WISER wavefront propagation simulation code: Advances and Applications

M. Manfredda*[a], A. Hafner[b], S. Gerusina[a], N. Mahne[c], A. Simoncig[a] , M. Zangrando[a,c] and L. Raimondi[a]

[a]Elettra – Sincrotrone Trieste – S.C.p.A., SS 14, km 163.5, 34149 Basovizza (TS), Italy;
[b]CERIC-ERIC, SS 14, km 163.5, 34149 Basovizza (TS), Italy;
[c]CNR-IOM – Istituto Officina dei Materiali, SS 14, km 163.5, 34149 Basovizza (TS), Italy

ABSTRACT

In this work, we report the advances in the development of WISER. WISER is a wave optics-based simulation library targeted at the simulation of the focusing performance of grazing X-Ray optics systems, which accounts for the metrological data of figure error and roughness power spectral density. First presented in 2016 (phase-I), WISER inherits and expands the mathematical concepts of its ancestor, WISE, originally conceived for X-Ray telescopes and then applied to free electron lasers. Thanks to its flexible framework, WISER easily allowed to simulate multi-element systems, as synchrotron and free electron laser beamlines. In phase-II (2020), WISER is further improved and it is delivered as fully integrable with OASYS, the graphical canvas gathering the mostly used X-Ray computation tools. In the following, we will illustrate the architecture of the library and present some examples of its applications.

Keywords: optics, simulation, x-ray, oasys, wiser, metrology

1. INTRODUCTION

The impact of in-house developed codes is of increasing importance in today's research, both for the capability of addressing specific problems and for the offered automation perspectives. However, often scientific codes, despite being a real mine of knowledge, are of scarce use for the community. Poor design, minimal generalization and cryptic shortcuts taken during the development process often limit their usability to few people, or to limited specific configurations. In addition, in most cases, a code is simply not thought so to be used by people other than the authors. Consequently, the gap-to-fill to make a collection of scripts evolving towards an organic ecosystem is often enormous, and it requires a combination of deep understanding of the physics and use of well-established programming paradigms. We decided to develop WISER to overcome these limitations: we started motivated by the previous outstanding experience of WISE [1,2], existing in a script-like form and already used to address challenging questions, with the aim of creating a totally new product, closer to a beamline design software. WISER's main feature is modularity, made possible by the extensive use of object oriented programming and of design patterns. Modularity, in turn, enables a better maintainability and progressive extensibility: this means that, even with a limited set of initial features (for instance of optical elements), WISER can easily answer to user community needs arising in the future.

The reasons for developing WISER have been essentially three. First, the relevance of predicting the focusing performance of a hard X-ray mirror. Such a capability, which is at the basis of design, manufacturing and polishing of the optics of telescopes, synchrotrons and free electron lasers, is not reached by any other freely available computing tools (or not with the same accuracy); second, the need of a versatile set of "building blocks", flexible enough to tackle potentially new scientific questions, and structured enough to be reusable; third, the recent development of OASYS[3,4], a working canvas whose aim is to gather all the most relevant software simulating X-ray beamlines and make them interact together. This is done by providing a simple modular graphic interface, which greatly propels the fruition and spread of the software and libraries contained in OASYS.

WISER is powered by three project libraries written in Python: *LibWiser*, which implements WISER entities (canvas, optical element, propagation engine, and more); *WofryWiser*, which adapts to WOFRY; and *OasysWiser*, which is the final implementation into OASYS. *LibWiser* enables to use WISER as standalone library, which requires extra-coding in Python, *WofryWiser* and *OasysWiser* enable to use WISER into OASYS, taking advantage of OASYS GUI. The lower

Advances in Computational Methods for X-Ray Optics V, edited by Oleg Chubar, Kawal Sawhney,
Proc. of SPIE Vol. 11493, 114930B · © The Authors. Published under a Creative Commons
Attribution CC-BY 3.0 License · doi: 10.1117/12.2568574

layer (*LibWiser)* provides the total control of WISER, whereas the hi-level layer (*OasysWiser*) allows the use of the most common features.

The source codes are available online:[5,6,7]

After a brief summary of the underlying physical principles, we will briefly describe the outline of *LibWiser*, and show its architecture. The description of *WofryWiser* and *OasysWiser* is not included, although some screenshot of OASYS implementation are shown.

2. PHYSICS

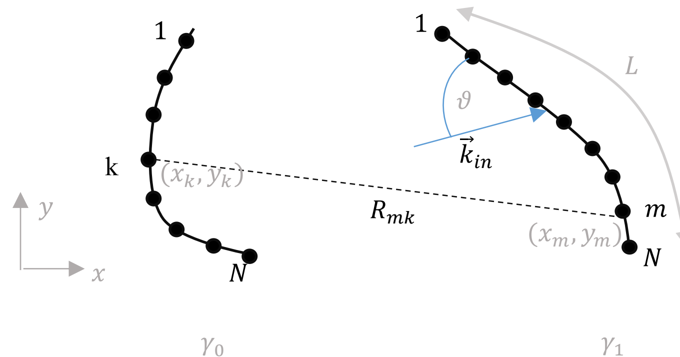
2.1 The Physical Basis

WISER is a wave propagation software, which enables to accurately simulate the electromagnetic field along a train of mirrors, accounting for the surface error metrology. WISER's goal is precision rather than performance. The propagation of light is modeled with fully coherent monochromatic light within Kirchoff's scalar diffraction theory, and the explicit evaluation of Huygens-Fresnel integral is carried out by recursive summation of complex exponentials (associated to spherical waves). Contrarily to many commonly used approaches in computational optics, FFT algorithm is not used. The attention devoted to metrology and the explicit summation of spherical waves are the main differences of WISER with respect to other well established – and more capillary – wavefront propagation software, such as SRW [8]. The surface error is modeled following the traditional classification of figure error and roughness: The first one is fed as a residual height profile with respect to the ideal curvature and the second one is fed as the Power Spectral Density of the surface height profile. Waive the use of the FFT makes computations longer, but it avoids the *a*-priori approximations which are necessary to make FFT-based propagators work when tilted planes are involved.

The integration is performed along the tangential (i.e. longitudinal) direction only. Such a choice is a consequence of the fact that, at the X-ray wavelengths, the mirrors are typically operated at grazing incidence, and sagittal scattering is negligible [1]. Each Optical Element (OE) is represented over a 1D-support, which is described in a Cartesian space by a couple of arrays (x_m, y_m). Ultimately, real-world optical elements are represented by their longitudinal section: ellipsoidal or Kirkpatrick-Baez mirror are represented by elliptic arc sections, paraboloid mirrors by parabolic arc sections, plane mirrors and detectors by line segment, and so on.

The theoretical basis of WISER can be found in [1], where we find the expression of the field propagated from the optical starting element γ_0 to the arrival element γ_1, which is written as:

$$\boldsymbol{E}(x'_m, y'_m) = \frac{L \sin\vartheta}{\sqrt{\lambda}} \sum_{k=1}^{N} \frac{\boldsymbol{E}(x_k, y_k)}{R_{mk}} exp(ikR_{mk})$$



where λ is the wavelength, $k = 2\pi/\lambda$, L is the total length of the arrival element γ_1, ϑ is the grazing angle comprised between the incident wave-vector $\vec{k}_{in}$ and the tangent at the mid-point if γ_1, R_{mk} the distance between the points and $\boldsymbol{E}$ is the complex field. Both elements sampled with *N* points.

3. ARCHITECTURE

WISER is written making extensive use of object oriented programming and design patterns with a manifold aims in mind: *a)* creating a flexible, modular code that can be progressively extended; *b)* giving the user a smooth, intuitive programming experience, also taking advantage of the auto-completion features of the modern IDE (Spyder, pyCharm); *c)* making the connection to OASYS simple and more natural.

The foundations of WISER are:

- The *BeamlineElements* (Foundation.py) class
- The *OpticalElement* (Foundation.py) class
- The member of Optics class (Optics.py), which are often referred to as *CoreOptics*.

An inclusion relation holds between these entities: a *BeamlineElements* objects contain a collection of *OpticalElement* objects, and every *OpticalElement* object contains a *CoreOptics* object. A sketch of WISER ecosystem is shown in Figure 1.

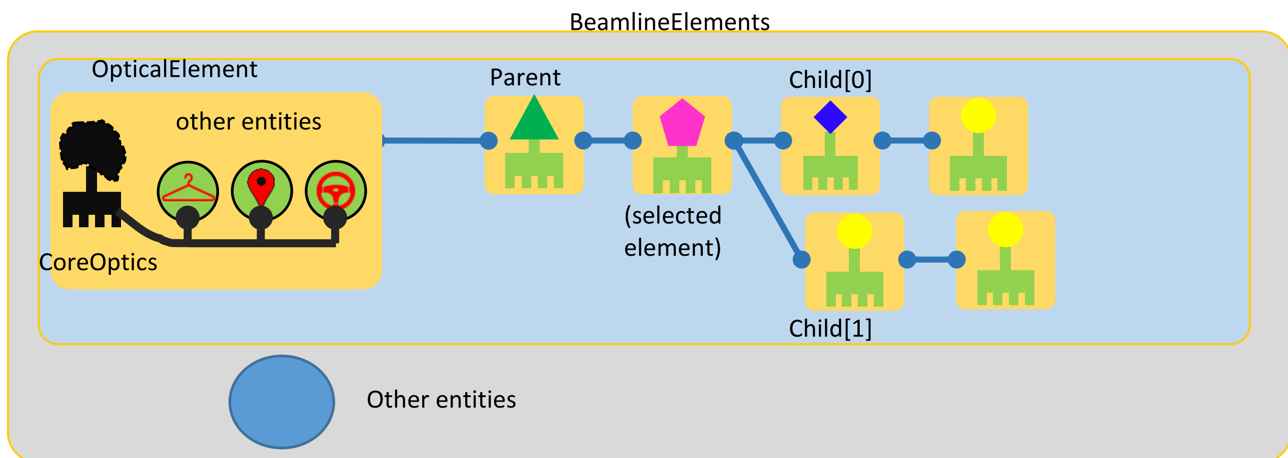


Figure 1. Layout of the Optics object class. For each class, the representative members are shown only.

3.1 BeamlineElements class

BeamlineElements represents a beamline. Its main roles are:

1. To collect the *OpticalElement* objects in a tree structure, thus providing some of the most common functionalities (indexing, appending, deleting, getting sequence of items, etc)
2. To act as *mediator* between the different optical element, handling:
    a. the positioning of the *OpticalElement* objects on the canvas (see)
    b. the field propagation.

As side note, the fact that *BeamlineElements* behaves like a tree structure, thus accepting nodes and forks, has been originally conceived for handling with devices such as split-and-delay line or spectrometers. However, presently such a functionality is not fully implemented yet.

3.2 OpticalElement class

OpticalElement is as an empty container whose tasks are:

1. Storing the positioning command in the *PositioningDirective* object
2. Storing the information about siblings (parent, child/children), the computed data and the computation settings
3. Storing the complex CoreOptics subsystem and implementing a *facade* pattern, to provide just the functionalities that are really needed in the *BeamlineElement* ecosystem.

A diagram of the *OpticalElement* class is shown in Figure 2.

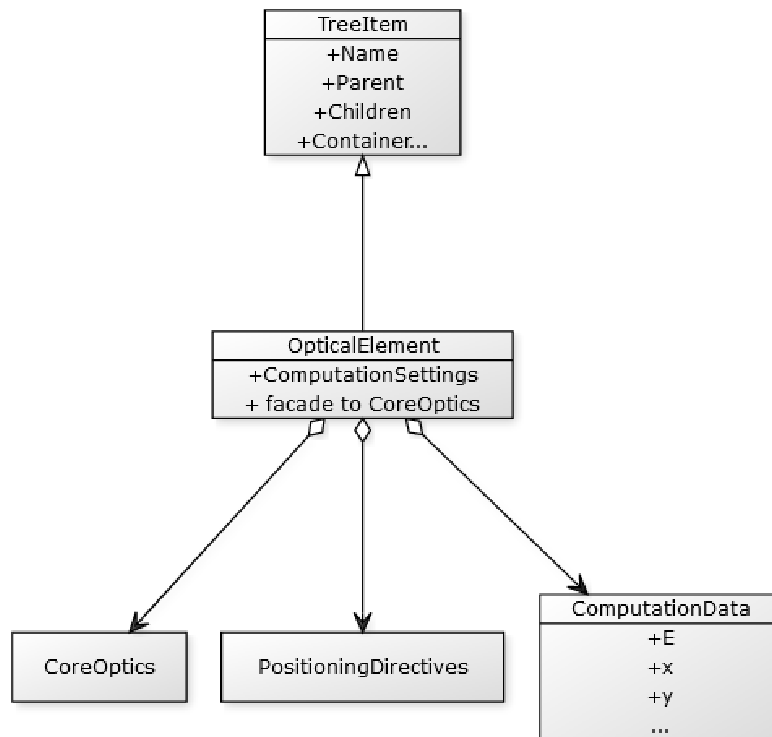


Figure 2 Diagram of the *OpticalElement* class.

3.3 Optics and CoreOptics

The *Optics* classes provide the true implementation of the real-world optical elements. In particular, the ultimate implementation is carried out by a subset of the *Optics* classes, which is informally called *CoreOptics*.

Eventually, the instances of a *CoreOptics* object are:

1. containing the specific code, which defines object behavior, namely how the coordinates within the canvas, the incident and the emerging vector are computed;
2. storing the specific parameters of the object to be created (e.g. the wavelength for a light source, or the incidence angle for a mirror, etc.);

3. storing the metrology information (figure error and roughness);
4. storing low-level computation settings;
5. providing, among the others, the following milestone attributes:
    a. *GetXY* (to get the *x,y* coordinates of the *CoreOptics* object),
    b. *EvalField*(*x,y, …)* to propagate the field from the present optical element to the coordinates *(x,y),*
    c. *RayInNominal*, *RayOutNominal*, i.e. the vectors (represented in WISER class) associated to incident and emerging scattering wave vector,

A layout of the class hierarchy is shown in Figure 3. The classes *Optics, OpticsNumerical, OpticsAnalytical* and *Mirror* acts as abstract blueprint for the "concrete" objects *SourceAnalytical*, *MirrorElliptic, MirrorSpherical, MirrorPlane, Slit, Detector*, etc. For these reason, the formers are marked as "<<abstract>>", understanding that they can not be initialized as standalone objects. The complexity of the *CoreOptics* classes depend on the optical element itself. For instance, the set of additional formulas required for handling the elliptic mirror are more involved than the ones required for plane mirrors. However, the autonomy of *CoreOptics* objects is rather limited, as the parent classes already implement the most challenging task: the positioning and rotation algorithms, and the propagation. For this reason, adding a new *CoreOptics* object is a straightforward operation end requires little or no maintenance to the main code.

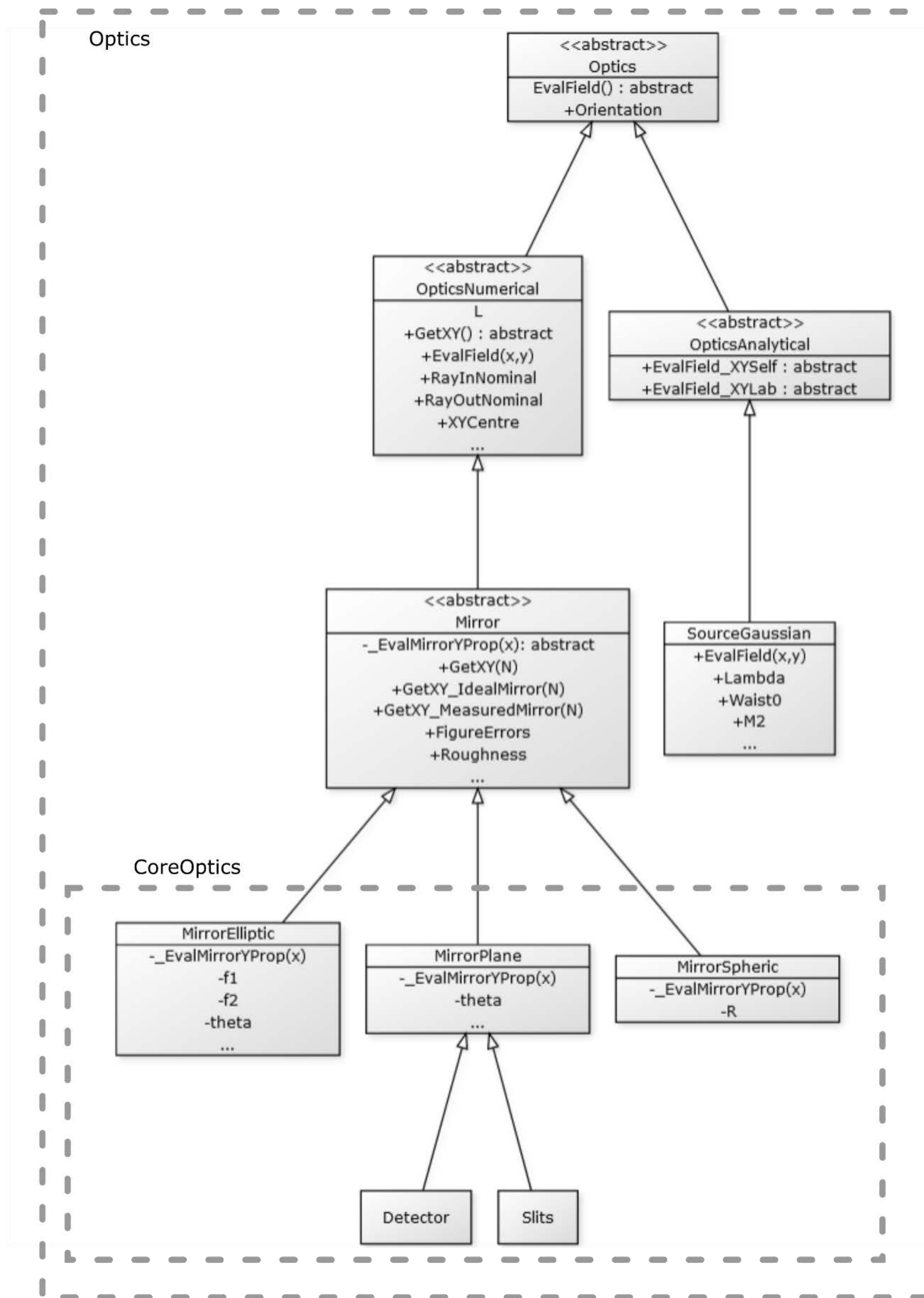


Figure 3 Diagram of the Optics object class. For each class, only the representative members are show.

4. EXAMPLE

A more thorough documentation of WISER, with examples, will be available online on the GIT repository [5]. Here we are just interested in presenting the general appearance of a client code that uses *LibWiser*. We will consider a simple example of a beamline composed by a FEL-like source, a plane mirror, a focusing ellipsoidal mirror and a detector. Parameters can be found in the code.

```
from LibWiser.must import *
from LibWiser.WiserImport import *

#---SOURCE
#------------------------------------------------------------
Lambda = 20e-9 # (m)
Waist0 = 150e-6# (m)
DeltaSource = 0
s = OpticalElement(
                    Name = 'Source',
                    IsSource = True,
                    CoreOpticsElement = Optics.SourceGaussian(
                                  Lambda = Lambda,
                                  Waist0 = Waist0,
                                  Orientation = Optics.OPTICS_ORIENTATION.HORIZONTAL),
                    PositioningDirectives = Foundation.PositioningDirectives(
                                  ReferTo = 'absolute',
                                  XYCentre = [0,0],
                                  Angle = 0)
                    )

#---Plane Mirror
#-------------------------------------------------------------
L = 0.4 # (m)
GrazingAngleDeg = 2 #deg
pm = OpticalElement(
                           Name = 'PM',
                           CoreOpticsElement = Optics.MirrorPlane(
                                  L = L,
                                  AngleGrazing = np.deg2rad(GrazingAngleDeg),
                                  Orientation = Optics.OPTICS_ORIENTATION.HORIZONTAL),
                           PositioningDirectives = Foundation.PositioningDirectives(
                                  ReferTo = 'source',
                                  PlaceWhat = 'centre',
                                  PlaceWhere = 'centre',
                                  Distance = 41.4427)
                                                )
```

```
#---Elliptic Mirror (or K-B)
#------------------------------------------------------------
f1 = 70
f2 = 1
kb = OpticalElement(
                          Name = 'kb',
                          CoreOpticsElement = Optics.MirrorElliptic(
                                 L = 0.4,
                                 f1 = 70,
                                 f2 = 1,
                                 Alpha = np.deg2rad(2),
                                 Orientation = Optics.OPTICS_ORIENTATION.HORIZONTAL),
                          PositioningDirectives = Foundation.PositioningDirectives(
                                 PlaceWhat = 'upstream focus',
                                 PlaceWhere = 'centre',
                                 ReferTo = 'source',
                                 Distance = 0
                                              ))
#---Detector
#------------------------------------------------------------
Detector = OpticalElement(
                          Name = 'Det',
                          CoreOpticsElement = Optics.Detector(
                                 L = DetectorSize,
                                 AngleGrazing = np.deg2rad(90),
                                 Orientation = Optics.OPTICS_ORIENTATION.HORIZONTAL),
                          PositioningDirectives = Foundation.PositioningDirectives(
                                 PlaceWhat = 'centre',
                                 PlaceWhere = 'centre',
                                 ReferTo = 'upstream',
                                 Distance = f2)
                                              )
```

Units are expessed in S.I..and angles are in radians. After instantiating the *CoreOptics* objects `s, pm, kb,Detector`, the *BeamlineElements* object can be instantiated and populated as well:

```
# Initialize and populate the BeamlineElements object
#------------------------------------------------------------
Beamline = Foundation.BeamlineElements()
Beamline.Append(s)
Beamline.Append(pm)
Beamline.Append(kb)
Beamline.Append(Detector)
Beamline.RefreshPositions()
```

The command `Beamline.RefreshPositions()` gives the instruction to traverse the chain of *OpticaElement* objects and physically assign the *XYCentre* attribute to each item. After the beamline is refreshed, it is possible to check the internal representation of WISER,by means of:

```
print(Beamline)

>[] ---- *[Source]*----[Plane Mirror]          DeltaZ=0.00 m, Z=0.00 m
>[Source] ---- *[Plane Mirror]*----[kb]        DeltaZ=41.44 m, Z=41.44 m
>[Plane Mirror] ---- *[kb]*----[Det(H)]        DeltaZ=28.56 m, Z=70.00 m
```

```
>[kb] ---- *[Det(H)]*----[]                DeltaZ=1.00 m, Z=71.00 m
```

The command iterates across the optical elements of `Beamline` and for each optical element displays its name (marked by double "*"), the name of its parent (to the left), the name of its first child (to the right). "DeltaZ" is the distance from the previous optical element, and "Z" is the cumulative distance from the source.

Another option for checking the internal representation of WISER is the `Beamline.Paint()` command, which produces a simplified sketch of the optical elements in the WISER canvas. Such a tool is just intended for debug purpose, and shall not be used for producing publication level graphics.

```
Beamline.Paint()
```

Figure 4 Example of the output of the *.Paint()* command.

Finally, the field can be computed onto all the optical elements by means of the `ComputeField` command.

```
Beamline.ComputeField()
```

For each optical element, the results are stored in

```
kb.ComputationData
```

which has the following attributes:

- Field: the complex field,
- Intensity: the squared modulus of the field,
- Lambda: the wavelength used,
- Hew: the Half Energy Width computed on the intensity
- NSamples: the number of samples used
- Name: a clone of the name of the optical element
- X,Y: the coordinates of the optical element (more precisely, of its *CoreObject* member)
- S: the longitudinal coordinate that runs along the optical element (used in plots).

The intensity at the detector can be displayed by means of:

```
plot(kb.ComputadionData.S, kb.ComputationData.Intensity)
```

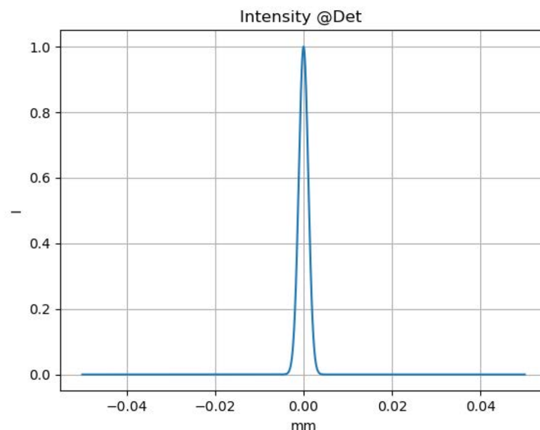


Figure 5 Example of the intensity computed at the nominal focal plane, with ideal mirror curvature.

The following code shows how to add a figure error to the surface profile of the optical element:

```
#h shall be an arbitrarily sized array containing the figure error.
#Different avenues of reading this kind of files, or synthetically generate h
are available. This topic will be tackled in further examples.

kb.CoreOptics.FigureErrorLoad(
                                    h = h,
                                    Step = 2e-3, # 2mm step
                                    AmplitudeScaling = -1) # reverse the profile

kb.CoreOptics.ComputationSettings.UseFigureError = True
Beamline.ComputeFields()
lw.ToolLib.CommonPlots.IntensityAtOpticalElement(kb) #useful helper function
```

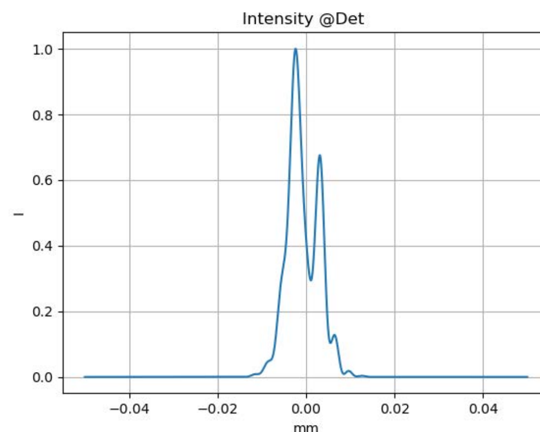


Figure 6 Example of intensity computed at the nominal focal plane, with non-ideal mirror curvature.

In the example shown, the field is computed at the nominal position of the detector. There may be a wide number of reasons, however, for which the true focal plane is shifted from its nominal position. Possible causes can be: the Gaussian nature of the source, the figure error itself, or the combination of the two [9]. WISER has dedicated functions to find the best focal spot by minimizing the spot size while performing a through-focus scan, which is already integrated in OASYS,

as it is shown in Figure 7.. In addition, it can be adapted to more sophisticate investigations which involve parametric scans. This make it an ideal tool for optical system performance analysis [10].

As an example, in Figure [],we display the parametric plot of the spot size (computed as the Half Energy Width) and of the focal shift as function of the wavelength for an ellipsoidal mirror illuminated by a Gaussian source.

Some of these functionalities are directly available into OASYS.

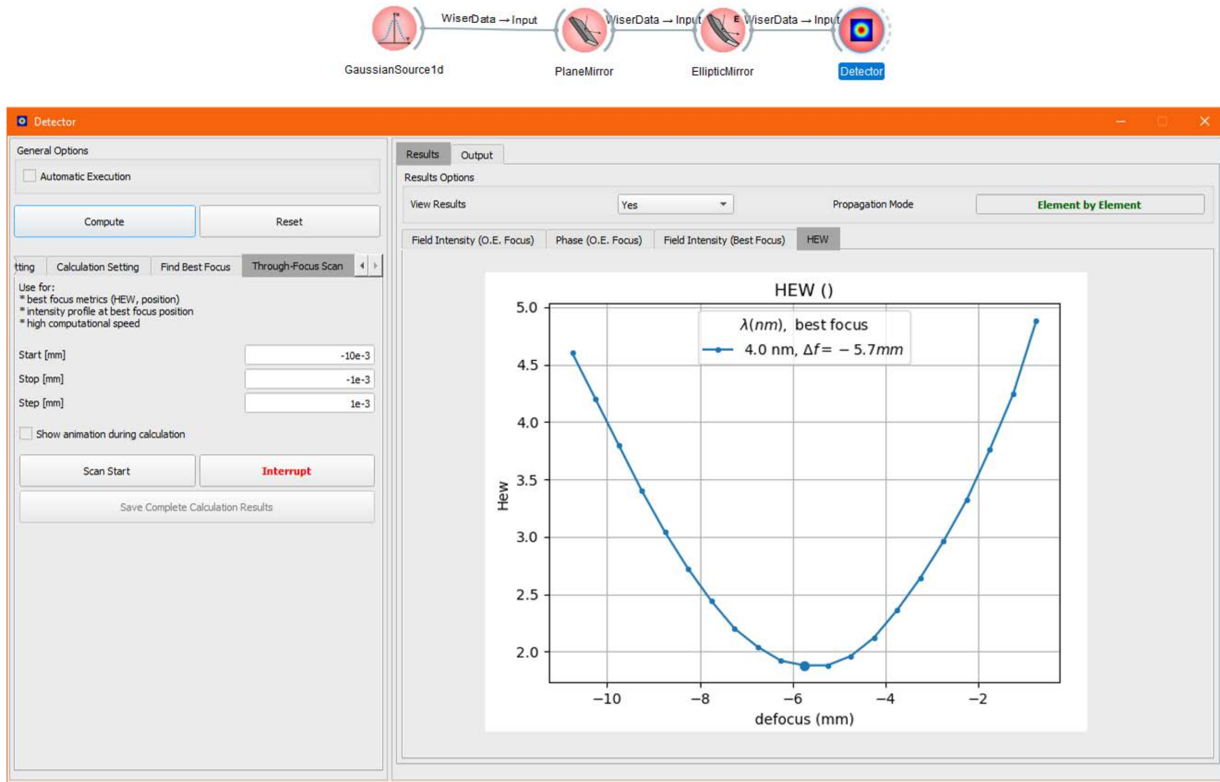


Figure 7 Screenshot of the "detector" widget in OASYS, which enables to find the best focus by minimizing the HEW. In the example considered, the best-focus is 5.7mm upstream with respect to the nominal focus. Simulation is performed at $\lambda = 4nm$. For the other parameters see *Figure 8*.

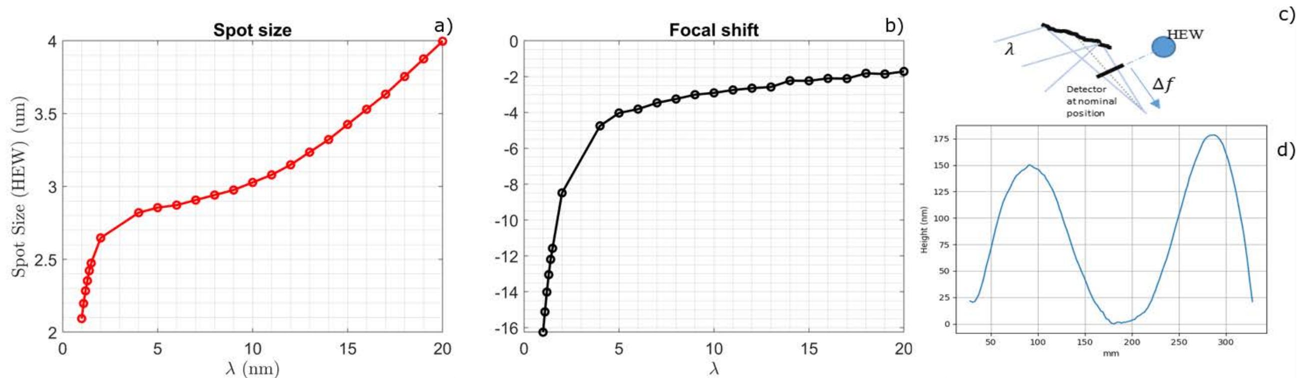


Figure 8 Example of some parametric scan which can be performed with WISER. The parameter is the source wavelength. a) dependence of the spot size as function of the wavelength; b) dependence of the focal shift as function of the wavelength; c) sketch of the conceptual layout; d) figure error used. The computation is performed with a Gaussian source (waist size=150 μm, $M^2 = 1$) illuminating an elliptic mirror ($f_1 = 88.5\ m$, $f_2 = 1.2\ m$, $\vartheta_i = 2°$).

5. CONCLUSIONS

WISER has been released both as stand-alone library and as add-on to OASYS. The development is far for being concluded. New optical elements will be implemented, such as gratings and monochromators, as well as new kind of sources, in order to better match the need of the X-ray optics community. Examples and updates will be soon available on the web [5], [11].

6. CREDITS

We thanks Luca Rebuffi for his dedicate support and advise on software design, Daniele Cocco for believing in WISER potentials and providing useful hints for improvement, and Daniele Spiga for the perseverance and knowledge he put in making the first version of WISE rise.

A.H. and this open access publication have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 823852 (Panosc).

REFERENCES

[1] L. Raimondi and D. Spiga, “Mirrors for X-ray telescopes: Fresnel diffraction-based computation of point spread functions from metrology,” *Astron. Astrophys.*, vol. 573, p. A22, 2015.

[2] L. Raimondi *et al.*, “Microfocusing of the FERMI@Elettra FEL beam with a K-B active optics system: Spot size predictions by application of the WISE code,” *Nucl. Instruments Methods Phys. Res. Sect. A Accel.*

[3] L. Rebuffi and M. Sanchez del Rio, “OASYS (OrAnge SYnchrotron Suite): an open-source graphical environment for x-ray virtual experiments,” p. 28, 2017.

[4] L. Rebuffi and M. Sanchez del Rio, “Interoperability and complementarity of simulation tools for beamline design in the OASYS environment,” p. 8, 2017.

[5] M.Manfredda, A. Hafner, L. Raimondi, “WISER, the wave propagation simulation code, reloaded”, https://github.com/oasys-elettra-kit/WISEr

[6] A. Hafner, M. Manfredda., L. Rebuffi, “OasysWiser”, https://github.com/oasys-elettra-kit/OasysWiser

[7] A. Hafner, M. Manfredda, L. Rebuffi, “WofryWiser”, https://github.com/oasys-elettra-kit/WofryWiser

[8] O. Chubar, P. Elleaume, "Accurate And Efficient Computation Of Synchrotron Radiation In The Near Field Region", proc. of the EPAC98 Conference, 22-26 June 1998, p.1177-1179

[9] M. Manfredda, L. Raimondi, N. Mahne, and M. Zangrando, “Focal shift induced by source displacements and optical figure errors,” *J. Synchrotron Radiat.*, vol. 26, pp. 1503–1513, 2019.

[10] L. Raimondi *et al.*, “Kirkpatrick-Baez active optics system at FERMI: System performance analysis,” 2019. *Spectrometers, Detect. Assoc. Equip.*, vol. 710, pp. 131–138, 2013.

[11] “WISER”, https://elettra.eu/wiser