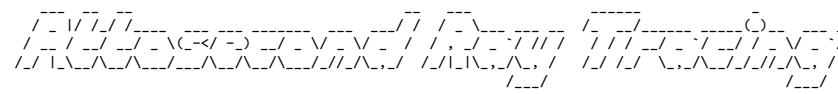
ART v0.4



Documentation

June 14, 2022

Anthony Guillaume[1], Charles Bourassin-Bouchet[2], and Stefan Haessler*

[1]Laboratoire d'Optique Appliquée, CNRS, Institut Polytechnique de Paris, France
[2]Laboratoire Charles Fabry, Institut d'Optique, CNRS, Université Paris-Saclay, France

Introduction

1.1 What is ART and what can it do?

ART stands for Attosecond Ray Tracing and is a free python code written by Anthony Guillaume, Stefan Haessler and Charles Bourassin-Bouchet. It does ray tracing calculations with a specific focus on ultrashort (femto- and attosecond) laser pulses. Therefore the code currently focuses on reflective optics, freely arrangeable including grazing incidence configurations.

Ten years ago, Charles made geometric optics calculations that demonstrated how sensitive attosecond pulses are to spatio-temporal distortions and how easily such distortions are picked up in the reflective grazing-incidence optical setups required to transport and refocus them [C. Bourassin-Bouchet *et al.* "How to focus an attosecond pulse". *Opt. Express* **21**, 2506 (2013); C. Bourassin-Bouchet *et al.* "Spatiotemporal distortions of attosecond pulses". *JOSA A* **27**, 1395 (2010)].

ART now makes these calculations avaible to the ultrafast optics community in a free and (hopefully) easily accessible python code.

1.2 Installation

You can get the ART code from GitHub. If you're using the *git* version control software, you can clone the code like so have to put it on github first...:

```
git clone https://github.com/mightymightys/AttosecondRaytracing.git
```

Maybe some words about forking it and contributing to further development or so...

1.2.1 Dependencies

The code requires Python 3.6 or newer and depends on the following libraries :

- NumPy, `https://numpy.org`
- Matplotlib, `https://matplotlib.org`
- Numpy-Quaternion, `https://github.com/moble/quaternion`

- *for 3D-rendering your optical configuration and the rays:*
  Mayavi, `https://docs.enthought.com/mayavi/mayavi`
- *for collecting results when looping over varying optical configurations:*
  Pandas, `https://pandas.pydata.org/`
  and tqdm, `https://pypi.org/project/tqdm/`

You can install those very easily if you use the anaconda/miniconda distribution by entering in the Anaconda prompt:

```
conda install -c anaconda numpy
conda install -c anaconda matplotlib
conda install -c conda-forge quaternion
conda install -c conda-forge mayavi
conda install -c anaconda pandas
conda install -c anaconda tqdm
```

Make sure these exist for your python version number. In particular the Mayavi package (or rather the VTK package that it depends on) is not always available for the most recent python version. The code is tested to run with *Miniconda3* version 2022.05, Conda 4.12.0, Python 3.9.7, on 64-bit Windows 10. Just in case, you can get older *Anaconda* versions from `https://repo.anaconda.com/archive/`, and older *Miniconda* versions from `https://repo.anaconda.com/miniconda/`.

Inside ART

ART is run from a launch script, for which a number of examples named `UseRayTracing_XXX.py` are provided. These can be edited and extended by the user and will serve as a start for deriving own launch scripts. These scripts call the actual code, organized in a number of modules in files all named `ModuleXXX.py` located in the folder `ART`. That same folder also contains the files `DoART.py` and `DoPLOTS.py`, which contain two standard code blocks called by the launch scripts. **To start ART, run a launch script in an IPython-console** (or e.g. in the *Spyder*-IDE).

The built-in analysis options and how to go further using ART's output data is described in section 2.5.

The impatient may immediately want to jump to section 3 that describes how to configure a simulation and walks you through the provided launch scripts.

In any case, keep in mind the following remarks:

- In the following, the (case-sensitive!) name of a `class` /`object or instance of that class` will be printed in blue, the name of a `function` / `class method` will be printed in green, and the name of a `variable` will be printed in red. (Strictly speaking, in Python every variable is an object. We make here the distinction with the classes and objects thereof which are specifically defined within ART.)
- **All distances are in millimeters.**
- All vectors and points in space are of the type `numpy.ndarray` with length 3.
- For every class and function described in the following, we will mention which module it is part of. In order to call these functions or classes you must first import the module and then call it by `ModuleXXX.FunctionOrClass`.

2.1 Light rays in ART

The light rays of geometric optics are described in ART by the class

`Ray`(`Point`=`numpy.ndarray`, `Vector`=`numpy.ndarray`, `Path`=`float`, `Number`=`int`, `Wavelength`=`float`, `Intensity`=`float`, `Incidence`=`float`),

contained in the module `ModuleOpticalRay.py`. A light ray is then defined by a `Point`

in space and direction `Vector`. These two properties are obviously essential for a `Ray`-object. However, it can be given a list of additional *optional* properties:

- `Path`: represents the optical path length covered by the ray *before it reached its starting point* `Point`, i.e. it keeps track of the length of the preceding ray path and is automatically determined by the program when creating new rays at each optical element.
- `Number`: is an integer that allows the program to link an incident to the corresponding reflected ray. The central ray of the light source (symmetry axis of the beam) by convention always carries the number 0 and the functions creating the sources attribute the numbers automatically.
- `Wavelength`: represents the wavelength associated with the rays and can be set by the users to implement cromatic effects.
- `Incidence`: is the angle of incidence (angle between the ray and the normal to the optics surface at the ray-optic intersection point) in radian of the ray on the optic, and is determined automatically by the program.
- `Intensity`: represents the light intensity (in arbitrary units) carried by the ray. It gives an indication of the energy distribution in the beam in simple cases and when the rays are mutually separated by many wavelengths such that interference can be neglected.

Advanced users may extend this list of properties to implement more advanced simulations.

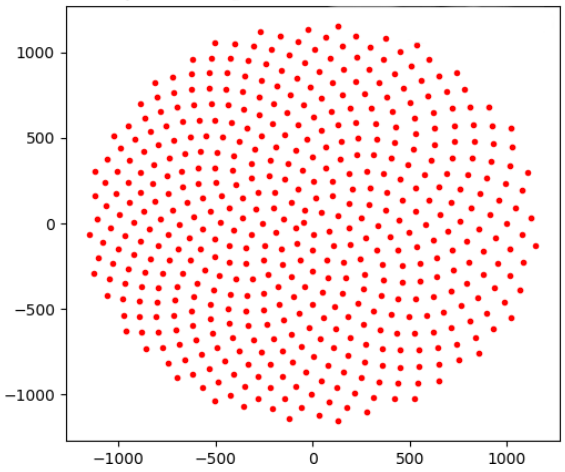
Note that the variable-names begin with a capital letter, as written above, but the corresponding object-properties begin with a lower-case letter. For example, to access a ray's number, you would write `Ray.number`.

2.2 Setting up the light source

`ModuleSource.py` contains the functions that create a python-list of rays representing the light source.

Plane Wave

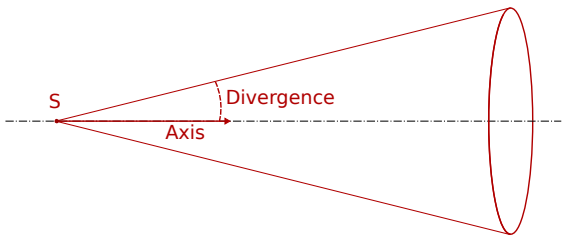
- The function `PlaneWaveDisk(Centre=numpy.ndarray, Axis=numpy.ndarray, Radius=float, NbRays=int)`
  creates a list of `NbRays` rays, all parallel to the vector `Axis` and whose source points all lie within the plane normal to `Axis` and containing the point `Centre`. `Radius` determines the radius of the circular beam of parallel light rays. The rays are uniformly distributed according to a Vogel spiral as shown below.



- The function PlaneWaveSquare(Centre=numpy.ndarray, Axis=numpy.ndarray, SideLength=float, NbRays=int)
  creates a list of ray's much as described above for PlaneWaveDisk except that the beam has a square cross-section of side length SideLength.

Point source

- The function PointSource(S=numpy.ndarray, Axis=numpy.ndarray, Divergence=float, NbRays=int)
  creates a list of NbRays rays, all starting from the same source point S and describing a cone. Axis is a vector (of arbitrary length) defining the symmetry axis of that cone. Divergence is the *half* apex angle of the cone in radian, i.e. analogy the divergence angle of a Gaussian beam. The rays are uniformly distributed according to a Vogel spiral, as shown above.



- The function GridRayList(Mirror, S=numpy.ndarray, NbRays=int)
  creates a list of NbRays rays, all starting from the same source point S and hit the surface of the Mirror (cf. description of these classes in section 2.3.2) in a uniformly distributed fashion. This supposes that the Mirror is centered at the origin [0,0,0] and its normal direction is the z-direction [0,0,1]. This is a way of simulating a point source that uniformly irradiates the first mirror in the optical setup.

Extended source

- ExtendedSource(S=numpy.ndarray, Axis=numpy.ndarray,
  Diameter=float ,Divergence=float, NbRays=int)
  creates a list of at least NbRays rays, forming an extended array of point sources, distributed according to a Vogel spiral over a disk with Diameter, normal to Axis and centered at point S. Each point source has Axis and Divergence as defined above for PointSource.

Adding more attributes

- The function ApplyGaussianIntensityDistribution(RayList=list of Rays,
  IntensityFraction=float$\in (0,1)$)
  applies a Gaussian intensity distribution to the intensity-property of the Rays in the list RayList. The ray RayList[0] takes the role of the symmetry axis of the intensity distribution. If the Ray-list represents a beam with finite divergence, then the intensity is attributed according to the angle between the ray and the symmetry axis, otherwise (in the plane-wave case) it is attributed according to the distance of the ray to the symmetry axis. The intensity (in arbitrary units) drops in a Gaussian dependence from 1 to a minimum value given by IntensityFraction at the border of the beam represented by the Ray-list.

2.3 Setting up optical elements

An optical element is described by the class

OpticalElement(Type, Position=numpy.ndarray, Normal=numpy.ndarray,
MajorAxis=numpy.ndarray),

contained in the module **ModuleOpticalElement.py**. It serves to connect the "lab-frame" in which the light rays are traced to the proper "element" coordinate frame in which an optic is defined. For the moment the implemented types are masks (cf. section 2.3.3) and different mirrors (cf. section 2.3.2).

The Position gives the optical element's center in the lab frame (what this center is, depends on the Type), the Normal is the lab-frame vector pointing against the direction that would be considered as normal incidence on the optical element (the z-direction in the element frame), and MajorAxis is the lab-frame vector of another distinguished axis (the x-axis in the element frame and therefore required to be perpendicular to Normal). This lets the user fix the optical element's rotation about Normal. Consequently, MajorAxis is of importance only for optical elements that are *not* rotationally symmetric about Normal, and corresponds, e.g. to the off axis direction for a parabolic mirror or the major axis of a toroidal or ellipsoidal mirror. See the figures in section 2.3.2 for illustrations of these vectors for the various implemented mirror types.

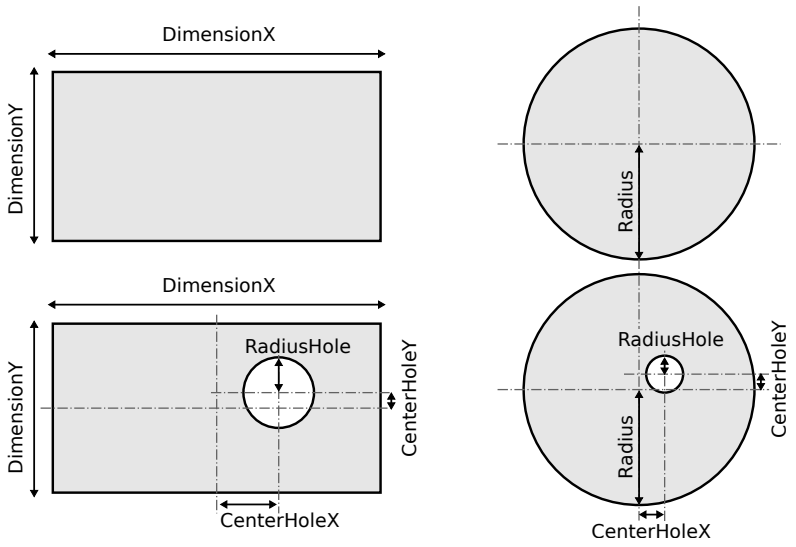
ART also features the function OEPlacement, described in section 3, which greatly facilitates setting up optical elements along a central light ray from a list of more easily

accessible paramters like distances and incidence angles.

2.3.1 Supports

The spatial extent and outer outer shape of optical elements is defined by a "support". Technically, the support is a section of the x-y-plane in the element coordinate frame. The implemented classes represent create round or rectangular shapes and may have a hole or not (as illustrated by the figure below), and are contained in the module **`ModuleSupport.py`**:

- `SupportRectangle(DimensionX=float, DimensionY=float)`
- `SupportRectangleHole(DimensionX=float, DimensionY=float, HoleX=float, HoleY=float, CenterHoleX=float,CenterHoleY=float)`
- `SupportRound(Radius=float)`
- `SupportRoundHole(Radius=float, RadiusHole=float, CenterHoleX=float, CenterHoleY=float)`



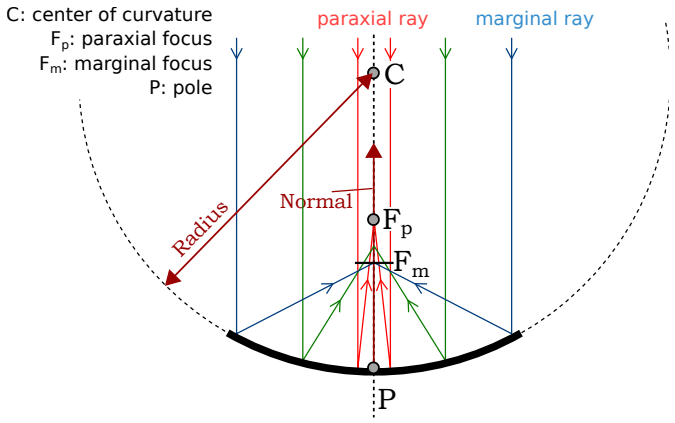
2.3.2 Mirrors

A mirror is a type of an optical element, defined by a support, taking the role of the mirror substrate, and by a surface shape, off of which rays get reflected according to the law of reflection. Rays that do not hit the support are not further propagated. The mirror center, fixed by the `Position` argument to the `OpticalElement` class, is the point on the mirror surface at the center of the mirror support, marked as point P in the schemes below.

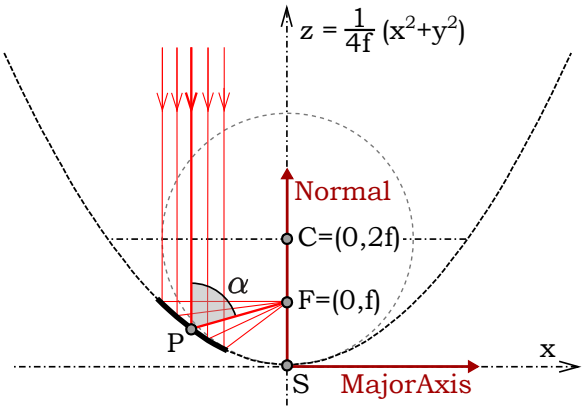
The different surface shape make up a classes of mirrors, representing types for an `OpticalElement`, and are contained in the module **`ModuleMirror.py`**. The implemented classes are

- `MirrorPlane(Support)`,
  where `Support` is an object of the support classes described above. The `Normal` direction is obviously the surface normal, and the `MajorAxis` has no effect other than rotating a rectangular support abouth the surface normal.

- `MirrorSpherical(Radius=float, Support)`,
  where `Support` is an object of the support classes described above and `Radius` is the radius of curvature of the spherical surface. The paraxial focal length is `Radius`/2. A positive/negative valued `Radius` yields a concave/convex mirror surface. As illustrated below, the `Normal` argument to the `OpticalElement` class is the surface normal at the pole point, and the `MajorAxis` has no effect other than rotating a rectangular support abouth the surface normal.

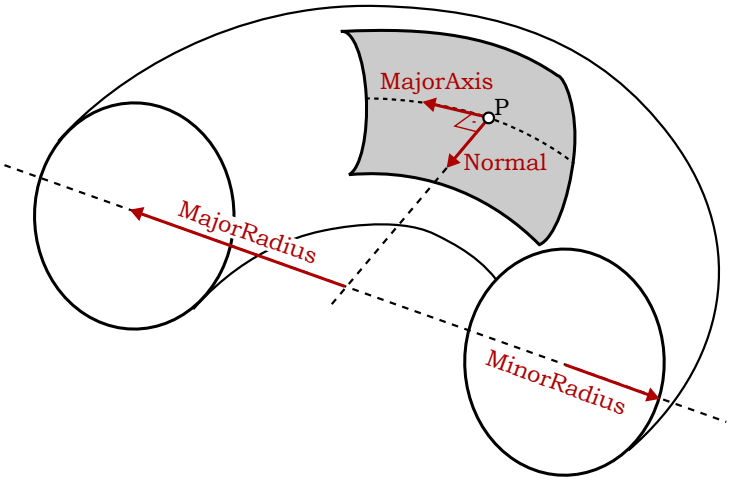


- `MirrorParabolic(SemiLatusRectum=float, OffAxisAngle==float, Support)`,
  where `Support` is an object of the support classes described above,
  `SemiLatusRectum` is the parabola-parameter of the same name, equal to twice the focal length f of the mother parabola, and the `OffAxisAngle` is the angle α represented the scheme below. The `Normal` argument to the `OpticalElement` class is the symmetry axis of the mother paraboloid and the `MajorAxis` is the off-axis direction.



- `MirrorToroidal`(`MajorRadius`=`float`, `MinorRadius`=`float`, `Support`), where `Support` is an object of the support classes described above and `MajorRadius` and `MinorRadius` are the radii of the torus as illustrated in the scheme below. The `Normal` argument to the `OpticalElement` class is the surface normal at the center of the support, and the `MajorAxis` the tangent to the torus along the circle with the major radius.

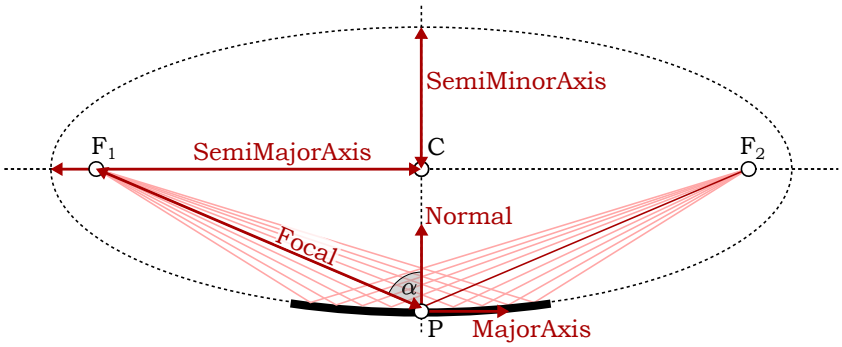
  Often, one rather wants to define the toroidal mirror in terms of its focal length and its optimal incidence angle instead of its two radii. In that case, the function `ReturnOptimalToroidalRadii`(`Focal`=`float`, `AngleIncidence`=`float` [deg]), also contained in **`ModuleMirror.py`**, returns the corresponding optimal major and minor radii (cf. example launch scripts).



- `MirrorEllipsoidal`(`SemiMajorAxis`=`float`, `SemiMinorAxis`=`float`, `Support`), where `Support` is an object of the support classes described above and `SemiMajorAxis` and `SemiMinorAxis` are the lengths defining the ellipsoid as illustrated in the scheme below. The `Normal` argument to the `OpticalElement` class is the surface

normal at the center of the support (along the minor axis), and the MajorAxis the tangent to the ellipsoid at that point, parallelto the major axis.

Often, one rather wants to define the ellpsoidal mirror in terms of its effective focal distance and incidence angle instead of the two axis lengths. In that case, the function ReturnOptimalEllipsoidalAxes(Focal=float, AngleIncidence=float [deg]), also contained in **ModuleMirror.py**, returns the corresponding optimal semi major and minor axis lengths. *An "off-axis" ellipsoidal mirror with different first and second focal distances is not yet implemented.*



- MirrorCylindrical(Radius=float, Support),
  where Support is an object of the support classes described above and Radius is the radius of curvature of the cylindrical surface. The paraxial focal length is Radius/2. A positive/negative valued Radius yields a concave/convex mirror surface. The Normal argument to the OpticalElement class is the surface normal at the center of the support, and the MajorAxis is parallel to the cylinder axis.

2.3.3 Masks

A mask is a type of an optical element, represented by a plane surface of the shape of its support which stops all rays that hit it, while all other rays continue their path unchanged. Its center is the point fixed by the Position argument to the OpticalElement class, its normal is the Normal argument and the MajorAxis argument is along the x-direction used for the definition of the support (cf. figure above). For setting the orientation angles of the masks, one can think of it like a plane mirror, only that its action is different. There is thus only a single class of masks:

- Mask(Support),
  where Support is an object of the support classes.

2.4 Launching the Raytracing calculation

1. Once the list of source light rays, ListofSourceRays, and the optical elements, OpticalElement1, OpticalElement2..., are defined, they are combined into a list

OpticalChain = [ListofSourceRays, OpticalElement1, OpticalElement2, ...].

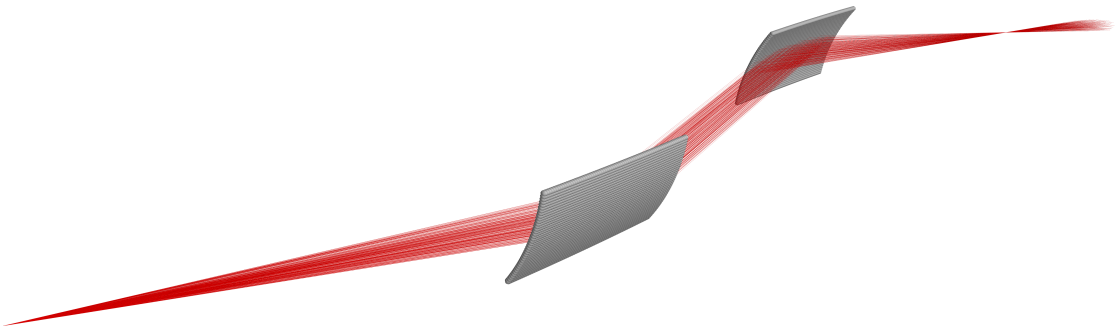
2. This list is taken as an argument by the function
RayTracingCalculation(OpticalChain=list)
contained in the **ModuleProcessing.py**, which will perform the actual ray tracing calculation and return the variable RayListHistory.

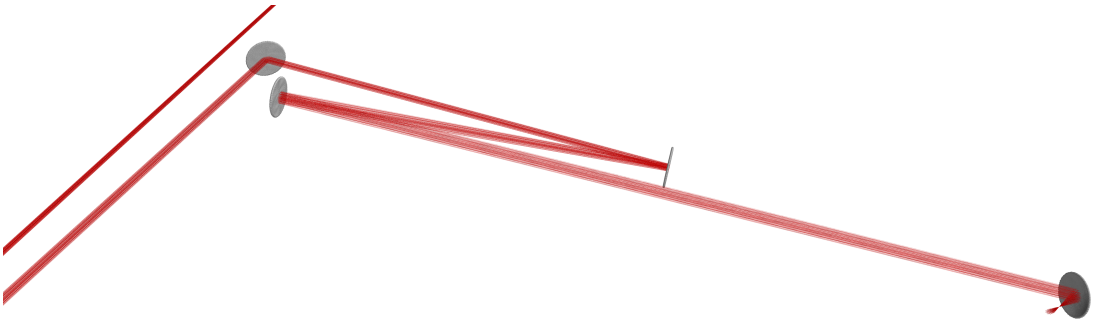
3. RayListHistory is a list of lists of rays, each list containing the rays bundle as it propagates from the source (RayListHistory[0]), after the first optical element (RayListHistory[1]), etc.. These ray lists can then be analysed as described in section 2.5.

2.5 Output data analysis

2.5.1 Rendering the complete optical chain and traced rays

The optical chain, described by the user-defined OpticalChain and the traced light rays contained in RayListHistory can be rendered in 3D by the the function
RayRenderGraph(RayListHistory=list of rays, OpticalChain=list,
EndDistance=float,maxRaysToRender=int)
contained in the module **ModuleAnalysisAndPlots.py**. Here, RayListHistory is the output of the ray-tracing calculation, OpticalChain is the list containing the light source and optical elements as described in section 2.4, EndDistance is the distance after the last optical element over which the rays will be drawn, and maxRaysToRender is the maximum number of rays to include in the rendering.
The generation of the image takes several typically about 10 seconds when including about 200 rays, which is significantly longer than the actual raytracing calculation for typically number of $\sim$ 1000 rays launched by the light source. It is therefore advisable to only generate this rendering when required and keep the number maxRaysToRender as low as possible. Two examples of renderings obtained with maxRaysToRender = 200 are show below.





2.5.2 Analysing rays in a detector plane

ART lets the user place a virtual detector —a plane in 3D space— in the ray path. The user first selects the ray bundle they want to have analysed by that detector by giving the corresponding index of `RayListHistory` in the variable `ReflectionNumber` = `float`. In most cases, the ray list after the complete optical chain, `RayListHistory[-1]`, is going to be the most interesting one, i.e. one would set `ReflectionNumber` = `-1`, but one may also choose a section inbetween two optical elements along the optical chain.

Setting up the detector

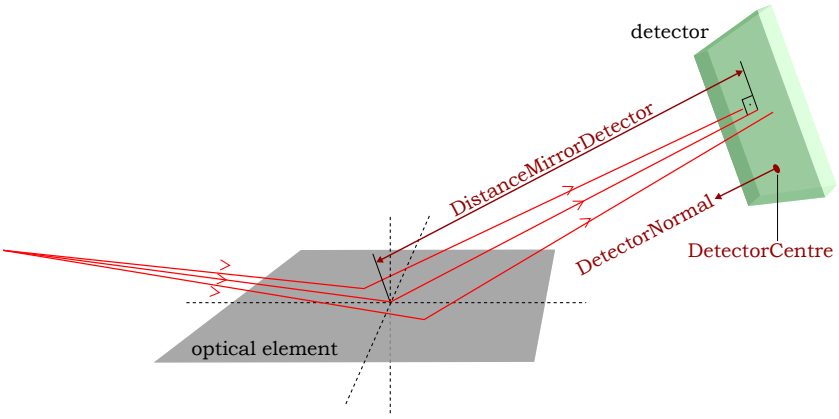
The detector is described by the class
`Detector(RefPoint=numpy.ndarray, Centre=numpy.ndarray,`
`Normal=numpy.ndarray)`
contained in the module **`ModuleDetector.py`**. Here, `Centre` and `Normal` are a point belonging to the detector plane and its normal vector, respectively. These are *optional* arguments when creating a `Detector`-instance, because they can later be set automatically by the method `autoplace` (cf. below). `RefPoint` is a point with respect to which the detector distance will be measured via the method `Detector.get_distance`. Per default, ART sets the center of the last optical element, `OpticalChain[ReflectionNumber].position` as the `RefPoint`.

Manually Creating an instance of the `Detector` and specifying its `Centre` and `Normal` corresponds to manually placing the detector.
This is done when the boolean `ManualDetector=True`, in which case the user has to specify 3D numpy-arrays `DetectorCentre` and `DetectorNormal` in their launch script, cf. section 3.

Automatic The detector plane can be automatically set to be perpendicular to the central ray of the bundle to be analysed at a distance `DistanceDetector` from the `RefPoint`. This is achieved by the method

`Detector.autoplace(RayList=List of Rays, DistanceDetector = float)`. In contrast to manually placing the detector, this can obviously only be done after the ray tracing calculation has executed.
This is the default behaviour of ART and the user has to specify `DistanceDetector` in their launch script, cf. section 3.

Moving the detector The detector can be moved by the methods
`Detector.shiftToDistance(NewDistance= float)`
and `Detector.shiftByDistance(DistanceShift= float)`,
which move the detector plane along its normal direction *to* or *by* the specified distances.



Automatic optimization of detector distance If the launch script sets the boolean `AutoDetectorDistance = True`, ART will automatically move the detector plane along its normal and find the distance that maximises an estimate for the obtained "intensity" by minimising the quantity $d^2\tau$, where d is the spatial standard deviation of the points where the rays impact the detector, and τ is the standard deviation of the total ray travel times to the detector, determined as the ray path lengths divided by the speed of light. The initially user-specified detector position is then used as a first guess for this search.

If the boolean `FindOptDistance = True`, ART will search for that same optimal detector distance and save it in the variables `OptDistance` and `OptDetectorCentre`. The corresponding optimal spot size (standard deviation of the points where the rays of `RayList` hit the detector plane) will be saved in `OptSizeSpot` and the optimal pulse duration (standard deviation of the the ray delays, cf. 2.5.2) in `OptDuration`. Unless `AutoDetectorDistance = True`, ART will however not move the detector to the found optimal distance.

If the boolean `FindBestDuration = True`, ART will move the detector plane along its normal and find the distance that minimizes the pulse duration (standard deviation of the the ray delays, cf. 2.5.2). The found distance is saved in the variable `DistanceBestDuration`, the minimized duration in `BestDuration` and the correspond-

ing spot size (standard deviation of the points where the rays of RayList hit the detector plane) in SizeBestDuration. ART will not move the detector to the found distance.

If the boolean FindBestSpot = True, ART will move the detector plane along its normal and find the distance that minimizes the spot size (standard deviation of the points where the rays of RayList hit the detector plane). The found distance is saved in the variable DistanceBestSpot, the minimized spot size in BestSize, and the corresponding duration (standard deviation of the the ray delays, cf. 2.5.2) in DurationBestSpot. ART will not move the detector to the found distance.

Getting the detector response

Once an object of the detector-class is defined, it provides provides the following basic methods to get its "response":

- Detector.get_PointList2DCentre(RayList=list of rays),
  where, as explained above, the RayList is the ray bundle to be analysed, e.g. RayList = RayListHistory[-1] for the one after the last optical element.
  The function returns the list ListPointDetector2DCentre of 2D coordinates in the detector plane of the points where the rays of RayList hit the detector, with the coordinate origin centered on the point cloud. The list index corresponds to that of the RayList.

- Detector.get_Delays(RayList=list of rays),
  where, as explained above, the RayList is the ray bundle to be analysed, e.g. RayList = RayListHistory[-1] for the one after the last optical element.
  The function returns the list DelayList of delays of the rays of RayList as they hit the detector plane, calculated as their total path length divided by the speed of light. The delays are relative to the mean "travel time", i.e. the mean path length of RayList divided by the speed of light. The list index corresponds to that of the RayList.

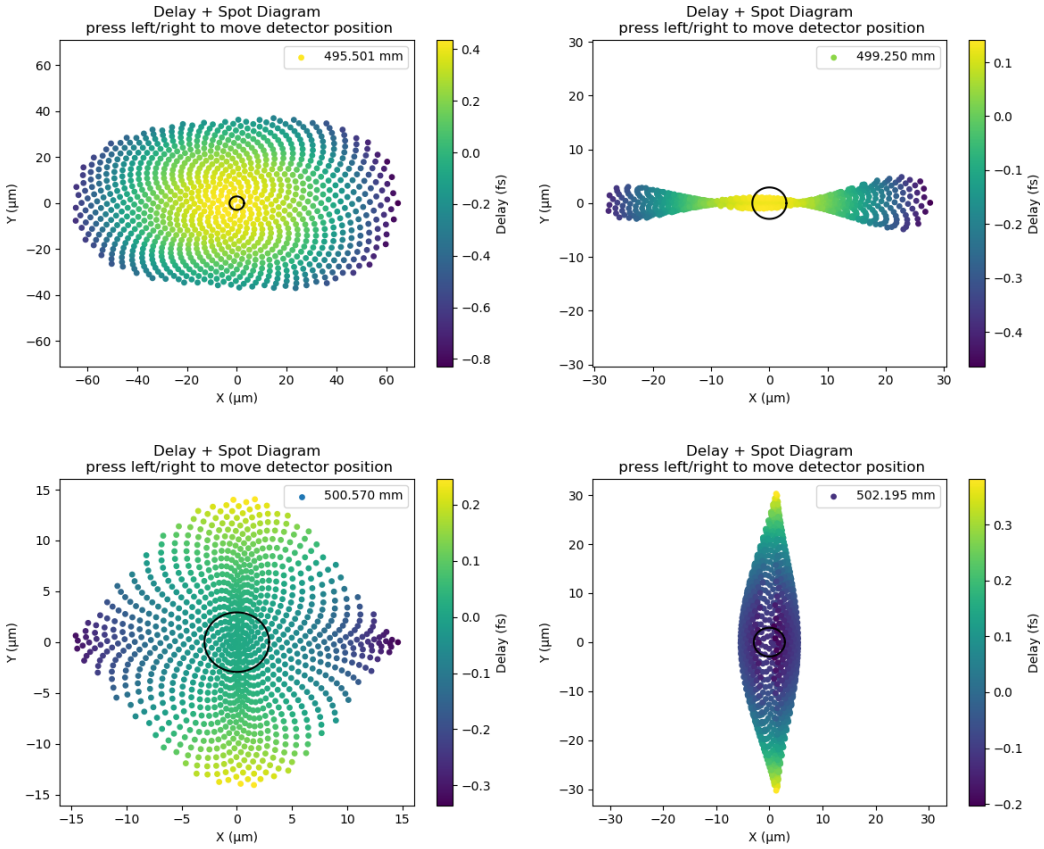
Plotting the detector response

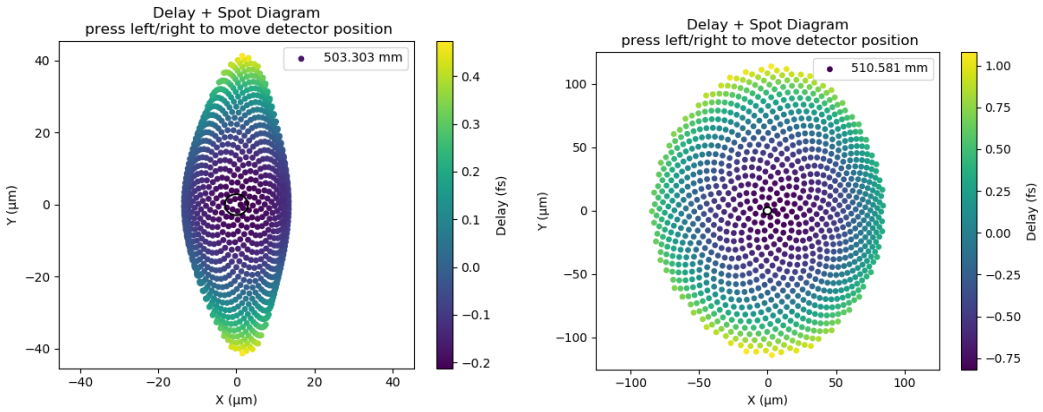
The following functions, contained in the module **ModuleAnalysisAndPlots**, generate a number of built-in graphical representations of ART's results:

- The function SpotDiagram(RayListAnalysed=list of rays, Detector,
  Wavelength=float, DrawAiryAndFourier=boolean, ColorCode=string)
  generates a spot diagram, i.e. a 2D representation of the impact points of the rays in the bundle RayListAnalysed on the detector plane Detector, with an optional additional property encoded by the color of the data points. The latter is given by the *optional* argument ColorCode which can be Delay, Intensity or Incidence (angle in degress on the preceeding optical element). The switch DrawAiryAndFourier decides whether or not the Airy disk size (calculated from the light Wavelength and the beam numerical aperture estimated for the rays in

the bundle `RayListAnalysed`) will be plotted as black circle. This can serve as a reference for the spatial scale from which on diffrative effects will outweight those desribed the ray optics treated by ART.

The generated figure is *interactive*, i.e. by pressing the left and right cursor keys, the detector plane is shifted along its normal so as to explore the ray bundle along its propagation. Examples obtained for a point source with 10 mrad divergence half-angle imaged by a single toroidal mirror in 2f–2f geometry and incidence angle misaligned by 0.01° are shown below.

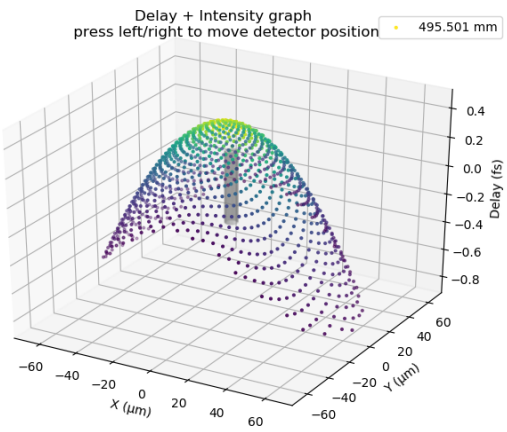


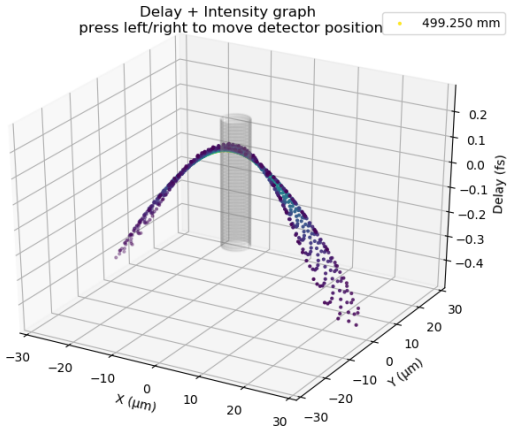


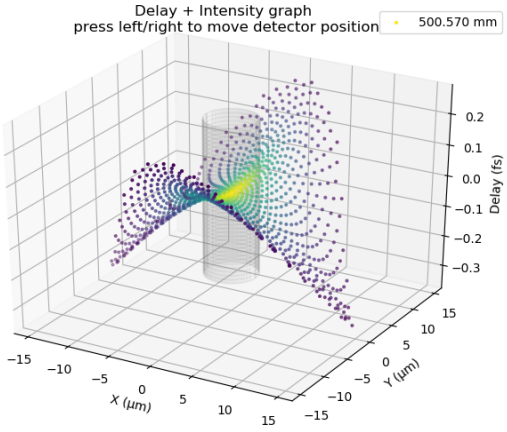
- The function DelayGraph(RayListAnalysed=list of rays, Detector, Wavelength=float, DeltaFT=float, DrawAiryAndFourier=boolean, ColorCode=string)

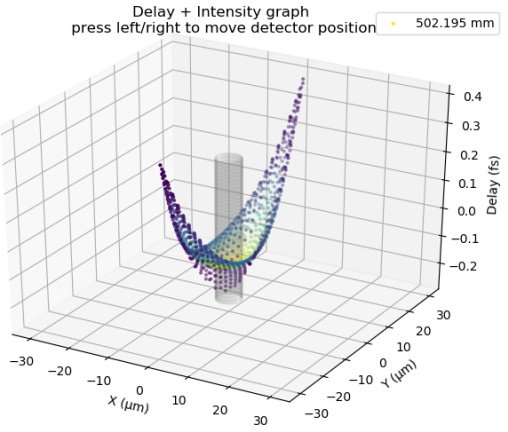
  generates 3D "ultrafast" spot diagram, i.e. in the first 2 dimensions the same impact points of the rays in the bundle RayListAnalysed on the detector plane Detector as in the spot diagram described above, and in the third dimension the ray delays calculated as their total path length divided by the speed of light. By default, the data point colour also encodes the ray delay, but it can optionally also encode the ray intensity or incidence angle on the last optical element by setting the *optional* argument ColorCode to Intensity or Incidence. The switch DrawAiryAndFourier decides whether or not a cylinder with a diameter of the Airy disk size (calculated from the light Wavelength and the beam numerical aperture estimated for the rays in the bundle RayListAnalysed) and height DeltaFT representing an ad hoc Fourier-limited pulse duration, will be plotted as black mesh. This can serve as a reference for the spatial scale from which on diffrative effects will outweight those desribed the ray optics treated by ART.

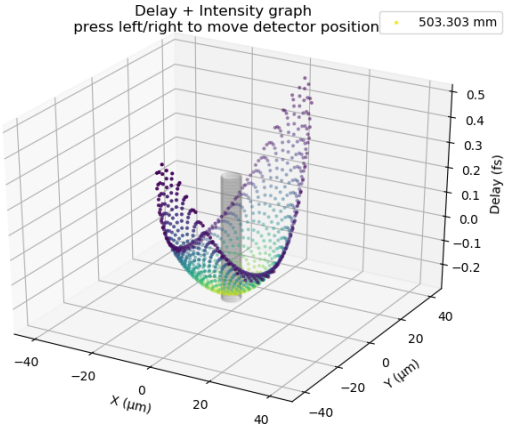
  The generated figure is *interactive*, i.e. by pressing the left and right cursor keys, the detector plane is shifted along its normal so as to explore the ray bundle along its propagation. Examples with the data point color encoding the ray "intensity" obtained for a point source with 10 mrad divergence half-angle imaged by a single toroidal mirror in 2f–2f geometry and incidence angle misaligned by 0.01° are shown below.

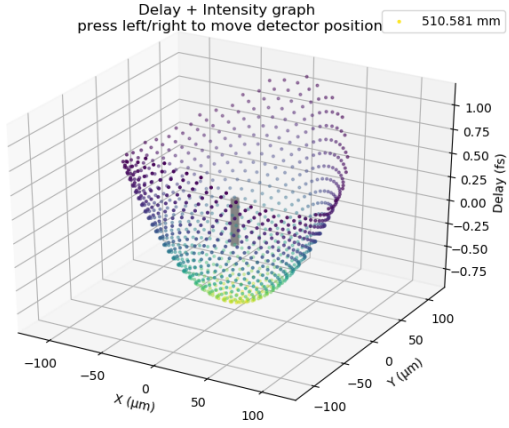

Delay + Intensity graph
press left/right to move detector position
495.501 mm
X (μm)
Y (μm)
Delay (fs)


Delay + Intensity graph
press left/right to move detector position
499.250 mm
X (μm)
Y (μm)
Delay (fs)


Delay + Intensity graph
press left/right to move detector position
500.570 mm
X (μm)
Y (μm)
Delay (fs)

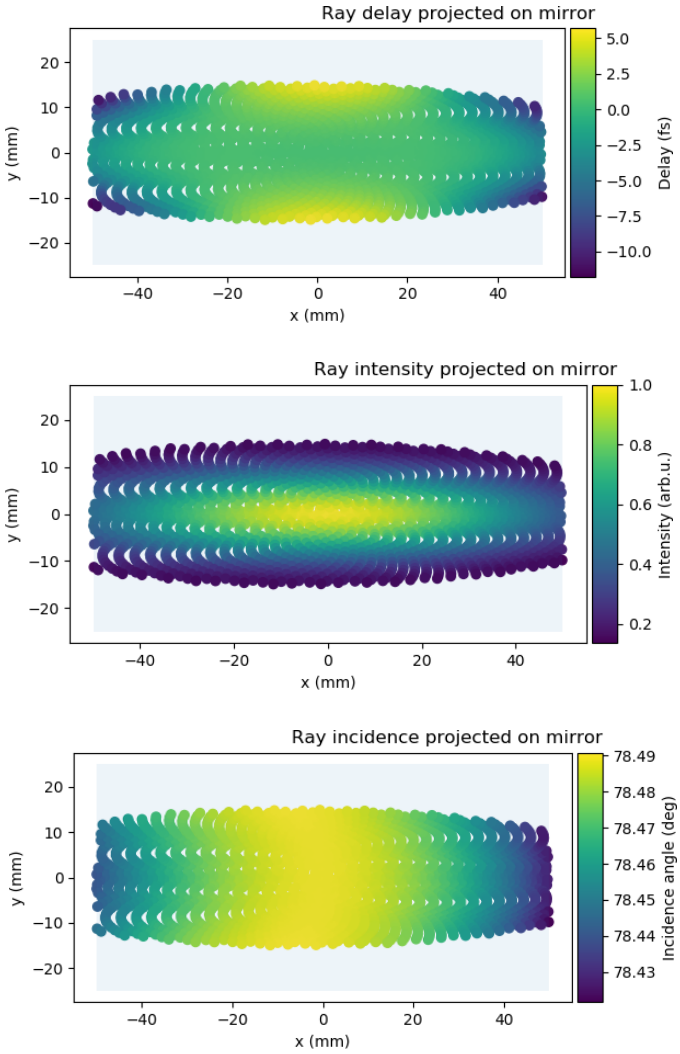

Delay + Intensity graph
press left/right to move detector position
502.195 mm
X (μm)
Y (μm)
Delay (fs)


Delay + Intensity graph
press left/right to move detector position
503.303 mm
X (μm)
Y (μm)
Delay (fs)


Delay + Intensity graph
press left/right to move detector position
510.581 mm
X (μm)
Y (μm)
Delay (fs)

- The function MirrorProjection(RayListAnalysed=list of rays, OpticalChain=list, ReflectionNumber=float, Detector, ColorCoded=string) generates a 2D plot of the projection of the impact points of the rays in RayListAnalysed on the surface of the optical element OpticalChain[ReflectionNumber]. The color of the data points can encode a property selected by the string ColorCoded, which can be 'Intensity', 'Incidence' or Delay.

  Examples obtained for a toroidal mirror imaging in 2f–2f geometry a point source with 30 mrad divergence half-angle and an incidence angle misaligned by 0.01° are shown below. The beam gets clipped because the mirror is too short.



Using ART — setting up a simulation

3.1 Automatic generation of an "optical chain"

While setting up the light source and optical elements "by hand" as described in sections 2.2 and 2.3 gives full freedom in defining one's setup, it can be cumbersome as it requires calculating in advance all the 3D coordinates and orientations of the setup's elements. Often, one rather thinks only of distances between elements along an optical beam path, and for each element of an incidence angle and possibly a rotation of the incidence plane from one optical element to the next. This approach is taken by the function

```
OEPlacement(Divergence=float, SourceSize=float, RayNumber=int,
     OpticsList=list of Mirrors, DistanceList=list of floats,
     IncidenceAngleList=list of floats,
     IncidencePlaneAngleList=list of floats)
```

contained in the module **ModuleProcessing.py**, which cautomatically creates an `OpticalChain` (cf. section 2.4) containing the light source and the successive optical elements.

`Divergence` is the divergence half-angle of the light source (cf. section 2.2) in radian. For a finite value, this creates a point source (if `SourceSize`=0) or extended source, while for `Divergence`=0, a plane wave with diameter `SourceSize` is created. `SourceSize` is the size in millimeters of the light source. `RayNumber` is the number of rays launched by the light source. The light source is launched into the direction of the optical beam path. `OEPlacement` applies a Gaussian intensity distribution dropping from 1 to $1/e^2$ from the centre to the edge of the ray bundle.

`OpticsList`=[`Mirror1`, `Mirror2`, ...] is a list of objects of the classes of optical elemets (cf. sections 2.3.2, 2.3.3) in their order of sucession along the optical beam path.

`DistanceList`=[d_1, d_2, ...] is the list of distances in millimetres of the optical elements along the optical beam path in their order of succession.

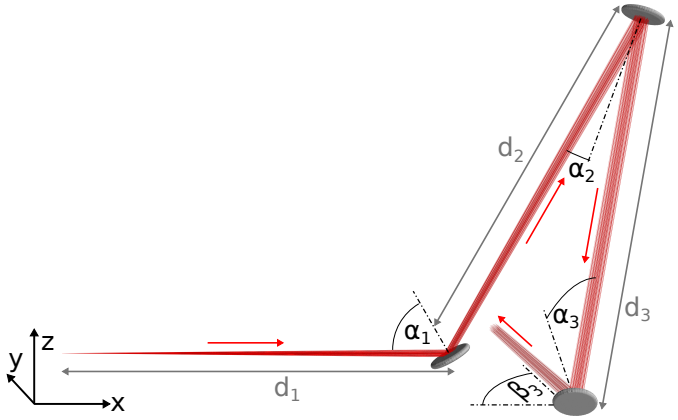
`IncidenceAngleList`=[α_1, α_2, ...] is the list of angles of incidence (between incident light axis and the optical element normal[3]) in degrees of the optical beam path on the optical elements in the order of sucession along the optical beam path. The sign of the angle determines the direction of reflection: a positive or negative value directs the beam

[3]Note that in the case of an off-axis parabolic mirror, this normal is the symmetry axis of the mother parabola, which is why perfect alignment of a focusing off-axis parabola is obtained with $\alpha = 0$, whereas a collimating off-axis parabola with off-axis angle θ is aligned when $\alpha = \theta$.

path to the left or the right, respectively.

IncidencePlaneAngleList=[β_1, β_2, ...] is the (optional) list of angles by which the plane of incidence is being rotated (by rotating the optical element normal about the incident optical beam axis) by the mirrors in the order of sucession along the optical beam path. The angle β_1 thus has no qualitative effect.

The following illustration shows an optical chain defined by DistanceList=[d_1 = 200, d_2 = 200, d_3 = 200], IncidenceAngleList=[α_1 = 60, α_2 = −10, α_2 = 45] and IncidencePlaneAngleList=[$\beta_1 = 0$, $\beta_2 = 0$, $\beta_3 = 90$]. In the third reflection, the plane of incidence is turned from the initial x-z-plane by 90° to the x-y-plane.



3.1.1 Subsequent modificaction of the alignment

The function OEPlacement creates a "perfectly" aligned chain of optics respecting the user-specified incidence angles with the optical elements all centered on the incident beam path and their major axes all lying in the plane of incidence. This can be modified afterwards, for example to simulate a misalignment, by picking out an optical element from the optical chain, e.g. the last one: pickedMirror=OpticalChain[-1], and then modifying the properties of this OpticalElement-object (cp. section 2.3): pickedMirror.position, pickedMirror.normal or pickedMirror.majoraxis.

The module **ModuleGeometry.py** contains some useful functions to help with this:

- RotationAroundAxis(Axis=numpy.ndarray, Angle=float, Vector=numpy.ndarray) rotates Vector by Angle in radians about Axis. For example, to tilt the major axis a bit about the normal:
  pickedMirror.majoraxis = ModuleGeometry.RotationAroundAxis(pickedMirror.normal, Angle [rad], pickedMirror.majoraxis).

- TranslationPoint(Point=numpy.ndarray, T=numpy.ndarray) simply translates Point by the vector T. For example, to shift the optical element by X mm along its major axis:
  pickedMirror.position = ModuleGeometry.TranslationPoint(pickedMirror.position, X*pickedMirror.majoraxis).

3.2 A launch script

In the following, we will go through an example launch-script which uses ART's automatic methods to set up, perform and analyse a simulation of a point source being reimaged in an f-f-f geometry by two grazing incidence toroidal mirrors, the second of which is rotated so as to flip the plane of incidence by 90°. This launch script is essentially the same as that in the provided `UseRayTracing_2toroidals_twisted.py`, apart from the loop over the rotation angle of the second mirror in that script.

- As usual in the beginning, we load some modules that we will use:

```
#%% Modules
import numpy as np

import ART.ModuleMirror as mmirror
import ART.ModuleSupport as msupp
import ART.ModuleProcessing as mp
```

- Then there is a block of boolean switches mainly selecting which plots to produce in the end:

```
#%%#############################################################################
""" User cell """
# print intermediate results and info in the console?
verbose = True

# look for detector distance with smallest spotsize?
FindBestSpot = False
# look for detector distance with shortest duration?
FindBestDuration = False
# look for optimal detector distance with "highest intensity"?
FindOptDistance  = False

# render optical elements and rays, and how many rays to render?
RayGraph = True
maxRaysToRender = 200

# Draw Airy spot and Fourier-limited duration in the following plots?
DrawAiryAndFourier = True

# produce an interactive spot diagram without color coding the spots?
SpotDiagram = False
# produce an interactive spot diagram with ray delays color coded?
DelaySpotDiagram = True
# produce an interactive spot diagram with ray intensities color coded?
IntensitySpotDiagram = False
# produce an interactive spot diagram with ray incidence angles color
↪  coded?
IncidenceSpotDiagram = False
```

```
# produce an interactive spot diagram with delays in 3rd dimension?
DelayGraph = False
# produce an interactive spot diagram with delays in 3rd dimension and ray
↪  intensities color coded?
IntensityGraph = True
# produce an interactive spot diagram with delays in 3rd dimension and ray
↪  incidence angles color coded?
IncidenceGraph = False

# produce a plot of the ray delays at the detector projected onto the
↪  mirror surface?
DelayMirrorProjection = True
# produce a plot of the ray intensities at the detector projected onto the
↪  mirror surface?
IntensityMirrorProjection = False
# produce a plot of the ray incidence angles at the detector projected
↪  onto the mirror surface?
IncidenceMirrorProjection = False
```

- Now we start setting up the optical configuration using the function `OEPlacement` to automatically create the `OpticalChain`. First, we define the source properties, with a finite divergence, i.e. representing a point source:

```
"""double twisted toroidal in f-d-f config """
#% source properties
Divergence = 20e-3  # half-angle in rad
Wavelength = 50e-6  # 50 nm, only for Airy spot size shown in some plots
DeltaFT = 0.5       # 500 as, only appears as a reference in some plots
NumberRays = 1000   # launch 1000 rays in the beginning
```

- Then we define a toroidal mirror (cp. section 2.3.2): first its support, an $80\times30\text{mm}^2$ rectangle, then the mirror itself, getting the torus radii for the desired focal length and design incidence angle from the helper function `ReturnOptimalToroidalRadii`:

```
#% mirrors
Support = msupp.SupportRectangle(80,30)
Focal = 250
ToroidalIncidence = 78.5 #in deg
OptimalMajorRadius, OptimalMinorRadius =
↪  mmirror.ReturnOptimalToroidalRadii(Focal, ToroidalIncidence)
ToroidalMirror = mmirror.MirrorToroidal(OptimalMajorRadius,
↪  OptimalMinorRadius, Support)
```

- Then we create the `OpticsList` containing twice the same mirror object, the `IncidenceAngleList` with the beam incidence angles on each of the two mirrors, the `IncidencePlaneAngleList` such that the second mirror flips the incidence plane by 90°, and finally the `DistanceList` fixing the distance from the source to

the first mirror and the distance from the first to the second mirror both to the mirrors' focal distance:

```
#% creating the optical chain
OpticsList = [ToroidalMirror, ToroidalMirror]

IncidenceAngleList = [ToroidalIncidence,-ToroidalIncidence] #in deg
IncidencePlaneAngleList = [0,90]

DistanceList = [1*Focal,1*Focal]

OpticalChain = mp.OEPlacement(Divergence,  NumberRays, OpticsList,
↪  DistanceList, IncidenceAngleList, IncidencePlaneAngleList)
```

- Now we are in principle ready to launch the ray tracing, but before we will define information for the dector, which we will let ART create automatically once the ray tracing has been done:

```
#% detector
ReflectionNumber = -1        # analyse ray bundle after last optical
↪  element
ManualDetector = False       # do not place the detector manually, i.e.
↪  let ART do it automatically
DistanceDetector = Focal  # first set the detector at this distance from
↪  the last optical element
AutoDetectorDistance = True  # but then search for the optimal detector
↪  distance and shift the detector there
```

- Now we execute the code containted in **DoART.py**, which executes the ray tracing calculation, sets up the detector and performs the basic analyses. This block can be looped over in order to simulate variations of the optical setup:

```
#%%#########################################################################
""" Do THE ACTUAL RAYTRACING calculations """
#############################################################################
exec( open("ART/DoART.py").read() )
```

- Finally we execute the code containted in **DoPLOTS.py**, which generates the built-in plots selected by the boolean switches in the beginning of the launch script:

```
#%%#########################################################################
"""  Go through the BUILT-IN PLOT OPTIONS """
#############################################################################
exec( open("ART/DoPLOTS.py").read() )
```

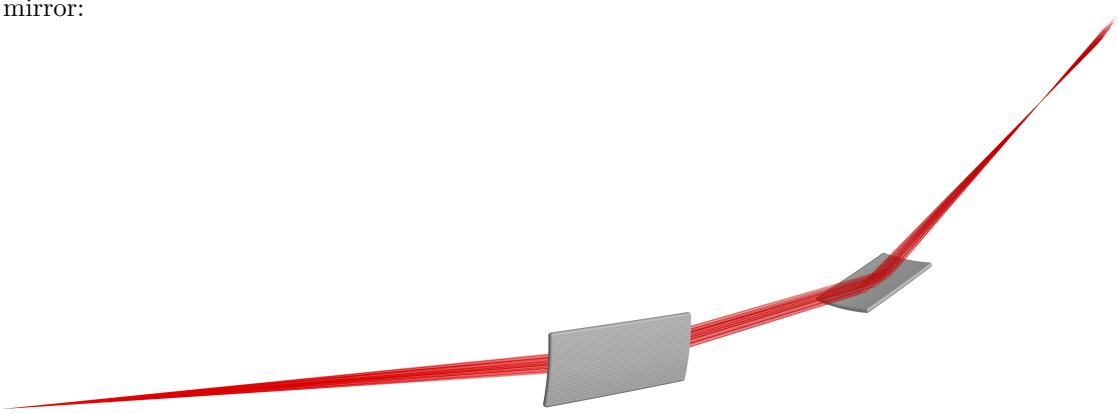
Launching this script in an IPython console produces the following output as well as the plots shown in section 4.1:

```
-----------------------------------------------------------------------------------------------
   ___ __  __                                    __   ___                 ______              _
  / _ |/ /_/ /____  ___ ___ _______  ___  ___/ /  / _ \___ ___ __  /_  __/______ _____(_)__  ___ _
 / __ / __/ __/ _ \(_-</ -_) __/ _ \/ _ \/ _  /  / , _/ _ `/ // /   / / / __/ _ `/ __/ / _ \/ _ `/
/_/ |_\__/\__/\___/___/\__/\__/\___/_//_/\_,_/  /_/|_|\_,_/\_, /   /_/ /_/  \_,_/\__/_/_//_/\_, /
                                                          /___/                            /___/

The optical setup has an energy transmission of 100.0 %.
At detector distance of 250.000 mm we get:
Spatial std :  143.031  mircon and min-max: 542.830 micron
Temporal std :  8.278e+00 fs and min-max :  4.618e+01 fs

Searching optimal detector position within [ 194.000 , 306.000 ] mm...

The optimal detector distance is 249.037 mm with
spatial std of 142.630 micron and temporal std of 8.247e+00 fs.
```

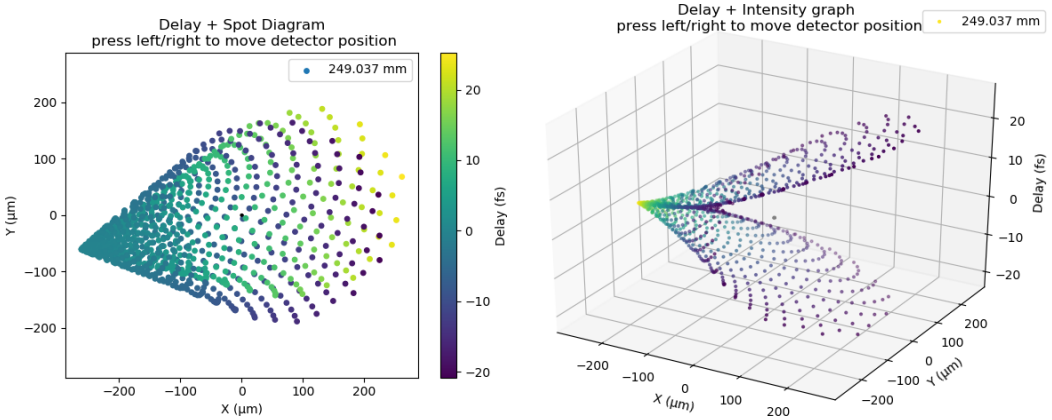
Examples

4.1 Two (twisted) toroidal mirrors

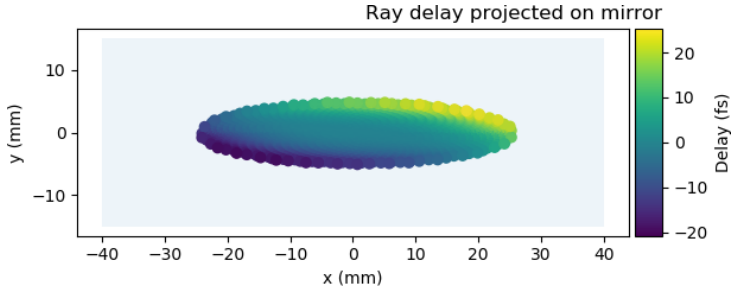
The first example is that simulated by the launch script detailed in the preceding section. The rendered image lets us verify that the simulated optical setup is indeed what we had in mind, and we can, qualitatively, see that as expected the beam is getting collimated by the first mirror (albeit with aberrations), and later refocused at about a the focal distance from the second mirror:



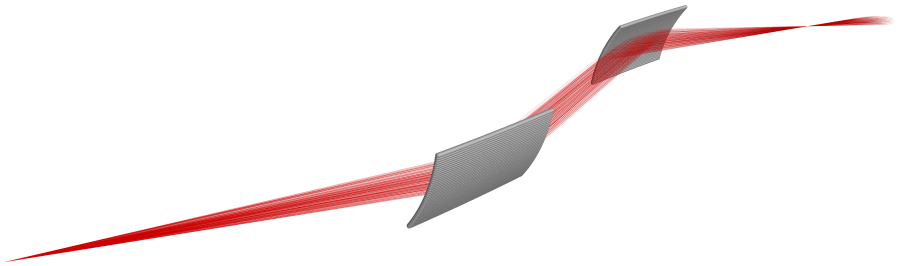
The spot diagrams obtained at the optimised detector distance show a pronounced coma aberration which leads to the temporal spread of 8.3 fs rms and a spatial spread of 140 µm, much larger than the Airy disk size which is nearly invisible in the plots:

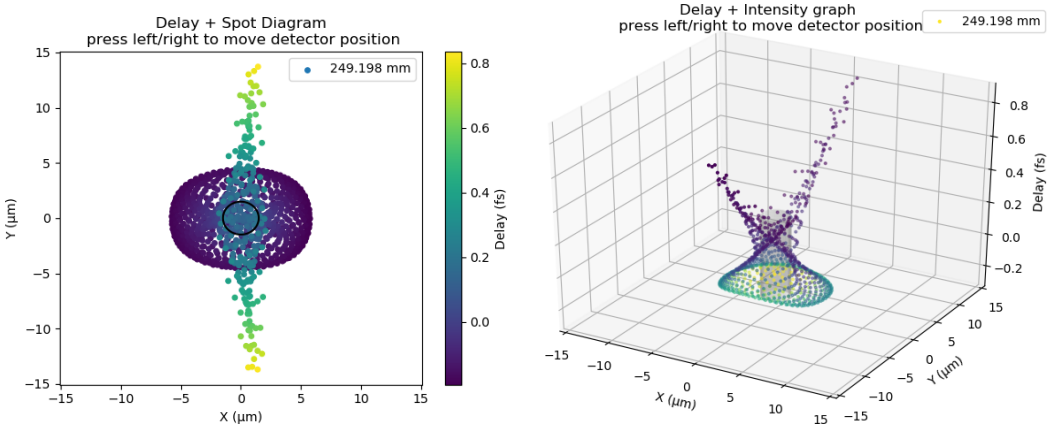


Projecting the ray delays onto the surface of the second mirror yields this figure, showing that (obviously) clipping the beam to a lower divergence would rapidly reduce the temporal spread in focus:

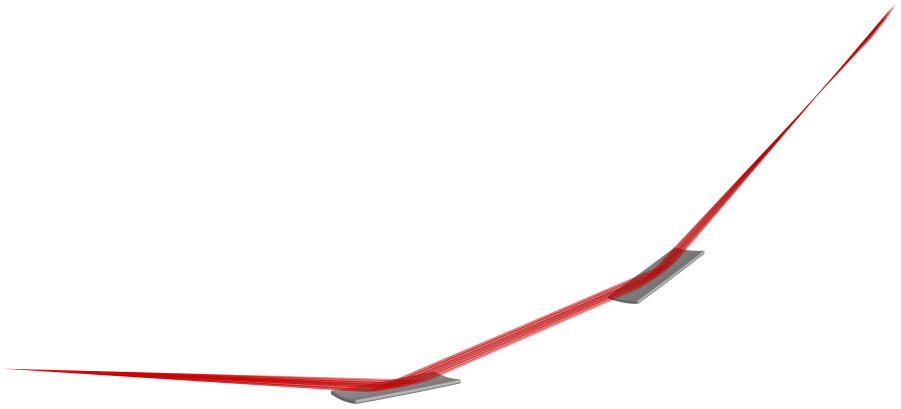


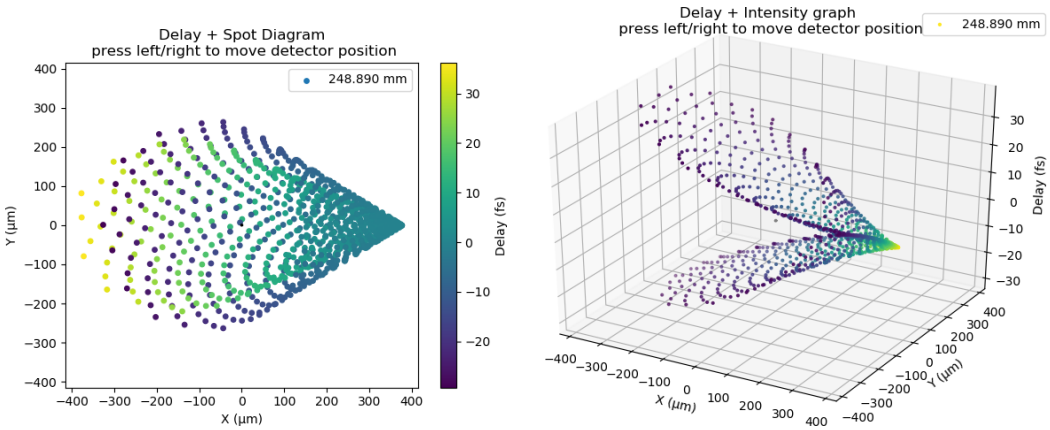
Apparently it is not a good idea to twist that second mirror. If instead we leave the plane of incidence unchanged but reflect in the opposite direction, which is achieved by setting `IncidenceAngleList` = [`ToroidalIncidence`, `-ToroidalIncidence`]
and `IncidencePlaneAngleList` = [0, 0], the coma is compensated and we get a much improved spatial and temporal spread of 4.6 µm rms and 220 as rms at the optimal focus with only higher-order aberrations left:





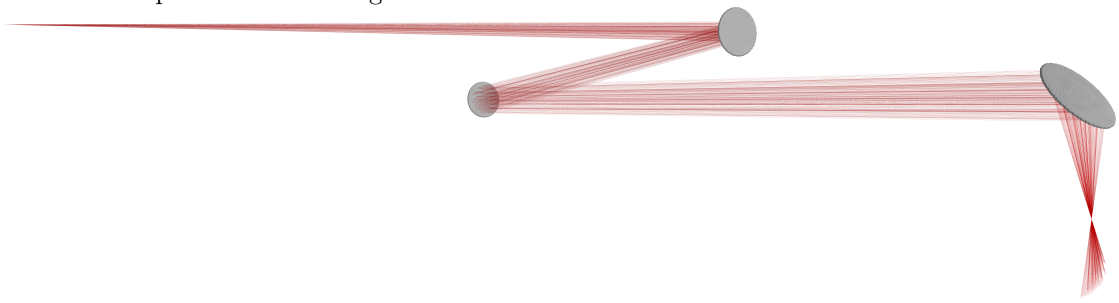
If on the other hand, we let the second mirror reflect in the same direction as the first, i.e. `IncidenceAngleList` = [`ToroidalIncidence`, `ToroidalIncidence`]
and `IncidencePlaneAngleList` = [0, 0], the coma is not compensated but adds up and we get again rather high spatial and temporal spreads of 200 µm rms and 12 fs rms at the optimal focus:





4.2 A collimating telescope followed by an off-axis parabola

As a second example, we will treat a telescope made from an f=-750 mm convex followed by an f=1250 mm concave spherical mirror, which collimates a beam (for example coming from a hollow-core fiber used for post-compression of a visible laser to few-cycle duration), followed by a 90°-off-axis parabola refocusing this beam:

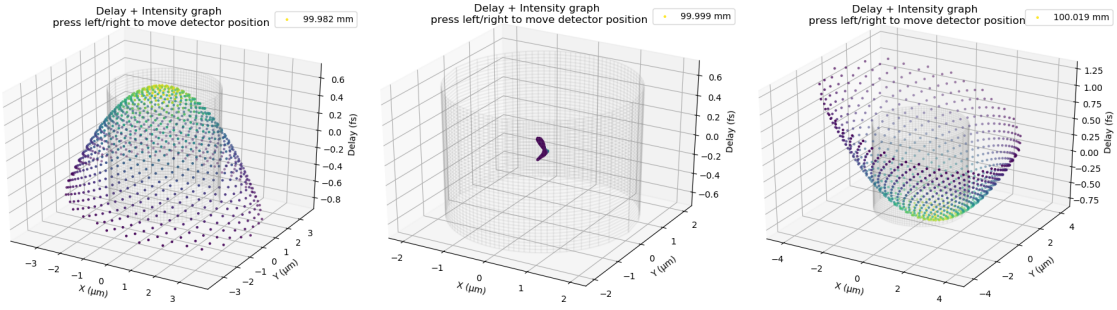


The optimal setup is created by the following section in a launch script:

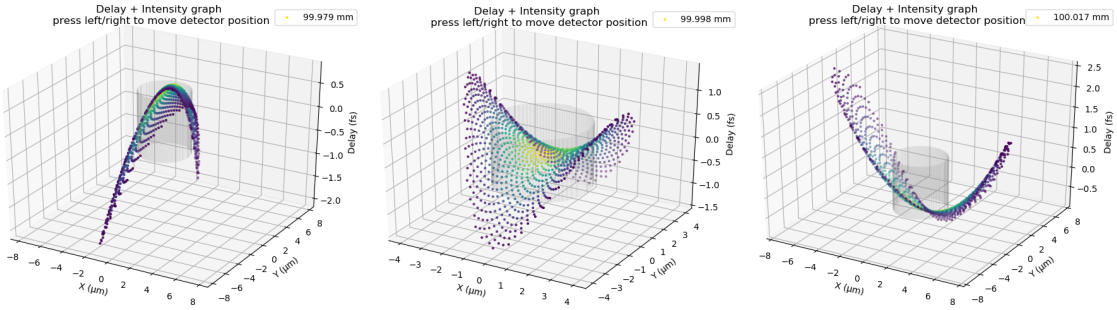
```
"""collimating telescope + off-axis parabola"""
#% source properties
Divergence = 2.2e-3 # half-angle in rad
```

```
Wavelength = 780e-6 # 780 nm, typical central WL for postcompressed Ti:Sa laser
DeltaFT = 1.3       # half-period of 780-nm wave (here we care for
↪  spatio-temporal distortions on the optical-cycle rather than envelope
↪  scale)
NumberRays = 1000   # launch 1000 rays in the beginning

#% mirrors
MirrorCX = mmirror.MirrorSpherical(-1500, msupp.SupportRound(25))
MirrorCC = mmirror.MirrorSpherical(2500, msupp.SupportRound(25))

offAxisAngle = 90     # in deg
SemiLatusRectum = 100 # in mm, SemiLatusRectum =  2*focal length of the "mother
↪  parabola"
Parabola = mmirror.MirrorParabolic(SemiLatusRectum, offAxisAngle,
↪  msupp.SupportRound(25) )

#% creating the optical chain
OpticsList = [MirrorCX, MirrorCC, Parabola]

DistanceList = [5000, 598, 1000]    # in mm
IncidenceAngleList = [5, 3.4, 0.0]   # in degrees, for the parabola this angle
↪  is relative to the mother parabola symmetry axis

OpticalChain = mp.OEPlacement(Divergence,  NumberRays, OpticsList,
↪  DistanceList, IncidenceAngleList)

#% detector
ReflectionNumber = -1 # analyse ray bundle after last optical element

ManualDetector = False # do not place the detector manually, i.e. let ART do it
↪  automatically
DistanceDetector = SemiLatusRectum/(1+np.cos(offAxisAngle/180*np.pi))  # first
↪  set the detector at this distance from the last optical element
AutoDetectorDistance = False  # but then search for the optimal detector
↪  distance and shift the detector there
```

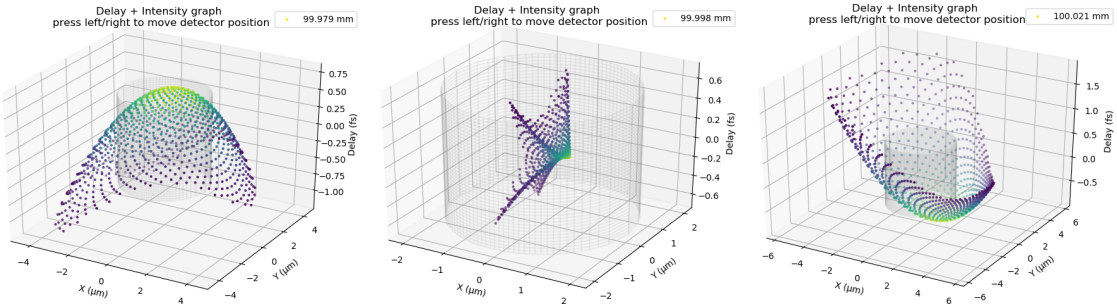
With this configuration, notably the optimised incidence angles of `IncidenceAngleList` = [5, 3.4, 0], these results around the focus are obtained:



The delay-resolved spot diagram is concentrated well within the Airy disk and sub-optical-cycle time scale and we can thus consider this as a practically perfect aberration-free focus. Note that for an off-axis parabola, the incidence angle is measured with respect to the symmetry axis of the mother parabola, i.e. 0° corresponds to the "perfect" alignment onto the focusing parabola. The couple of incidence angles on the two telescope mirrors are optimized such that the astigmatism introduced by the first (as any spherical mirror at non-zero incidence angle) is compensated by the second. This beomes clear when they are set to the same incidence angle instead, `IncidenceAngleList` = [5, 5, 0]: the results shows clearly apparent astigmatism leading to a spatial spread beyond the Airy disk size and a significant temporal spread compared to the optical cycle scale:



One may try to compensate this astigmatism introduced by the telescope by a slight misalignment of the incidence angle on the parabola by 0.01°, e.g. `IncidenceAngleList` = [5, 5, 0.01]. This indeed compensates the astigmatism, but introduces a weak coma aberration:



At the best focus, the delay-resolved spot diagram however still remains within the Airy disk and sub-optical-cycle time scale, so this coma aberration is most likely negligible. A full wave-optics calculation would be required to quantify the difference between this case and the optimized `IncidenceAngleList` = [5, 3.4, 0] shown above.