needed if no waveguide
            num_gd += 1

    for i in range(peaks.shape[0]):
        P[i, :] = bpm.power_guide(x_beg[i, :], x_end[i, :])
        ax1.plot(z_disp, P[i, :], label='P'+str(i))
    plt.legend()
    plt.show()
#    plt.savefig("splitter_power.png", bbox="tight", dpi=250)
    print("Input power = %.2f \nOutput powers = %.2f and %.2f" % (P[0, 0],
                                                                  P[2, -1],
                                                                  P[3, -1]))
    print("Losses = %.2f" % (P[0, 0]-P[2, -1]-P[3, -1]))


def benchmark_kerr():
    """Kerr benchmark by looking at the critical power at which a beam become
    a soliton. Several test were done with this function and for now, it seems
    that the critical power find by the simulations is about 85% of the
    theorical value. Meaning a 15% error. Further tests are needed to
    understand the observed differences."""
    width = 0
    no = 1
    wavelength = 4
    delta_no = 0.000
    length_z = 1000
    dist_z = 0.5
    nbr_z_disp = 200
    dist_x = 0.05
    length_x = 500
    nbr_p = 1
    p = 20
    fwhm = 12
    theta_ext = 0
    n2 = 2.44e-20
    alpha = 1.8962  # gaussian beam

    # See Self-focusing on wiki
    P_c = alpha*(wavelength*1e-6)**2/(4*pi*no*n2)
    print("critical power = %.2e W" % P_c)

#    irrad = 5e3*1e13
#    power = irrad * (fwhm*1e-6)**2 * pi / (4*np.log(2))
#    print("power", power)

    # See Gaussian_beam on wiki on beam power
    w = fwhm*1e-6 / sqrt(2*log(2))
    irrad_cri = 2*P_c/(pi*w**2)
    print("Critical irradiance %.2e W/m^2" % irrad_cri)

    bpm = Bpm(no, wavelength,
              length_z, dist_z, nbr_z_disp,
              length_x, dist_x)

    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()

    shape = bpm.gauss_guide(width, 4)
```

(continues on next page)

(continued from previous page)

```
    sweep = (1,
             0.2*irrad_cri, 0.5*irrad_cri, 0.7*irrad_cri, 0.75*irrad_cri,
             0.8*irrad_cri, 0.85*irrad_cri,  # 0.85 seem correct
             0.9*irrad_cri, irrad_cri
             )

    for i, irrad in enumerate(sweep):
        power = irrad/2 * (pi*w**2)
        print(i+1, "/", len(sweep), "\tpower= %.2e W" % power, sep="")

        [peaks, dn] = bpm.create_guides(shape, delta_no, nbr_p, p)

        field = bpm.gauss_light(fwhm)

        [progress_pow] = bpm.init_field(field, theta_ext, irrad)

        [progress_pow] = bpm.main_compute(dn, n2=n2, kerr_loop=2,
                                          disp_progress=False)

        x_pos = int(length_x/2/dist_x)
        z_disp = np.linspace(0, length_z/1000, nbr_z_disp+1)

        plt.figure("Benchmark kerr")
        plt.title(r"Central irradition for a theorical $P_c$ = %.2e W" % P_c)
        plt.xlabel("z (µm)")
        plt.ylabel(r"I (a.u)")
#        plt.ylim(-0.05, 2)
        plt.plot(z_disp*1e3,
                 progress_pow[:, x_pos]/progress_pow[0, x_pos],
                 label="Power=%.2e" % power)
        plt.legend()
        plt.show()
#        plt.savefig("kerr_irrad.png", bbox="tight", dpi=250)

#        x_beg = np.array([None]*nbr_z)
#        x_end = np.array([None]*nbr_z)
#        P = np.zeros(nbr_z_disp+1)
#        x_beg, x_end = bpm.guide_position(peaks, 0, p)
#        P = bpm.power_guide(x_beg, x_end)

#        plt.figure("Benchmark kerr2")
#        plt.title("Power for a theorical critical power = %.2e" % P_c)
#        plt.xlabel("z (µm)")
#        plt.ylabel(r"Power (a.u)")
#        plt.ylim(-0.05, 2)
#        plt.plot(z_disp, P, label="Power=%.2e" % power)
#        plt.legend()
#        plt.show()

#        xv, zv = np.meshgrid(x, z_disp)
#        plt.figure("check borders")
#        plt.title("check if beam don't reach the borders")
#        plt.pcolormesh(xv, zv, progress_pow, cmap='gray')
#        plt.show()

        plt.figure("Benchmark beam profil")
```

(continues on next page)

(continued from previous page)

```
        plt.title("Beam profil at z=%.0f µm" % length_z)
        plt.xlabel("x (µm)")
        plt.ylabel(r"I/$I_0$")
        plt.xlim(-30, 30)
        plt.plot(x, progress_pow[-1]/np.max(progress_pow[0]),
                 label="Power=%.2e" % power)
        plt.legend()
        plt.show()
#        plt.savefig("kerr_power.png", bbox="tight", dpi=250)


def stability():
    """Show the possible BPM approximations for implementing a refractive
    index variation"""

    width = 6
    no = 2.14
    wavelength = 1.55
    delta_no = 0.0014
    length_z = 200
    dist_z = 1
    nbr_z_disp = 1
    dist_x = 0.2
    length_x = 1000

    bpm = Bpm(no, wavelength,
              length_z, dist_z, nbr_z_disp,
              length_x, dist_x)

    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()

    shape = bpm.squared_guide(width)

    nbr_p = 1
    p = 0

    [peaks, dn] = bpm.create_guides(shape, delta_no, nbr_p, p)

    fwhm = 6
    lo = 1.5

    field = bpm.gauss_light(fwhm)
    irrad = 1
    theta_ext = 0

    [progress_pow] = bpm.init_field(field, theta_ext, irrad)

    nbr_step = nbr_z-1

    # Need to overwrite those variables due to changes
    theta = asin(sin(radians(theta_ext)) / no)  # angle in the guide
    nu_max = 1 / (2 * dist_x)  # max frequency due to sampling
    # Spacial frequencies over x (1/µm)
    nu = np.linspace(-nu_max,
                     nu_max * (1 - 2/nbr_x),
                     nbr_x)
    intermed = no * cos(theta) / lo
```

(continues on next page)

(continued from previous page)

```
    fr = -2 * pi * nu**2 / (intermed + np.sqrt(
        intermed**2
        - nu**2
        + 0j))

    bpm.phase_mat = fftshift(np.exp(1j * dist_z * fr))
    bpm.phase_mat_demi = fftshift(np.exp(1j * dist_z / 2 * fr))
    bpm.nl_mat = bpm.ko * bpm.dist_z * dn
    # End overwrite

    field = bpm.field
    for i in range(nbr_step):
        field = ifft(bpm.phase_mat * fft(field))
        field *= np.exp(1j * bpm.nl_mat[nbr_step, :])
    test1 = field

    field = bpm.field
    for i in range(nbr_step):
        field = ifft(bpm.phase_mat_demi * fft(field))
        field *= np.exp(1j * bpm.nl_mat[nbr_step, :])
        field = ifft(bpm.phase_mat_demi * fft(field))
    test2 = field

    field = bpm.field
    for i in range(nbr_step):
        field *= np.exp(1j * bpm.nl_mat[nbr_step, :])
        field = ifft(bpm.phase_mat * fft(field))
    test3 = field

    plt.figure("Power")
    plt.title("Power: Impact of the chosen algorithm on the free propagation")
    plt.xlim(-20, 20)
#    plt.ylim(-1, 350)

    plt.plot(x, abs2(test1), label='dz+lens')
    plt.plot(x, abs2(test2), label='dz/2+lens+dz/2')
    plt.plot(x, abs2(test3), label='lens+dz')
    plt.legend()
    plt.show()
    plt.figure("Phase")
    plt.title("Phase: Impact of the chosen algorithm on the free propagation")
    plt.xlim(-30, 30)
    plt.plot(x, np.angle(test1), label='dz+lens')
    plt.plot(x, np.angle(test2), label='dz/2+lens+dz/2')
    plt.plot(x, np.angle(test3), label='lens+dz')
    plt.legend()
    plt.show()

    # Old results showing possibilities to reduce the computation times
#    field = bpm.field
#    field = ifft(bpm.phase_mat_demi * fft(field))
#    field *= np.exp(1j * bpm.nl_mat[0, :])
#    field = ifft(bpm.phase_mat * fft(field))
#    field *= np.exp(1j * bpm.nl_mat[0, :])
#    field = ifft(bpm.phase_mat_demi * fft(field))
#    test4 = field
#
```

(continues on next page)

(continued from previous page)

```
#     field = bpm.field
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     field *= np.exp(1j * bpm.nl_mat[0, :])
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     field *= np.exp(1j * bpm.nl_mat[0, :])
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     test5 = field

#     plt.figure("Power 2")
#     plt.title("Power: Same algorithme optimized")
#     plt.xlim(-20, 20)
#     plt.plot(x, abs2(test4), label='dz/2+lens+dz+lens+dz/2')
#     plt.plot(x, abs2(test5), label='(dz/2+lens+dz/2)*2')
#
#     plt.legend()
#     plt.show()
#     plt.figure("Phase 2")
#     plt.title("Phase: Same algorithme optimized")
#     plt.xlim(-30, 30)
#     plt.plot(x, np.angle(test4), label='dz/2+lens+dz+lens+dz/2')
#     plt.plot(x, np.angle(test5), label='(dz/2+lens+dz/2)*2')
#     plt.legend()
#     plt.show()
#
#     field = bpm.field
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     field *= np.exp(1j * bpm.nl_mat[0, :])
#     field = ifft(bpm.phase_mat * fft(field))
#     test6 = field
#
#     field = bpm.field
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     field *= np.exp(1j * bpm.nl_mat[0, :])
#     field = ifft(bpm.phase_mat_demi * fft(field))
#     test7 = field
#
#     plt.figure("Power 3")
#     plt.title("Power: Approximation if uses loop over lens+dz")
#     plt.xlim(-20, 20)
#     plt.plot(x, abs2(test6), label='dz/2+lens+dz')
#     plt.plot(x, abs2(test7), label='dz/2+lens+dz/2')
#
#     plt.legend()
#     plt.show()
#     plt.figure("Phase 3")
#     plt.title("Phase: Approximation if uses loop over lens+dz")
#     plt.xlim(-30, 30)
#     plt.plot(x, np.angle(test6), label='dz/2+lens+dz')
#     plt.plot(x, np.angle(test7), label='dz/2+lens+dz/2')
#     plt.legend()
#     plt.show()


def test_kerr():
    """More test than example. Show different approximations for the BPM
    implementation of the Kerr effect."""
```

(continues on next page)

(continued from previous page)

```
no = 1
lo = 1.5
length_z = 50
dist_z = 0.1
nbr_z_disp = 1
dist_x = 0.01
length_x = 400

bpm = Bpm(no, lo,
          length_z, dist_z, nbr_z_disp,
          length_x, dist_x)

[length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()

dn = np.zeros((nbr_z, nbr_x))

fwhm = 6

field = bpm.gauss_light(fwhm)
irrad = 260e13  # if too high, see big difference between method
theta_ext = 0

[progress_pow] = bpm.init_field(field, theta_ext, irrad)

# Need to overwrite those variables due to changes
theta = asin(sin(radians(theta_ext)) / no)  # angle in the guide
nu_max = 1 / (2 * dist_x)  # max frequency due to sampling
# Spacial frequencies over x (1/µm)
nu = np.linspace(-nu_max,
                 nu_max*(1 - 2/nbr_x),
                 nbr_x)
intermed = no * cos(theta) / lo
fr = -2 * pi * nu**2 / (intermed + np.sqrt(
    intermed**2
    - nu**2
    + 0j))

bpm.phase_mat = fftshift(np.exp(1j * dist_z * fr))
bpm.phase_mat_demi = fftshift(np.exp(1j * dist_z / 2 * fr))
bpm.nl_mat = bpm.ko * bpm.dist_z * dn
# End overwrite

kerr_loop = 10
chi3 = 10 * 1E-20

nbr_step = nbr_z-1

print("\n dz/2+lens+dz/2")
field = bpm.field
for i in range(nbr_step):
    # Linear propagation over dz/2
    field = ifft(bpm.phase_mat_demi * fft(field))

    # Influence of the index modulation on the field
    field = field * np.exp(1j * bpm.nl_mat[nbr_step, :])  # No changes if
    # no initial guide (exp=1)
```

(continues on next page)

(continued from previous page)

```
        # Linear propagation over dz/2
        field = ifft(bpm.phase_mat_demi * fft(field))

        cur_pow = bpm.epnc * abs2(field)

    field_ref = field
    cur_ref = cur_pow

    print("\n dz+kerr")
    field = bpm.field
    for i in range(nbr_step):
        # Linear propagation over dz
        field = ifft(bpm.phase_mat * fft(field))

        # Influence of the index modulation on the field
        field_x = field * np.exp(1j * bpm.nl_mat[nbr_step, :])

        cur_pow = bpm.epnc * abs2(field_x)

        for k in range(1):  # insensitive to correction loop
            prev_pow = cur_pow
            # influence of the beam intensity on the index modulation
            dn_temp = dn[nbr_step, :] + (3*chi3)/(8*no)*(prev_pow/bpm.epnc)
            nl_mat = bpm.ko * bpm.dist_z * dn_temp
            # influence of the index modulation on the field
            field_x = field * np.exp(1j*nl_mat)  # No changes for the pow

            cur_pow = bpm.epnc * abs2(field_x)

        try:
            bpm.variance(prev_pow, cur_pow)
        except ValueError as error:
            print(error)

        field = field_x

    field_1 = field
    cur_1 = cur_pow
    dn_1 = dn_temp

    print("\n dz/2+kerr+dz/2")
    field = bpm.field
    for i in range(nbr_step):
        # Linear propagation over dz/2
        field = ifft(bpm.phase_mat_demi * fft(field))

        # Influence of the index modulation on the field
        field_x = field * np.exp(1j * bpm.nl_mat[nbr_step, :])

        # Linear propagation over dz/2
        field_x = ifft(bpm.phase_mat_demi * fft(field_x))

        cur_pow = bpm.epnc * abs2(field_x)

        for k in range(kerr_loop):
            prev_pow = cur_pow
            # influence of the beam intensity on the index modulation
```

(continues on next page)

(continued from previous page)

```
            dn_temp = dn[nbr_step, :] + (3*chi3)/(8*no)*(prev_pow/bpm.epnc)
            nl_mat = bpm.ko * bpm.dist_z * dn_temp

            # influence of the index modulation on the field
            field_x = field * np.exp(1j * nl_mat)
            # Linear propagation over dz/2
            field_x = ifft(bpm.phase_mat_demi * fft(field_x))

            cur_pow = bpm.epnc * abs2(field_x)

        try:
            bpm.variance(prev_pow, cur_pow)
        except ValueError as error:
            print(error)

        field = field_x

    field_2 = field
    cur_2 = cur_pow
    dn_2 = dn_temp

    print("\n kerr+dz")
    field = bpm.field
    for i in range(nbr_step):
        # Influence of the index modulation on the field
        field_x = field * np.exp(1j * bpm.nl_mat[nbr_step, :])

        # Linear propagation over dz
        field_x = ifft(bpm.phase_mat * fft(field_x))

        cur_pow = bpm.epnc * abs2(field_x)

        for k in range(kerr_loop):
            prev_pow = cur_pow
            # influence of the beam intensity on the index modulation
            dn_temp = dn[nbr_step, :] + (3*chi3)/(8*no)*(prev_pow/bpm.epnc)
            nl_mat = bpm.ko * bpm.dist_z * dn_temp

            # influence of the index modulation on the field
            field_x = field * np.exp(1j * nl_mat)
            # Linear propagation over dz
            field_x = ifft(bpm.phase_mat * fft(field_x))

            cur_pow = bpm.epnc * abs2(field_x)

        try:
            bpm.variance(prev_pow, cur_pow)
        except ValueError as error:
            print(error)

        field = field_x

    field_3 = field
    cur_3 = cur_pow
    dn_3 = dn_temp

    plt.figure(num="Impact order kerr")
```

(continues on next page)

(continued from previous page)

```
    ax1 = plt.subplot(211)
    ax2 = plt.subplot(212)

    ax1.set_title("phase: comparison")
    ax2.set_title("power: comparison")

    ax1.set_xlim(-15, 15)
    ax2.set_xlim(-15, 15)

    ax1.plot(x, np.angle(field_ref), label="no kerr")
    ax1.plot(x, np.angle(field_1), label="dz+kerr")
    ax1.plot(x, np.angle(field_2), label="dz/2+kerr+dz/2")
    ax1.plot(x, np.angle(field_3), label="kerr+dz")

    ax2.plot(x, cur_ref, label="no kerr")
    ax2.plot(x, cur_1, label="dz+kerr")
    ax2.plot(x, cur_2, label="dz/2+kerr+dz/2")
    ax2.plot(x, cur_3, label="kerr+dz")

    ax1.legend(loc="upper right")
    ax2.legend(loc="upper right")

    plt.show()

    dn_ref = dn[nbr_step, :]

    plt.figure(num="Impact on dn order kerr")

    ax1 = plt.subplot(111)

    ax1.set_title("dn: comparison")

    ax1.set_xlim(-15, 15)

    ax1.plot(x, dn_ref, label="no kerr")
    ax1.plot(x, dn_1, label="dz+kerr")
    ax1.plot(x, dn_2, label="dz/2+kerr+dz/2")
    ax1.plot(x, dn_3, label="kerr+dz")

    ax1.legend(loc="upper right")

    plt.show()


def show_grid():
    """Show the computation grid and the displayed grid"""
    length_z = 10
    dist_z = 1
    nbr_z_disp = 6
    length_x = 10
    dist_x = 1
    bpm = Bpm(1, 1, length_z, dist_z, nbr_z_disp, nbr_z_disp, dist_x)
    print("Initial values")
    print("length_z=%f, nbr_z=%f, nbr_z_disp=%f, length_x=%f, nbr_x=%f" % (
            length_z, length_z/dist_z, nbr_z_disp, length_x, length_x/dist_x))
```

(continues on next page)

(continued from previous page)

```
    [length_z, nbr_z, nbr_z_disp, length_x, nbr_x, x] = bpm.create_x_z()
    print("Corrected values")
    print("length_z=%f, nbr_z=%f, nbr_z_disp=%f, length_x=%f, nbr_x=%f" % (
            length_z, nbr_z, nbr_z_disp, length_x, nbr_x))
    dn = np.zeros((nbr_z, nbr_x))

    z = np.linspace(0, length_z, nbr_z)
    z_disp = np.linspace(0, length_z, nbr_z_disp+1)
    dn_disp = np.linspace(0, nbr_z-1, nbr_z_disp+1, dtype=int)
    xv1, zv1 = np.meshgrid(x, z)
    xv2, zv2 = np.meshgrid(x, z_disp)

    plt.figure()
    ax1 = plt.subplot(121)
    ax1.set_title("Computation grid")
    ax1.set_xlabel("x (µm)")
    ax1.set_ylabel("z (µm)")
    ax1.pcolormesh(xv1, zv1, dn, cmap='gray', edgecolor="w")

    ax1.annotate("", xy=(-dist_x, 1.8*dist_z), xytext=(0., 1.8*dist_z),
                 arrowprops=dict(arrowstyle="<|-|>", connectionstyle="arc3",
                                 color='w'))
    ax1.text(-dist_x/2, 1.3*dist_z, "dist_x",
             {'color': 'black', 'fontsize': 12, 'ha': 'center', 'va': 'center',
              'bbox': dict(boxstyle="round", fc="white", ec="black", pad=0.2)})

    ax1.annotate("", xy=(-1.2*dist_x, 2*dist_z), xytext=(-1.2*dist_x, 3*dist_z),
                 arrowprops=dict(arrowstyle="<|-|>", connectionstyle="arc3",
                                 color='w'))
    ax1.text(-1.5*dist_x, 2.5*dist_z, "dist_z",
             {'color': 'black', 'fontsize': 12, 'ha': 'right', 'va': 'center',
              'bbox': dict(boxstyle="round", fc="white", ec="black", pad=0.2)})

    ax2 = plt.subplot(122)
    ax2.set_title("Displayed grid")
    ax2.set_xlabel("x (µm)")
    ax2.pcolormesh(xv2, zv2, dn[dn_disp], cmap='gray', edgecolor="w")

    ax2.annotate("", xy=(-dist_x, 1.85*bpm.pas), xytext=(0., 1.85*bpm.pas),
                 arrowprops=dict(arrowstyle="<|-|>", connectionstyle="arc3",
                                 color='w'))
    ax2.text(-dist_x/2, 1.6*bpm.pas, "dist_x",
             {'color': 'black', 'fontsize': 12, 'ha': 'center', 'va': 'center',
              'bbox': dict(boxstyle="round", fc="white", ec="black", pad=0.2)})

    ax2.annotate("", xy=(-1.2*dist_x, 2*bpm.pas), xytext=(-1.2*dist_x, 3*bpm.pas),
                 arrowprops=dict(arrowstyle="<|-|>", connectionstyle="arc3",
                                 color='w'))
    ax2.text(-1.5*dist_x, 2.5*bpm.pas, "pas",
             {'color': 'black', 'fontsize': 12, 'ha': 'right', 'va': 'center',
              'bbox': dict(boxstyle="round", fc="white", ec="black", pad=0.2)})

    plt.show()
#    plt.savefig("grid_definition.png", bbox="tight", dpi=250)
```

5.5 Release note

5.5.1 Todo (not sure)

Change the whole code structure to have a more readable and convinient one with a windows, waveguide, beam, propagation distinct class.

Add an autosave with possibility to go back to previous parameters choices.

Fixe the plotting issue when using tight layout.

5.5.2 Version 1.11

bpm.py and user_interface.py

Waveguides can have a finite z dimension.

Remove the losses menu to incorporate the losses in the refractive index.

Correct an error in the Kerr calculation. Previous results were wrong by a factor 2eta/no = 753/no.

Add the nonlinear refractive index n2 as a alternative to chi3.

Change most of the operations into numpy ones to speed up the computation.

Add a option to select which waveguide width and dn are used to find the corresponding optical mode.

Copy the main_compute function into the user_interface to be able to display the computation progression onto the interface.

The power in each waveguides can be computed and displayed without recomputing the whole propagation.

Variables chi3, n2 and irrad are now defined by a significand and a exponent, allowing to choose a larger range in the interface version.

Correct an error in the power calculation for curved waveguides. Previous results were wrong for high curvature factor.

Change the linestyle in the power display to have more control over it.

examples.py

Update examples.

Add a multimodal beam splitter example.

Add an attempt of benchmarking the Kerr effect.

5.5.3 Version 1.1

Implement a waveguide creation menu based on the existing beam menu. It is now possible to create as many waveguides as wanted, with different parameters for each one.

5.6 MIT License

5.7 Contact

The Beampy module was mainly developed by Jonathan Peltier.

And the interface was developed at first by Marcel Soubkovsky.

For any questions about Beampy please contact us at:

Jonathan Peltier: jonathanp57@outlook.fr, https://github.com/Python-simulation

Marcel Soubkovsky: https://github.com/marcelrsoub

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex

Last edit: Dec 31, 2020 for the version 1.11

PYTHON MODULE INDEX

b

INDEX

O

P

R

S

T

U

V