ToFu Documentation

Release alpha

D. Vezinet, A. Ratnani

December 12, 2014

1 Overview **3**

2 ToFu_Geom **7**
2.1 Getting started with ToFu_Geom . . . 8
2.2 The Tor object class . . . 8
2.3 The LOS object class . . . 12
2.4 The GLOS object class . . . 17
2.5 The Detect and Apert object classes . . . 19
2.6 The GDetect object class . . . 32

3 ToFu_Mesh **41**
3.1 Getting started with ToFu_Mesh . . . 42
3.2 The Mesh1D, Mesh2D and Mesh3D object classes . . . 43
3.3 The BaseFunc1D, BaseFunc2D and BaseFunc3D object classes . . . 46

4 ToFu_MatComp **59**
4.1 Getting started with ToFu_MatComp . . . 59
4.2 Limits to the LOS approximation for the geometry matrix computation . . . 72

5 ToFu_Inv **75**
5.1 Getting started with **ToFu_Mesh** . . . 76
5.2 The Tor object class . . . 76

6 Indices and tables **79**

Contents:

CHAPTER

ONE

OVERVIEW

(This project is not finalised yet, work in progress...)

ToFu, which stands for "TOmography for FUsion" is a python package (with parts in C/C++) providing all necessary tools for tomography diagnostics for the Fusion community, it is particularly relevant for X-ray and bolometer diagnostics on Tokamaks. On of the objectoves is to provide a common tool for tomographic inversions, with both accurate methods and enough flexibility to be easily adapted to any Tokamak and to the specific requirements of each user. The main language (Python) has been chosen for its open-source philosophy, for its object-oriented capacities, and for the good performance / flexibility ratio that it offers. The architecture of the **ToFu** package is intended to be modular to allow again for maximum flexibility and to facilitate customisation and evolutivity from the users.

ToFu: provides in particular, but not only, the main following functionnalities :

- Using the 3D geometry of the diagnostic (positions of detectors and apertures are provided as inputs) to compute quantities of interest (e.g.: the optimal line of sight, the exact etendue..). This is done by the module ToFu_Geom.
- Building of a variable grid size mesh for spatial discretisation of the solution (i.e. emissivity field) on which B-splines of any degree can be added to serve as Local Basis Functions. This is done by the module ToFu_Mesh.
- Computing of the geometry matrix associated to a set of detectors and a set of basis functions, both with a full 3D approach or with a Line Of Sight (LOS) approximation. This is done by the module ToFu_MatComp, which uses both ToFu_Geom and ToFu_Mesh.
- Computing tomographic inversions based on the constructed geometry matrix and Phillips-Tikhonov inversion with a choice of objective functionals (among which first order and second order derivatives or Fisher information, and more to come). This is done by the module ToFu_Inv, which uses the matrix computed by ToFu_MatComp.
- Visualizing, exploring and interpreting the resulting inversions using a built-in Graphic User Interface.

The joint use of a full 3D approach and of regular basis functions (B-splines) allows for advanced functionalities and flexibility,

- Accurate computation of etendue and geometry matrix.
- Exact differential operators (provided sufficient degree of the basis function) instead of discretised operators (this feature and the previous one aim at improving the accuracy of tomographic inversions).
- Accurate description of toroidal-viewing detectors with potentially large viewing cones and for which the LOS approximation cannot be used.
- Making possible 3D inversions (provided the geometrical coverage of the plasma volume is sufficient, for example thanks to toroidal-viewing detectors).
- Enabling proper taking into acccount of anisotropic radiation (for example due to fast electrons due to disruptions).

The ToFu package has built-in mesh and B-spline definitions, however, if used alone, it can only create and handle rectangular r

Rational B-Splines, or NURBS, curves) on which it can also add several different types of regular basis functions. It is a next-gen solution for optimisation of plasma-physics simulation codes. Hence, the final idea is that the same mesh and tools can be used for running CPU-expensive plasma physics simulations and, from their output, to compute the associated simulated measurements on any radiation diagnostics. This synthetic diagnostic approach is aimed at facilitating direct comparisons between simulations and experimental measurements and at providing the community with flexible and cross-compatible tools to fit their needs. Plasma physics codes that are planning on using **Pigasus** in a near future include in particuler **JOREK** (in its **Django** version) and **GISELA** (**CELALIB** in its next version). More information about **Pigasus** (lien), **JOREK** (lien) and **GISELA** can be found on their respective pages.

In order to avoid too much dependency issues, the **ToFu** package resorts to widely used Python libraries like scipy, numpy and matplotlib. Whenever it was possible, the idea was either to use a very common and accessible library or to have built-in methods doing the job. It can be run as a stand-alone on an offline computer (i.e.: on a laptop while travelling), in an online mode (using a central database on the internet) and with or without **Pigasus** (keeping in mind that only rectangular mesh can be created without it).

For faster computation, some modules and/or methods are coded with Cython or Boost.Pyton. It is also intended to be MPI and OpenMP parallelized.

The general architecture is briefly represented in the following figure:

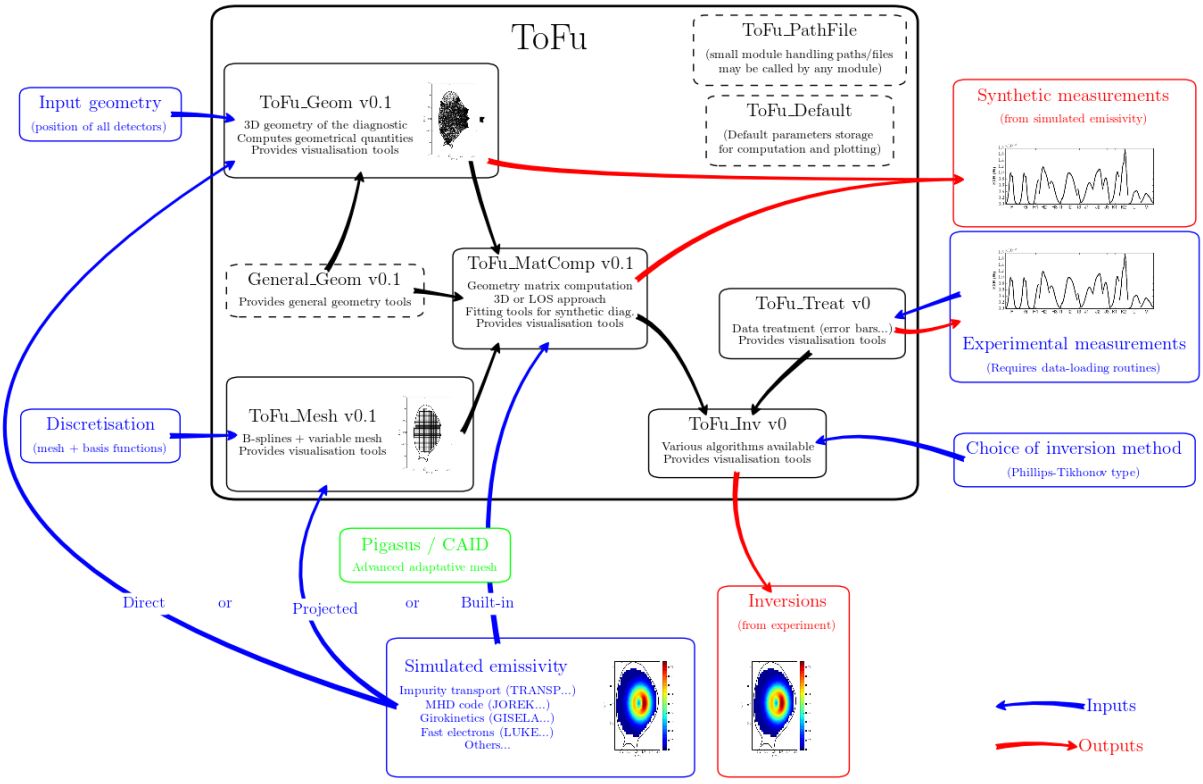

Figure 1.1: Modular architecture of ToFu, with its main modules.

This general overview shows all the **ToFu** modules and their main functionnalities and dependancies. Particularly important are the modules **ToFu_Geom**, **ToFu_Mesh** and **ToFu_MatComp** which provide all necessary tools to pre-calculate the geometry matrix which is a key feature of the two main uses of **ToFu**.

On the one hand, **ToFu** can be used as a synthetic diagnostic since from a simulated emissivity field it can compute

the corresponding synthetic measurements for comparison with experimental measurements. This, as illustrated below, can be done in different ways depending on whether the simualted is used directly as a function, projected on a predefined mesh of the plasma volume, or if the simulated emissivity itself was computed on a mesh using the **Pigasus/CAID** code suite which is directly compatible with **ToFu**. These three possibilities are illustrated in the following figure:

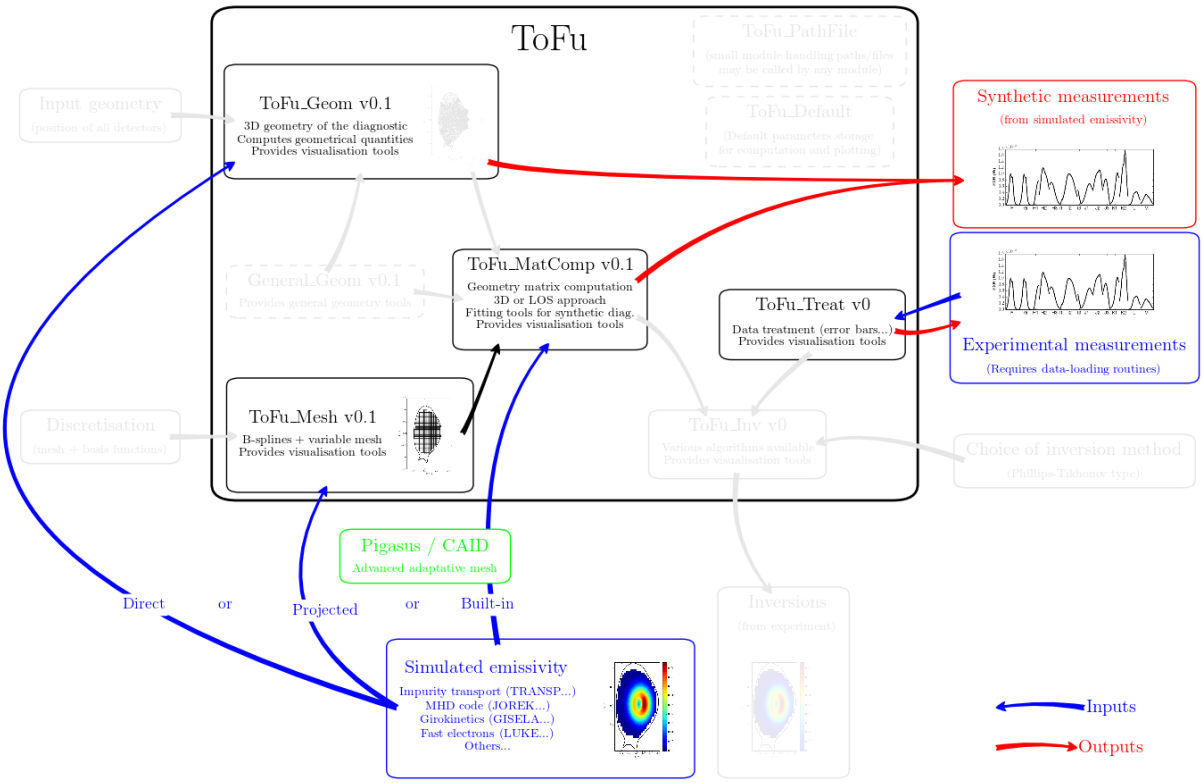

Figure 1.2: Modular architecture of ToFu, with its main modules for synthetic diagnostics.

On the other hand, **ToFu** can be used the other way around : use the experimental measurements to compute a reconstructed experimental emissivity field via a tomographic inversion, for comparisopn with a simulated emissivity field or simply for getting an idea of what the emissivity field looks like, which is illustrated in the following figure:

The following will go into further details regarding each module.

ToDo list:

- Rest of documentation, with relevant references (like :cite:Ingesson08FST) and figures
- Tutorial
- ToFu_Inv
- GUI (one for each module)
- Accelerate existing modules with Cython, Boost.Python + Parallelization
- Use it to do some physics at last !!!

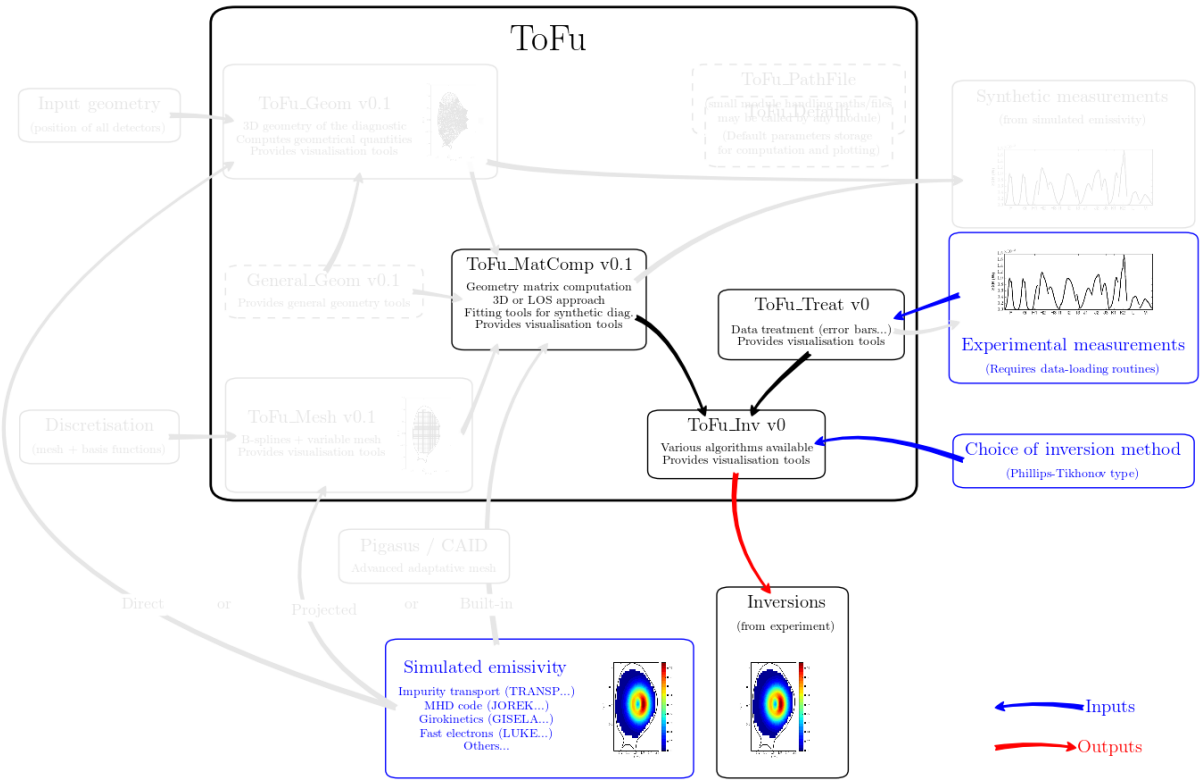

Figure 1.3: Modular architecture of ToFu, with its main modules for tomography.

CHAPTER

TWO

TOFU_GEOM

(This project is not finalised yet, work in progress...)

ToFu_Geom, is the first ToFu-specific module, it is dedicated to handling the 3D geometry of the diagnostic of interest. It defines 6 objects classes and many functions used as objects methods. It resorts to a module called **General_Geom**, which is not ToFu-specific (i.e.: it mostly conatins functions and has no reference to ToFu objects), which should be entirely re-written using Cython for faster computation. As all the other ToFu-specific modules, **ToFu_Geom** not only defines computing methods but also a variety of plotting methods that can be used to visualise various aspects and characteristics of the diagnostics as well as for debugging. This section will first give a general presentation of the **ToFu_Geom** module and will then give a tutorial for building your own diagnostic.

ToFu is designed for handling passive radiation detectors (e.g.: bolometer foils, semi-conductor diodes or gas detectors), which can be placed behind an arbitrary number of collimating apertures of any shape and orientation. This goes also for the detector, represented by its active surface (the only constraint for apertures and detectors - in the current version - is that each must be represented by a planar polygon, but they do not have to be co-planar). Each detector is thus associated to a list of apertures through which it "sees" a certain volume. The volume of interest is limited, in the case of a Tokamak, to a chamber (i.e.: the vacuum vessel) represented in **ToFu** by a toroid, itself defined by a reference 2D polygon (usually the best possible representation of the inner walls of the Tokamak) which is then expanded toroidally. The volume "seen" by each detector is then the fraction of the toroid that it can "see" directly through its various apertures. On most fusion devices, such passive radiation detectors are located in a poloidal cross-section and arranged so that their cone of vision is very thin, such that it can be represented by a simple line (called a Line Of Sigh, or LOS) and an etendue. **ToFu_Geom** allows for a full 3D description of the whole system, and also for an accurate computation of the geometrically optimal LOS and its associated etendue value. Hence, it is possible to do everything with the two approaches (full 3D or LOS) and quatify the error due to the LOS approximation, if any.

This short introduction gives the key points addressed by **ToFu_Geom**, which can be summarized by listing the 7 object classes and their meaning :

Table 2.1: The object classes in ToFu_Geom

Name	Description	Inputs needed
ID	An identity object that is used by all **ToFu** objects to store specific identity information (name, file name if the object is to be saved, names of other objects necessary for the object creation, date of creation, signal name, signal group, version...)	By default only a name (a character string) is necessary, A default file name is constructed (including the object class and date of creation), but every attribute can be modified and extra attribute can be added to suit the specific need of the the data acquisition system of each fusion experiment or the naming conventions of each laboratory.
Tor	The limits of the toroidal chamber	A 2D polygon in (R,Z) coordinates
LOS	A LOS, can be defined by the user for tests, but usually defined by the Detect object as an output	A Tor object, a starting point and a unitary vector indicating the direction of observation (the end point is computed), both in 3D (X,Y,Z) coordinates
GLOS	A group of LOS objects, with a name (useful for defining cameras which are sets of detectors with a common aperture and a commom name)	A list of LOS objects
Aperture	An aperture, represented by a planar polygon	A Tor object and a planar polygon in 3D (X,Y,Z) coordinates
Detect	A detector, represented by its planar active surface, computed a geometrically optimal LOS as an output	A Tor object, a planar polygon in 3D (X,Y,Z) coordinates, and a list of Aperture objects
GDetect	A group of Detect objects, useful for defining cameras	A list of Detect objects

The following will give a more detailed description of each object and its attributes and methods through a tutorial at the end of which you should be able to create your own diagnostics and access its main geometrical characteristics (the will be computed automatically).

2.1 Getting started with ToFu_Geom

Once you have downloaded the whole **ToFu** package (and made sur you also have scipy, numpy and matplotlib, as well as a free polygon-handling library called **Polygon** which can be downloaded at http://www.j-raedler.de/projects/polygon/, just start a python interpreter and import **ToFu_Geom** (we will always import **ToFu** modules 'as' a short name to keep track of the functionalities of each module). To handle the local path of your computer, we will also import the small module called **ToFu_PathFile**, and **matplotlib** and **numpy** will also be useful:

```
import numpy as np
import matplotlib.pyplot as plt
import ToFu_Geom as TFG
import ToFu_PathFile as TFPF
import os
import cPickle as pck # for saving objects
```

The os module is used for exploring directories and the cPickle module for saving and loading objects.

2.2 The Tor object class

To define the volume of the vacuum chamber, you need to know the (R,Z) coordinates of its reference polygon (in a poloidal cross-section). You should provide it as a (2,N) numpy array where N is the number of points defining the polygon. To give the Tor object its own identity you should at least choose a name (i.e.: a character string). For more elaborate identification, you can define an ID object and give as an input instead of a simple name. You can also provide the position of a "center" of the poloidal cross-section (in 2D (R,Z) coordinates as a (2,1) numpy array) that

will be used to compute the coordinates in transformation space any LOS using this Tor object (and the sinogram of any scalar emissivity field using this Tor object). If not provided, the center of mass of the reference polygon is used as a default "center".

In the following, we will use the geometry of ASDEX Upgrade as a example. We first have to give a reference polygon ('PolyRef' below) as a (2,N) numpy array in (R,Z) coordinates.

```
theta = np.linspace(0,2*np.pi,100)
Rcoo = 1.5 + 0.75*np.cos(theta)
Zcoo = 0.75*np.sin(theta)
PolyRef = np.array([Rcoo,Zcoo])
Tor1 = TFG.Tor('Tor_Example', PolyRef)
print Tor1
```

Alternatively, you can store PolyRef in a file and save this file locally, or use one of the default tokamak geometry stored on the **ToFu** database where Tor input polygons are stored in 2 lines .txt files (space-separated values of the R coordinates on the first line, and corresponding Z coordinates on the second line). Here, we use the default ASDEX Upgrade reference polygon stored in AUG_Tor.txt.

```
RP = TFPF.Find_Rootpath()
PathFile = RP + '/Inputs/AUG_Tor.txt'
PolyRef = np.loadtxt(PathFile, dtype='float', comments='#', delimiter=None, converters=None, skiprows
Tor2 = TFG.Tor('AUG',PolyRef)
print Tor2
```

We now have created two Tor objects, and **ToFu_Geom** has computed a series of geometrical characteristics that will be useful later (or that simply provide general information). In particular, we have access to the following attributes :

Table 2.2: The attributes of a Tor object

Attribute	Description
self.ID	The ID class of the Tor object
self.Poly	The reference polygon used to create the Tor object, as a (2,N) numpy array, where N is the number of points (the last one being identical to the first one)
self.BaryP	The barycenter of self.Poly
self.Surf	The surface of self.Poly
self.BaryS	The center of mass of self.Poly
self.Vect	The 2D vectors representing the edges of self.Poly as a (2,N) numpy array
self.Vin	The normalised 2D vectors oriented towards the inside of self.Poly for each edge
self.PRMin, self.PRMax	The points of self.Poly with the maximum (resp. minimum) R coordinate, as a (2,1) numpy array (one for PRMin, one for PRMax)
self.PZMin, self.PZMax	The points of self.Poly with the maximum (resp. minimum) Z coordinate, as a (2,1) numpy array (one for ZPMin, one for PZMax)
self.ImpRZ	The (R,Z) coordinates of the point used for computing the impact factor (i.e. the coordinates in projection space(lien)) of the LOS objects using this Tor and of the enveloppe of this Tor (default is self.BaryS)
self.Imp_EnvTheta	The discretized values used for computing the enveloppe of Tor in projection space (where theta is in [0,pi], (lien))
self.Imp_EnvMinMax	The enveloppe of Tor in projection space (lien) (i.e.: the - algebraic - minimum and maximum impact factor of the reference polygon for each value of self.Imp_EnvTheta)

In addition to these attributes, the Tor object has a number of built-in methods that can be used to visualise its characteristics. As in the whole **ToFu** package, the object methods used for plotting always begin with "self.plot...", where the name of the method after "plot..." is relatively explicit. All the plotting methods are based on matplotlib, and in order to allow for flexibility and customization, you can either pass as input an already existing matplotlib axes on which to plot, or use a predefined default axes (simply by not specifying any axes). Similarly, extensive use of keyword argumants with default values is made, thus all plotting options are customizable since you can pass a dictionnary for

element to be plotted (see the detailed documentation of each method to know which kwarg to use for which element).

As an example, you can plot the reference polygon of ASDEX Upgrade in both a poloidal and a toroidal projection, using the default axes (defined in **ToFu_Geom**) :

```
axP, axT = Tor2.plot_AllProj(Elt='PI')
#plt.show()
```

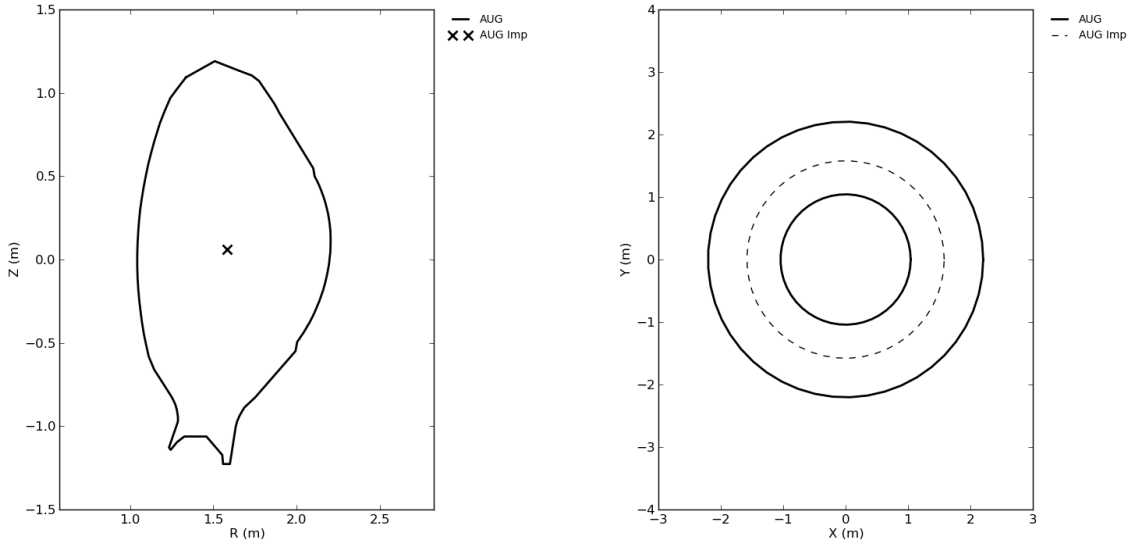

Figure 2.1: Poloidal and Toroidal projections of the reference polygon of ASDEX Upgrade

Here we used the keyword argument 'Elt' to specify which elements which wanted to plot. We provided a string in which each letter is a code for an element. Here 'P' stands for the reference polygon and 'I' for the point used for computing the impact parameter of the enveloppe. We can then re-use the axes of the poloidal projection to plot the vectors defining the edges and the inner side of the reference polygon:

```
axP = Tor2.plot_PolProj_Vect(ax=axP)
#plt.show()
```

(for some mysterious reason it is not working on my Linux station, but it does work on my macbook, as it should)

We can also plot a 3D representation of the reference polygon, and specify that we only want to plot a fraction of it, between pi/4 and 7pi/4:

```
ax3 = Tor2.plot_3D_plt(thetaLim=(np.pi/4.,7.*np.pi/4.))
#plt.show()
```

We can also visualise the enveloppe of ASDEX Upgrade in the projection space (lien), in 2D or 3D, with a color of our choosing :

```
axImp = Tor2.plot_Impact_PolProj()
axImp3 = Tor2.plot_Impact_3D()
#plt.show()
```

Feel free to explore the various keyword arguments of each method. This Tor object can then be used as a limit to the volume that can be detected by each LOS or Detect object.

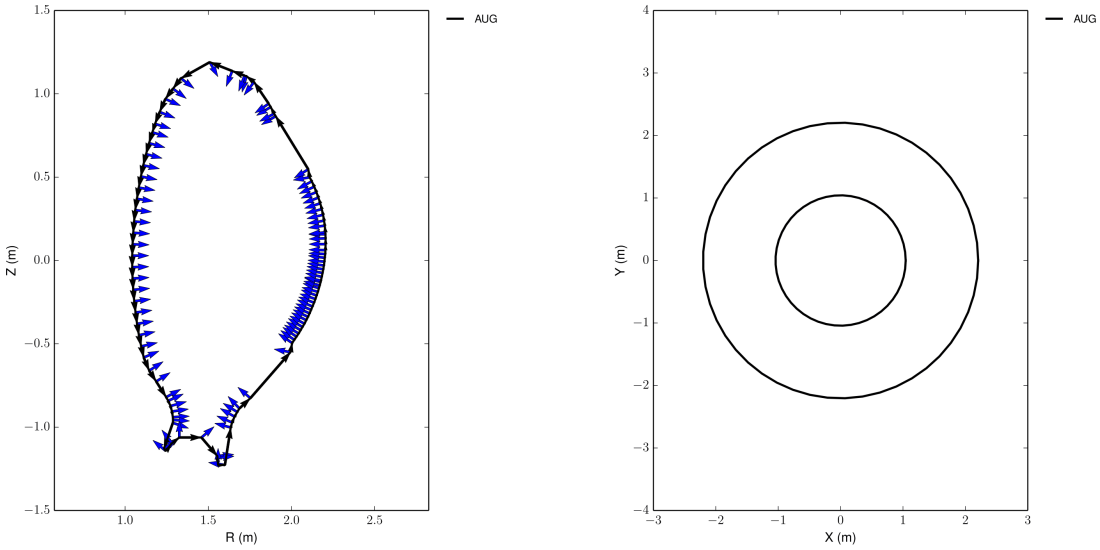

Figure 2.2: Vector representation of the reference polygon of ASDEX Upgrade

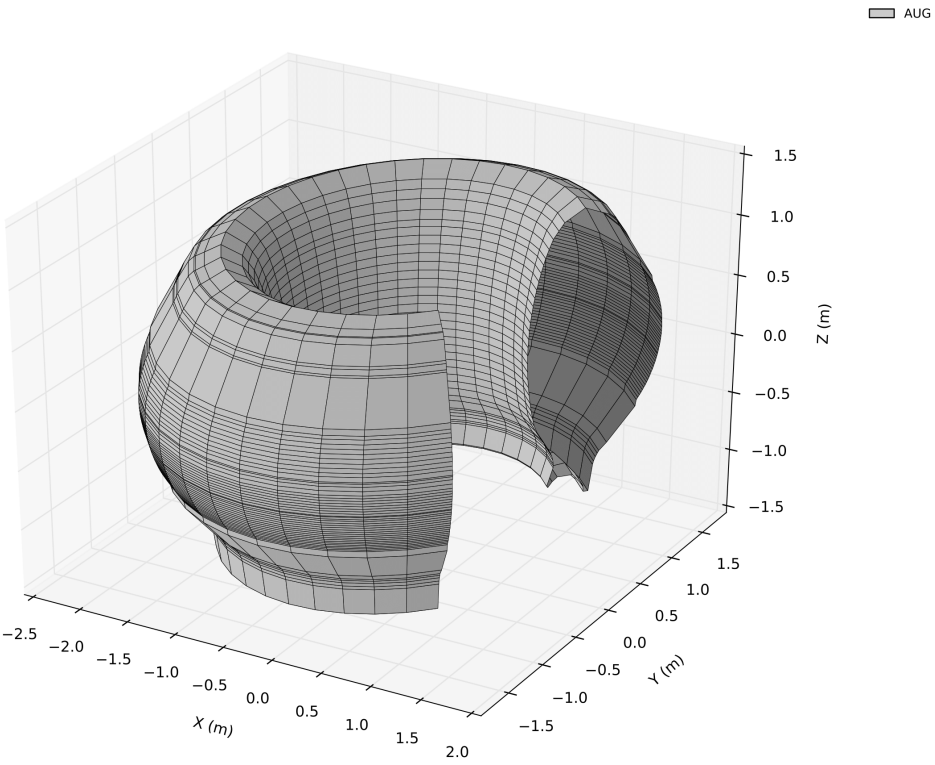

Figure 2.3: 3D representation of the reference polygon of ASDEX Upgrade

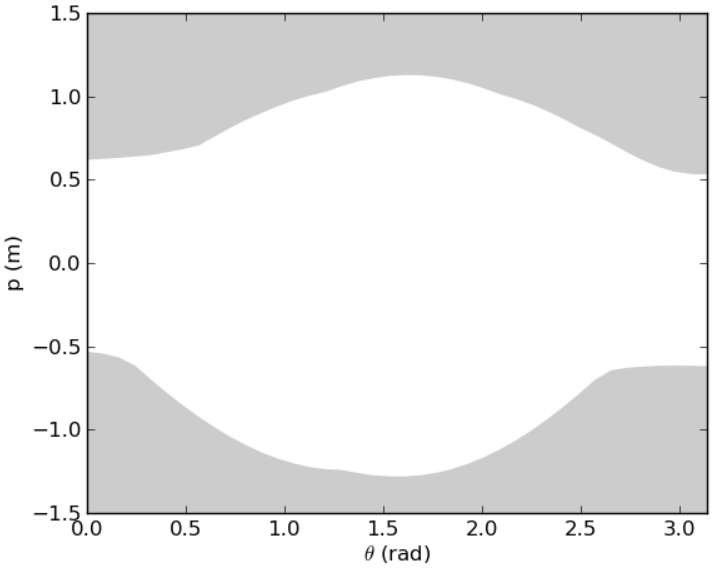

Figure 2.4: Representation in projection space of the reference polygon of ASDEX Upgrade

2.3 The LOS object class

Since most tomography users in the fusion community are familiar with the LOS approximation (which gives satisfactory results in most usual situations), we choose to provide in **ToFu** the two extremes of the spectrum : a pure LOS approximation, and a full 3D approach. Any attempt to compute the geometry matrix with an "advanced" or "improved" LOS approximation (i.e.: taking into account finite beam width, using anti-aliasing techniques with pixels...) can be considered to fall somewhere between these two extremes, and since every user has his own recipes, we do not provide any except the two extreme approaches. Obviously, all users can download **ToFu** and add there own recipe in their local version (this should be done in the **ToFu_MatComp** module). Hence, a pure LOS object exists in **ToFu**, and can be defined with minimum knowledge of the diagnostics : only a point (D) and a unitary vector (u) are necessary for each LOS. The unitary vector shall be pointing towards the direction of observation (i.e.: towards the interior of the vacuum chamber). nce a LOS id defined, **ToFu** automatically computes a series of points of interest. Indeed, if a Tor object is provided to the LOS object, we can determine the first point of entry into the Tor volume (PIn), and the point were the LOS gets out of it (POut). We can also determine the point on the LOS with minimum R-coordinate (PRMin, which is usually PIn or POut except when the LOS has a strong toroidal inclination, in which case PRMin is somewhere in the middle of the LOS). If the LOS object has a RZImp (by default the RZImp of the associated Tor object), then the impact parameter of the LOS with respect to this RZImp can be computed (has well as its two associated angles), and the LOS can be represented in projection space.

Hence, a LOS object has the following attributes :

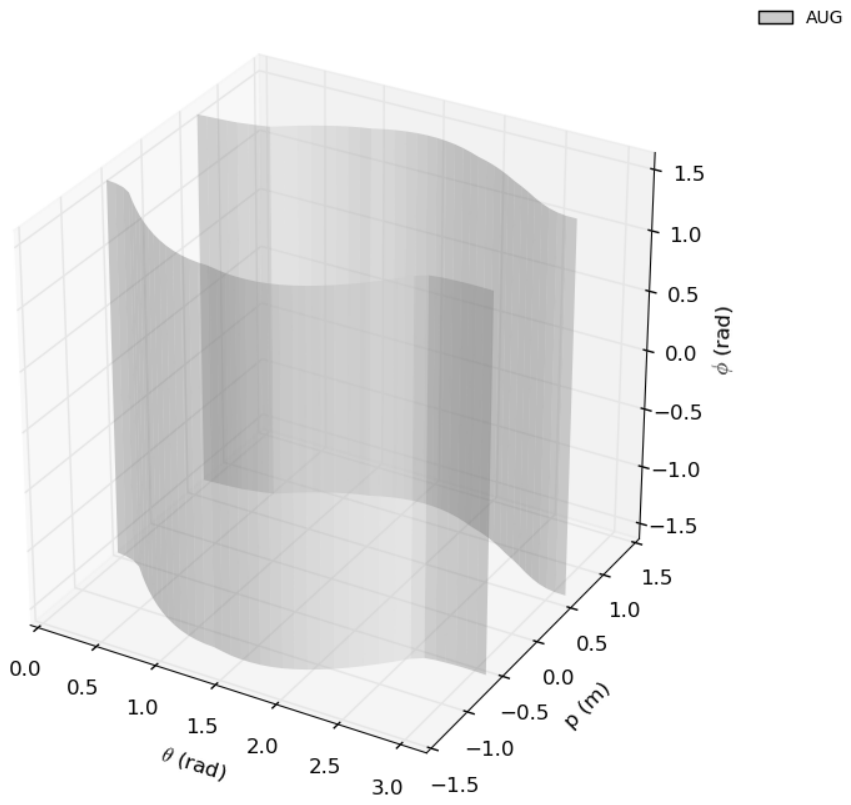

Figure 2.5: Representation in porojection space of the reference polygon of ASDEX Upgrade in 3D, this representation may be usefull when there are LOS which are not contained in a poloidal cross-section, as we will see later

Table 2.3: The attributes of a LOS object

Attribute	Description
self.ID	The ID class of the object
self.D	The starting point of the LOS in 3D (X,Y,Z) coordinates (usually the center of mass of the associated detector or the center of mass of the collimating slit for fan-like cameras)
self.u	The unitary direction vector in 3D (X,Y,Z) coordinates (oriented towards the interior of the associated Tor)
self.Tor	The associated Tor object
self.PIn, self.POut, self.PRMin	The particular points, in 3D (X,Y,Z) coordinates, on the LOS associated to the Tor enveloppe : the point at which the LOS enters the volume, the point at which it exits it, and the point of minimum R-coordinate
self.kPIn, self.kPOut, self.kPRMin	The length on the LOS (from self.D) of self.PIn, self.POut and self.PRMin
self.RMin	The value of the R-coordinate of self.PRMin
self.PolProjAng	An estimate of the angle by which the LOS is distorted in its poloidal projection. Value of 0 means the LOS is already in a poloidal cross-section and remains a straight line.
self.PplotOut, self.PplotIn	The points along the LOS used to plot its poloidal projection, either the whole LOS (self.PplotOut, from self.D to self.POut) or only the part which is inside the Tor volume (self.PplotIn, from self.PIn to self.POut)
self.ImpRZ	The 2D (R,Z) coordinates used to compute the impact factor of the LOS (i.e. its coordinates in projection space), by default self.ImpRZ = self.Tor.ImpRZ
self.ImpP, self.ImpPk, self.ImpPr, self.ImpPTheta	The point on the LOS which is closest to the self.ImpRZ (the "impact point") is self.ImpP, and its distance from self.D ((self.ImpP-self.D).(self.u) = self.ImpPk) is self.ImpPk. Its small (geometric) radius from self.ImpRZ is self.ImpPr and its toroidal angle is self.ImpPTheta
self.Impp, self.Imptheta, self.Impphi	The coordinates of the LOS in projection space, where self.Impp is the (positive or negative) impact parameter, self.Imptheta is the projection angle in a poloidal cross-section and self.Impphi is the deviation angle from the poloidal crosss-section (the reference poloidal cross-section being the one which includes self.ImpP).

Defining a LOS object only requires a start point and a unitary vector indicating the viewing direction (both in 3D (X,Y,Z) coordinates), as well as an associated Tor object. As an example, we can define Los, a LOS object as follows:

```
D = np.array([2.,-2.,1.]).reshape((3,1))     # Defining D (starting point)
uu = np.array([-0.2,1.,-0.8])                # Defining uu (vector for direction of LOS)
uu = (uu/np.linalg.norm(uu)).reshape((3,1)) # Normalising uu
Los = TFG.LOS('LOS 1',D,uu,T=Tor2)           # Creating a LOS object using a pre-defined Tor object (
print Los
```

Note that if you define a LOS objects that does not intersect the Tor volume, you will get an error message telling you that the code could not find a PIn or POut point (both are necessary). All the geometric characteristics of Los has now been computed (the coordinates in projection space have been computed using the center of mass of the reference polygon of Tor2 as default, but they can be re-computed with another reference point, as we will see later). The built-in routines can be used to visualise Los, and we specify, thanks to the keyword argument 'PDIOR' that we not only want to see the LOS itself but also the position of the particular points that were computed or that were used for its definition (self.D, self.PIn, self.POut and self.PRMin => DIOR). In order to better visualise it, we plot it both in poloidal and toroidal projections, re-using a set of axes on which we first plot Tor2:

```
axP, axT = Tor2.plot_AllProj()
axP, axT = Los.plot_AllProj(Elt='LDIORr',EltTor='PI')
#plt.show()
```

We used again here the 'Elt' keyword argument to specify that we want to plot the LOS itself ('L'), the particular

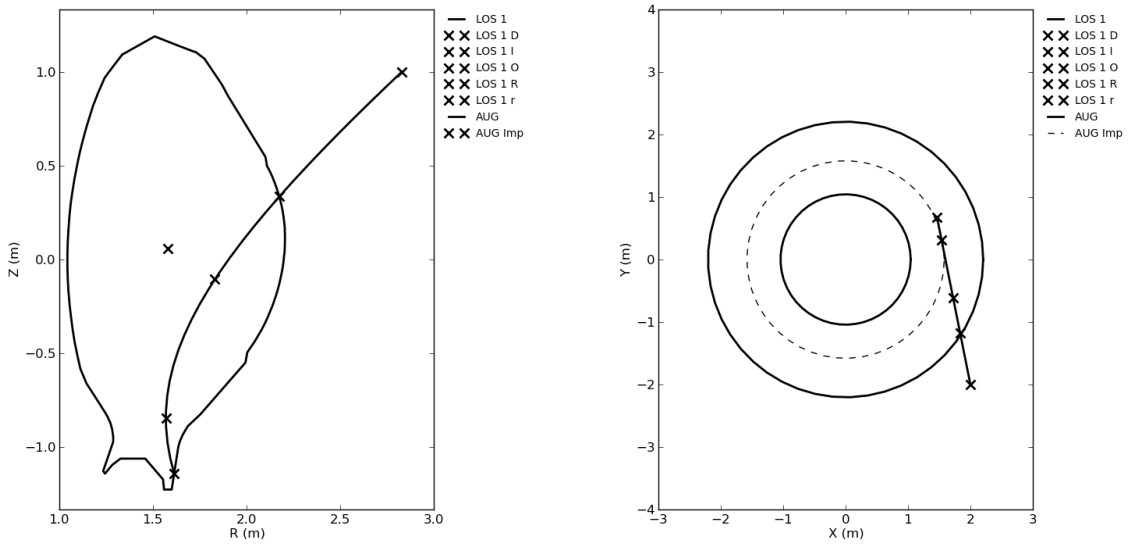

Figure 2.6: Poloidal and toroidal projection of a user-defined LOS object, with points of interest and parent Tor object

points self.D ('D'), self.PIn ('I'), self.POut('O'), self.PRMin ('R') and self.ImpP ('r'). In fact, since the LOS object has a Tor attribute, the possibly of plotting the Tor object at the same time as the LOS object is provided in the same method, through the kwdarg 'EltTor' (just provide the same letters as for a Tor object).

Like for the Tor object, the LOS object can also be plotted in 3D using the plot_3D_plt() method:

```
ax3 = Los.plot_3D_plt(Elt='LDIORr',EltTor='PI',thetaLim=(0.,7.*np.pi/4.),MdictR={'c':'b','marker':'o
#plt.show()
```

Where we have plotted the associated Tor object using the kwdarg 'EltTor' and changed the dictionaries for the self.PRMin and self.ImpP points. But generally matplotlib is not the best library for 3D plots with several objects, hence, mayavi is currently being considered for implementation since it is much more adapted to this particular task.

Also, the coordinates of Los in projection space can be plotted on the same graph as the enveloppe of Tor2 was plotted, in 2D or 3D (3D being relevant to take into account the fact that Los does not lie in a poloidal cross section). Beware that these coordinates depend on the reference point chosen. To illustrate this, we compute the impact parameter of Los with the default reference point (i.e.: the center of mass of its associated Tor object) in blue and with a different, arbitrary, reference point in red:

```
mdict = {'ls':'None','marker':'o','c':'b'}
axImp = Los.plot_Impact_PolProj(ax=axImp,Mdict=mdict)    # Plot coordinates (impact parameter and angl
RefP2 = np.array([1.5,-0.05]).reshape((2,1))
Los.set_Impact(RefP2)                                    # Compute the new coordinates (impact paramet
mdict = {'ls':'None','marker':'o','c':'r'}
axImp = Los.plot_Impact_PolProj(ax=axImp,Mdict=mdict)    # Plot new coordinates (impact parameter and
#plt.show()
```

N.B.: In fact the enveloppe of Tor2 changes also when we change the reference point, but only the first enveloppe is displayed here for clarity. Now we know how to build a LOS object, get its main geometrical characteristics and plot it. But most tomography diagnotics rely on tens or hundreds of different LOS which, in the fusion community, are usually grouped in what is called "cameras". A "camera" is typically a set of several detectors which share a common aperture in a fan-like arrangement, which is a good compromise between room saving (access is scarce around Tokamaks) and good geometrical coverage. Hence, a LOS can be defined for each detector as the line that runs through its center of mass and through the center of mass of its associated aperture. The fan-like arrangement means that all LOS belonging

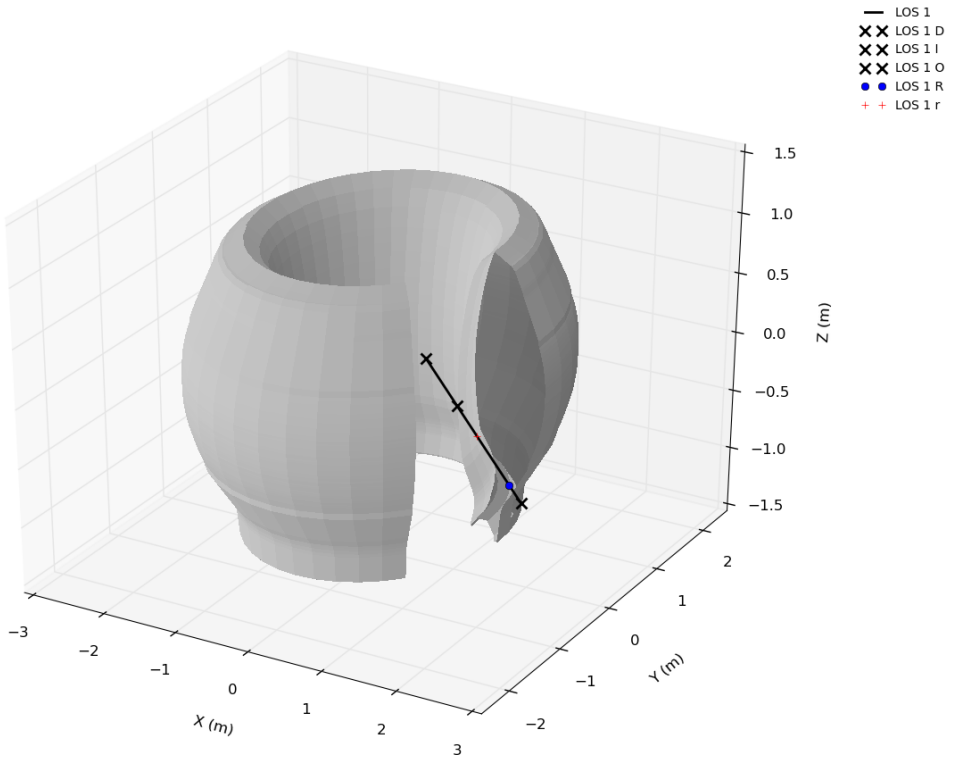

Figure 2.7: 3D plot of a user-defined LOS object, with points of interest and parent Tor object

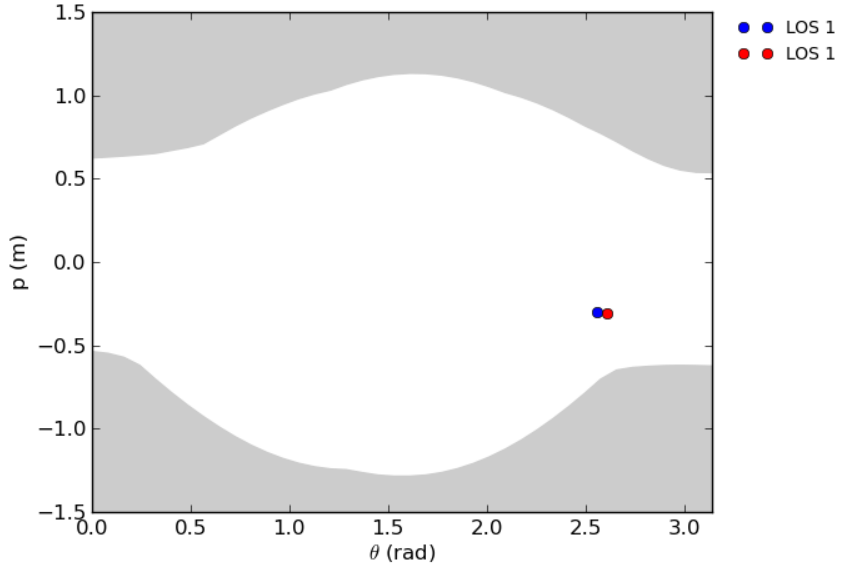

Figure 2.8: Coordinates in projection space of Los, with respect to two different reference points

to the same camera will cross at their common aprture. To this purely goemterical consideration, one must add the data acquisition system which often treats data from a camera as a group of signals identified by a common nomenclature. For these reasons, it is useful to define an object embodying the notion of "camera", which is simply done by the GLOS object class.

ToDo : add visualisation options for for the reference points and LOS.ImpP in physical space (add kwdarg in existing functions)

2.4 The GLOS object class

The GLOS object class (where GLOS stands for Group of Lines Of Sight) is simply a list of LOS objects with a common ID class (i.e.: a common name and other identity features). It is useful for fast and easy handling of a large number of LOS.

Table 2.4: The attributes of a GLOS object

Attribute	Description
self.ID	The ID class of the object
self.LLOS, self.nLOS	The list of LOS objects contained in this group, and the number of LOS (self.nLOS = len(self.LLOS))
self.Tor	The Tor object common to all LOS of self.LLOS

The methods of a GLOS object can be seperated in two categories. First, all the LOS objects methods are reproduced in such a way as to handle all the LOS contained in GLOS (for example with "for" loops). These include in particular the plotting methods. Second, some methods are provided to facilitate selection of sub-sets of LOS in the GLOS objects and handle them. For example, we can create two cameras of respectively 10 amd 15 LOS:

```
P1, P2 = np.array([2.,-2.,1.]).reshape((3,1)), np.array([0.5,-0.5,-0.5]).reshape((3,1))    # Creating
n1, n2 = 10, 15                                                                            # Reminding
phi1, phi2 = np.pi/7., np.pi/4.
theta1, theta2 = np.linspace(np.pi/8.,np.pi/4.,n1), np.linspace(5.*np.pi/6.,6.5*np.pi/6.,n2)
uu1 = np.array([-np.sin(phi1)*np.cos(theta1),np.cos(phi1)*np.cos(theta1),-np.sin(theta1)])  # Creati
uu2 = np.array([-np.sin(phi2)*np.cos(theta2),np.cos(phi2)*np.cos(theta2),-np.sin(theta2)])
LLos1 = [TFG.LOS("Los1"+str(ii),P1,uu1[:,ii:ii+1],T=Tor2) for ii in range(0,n1)]           # Creating
LLos2 = [TFG.LOS("Los2"+str(ii),P2,uu2[:,ii:ii+1],T=Tor2) for ii in range(0,n2)]
GLos1, GLos2 = TFG.GLOS("Cam1",LLos1), TFG.GLOS("Cam2",LLos2)                              # Creating t
print GLos1, GLos2
```

We can then plot their poloidal and toroidal projections (without the particular points) , on top of the reference polygon of Tor2:

```
axP, axT = GLos1.plot_AllProj(Ldict={'c':'b'},Elt='L',EltTor='PI',Lplot='In')
axP, axT = GLos2.plot_AllProj(axP=axP,axT=axT,Ldict={'c':'r'},Elt='L')
#plt.show()
```

Notice here that we used the keyword argument "LPlot" to specify that the LOS of the first camera should only be plotted inside the Tor volume (i.e.: from PIn to POut) whereas tyhe default is to plot the entire LOS (Lplot='Tot', which plots the LOS from D to POut). We also used the "Ldict" kwdarg to specify a dictionnary for the plotting command.

Like the LOS objects, a GLOS object enables you to plot the coordinates in projection space of all the LOS it contains (in 2D or 3D):

```
axImp = Tor2.plot_Impact_PolProj()
axImp = GLos1.plot_Impact_PolProj(ax=axImp,Mdict={'ls':'None','marker':'o','c':'b'})
axImp = GLos2.plot_Impact_PolProj(ax=axImp,Mdict={'ls':'None','marker':'x','c':'r'})
plt.show()
```

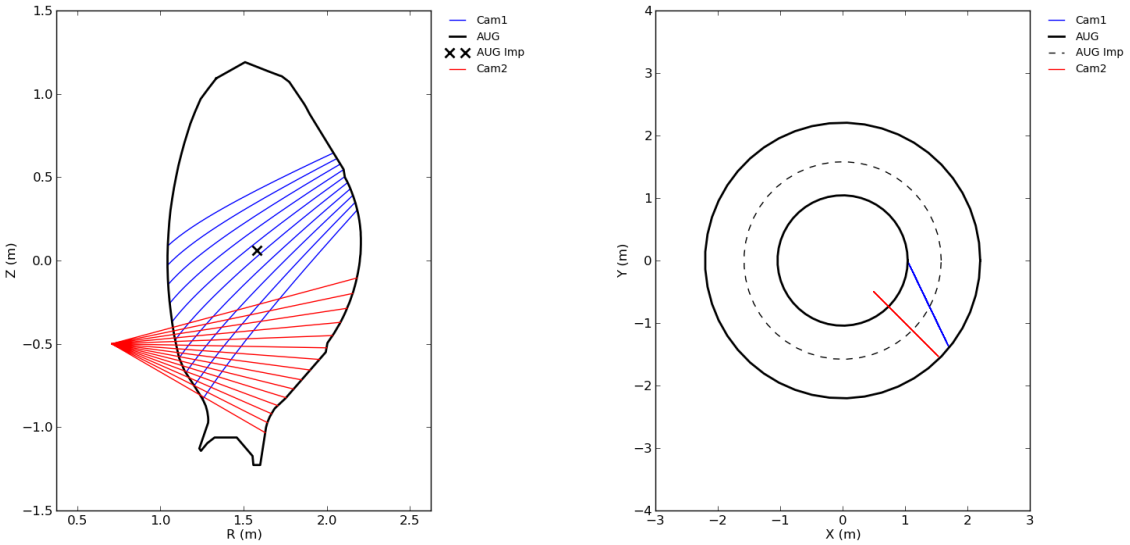

Figure 2.9: Poloidal and toroidal projections of two arbitrary cameras, with differenmt plotting options

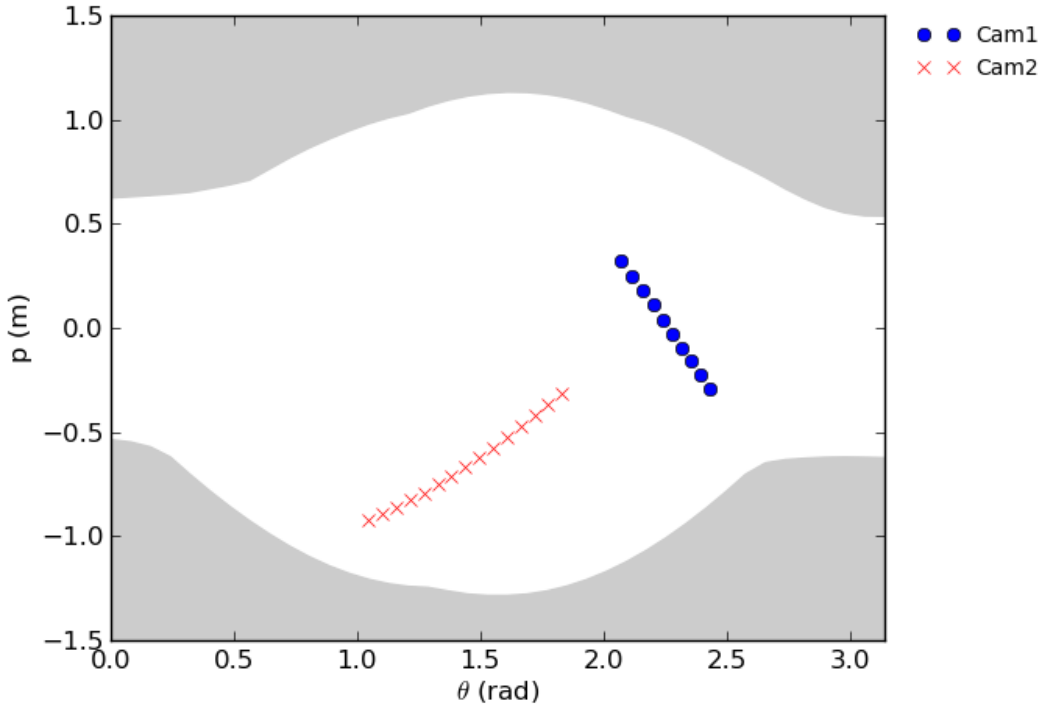

Figure 2.10: Coordinates in projection space of the two cameras, with the Tor enveloppe

In adition to these methods which echo the methods of the LOS class, the GLOS class offers tools to select subsets of the list of LOS from which it was created. This is were the relevance of the ID class starts to show, indeed, besides the Name that you gave to your objects, you might want to store data which is both specific to these objects and to your needs or to the naming conventions of your laboratory. For example, you might want to know enter the signal code associated to each detector, or the age of each detector (to have an idea of the effect of ageing on its performance)... Hence, when you create an object - like a LOS for example - you can add extra attributes to its ID class. These attributes are anything that you consider helpful to identify / discriminate a particular object. In the following example, we re-create a camera, but we add the code signal ("Signal" + number of LOS) and age (between 0 and 3 years) of each detector to the ID class of its LOS:

```
for ii in range(0,n1):
    GLos1.LLOS[ii].Id.Code = "Signal"+str(ii*10)
    GLos1.LLOS[ii].Id.Age = 3.*ii/n1
```

We can now ask the GLOS object to give us a list of its LOS that match a criteria of our choosing. There are two methods to do this. They take the same arguments, but the first one will return a numpy array of boolean indices (for later use if you need it), while the second one directly returns a list of LOS objects (and uses the first one). For example, we use the first one to get the indices of LOS with a signal code equal to "Signal0" or "Signal50", and the second one to get a list of LOS aged less than a year:

```
    ind = GLos1.get_ind_LOS('Code',"in ['Signal0','Signal50']")
subLLos = GLos1.pick_LOS('Age','<=1.')
print ind, subLLos
```

The flexibility is provided through the use of eval() which allows for string expressions. These methods are intended to provide the necessary flexibility for quick adaptation to your specific needs. Depending on return on experience, it may also evolve or be developped further. Alternatively, you can also build a list of the attributes you are interest in and then use the list() methods to get the indices you want:

```
LAttr = [los.Id.Code for los in GLos1.LLOS]
ind = [LAttr.index('Signal0'), LAttr.index('Signal50')]
print ind
```

As was already said, **ToFu** provides you with the possibility of defining and using LOS if you wish, however, its main features reside in the 3D description of the diagnostic, of which the LOS description is just an approximation (which can be sufficient for your needs, depending on the geometry of your system, on the physics at play and on the accuracy that you want to acheive). In the following, we introduce the Detect class which is used to handle the 3D geometry of the problem. Once a Detect object is created, it can be associated to several Aperture objects to determine its 3D field of view. Consequently, the geometrically optimal LOS can also be computed and the associated LOS object can be easily produced on demand, we then generally advise to directly create Detect objects, of which LOS objects can be seen as a subproduct.

2.5 The Detect and Apert object classes

In addition to what as been said above, creating dierctly a Detect object instead of a LOS object will provide you with the ability to compute an accurate value of the etendue associated to the LOS approximation (link to definition of etendue and why it is important for proper use of LOS approximation). In its current version, **ToFu** handles apertures as 3D planar polygons which, to this day, have the following attributes:

Table 2.5: The attributes of an Apert object

Attribute	Description
self.ID	The ID class of the object
self.Poly, self.PolyN	A (3,N) numpy array representing a planar polygon in 3D (X,Y,Z) coordinates, and the number of points that this polygon is comprised of.
self.BaryP, self.BaryS, self.S, self.nIn	The barycenter of self.Poly and its center of mass, its surface and the normalised vector perpendicular to the plane of self.Poly and oriented towards the interior of the Tor volume.
self.Tor	The Tor object associated to the Detect object

The Apert object is mainly used as a computing intermediate for the Detect object. However, it does come along with some key plotting methods aimed at giving you an idea of its geometry in the usual projections (poloidal and toroidal) and in 3D.

Similarily, **ToFu** handles apertures as 3D planar polygons (i.e.: the polygon embodying the active surface of the detector) which, to this day, have the following attributes:

Table 2.6: The attributes of a Detect object

Attribute	Description
self.ID	The ID class of the object
self.Poly, self.PolyN	A (3,N) numpy array representing a planar polygon in 3D (X,Y,Z) coordinates, and the number of points that this polygon is comprised of.
self.BaryP, self.BaryS, self.S, self.nIn	The barycenter of self.Poly and its center of mass, its surface and the normalised vector perpendicular to the plane of self.Poly and oriented towards the interior of the Tor volume
self.Tor	The Tor object associated to the Detect object
self.LApert	A list of Apert objects associated to the Detect object
self.LOS	A LOS object corresponding to the geometrically optimal LOS going through self.BaryS and through the center of mass of the intersection of all the associated Apert objects. Its value is 'Impossible' if the geometry you built does not allow for the existence of a LOS (i.e.: if the volume inside Tor viewed by the detector through its apertures is zero).
self.LOS_TorAngRef	The value of toroidal angle corresponding to the position of the middle of the LOS (between self.LOS.PIn and self.LOS.POut), used by plotting methods, can be different from the toroidal angle of the detector if the LOS does not stand in a poloidal cross-section.
self.LOS_Etend_0Approx, self.LOS_Etend_0ApproxRev, self.LOS_Etend_PerpSamp, self.LOS_Etend_Perp, self.LOS_Etend_RelErr	Values of the etendue, computed respectively using a fast 0th order approximation, a 0th order approximation reversed, a sampled integral in a plane perpendicular to the LOS, an adaptative integral in a plane perpendicular to the LOS. The last attribute is the relative error tolerance used for the adaptative computation of the integral (default is 0.01 %).
self.Span_R, self.Span_Theta, self.Span_Z, self.Span_k	The tuples indicating the min and max values of the cylindrical (R,Theta,Z) coordinates inside which the viewing cone of the Detect object can be found. These are limits that define a box inside which the viewing cone is found, they do not give the viewing come itself. The Span_k attribute corresponds to the span of the component along self.LOS.u that can be reached inside the viewing cone (estimated by sampling the viewing cone into more than 1000 LOS - the exact number depends on self.Poly and on the shapes of the apertures and can be tuned by parameters).
self.Span_NEdge, self.Span_NRad	The parameters that were used for computing the span in cylindrical coordinates of the system. The first one quantifies the number of extra points added on the polygon edges, and the second one the number of extra points added in the radial direction.
self.Cone_PolyPol, self.Cone_PolyTor	The poloidal and toroidal projectiosn of the 3D viewing cone of the {detector+apertures} system. These projected polygons are useful for visualising the detected volume (or rather its projections) and for fast discrimination of points which are inside / outside of this detected volume (i.e.: fast computation of integrated signal)

Now we are going to create two arbitrary Apert objects and one Detect object to show how it is done and what information it gives access to. As already mentioned, the various Apert objects associated to a Detect object must be planar polygons, but they do not need to be coplanar, and they can have any arbitrary shape, hence:

```
d1, d2, d3 = 0.02, 0.02, 0.02
C1, C2, C3 = np.array([1.56,-1.49,0.75]), np.array([1.52,-1.38,0.70]), np.array([1.60,-1.60,0.80])
C1, C2, C3 = C1.reshape((3,1)), C2.reshape((3,1)), C3.reshape((3,1))
n1, n2, n3 = np.array([0.1,-2.,0.5]), np.array([1.,-1.,0.8]), np.array([1.,-1.,0.])
n1, n2, n3 = n1/np.linalg.norm(n1), n2/np.linalg.norm(n2), n3/np.linalg.norm(n3)
e11, e21, e31 = np.cross(n1,np.array([0.,0.,1.])), np.cross(n2,np.array([0.,0.,1.])), np.cross(n3,np
e11, e21, e31 = e11/np.linalg.norm(e11), e21/np.linalg.norm(e21), e31/np.linalg.norm(e31)
e12, e22, e32 = np.cross(n1,e11), np.cross(n2,e21), np.cross(n3,e31)
Poly1 = d1*np.array([[-1, 1, 0.],[-1, -1, 1]])
Poly2 = d2*np.array([[-1, 1, 1.5, 0., -1.5],[-1, -1, 0., 1, 0.]])
Poly3 = d3*np.array([[-1., 1, 1, -1,],[-1., -1., 1., 1.]])
Poly1 = np.dot(C1,np.ones((1,Poly1.shape[1]))) + np.dot(e11.reshape((3,1)),Poly1[0:1,:]) + np.dot(e12
Poly2 = np.dot(C2,np.ones((1,Poly2.shape[1]))) + np.dot(e21.reshape((3,1)),Poly2[0:1,:]) + np.dot(e22
Poly3 = np.dot(C3,np.ones((1,Poly3.shape[1]))) + np.dot(e31.reshape((3,1)),Poly3[0:1,:]) + np.dot(e32

Ap1, Ap2 = TFG.Apert('Apert1', Poly1, T=Tor2), TFG.Apert('Apert2', Poly2, T=Tor2)
D1 = TFG.Detect('Detect1', Poly3, T=Tor2, LApert=[Ap1,Ap2], CalcEtend=True, CalcCone=True)
print Ap1, Ap2, D1
print D1.LOS, D1.LOS_Etend_Perp
```

Note that the computation may take some time (several seconds) due to the accurate computation of the etendue. If you do not need the etendue, you can avoid its computation using the kwdarg 'CalcEtend'=False (default value is True). Once we can check that the constructed geometry is relevant (i.e.: that it allows for a non-zero detected volume, which means that a LOS should exist), we can plot the associated Detect elements and LOS:

```
axP, axT = D1.plot_AllProj(Elt='PV', EltApert='PV', EltLOS='LDIORr', EltTor='PI')
#plt.show()
```

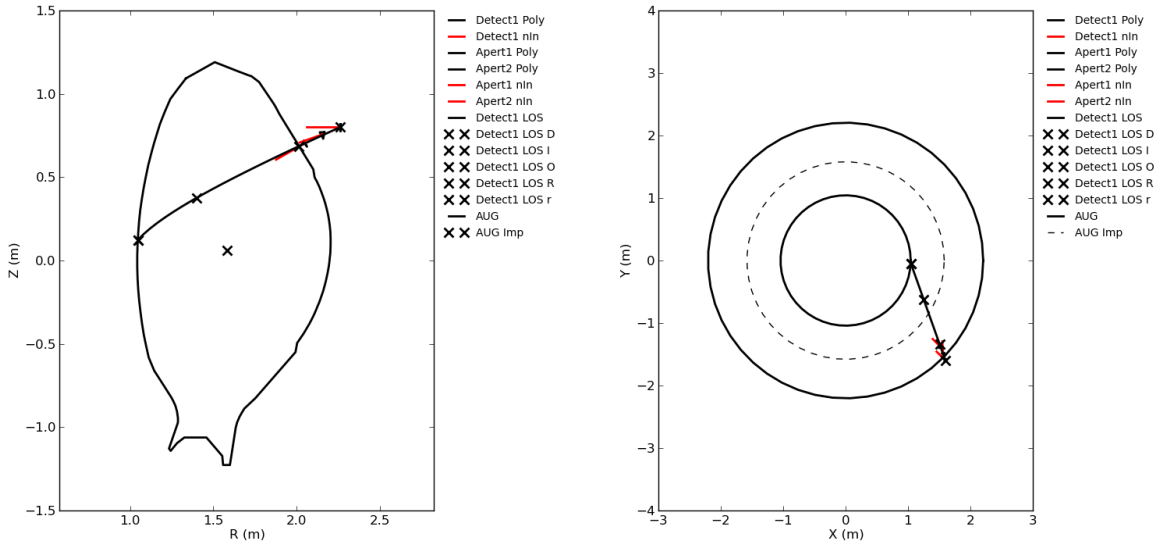

Figure 2.11: Poloidal and toroidal projections of a Detect object with all its associated objects (2 Apert objects, a Tor object, and a subsequent LOS object)

As said earlier, the three polygons do not have to be coplanar, as is visible on the next figure on which we only plotted the two Apert objects and the Detect object (with their perpendicular vectors), as well as the start point of the LOS and its entry point into the Tor volume (in blue):

```
ax3 = D1.plot_3D_plt(Elt='PV', EltApert='PV', EltLOS='DI', EltTor='',MdictI={'c':'b','marker':'o','ls
#plt.show()
```

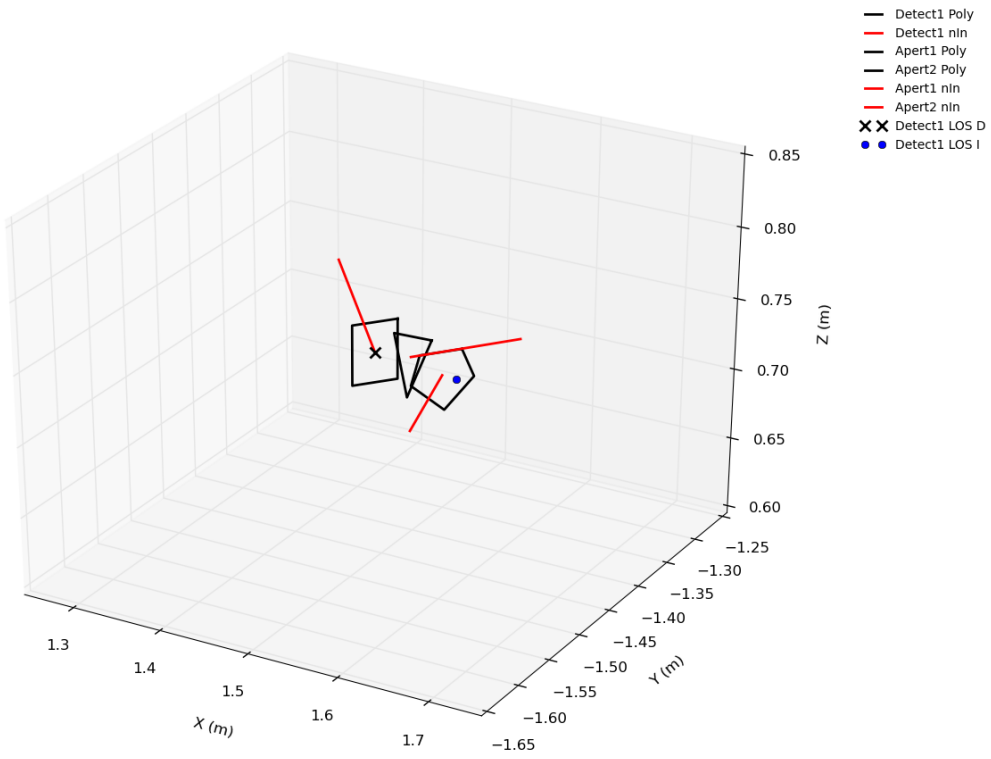

Figure 2.12: 3D plot of an arbitrary Detect object with two non-coplanar Apert objects

Since the Detect object has a LOS object as an attribute, all the LOS methods are accessible via this LOS attribute, making it easy to plot the coordinates in projection space of the LOS of this particular Detect object.

We saw that the etendue is computed automatically when the Detect object is defined. This is done via numerical integration, on a plane perpendicular to the goemetrically optimal LOS, of the solid angle subtended by the Detect and its Apert objects. **ToFu_Geom** thus has built-in routines to compute that solid angle from any point in the 3D Tor volume. This will also be useful to compute the total signal received by the detector from a given radiation field. Of course, when taken on a plane perpendicular to the geometrically optimal LOS, the solid angle decreases as we get to the edge of the viewing cone. You can visualise the solid angle on any plane perpendicular to the LOS simply by choosing its relative position on the LOS via the 'Ra' kwdarg of the following method (0 and 1 corresponding respectively to the PIn and POut points of the LOS):

```
ax = D1.plot_SAng_OnPlanePerp(Ra=0.5)
#plt.show()
```

The value in parenthesis in the title is a ratio (here 1 %) used to plot make sure the plot includes the entirety of the viewing cone in this plane (i.e.: the plotting surface is 1 % larger than the estimated support of the viewing cone). The reason why this surface has no easily reckognisable shape is due to the fact that it comes from a system consisting of 3 non-coplanar polygons with various shapes. If we had used a square detector with a coplanar square aperture, the square shape would have been visible on the iso-contours of the solid angle.

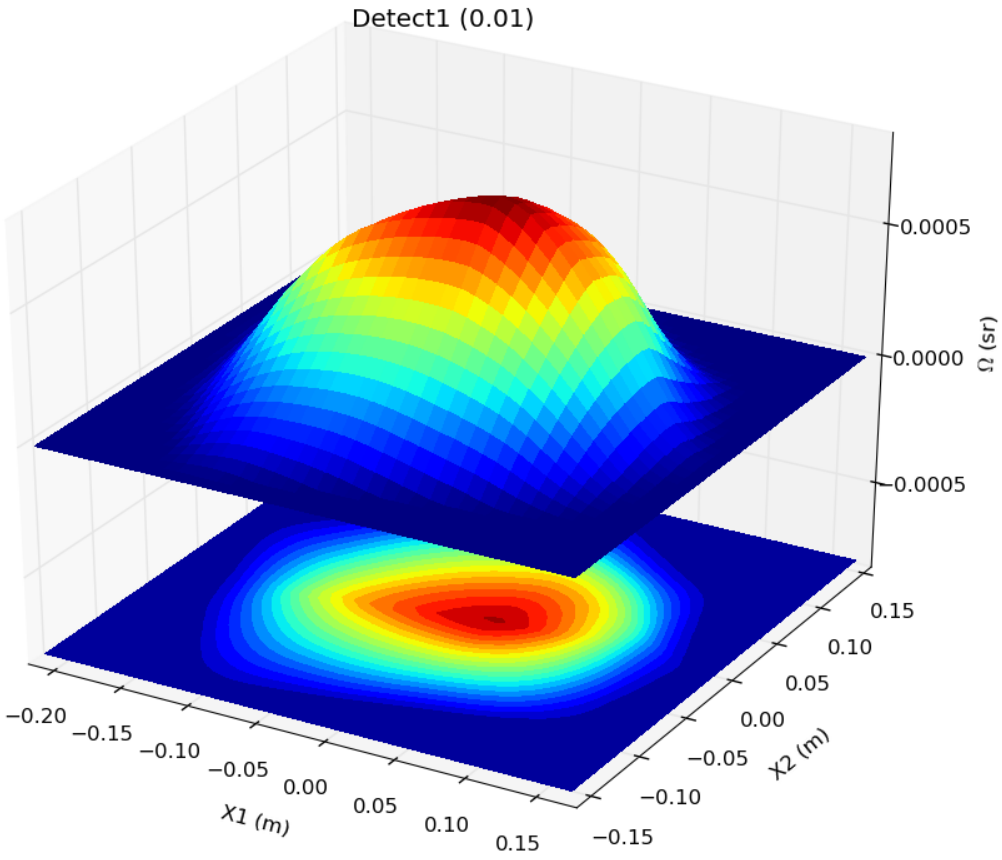

Figure 2.13: Surface plot of the solid angle subtended by the {Detector + Apertures} system as seen from points on a plane perpendicular to the system's LOS and placed at mid-length of the LOS

Similarily, it is possible to simply plot the evolution of the etendue (solid angle integrated on the plane) as a function of the point's distance on the LOS (to check that it remains constant), using three different integration methods (two via discretisation and one via an adaptative algorithm), this may take a while because the etendue has to computed Nx3 times (3 times for each point):

```
ax = D1.plot_Etend_AlongLOS(NP=5, Colis=False, Modes=['simps','trapz','quad'], Ldict=[{'c':'k','ls':
#plt.show()
```

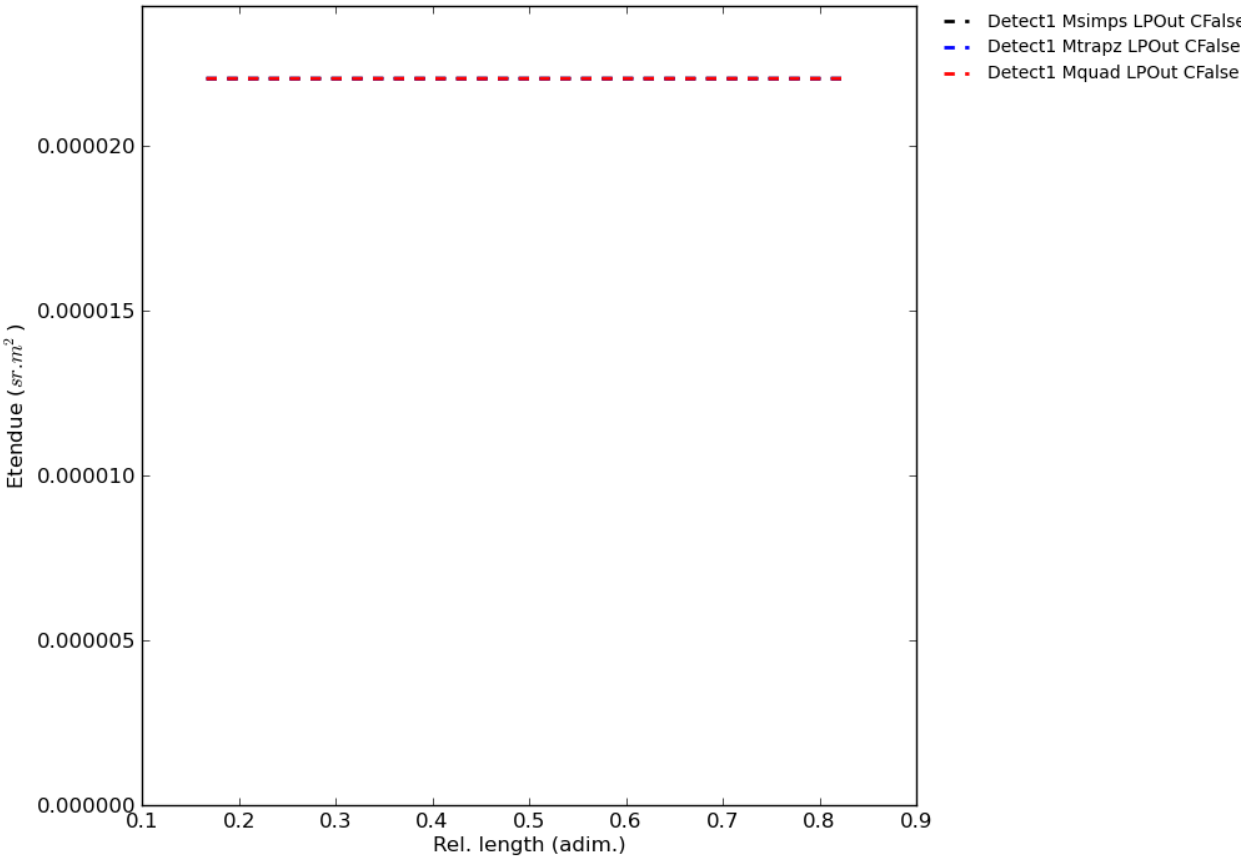

Figure 2.14: Etendue of the {Detector + Apertures} system as a function of the relative distance along the LOS (from 0 = self.LOS.PIn to 1 = self.LOS.POut), with three different integration methods using their defaults settings

We can see that the default settings used for each method are sufficient to give an accurate computation of the etendue that remains constant along the LOS, as it should.

Now, in order to explore further the geometry of the system, we can plot the value of the solid angle in any poloidal plane (respectively horizonntal plane) intersecting the viewing cone, we can then visualise the viewing cone:

```
axSAP, axNbP = D1.plot_PolSlice_SAng()
axSAP, axNbP = D1.Tor.plot_PolProj(ax=axSAP), D1.Tor.plot_PolProj(ax=axNbP)
axSAT, axNbT = D1.plot_TorSlice_SAng()
axSAT, axNbT = D1.Tor.plot_TorProj(ax=axSAT), D1.Tor.plot_TorProj(ax=axNbT)
#plt.show()
```

By default the poloidal slice is the plane which interssects the LOS at mid-length, by you can choose any toroidal

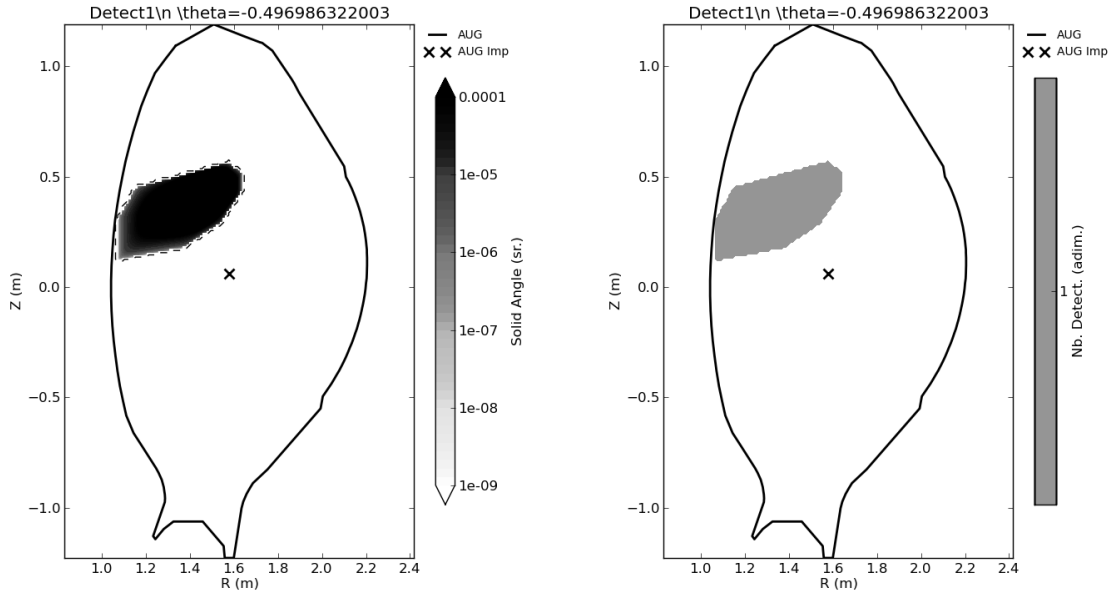

Figure 2.15: (Left) Contour plot of the solid angle subtended by the {Detector+Apertures} system (Right) Number of detectors that can "see" each point of the same poloidal slice (this will be useful for systems with several detectors)

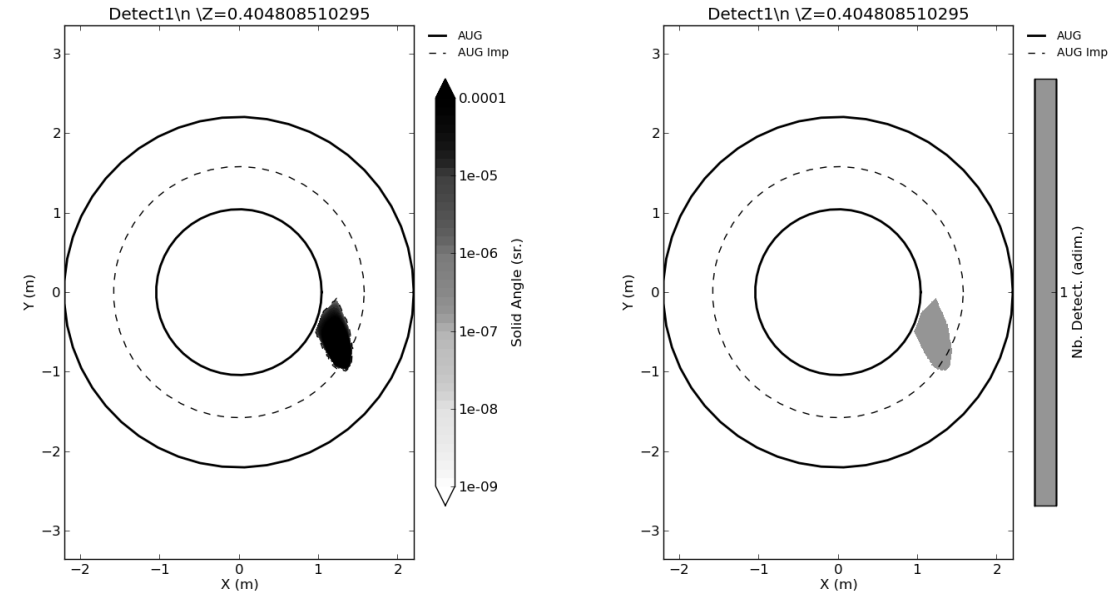

Figure 2.16: (Left) Contour plot of the solid angle subtended by the {Detector+Apertures} system (Right) Number of detectors that can "see" each point of the same horizontal slice (this will be useful for systems with several detectors)

angle by using the "Theta" kwdarg. Note that the above plots are poloidal (resp. horizontal) *slices*, not *projections*. In its current version, **ToFu_Geom** only allows to plot *projections* by computing the solid angle for several discrete *slices* (25 by default, plus particular slices including self.LOS.PIn, self.LOS.POut and the mid-length point) close to each other and plotting the maximum value for each points (computation is very long in the current, non-optimised, python-only version, typically 20-30 min for 10 slices):

```
axSAP, axNbP = D1.plot_PolProj_SAng()
axSAP, axNbP = D1.Tor.plot_PolProj(ax=axSAP), D1.Tor.plot_PolProj(ax=axNbP)
axSAT, axNbT = D1.plot_TorProj_SAng()
axSAT, axNbT = D1.Tor.plot_TorProj(ax=axSAT), D1.Tor.plot_TorProj(ax=axNbT)
#plt.show()
```

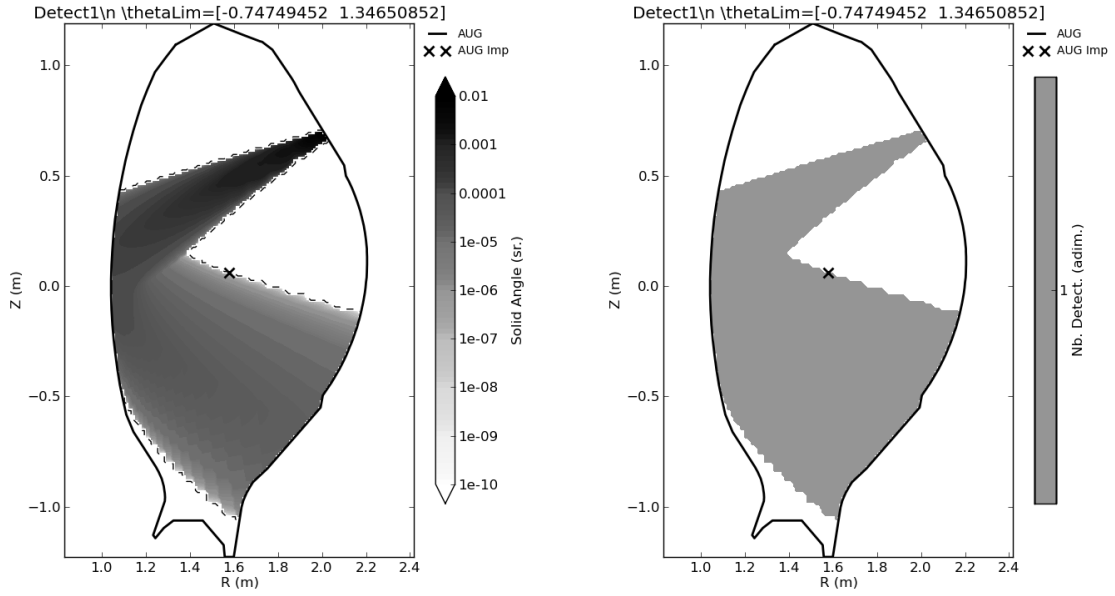

Figure 2.17: (Left) Contour plot of the solid angle subtended by the {Detector+Apertures} system in a poloidal projection (Right) Number of detectors that can "see" each point (this will be useful for systems with several detectors)

Notice that there is a collision-detection routine in the ray tracing code that takes into account the fact that the viewing cone is limited by the Tor instance. This caveat can be de-activated by using the "Colis" kwdarg (=True by default), as illustrated in the following:

```
axSAP, axNbP = D1.plot_PolProj_SAng(Colis=False)
axSAP, axNbP = D1.Tor.plot_PolProj(ax=axSAP), D1.Tor.plot_PolProj(ax=axNbP)
axSAT, axNbT = D1.plot_TorProj_SAng(Colis=False)
axSAT, axNbT = D1.Tor.plot_TorProj(ax=axSAT), D1.Tor.plot_TorProj(ax=axNbT)
#plt.show()
```

These plotting commands give a pretty good idea of the fraction of the Tor volume which is seen by the detector through its associated apertures. It is actually these functionalities (plotting poloidal and toroidal projections of the solid angle) that are used to extract the poloidal and toroidal projections of the viewing cone as two sets of 2D polygons (i.e.: the 0 iso-contours of the solid angle projections). These two projected polygons can be simply plotted by adding 'C' (like 'cone') to the "Elt" kwdarg of the plot_PolProj and plot_TorProj plotting methods:

```
axP, axT = D1.plot_AllProj(Elt='PVC', EltApert='PV', EltLOS='LDIORr', EltTor='PI')
#plt.show()
```

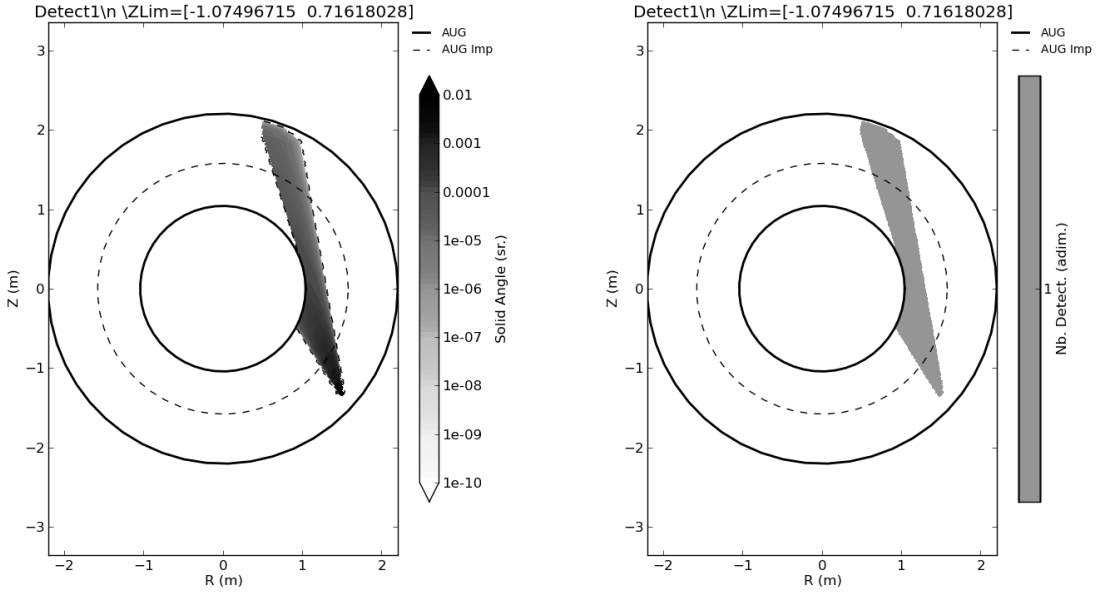

Figure 2.18: (Left) Contour plot of the solid angle subtended by the {Detector+Apertures} system in a horizontal - or toroidal - projection (Right) Number of detectors that can "see" each point (this will be useful for systems with several detectors)

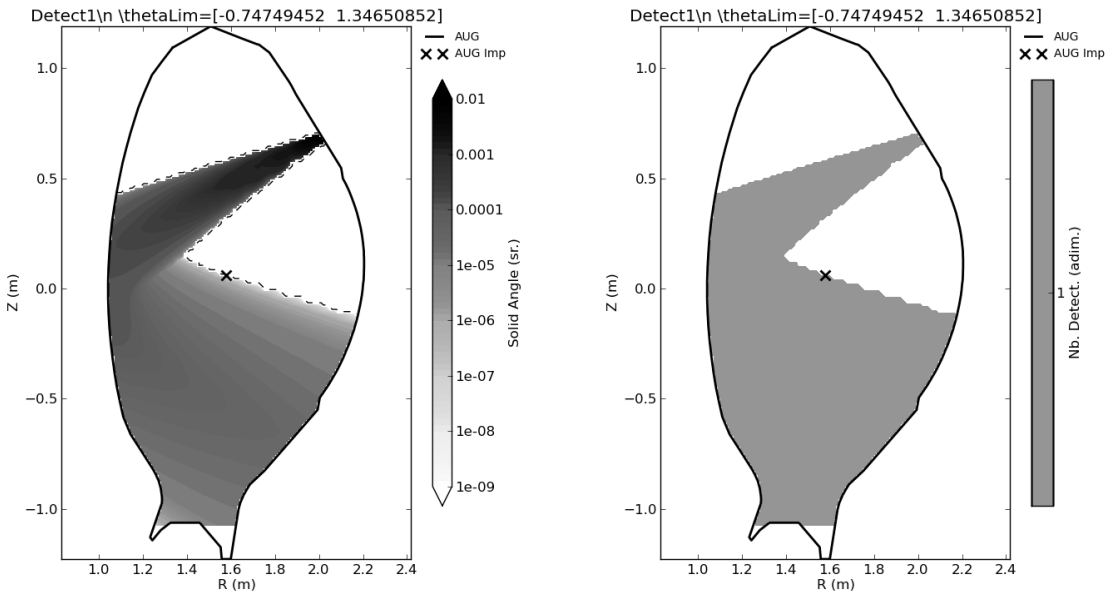

Figure 2.19: (Left) Contour plot of the solid angle subtended by the {Detector+Apertures} system in a poloidal projection with de-activated collision detection (Right) Number of detectors that can "see" each point (this will be useful for systems with several detectors)

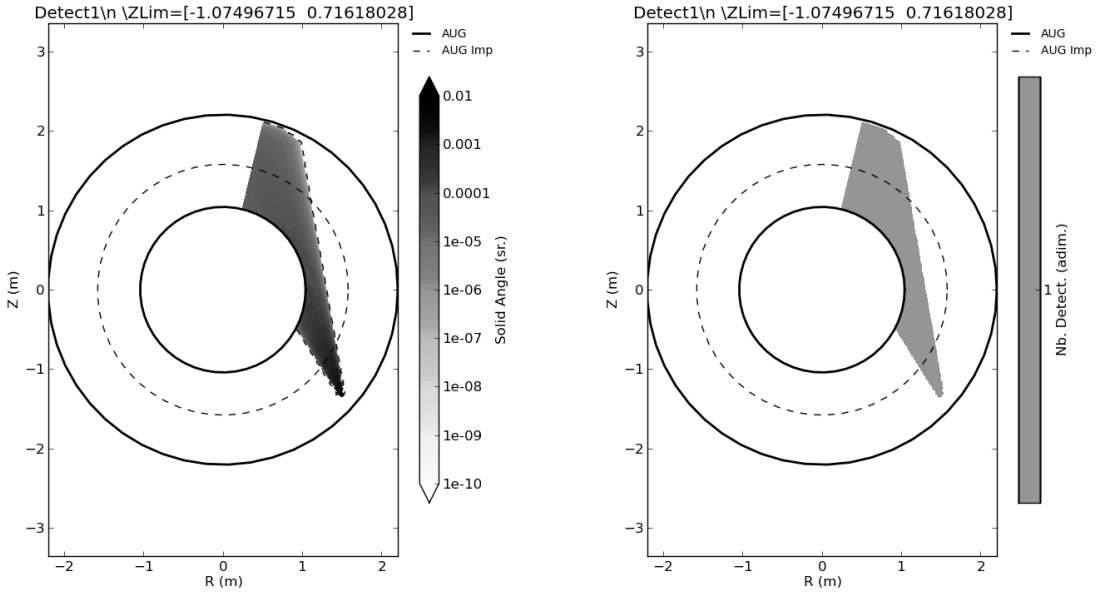

Figure 2.20: (Left) Contour plot of the solid angle subtended by the {Detector+Apertures} system in a horizontal - or toroidal - projection with de-activated collision detection (Right) Number of detectors that can "see" each point (this will be useful for systems with several detectors)

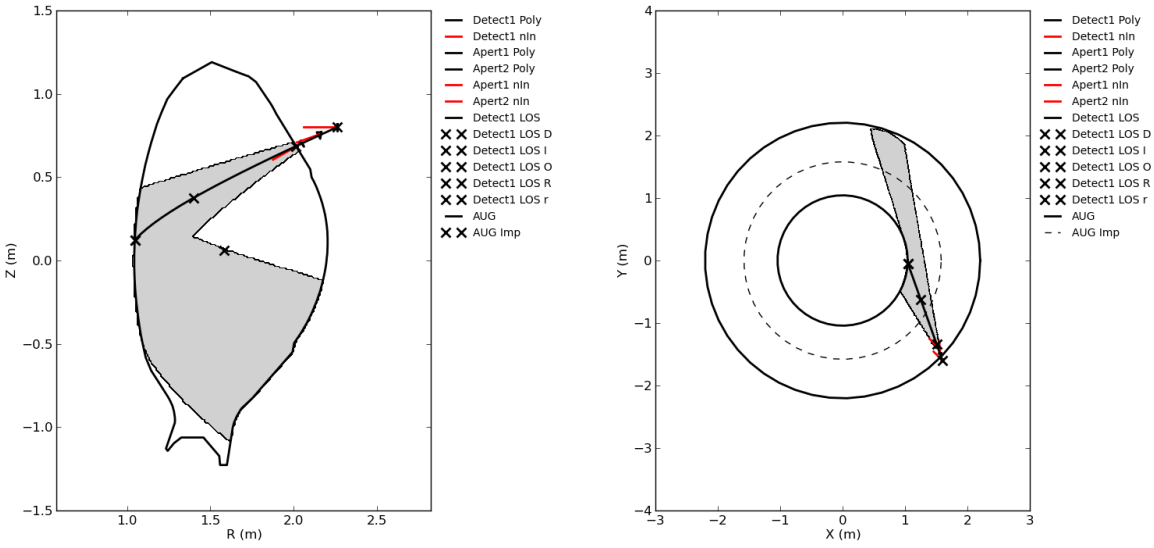

Figure 2.21: Poloidal and toroidal projections of Detect elements, this time including the projected polygons of the viewing cone

If you want to use the LOS approximation, you have to make sure it is valid. This approximation relies on several assumptions, one of which is that the etendue must remain constant along the LOS. We confirmed this in our case when we plotted it. However, we did not take into account the fact that a fraction only of the viewing stops where the LOS stops, and that the other fraction continues its way into the vacuum chamber. This means that there will be contributons to the signal which are not taken into account by the current LOS. An option could be to artificially extend the LOS through the central solenoid to the far end of the viewing cone, but this would still be unsufficient since the etendue that should be used for this extended part of the LOS is lower than the one we computed for the first part of the LOS. This type of situations, in which a fraction of the viewing cone is obstructed, corresponds to situations in which the etendue is in fact not constant along the entirety of the *extended* LOS (i.e.: extended to the far end of the viewing cone), as illustrated below. It reveals the limits to the LOS approximation and the advantages of a 3D description of the geometry.

```
ax = D1.plot_Etend_AlongLOS(NP=14, Length='kMax', Colis=True, Modes=['simps'], PlotL='abs', Ldict=[{
ax = D1.plot_Etend_AlongLOS(ax=ax, NP=14, Length='kMax', Colis=False, Modes=['simps'], PlotL='abs', L
ax = D1.plot_Etend_AlongLOS(ax=ax, NP=6, Length='POut', Colis=True, Modes=['simps'], PlotL='abs', Ld
plt.show()
```

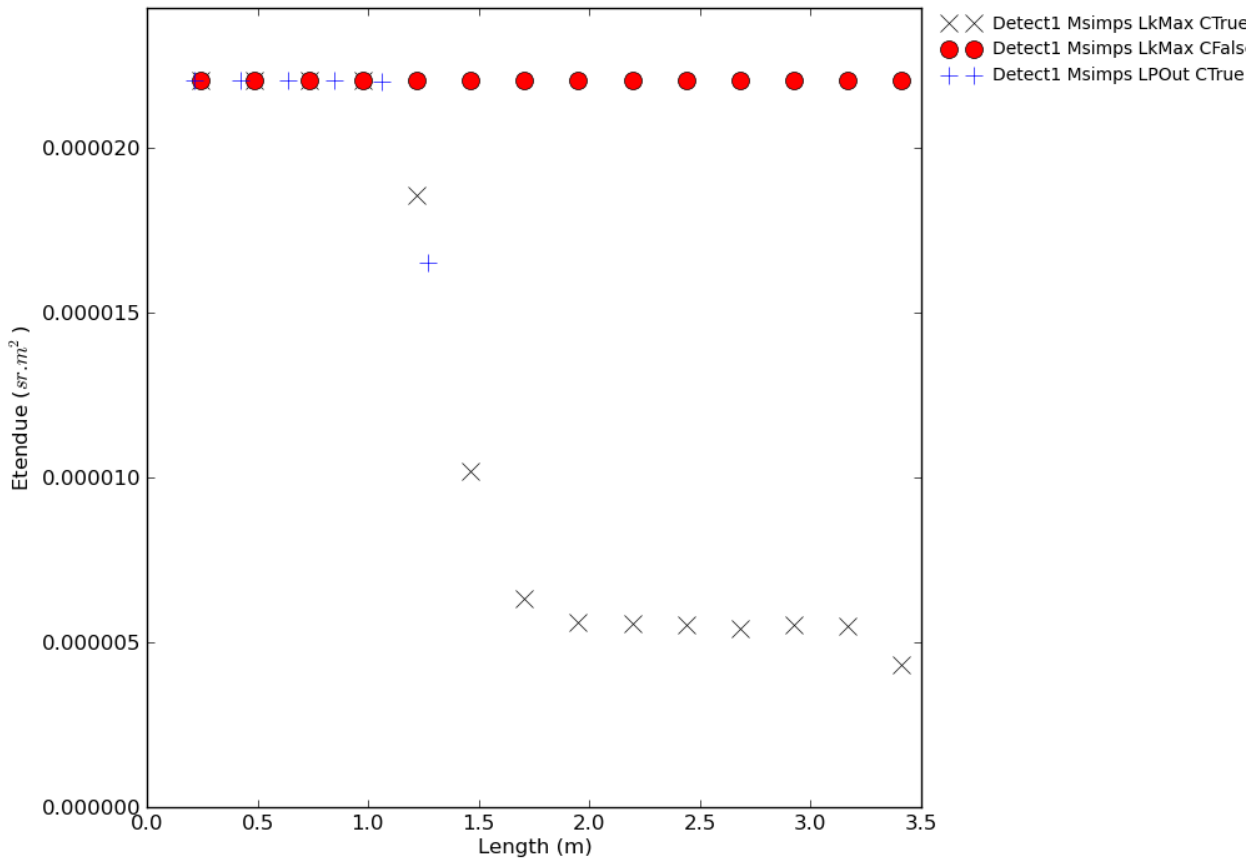

Figure 2.22: Etendue of the {Detector + Apertures} system as a function of the relative distance along the *extended* LOS, with and without taking into account collisions, and along the former LOS.

In addition to this effect, it also possible to visualise the difference between the LOS approximation and the real viewing cone by plotting the contour of the viewed volume in projection space, as illustrated below:

```
axImp = D1.plot_Impact_PolProj(Elt='DLT')
plt.show()
```

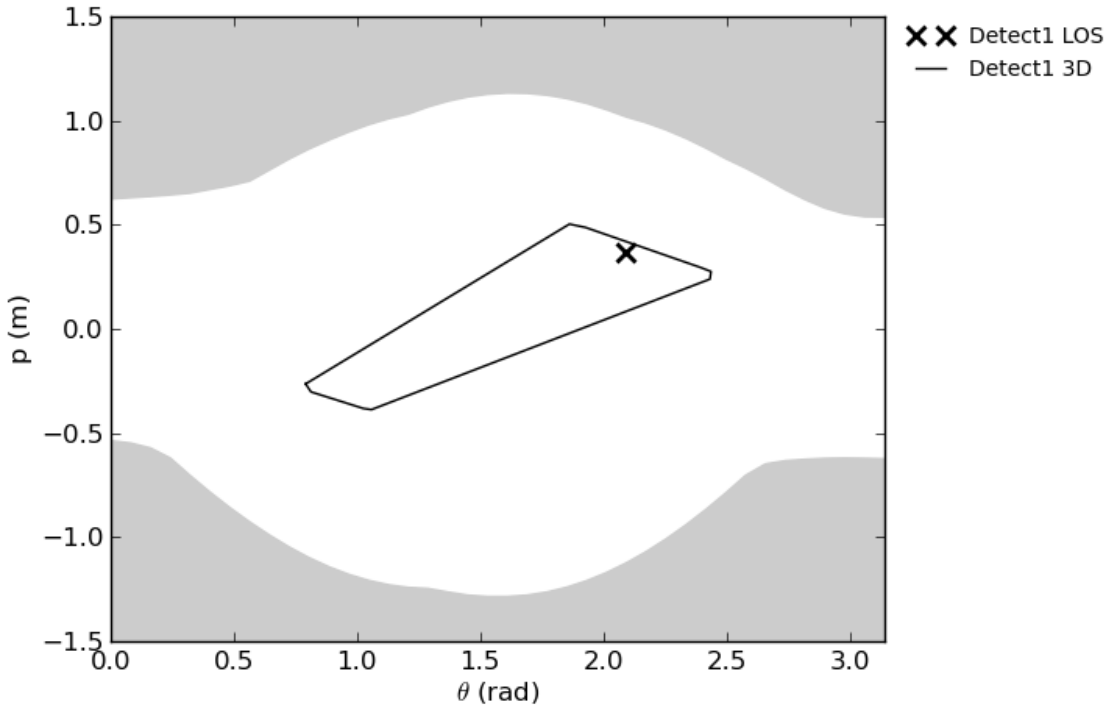

Figure 2.23: Etendue of the {Detector + Apertures} system as a function of the relative distance along the *extended* LOS, with and without taking into account collisions, and along the former LOS.

The more the area delimited by the contour is small, the better is the LOS approximation. We can clearly see here that the difference is significant. But it could nevertheless still be valid if the tomogram of the observed emissivity field was constant on this area (which is not the case in most standard situations).

More that visualisation or computation of the etendue, knowing the two projected polygons of the viewing cone is helpful for faster integration of signal in a synthetic diagnostic approach. Indeed, we know that all points which are not in both projected polygons are necessarily outside of the viewing cone. Hence, they can be used for fast discrimination of points which are useless for the signal.

Hence, the total incoming power on the detector for a given spectrally-integrated 3D emissivity field (provided as an input function of the position as a (3,1) numpy array) can be computed. As for the computation of the etendue, you can choose between three integration methods (via the "Mode" kwdarg), among which two discretisation methods and an adaptative algorithm (computation may be very long for high resolution discretisation with Colis=True). The following example shows a simple gaussian profile, constant on the toroidal direction:

```
def Emiss1(Points):
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Z = Points[2,:]
    Val = np.exp(-(R-1.68)**2/0.20**2 - (Z-0.05)**2/0.35**2) - 0.50*np.exp(-(R-1.65)**2/0.08**2 - (Z-
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return 1000.*Val
RR, ZZ = np.linspace(Tor2.PRMin[0], Tor2.PRMax[0],100), np.linspace(Tor2.PZMin[1], Tor2.PZMax[1],200)
RRf, ZZf = np.ones((200,1))*RR, ZZ.reshape((200,1))*np.ones((1,100))
```

```
Val = Emiss1(np.array([RRf.flatten()*np.cos(0), RRf.flatten()*np.sin(0), ZZf.flatten()]))
ax = Tor2.plot_PolProj(Elt='P')
Val[~Tor2.isinside(np.array([RRf.flatten(),ZZf.flatten()]))] = np.nan
ax.contourf(RRf, ZZf, Val.reshape((200,100)),50)
#plt.show()
SigLOS1, SigLOS2 = D1.calc_Sig(Emiss1, Method='LOS', Mode='quad'), D1.calc_Sig(Emiss1, Method='LOS',
SigCol = D1.calc_Sig(Emiss1, Colis=True, Mode='simps')
SigNoC = D1.calc_Sig(Emiss1, Colis=False, Mode='simps')
print "Signals :", SigLOS1, SigLOS2, SigCol, SigNoC

'Signals : [  8.34905419e-08] [  6.33159209e-08]'
```

As one can expect, the signal is higher when collisions with the Tor boundary are not considered because of the contribution from the plasma volume which should be hidden behind the central solenoid.

This direct approach is most accurate (provided sufficient discretisation of the integral) since it does not rely on a generic pre-defined spatial discretisation of the 3D emissivity on a mesh. Such discretisation is nonetheless necessary for tomographic inversions and allows for much faster synthetic diagnostic computation since the input emissivity function can be projected on so-called 'basis functions' with pre-computed contributions (via the so-called geometry matrix) to each detector. Spatial discretisation is addressed in the **ToFu_Mesh** module and the computation of the geometry matrix (both with a 3D and a LOS approach) is addressed in the **ToFu_MatComp** module.

But before, let us describe the last object class of **ToFu_Geom**, which is the Detect equivalent of the GLOS object class.

2.6 The GDetect object class

The GDetect object class provides an easy way to handle groups of detectors which have some features in common, like the GLOS object class does for LOS objects. It is basically a list of Detect objects with a common name and adapted methods for easily computing and plotting the characteristics of all detectors it contains with a single-line command. It also comes with selection methods to extract a sub-set of its Detect objects.

Table 2.7: The attributes of a GDetect object

Attribute	Description
self.ID	The ID class of the object
self.LDetect, self.nDetect	A list of Detect objects, which should have the same Tor object, and the number of Detect object it contains
self.BaryP, self.BaryS, self.S, self.nIn	The barycenter of self.Poly and its center of mass, its surface and the normalised vector perpendicular to the plane of self.Poly and oriented towards the interior of the Tor volume
self.Tor	The Tor object associated to the Detect object
self.LApert	A list of Apert objects associated to the Detect object

Naturally, the methods are similar to both the GLOS object and Detect object class. In the following, the GDetect object class is illustrated with the geometry of the F camera of the SXR diagnostic of ASDEX Upgrade. Once it is loaded as a **ToFu_Geom** GDetect instance, we can use the built-in methods to explore its characterisrics, like the etendue of each detector it is comprised of:

```
pathfileext = './Objects/TFG_GDetect_AUG_SXR_F_D20141202_T230455.pck'
with open(pathfileext, 'rb') as input:
    F = pck.load(input)

# Plot etendues
ax = F.plot_Etendues()
plt.show()
```

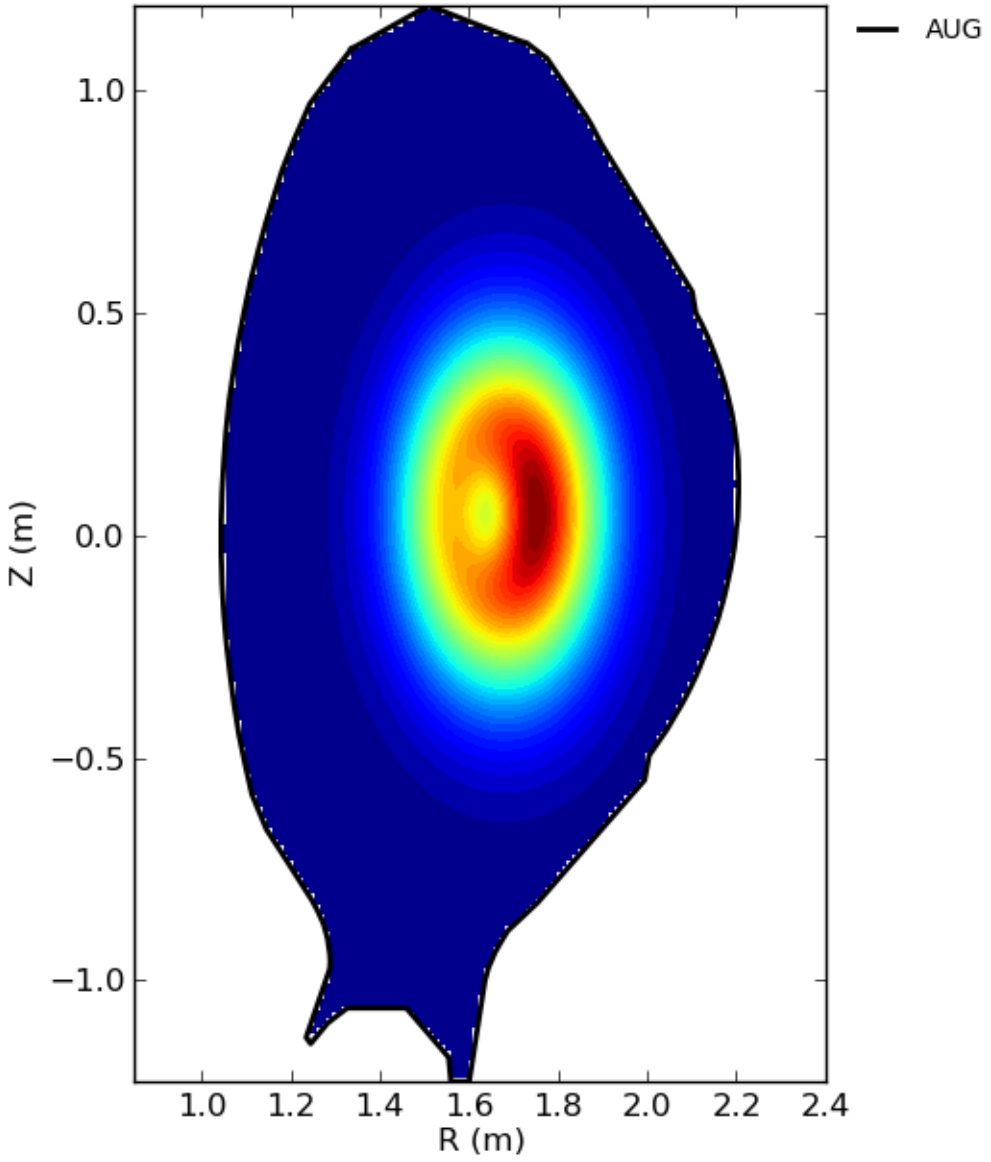

Figure 2.24: Fake (double gaussian) SXR emissivity field (toroidally invariant) which resembles one of the typical cases of ASDEX Upgrade

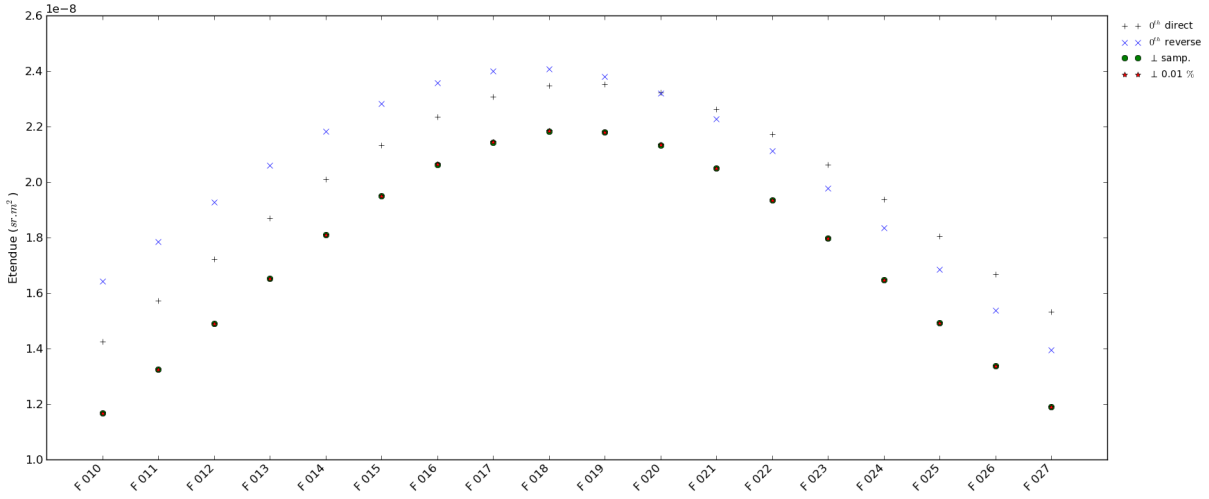

Figure 2.25: Etendues of the detectors composing the F camera of ASDEX Upgrade, computed using the usual 0-order approximation in both ways (direct and reverse), and using a complete integration with an adaptative algorithm (with reltive error < 0.01 %) and a sampling algorithm.

We can also visualise the lines of sight and projected viewing cones of all the detectors. In the following example, we use the 'Elt'-type kwdrg to specify that we first want to plot the viewing cone and the polygon constituting the detectors ('CP'), with the polygons of the apertures ('P') and the reference polygon of the Tor ('P'), but no LOS (''). Then we plot the LOS ('L') but not the viewing cones.

```
axP1, axT1 = F.plot_AllProj(Elt='CP',EltApert='P',EltLOS='',EltTor='P')
axP2, axT2 = F.plot_AllProj(Elt='P',EltApert='P',EltLOS='L',EltTor='P')
plt.show()
```

We can also select one particular detector to plot it only. To do this we can use the dedicated routine which return the index of a detector reckognizable by one its ID attributes (its name, its signal code, its savename or any ID attribute that you have previously passed).

```
ind = F.get_ind_Detect(IDAttr='Name',IDExp="=='F_021'").nonzero()[0]
axP3, axT3 = F.LDetect[ind].plot_AllProj(Elt='CP',EltApert='',EltLOS='L',EltTor='P')
plt.show()
```

It is also interesting to plot the LOS and viewing cones in projection space, to see how a realistic diagnotic looks like in this representation and see how far we are from a pure LOS (specifying we want the LOS 'L', the viewing cone 'C' and the Tor enveloppe 'T'):

```
ax = F.plot_Impact_PolProj(Elt='CLT')
plt.show()
```

We can see that the surfaces corresponding to the viewing cone are resonably small (and quite elongated), which is an indication that the LOS approximation seems a reasonable hypothsesis from a purely geometrical point of view, but of course, in practice it also depends on the nature / shape of the observed emissivity field.

When it comes to computing the signal of each detector associated to an arbitrary input emissivity field, one must keep in mind that while the LOS approximation allows for fast but approximate computation, a full-3D approach ives an accurate result, but is much slower. While fractionaof second is sufficient for a LOS computation, several minutes can be necessary for each detector for a full 3D computation. Of course, it depends on the volume which is inside the viewing cone and on the level of accuracy to be obtained. The method used is simple sampling in cartesian coordinates of the viewing cone. The default is a uniform grid of 5mmx5mmx5mm, which appears sufficient for most

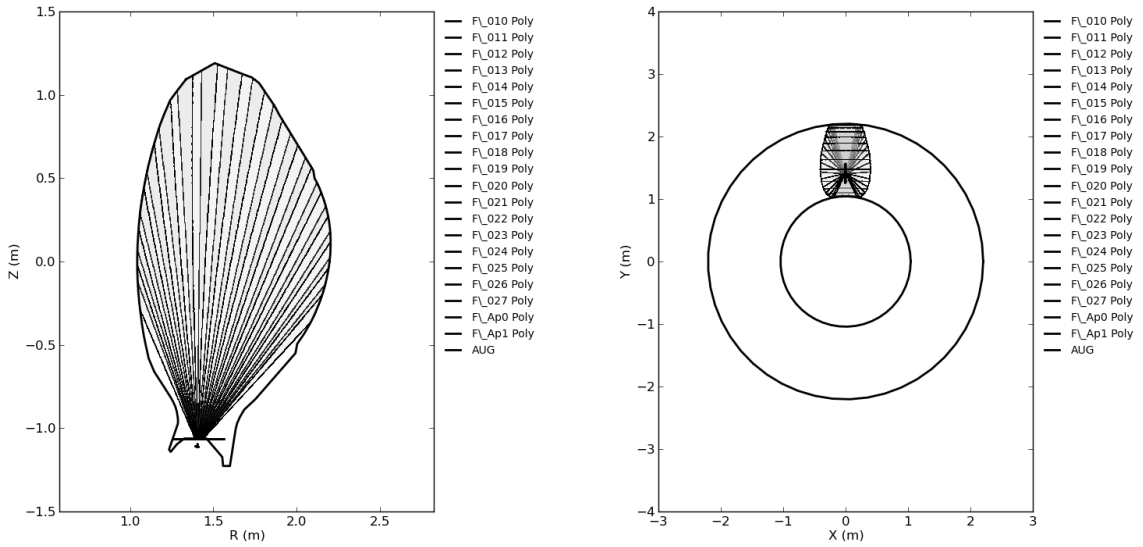

Figure 2.26: Poloidal and toroidal projections of the geometry of F, with the viewing cones

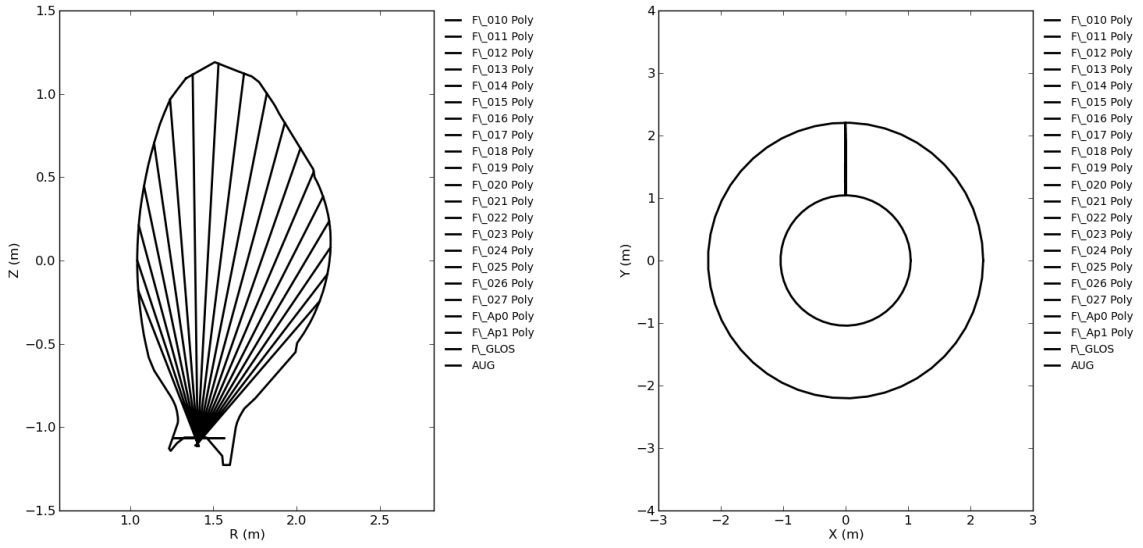

Figure 2.27: Poloidal and toroidal projections of the geometry of F, with the LOS

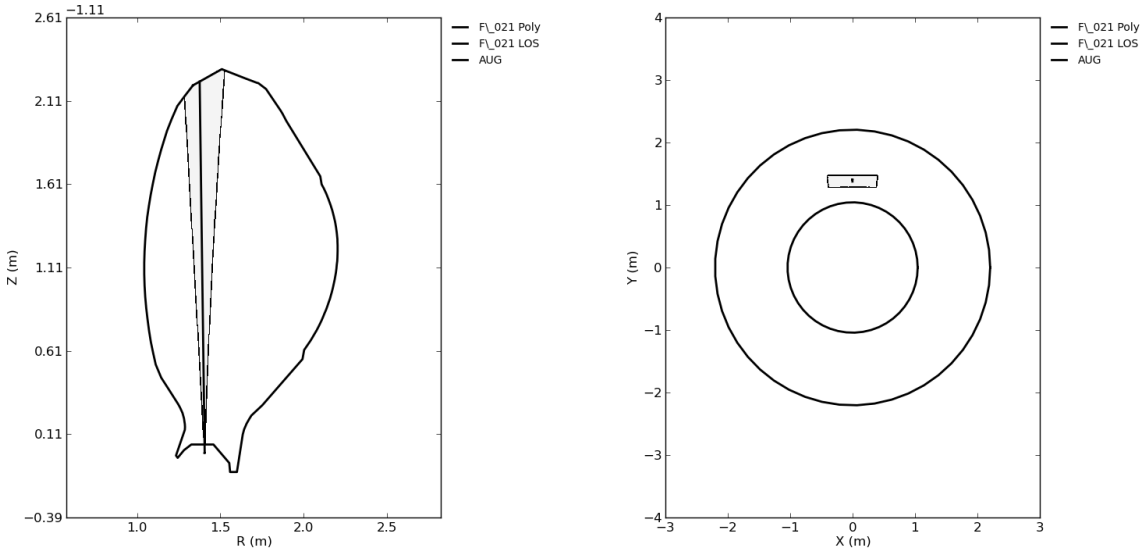

Figure 2.28: Poloidal and toroidal projections of the geometry of of one particular detector of F

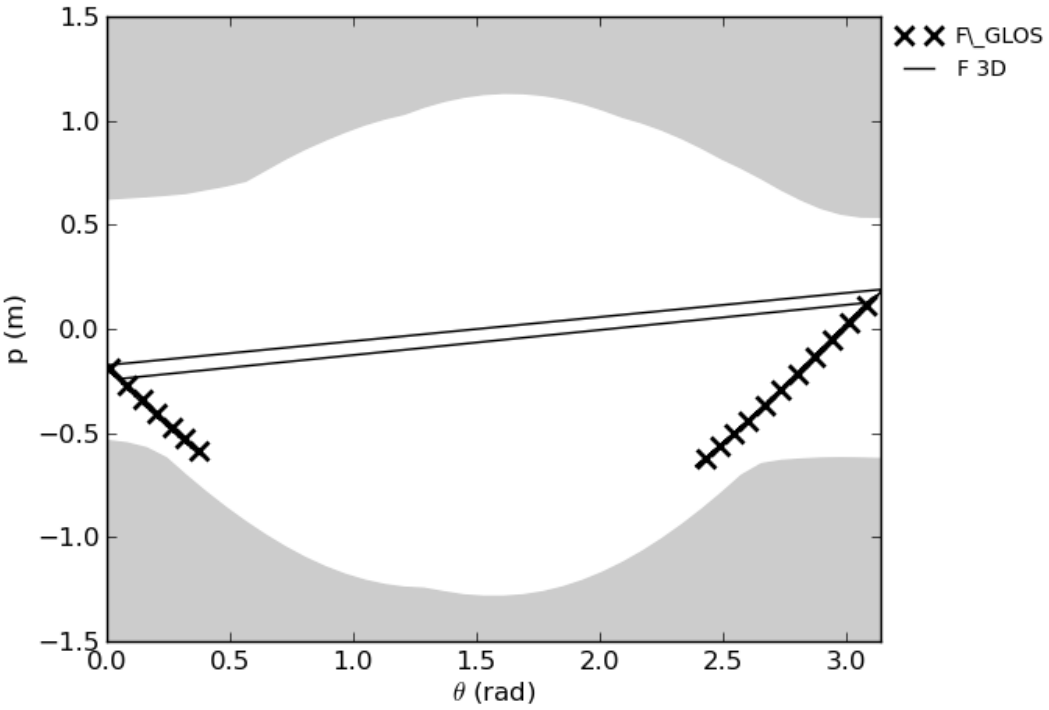

Figure 2.29: Representation in projection space of both the LOS and the viewing cones of F, with the Tor enveloppe. One detector spans from theta values close to pi to values close to 0, which explains the boundaries of its associated cone streching from one end of the graph to the other (in reality, it should be separated in two polygons on this graph).

standard cases. But the user can choose his own grid size by using the 'dX12' (=[0.005,0.005] by default, in the plane perpendicular to the LOS) and 'ds' (=0.005 by default, along the LOS). Since the user may often need to evaluate the signal not only once but several times for each detector (fro example to plot the time evolution of the signal), it is possible to store a pre-computed grid (the solid angle, which is the longest value to calculate, is pre-computed) and use it for all the successive computations (the pre-computed solid angle is then simply multiplied by the local value of the input emissitivity and integration is performed by summation and multiplication by the elementary volume).

An example is given below, where three input emissivity fields are provided. The first one is toroidally constant, the local maximum of the second one rotates as if it were a hot spot on the q=1 surface, and the last one is toroidally constant but has an anisotropic radiation (it radiates 100 times more in the toroidal direction).

```
# Define toroidally constant emissivity
def Emiss1(Points):
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Z = Points[2,:]
    Val = np.exp(-(R-1.68)**2/0.20**2 - (Z-0.05)**2/0.35**2) - 0.50*np.exp(-(R-1.65)**2/0.08**2 - (Z-
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return 1000.*Val

# Define toroidally variable emissivity
def Emiss2(Points):
    ROff = 0.05
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Theta = np.arctan2(Points[1,:],Points[0,:])
    Z = Points[2,:]
    CentR = 1.68+ROff*np.cos(Theta)
    CentZ = 0.05+ROff*np.sin(Theta)
    Val = np.exp(-(R-1.68)**2/0.20**2 - (Z-0.05)**2/0.35**2) - 0.50*np.exp(-(R-CentR)**2/0.08**2 - (Z
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return 1000.*Val

# Define anisotropic emissivity
def Emiss3(Points, Vect):
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Theta = np.arctan2(Points[1,:],Points[0,:])
    Z = Points[2,:]
    Cos = -np.sin(Theta)*Vect[0,:] + np.cos(Theta)*Vect[1,:]
    Sin2 = Vect[2,:]**2 + (np.sin(Theta)*Vect[1,:] + np.cos(Theta)*Vect[0,:])**2
    sca = 100.*Cos**2+1.*Sin2
    Val = np.exp(-(R-1.68)**2/0.20**2 - (Z-0.05)**2/0.35**2) - 0.50*np.exp(-(R-1.65)**2/0.08**2 - (Z-
    Val = Val*sca
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return 1000.*Val
```

Since we know we are going to use the same grid several times, we pre-compute it (using the default parametrisation), the pre-computed matrix is then automatically assigned as a new attibute of each Detect object (this may take 2-5 min for each detector ToDo : implement a full C-version of the bottleneck routines for faster computation):

```
# Pre-compute the grid
F.set_SigPreCompMat()
```

And then we compute the LOS and 3D signals, specifying that we want to use the pre-computed grid for faster computation (now the computation should take less than a second for each detector):

```
Sig1, Sig2 = F.calc_Sig(Emiss1, Method='LOS',Mode='quad'), F.calc_Sig(Emiss1, Method='Vol',Mode='simp
ax = F.plot_Sig(Sig1)
```

```
ax.plot(np.arange(1,F.nDetect+1), Sig2 ,label='Vol', c='r')
plt.show()
```

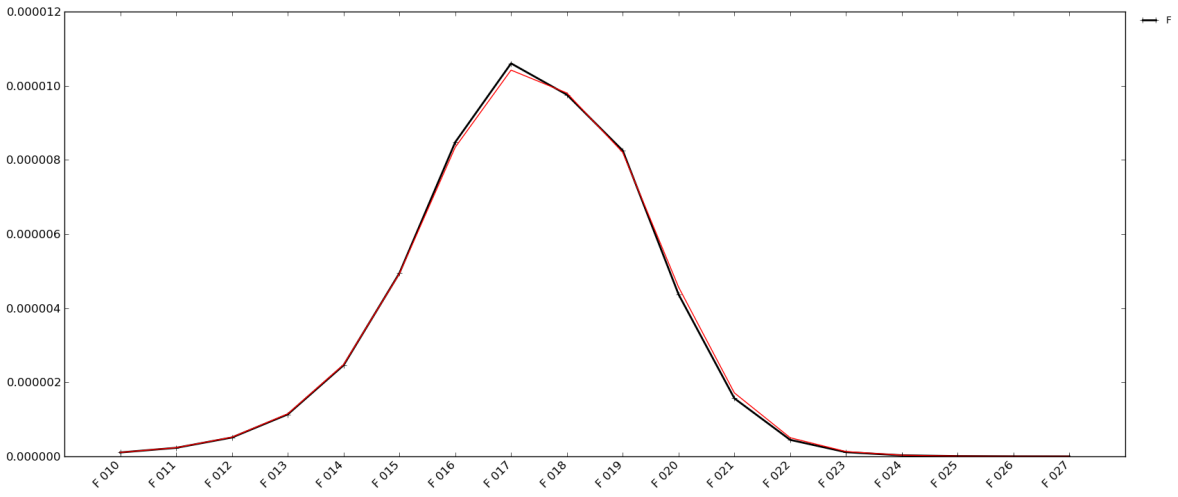

Figure 2.30: The integrated signals of camera F with a toroidally constant input emissivity (both with a LOS and 3D approach)

It can be seen that even for toroidally constant emissivity, there are some small differences between the pure LOS integration and the full 3D computation (of the order of 1-2 % in the most central LOS, and up to 10 % near the edge). In order to check that these differences are real and are not due to discretization errors or bad implementation of the 3D integrating algorithm, we can do the following: we provide as an input an emissivity field that only varies with Z. Indeed the F camera is mostly looking upward, hence, if the emissivity field only changes with Z, the validity of the LOS approximation should be very good and the difference the LOS and 3D integrations should be minimal since the emissivity is indeed quasi-constant on planes perpendicuar to the LOS.

```
def EmissZ(Points):
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Z = Points[2,:]
    Val = np.exp(-(Z-0.05)**2/0.35**2)
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return 1000.*Val
```

We can see here indeed that the agreement is particularly good for the most central LOS (which are the most vertical, hence the LOS approximation holds mostly for them), and less for the edge LOS which are more and more inclinated with respect to the iso-emissivity surfaces.

We can now try to do the same for the second input emissivity (with “m=1-like” perturbation):

```
Sig1, Sig2 = F.calc_Sig(Emiss2, Method='LOS',Mode='quad'), F.calc_Sig(Emiss2, Method='Vol',Mode='simp
ax = F.plot_Sig(Sig1)
ax.plot(range(1,F.nDetect+1), Sig2 ,label='Vol', c='r')
ax.figure.savefig(RP+"/../doc/source/figures_doc/Fig_Tutor_ToFuGeom_GDetect_Sig2.png",frameon=None,bb
plt.show()
```

We observe that the change with respect to the toroidally constant emissivity is not dramatic, which can be explained by the averaging effect of the cone of sight.

And finally for the anisotropic emissivity, note that in this case we have to specify to the method that the emissivity is

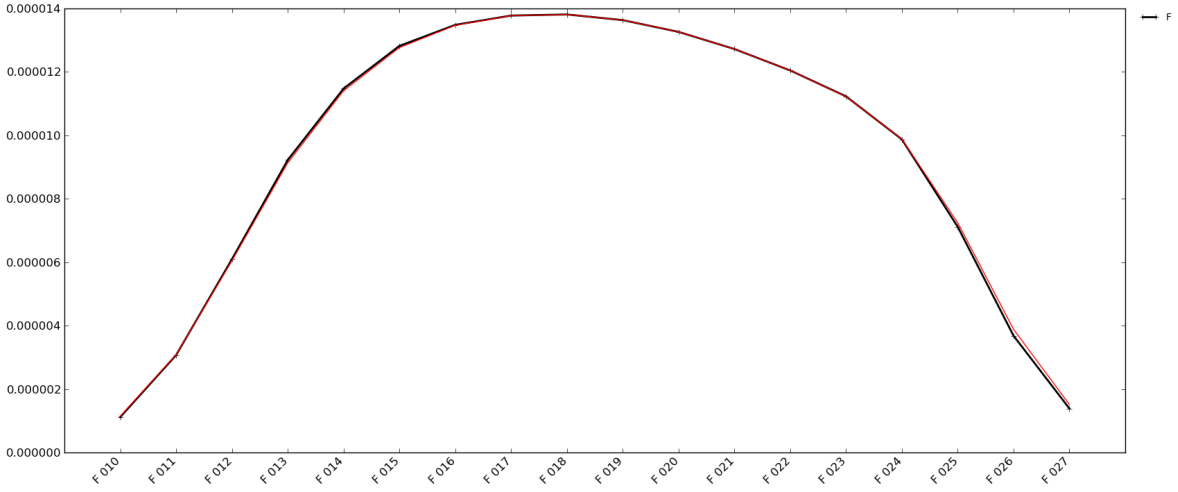

Figure 2.31: The integrated signals of camera F with a horizontally constant input emissivity (both with a LOS and 3D approach)

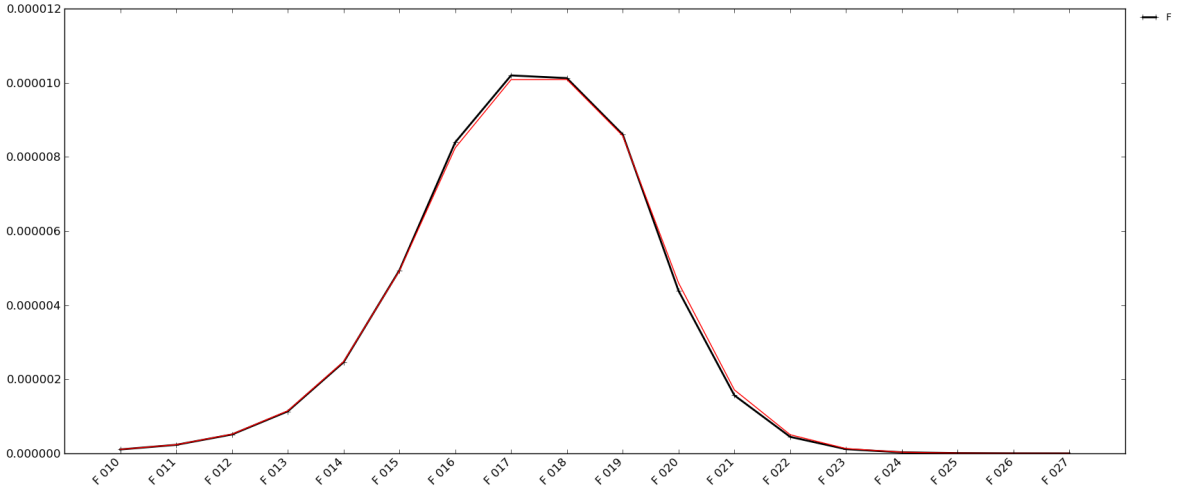

Figure 2.32: The integrated signals of camera F with a toroidally varying input emissivity (both with a LOS and 3D approach)

anisotropic.

```
Sig1, Sig2 = F.calc_Sig(Emiss3, Ani=True, Method='LOS',Mode='quad'), F.calc_Sig(Emiss3, Ani=True, Met
ax = F.plot_Sig(Sig1)
ax.plot(range(1,F.nDetect+1), Sig2 ,label='Vol', c='r')
ax.figure.savefig(RP+"/../doc/source/figures_doc/Fig_Tutor_ToFuGeom_GDetect_Sig3.png",frameon=None,bb
plt.show()
```

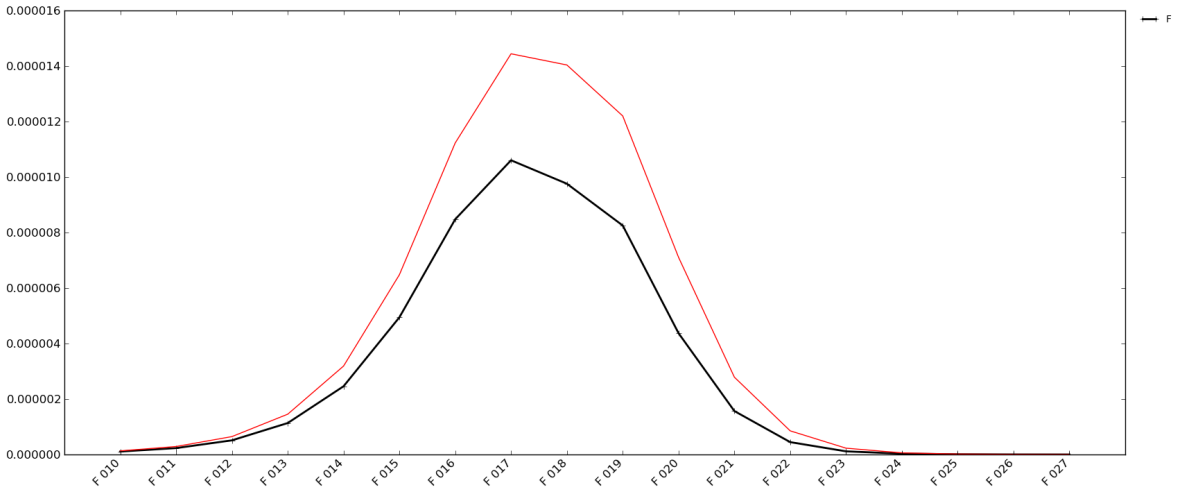

Figure 2.33: The integrated signals of camera F with an anisotropic input emissivity (both with a LOS and 3D approach)

Unsurprisingly, this case displays the most dramatic differences between the LOS approach, intrinsically limited, and the 3D computation. The observed differences range from a few percents (very small anisotropy) to several orders of magnitude (purely forward-domiated radiation). The next challenge is to determine whether we will able to reconstruct such anisotropies in an inverse-problem approach, which will be addressed in **ToFu_MatComp** and **ToFu_Inv**.

CHAPTER

THREE

TOFU_MESH

(This project is not finalised yet, work in progress...)

ToFu_Mesh, is a ToFu module aimed at handling spatial discretisation of a 3D scalar field in a vacuum chamber (typically the isotropic emissivity of a plasma). Such discretisation is done using B-splines of any order relying on a user-defined rectangular mesh (possibily with variable grid size). It is particularly useful for tomographic inversions and fast synthetic diagnostics.

It is designed to be used jointly with the other **ToFu** modules, in particular with **ToFu_Geom** and **ToFu_MatComp**. It is a ToFu-specific discretisation library which remains quite simple and straightforward. However, its capacities are limited to rectangular mesh and it may ultimately be percieved as a much less powerful version of **PIGASUS/CAID**. Users who wish to use **ToFu** only for tomographic inversions may find **ToFu_Mesh** sufficient for thir needs, others, who wish to use a synthetic diagnostic approach, and/or to use **ToFu_Mesh** jointly with plasma physics codes (MHD...) may prefer using **PIGASUSCAID** for spatial discreatisation.

Hence, **ToFu_Mesh** mainly provides two object classes : one representing the mesh, and the other one (which uses the latter) representing the basis functions used for discretisation:

Table 3.1: The object classes in ToFu_Geom

Name	Description	Inputs needed
ID	An identity object that is used by all **ToFu** objects to store specific identity information (name, file name if the object is to be saved, names of other objects necessary for the object creation, date of creation, signal name, signal group, version...)	By default only a name (a character string) is necessary, A default file name is constructed (including the object class and date of creation), but every attribute can be modified and extra attribute can be added to suit the specific need of the the data acquisition system of each fusion experiment or the naming conventions of each laboratory.
Mesh1D, Mesh2D, Mesh3D	1D, 2D and 3D mesh objects, storing the knots and centers, as well as the correspondence between knots and centers in both ways. The higher dimension mesh objects are defined using lower dimension mesh objects. The Mesh 2D object includes an enveloppe polygon. They all include plotting methods and methods to select a subset of the total mesh. The Mesh 3D object is not finished.	A numpy array of knots, which can be defined using some of the functions detailed below (for easy creation of linearly spaced knots with chosen resolution).
BaseFun1D, BaseFunc2D, BaseFunc3D	1D, 2D and 3D families of B-splines, relying on Mesh1D, Mesh2D, Mesh3D objects, with chosen degree and multiplicity for each dimension. Includes methods for plotting, for determining the support and knots and centers associated to each basis function, as well as for computing 1st, 2nd or 3rd order derivatives (as functions), and local value (summation of all basis functions or their derivatives at a given point and for given weights). Includes methods for computing integrals of derivative operators...	A Mesh object of the adapted dimension, and a degree value.

The following will give a more detailed description of each object and its attributes and methods through a tutorial at the end of which you should be able to create your own mesh and basis functions and access its main characteristics.

3.1 Getting started with ToFu_Mesh

Once you have downloaded the whole **ToFu** package (and made sur you also have scipy, numpy and matplotlib, as well as a free polygon-handling library called Polygon which can be downloaded at), just start a python interpreter and import **ToFu_Geom** and **ToFu_Mesh** (we will always import **ToFu** modules 'as' a short name to keep track of the functionalities of each module). To handle the local path of your computer, we will also import the small module called **ToFu_PathFile**, and **matplotlib** and **numpy** will also be useful:

```
import numpy as np
import matplotlib.pyplot as plt
import ToFu_Geom as TFG
import ToFu_Mesh as TFM
import ToFu_PathFile as TFPF
import os
import cPickle as pck # for saving objects
import ToFu_Defaults as TFD
```

The os module is used for exploring directories and the cPickle module for saving and loading objects.

3.2 The Mesh1D, Mesh2D and Mesh3D object classes

In this section, we describe the Mesh objects starting from the unidimensional to the 3D version.

Table 3.2: The attributes of a Mesh1D object

Attribute	Description
self.ID	The ID class of the object
self.NCents, self.NKnots	The number of mesh elements or centers (resp. knots) of the object (typically self.NKnots = self.NCents+1)
self.Cents, self.Knots	The coordinates of the centers and knots themselves, as two numpy arrays
self.Lengths, self.Length, self.BaryL, self.BaryP	The length of each mesh element, the total length of the mesh and the center of mass of the mesh (i.e.: weight by the respective length of each mesh element), and the barycenter of the self.Cents
self.Cents_Knotsind, self.Knots_Centsind	The index arrays used to get the correspondence between each mesh element (resp, each knot) and its associated knots (resp. its associated mesh elements)

Table 3.3: The attributes of a Mesh2D object

Attribute	Description
self.ID	The ID class of the object
self.MeshR, self.MeshZ	The two Mesh1D objects used to create this Mesh2D object
self.NCents, self.NKnots	The number of mesh elements or centers (resp. knots) of the object (typically self.NKnots = self.NCents+1)
self.Cents, self.Knots	The coordinates of the centers and knots themselves, as two numpy arrays
self.Surfs, self.Surf, self.VolAngs, self.VolAng, self.BaryV, self.BaryS, self.BaryL, self.BaryP	The surface of each mesh element, the total surface of the mesh, the volume per unit angle of each mesh element, the total volume per unit angle, the volume barycenter of the mesh (i.e. taking into account not only the surface repartition but also the toroidal geometry), the center of mass of the mesh (i.e.: weight by the respective surface of each mesh element), the middle point (the average between the extreme (R,Z) coordinates) and the barycenter of all the self.Cents
self.Cents_Knotsind, self.Knots_Centsind	The index arrays used to get the correspondence between each mesh element (resp, each knot) and its associated knots (resp. its associated mesh elements)
self.BoundPoly	The boundary polygon of the mesh, useful for fast estimation whether a point lies inside the mesh support or not.

In an experiment-oriented perspective, **ToFu_Mesh** comes with simple functions to help you quickly define an optimal 1D grid, with explicit parametrisation of the spatial resolution on regions of interest. For example, if you want to define a 1D grid with a 5 cm resolution near the first end, that gradually refines to 1 cm at a given point, stays 1 cm for a given length and is then gradually enlarged to 6 cm at the other end, you just have to feed in the points of interest and their associated resolution to the *LinMesh_List* function, as a two lists of corresponding (start,end) tuples.

```
#Knots, Res = TFM.LinMesh_List([(1.,1.5),(1.5,1.8),(1.8,2.)], [(0.06,0.02),(0.02,0.02),(0.02,0.08)])
#print Res
#print Knots

# [(0.0569230769230769, 0.02), (0.02, 0.02), (0.02, 0.07999999999999999)]
# [ 1.          1.05692308  1.11076923  1.16153846  1.20923077  1.25384615  1.29538462  1.33384615
```

You can then feed the resulting knots numpy array to the Mesh1D object class and use this object methods to access all the features of interest of the created mesh:

```
M1 = TFM.Mesh1D('M1', Knots)
ax1 = M1.plot(Elt='KCN')
ax2 = M1.plot_Res()
#plt.show()
```

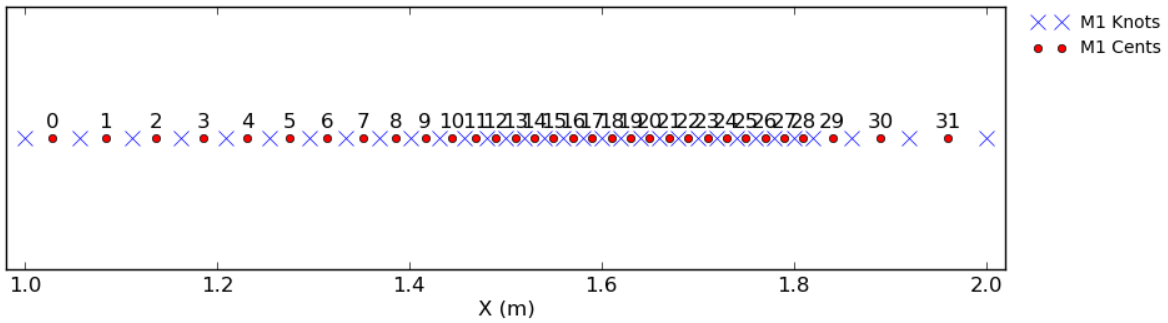

Figure 3.1: Arbitrary 1D mesh with customized resolution in chosen regions

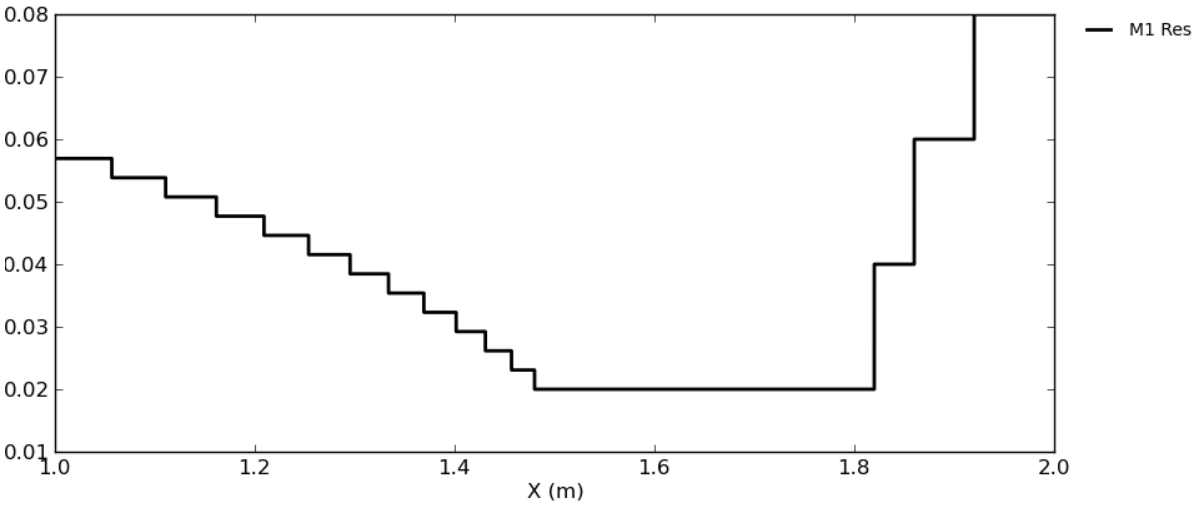

Figure 3.2: Local spatial resolution of the created 1D mesh

It can seen that the algorithm tried to render a mesh with the required resolution, even though it had to decrease it slightly around the first point, where it is lower than the required 6 cm (this is necessary due to the necessity to the number of mesh elements which must be an integer, thus leading to rounding). This is shown also in the *Res* variable which returns the actual resolution. Like for the **ToFu_Geom** plotting routines, the 'Elt' keyword argument provides you with the possibility of choosing what is going to be plotted (the knots 'K', the centers 'C' and/or the numbers 'N').

The Mesh2D object class relies on the same basics, except that its multi-dimensional nature means that it has extra methods for easy handling of mesh elements. Let us for example create a coarse 2D mesh using 2 different 1D mesh objects:

```
PolyRef = np.loadtxt(RP + '/Inputs/AUG_Tor.txt', dtype='float', comments='#', delimiter=None, convert
AUG = TFG.Tor('AUG',PolyRef)
KnotsR, ResR = TFM.LinMesh_List([(AUG.PRMin[0,0],1.5),(1.5,1.75),(1.75,AUG.PRMax[0,0])], [(0.06,0.02)
KnotsZ, ResZ = TFM.LinMesh_List([(AUG.PZMin[1,0],-0.1),(-0.1,0.1),(0.1,AUG.PZMax[1,0])], [(0.10,0.02)
M2 = TFM.Mesh2D('M2', [KnotsR,KnotsZ])
#ax = M2.plot(Elt='MBKC')
#plt.show()
```

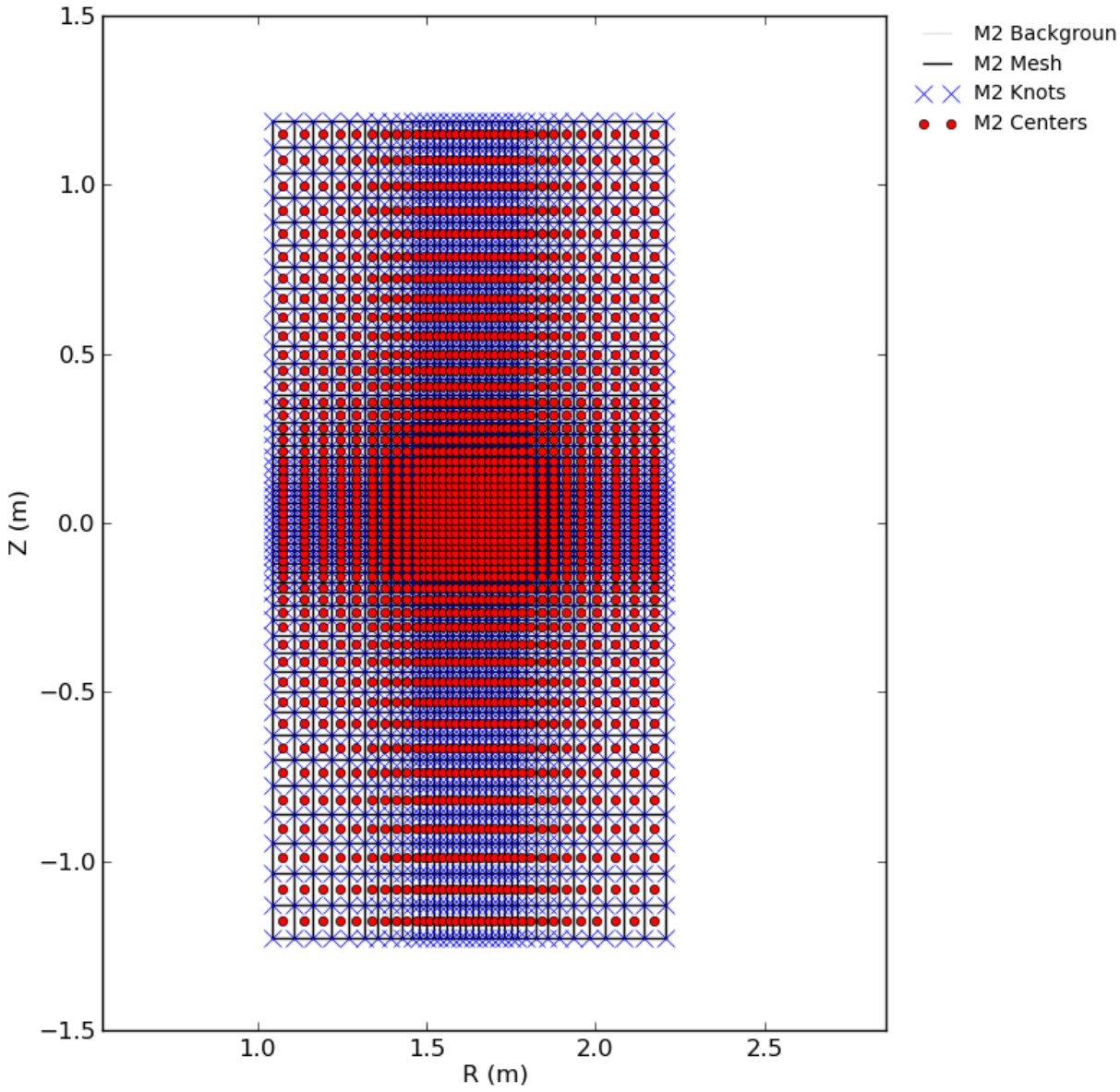

Figure 3.3: Arbitrary 2D mesh with customized resolution in chosen regions

The Mesh2D class comes with a method to automatically create another Mesh2D object that can be seen as a sub-mesh (only the elements lying inside an input polygon are kept, the rest being memorized only as 'Background'). In our example, we can use a specific method of the TFG.Tor object class to create a smooth convex polygon lying inside the Tor enveloppe (see the kwdargs for customization of the smoothing and offset) to concentrate on the region where

most SXR radiation comes from:

```
Poly = AUG.get_InsideConvexPoly(Spline=True)
M2bis = M2.get_SubMeshPolygon(Poly, NLim=2)
#ax = AUG.plot_PolProj(Elt='P')
#ax = M2bis.plot(Elt='BM', ax=ax)
#ax1, ax2, ax3, axcb = M2bis.plot_Res()
#plt.show()
```

Here, the 'NLim' kwdarg is used to specifiy how many corners of a mesh element must lie inside the input polygon so that this mesh element can be counted in.

Now, the Mesh2D object class provides tools to easily select and plot chosen elements of the 2D mesh. For example, if you want to get the coordinates of the four knots associated to the mesh element number 50, you can use the attribute 'Centers_Knotsind' to get them, and then plot them:

```
Knots50 = M2bis.Knots[:,M2bis.Cents_Knotsind[:,50].flatten()]
print Knots50
ax = M2bis.plot_Cents(Ind=50, Elt='BMKC')
#plt.show()

# [[ 1.69230769  1.71153846  1.71153846  1.69230769]
#  [-0.94421053 -0.94421053 -0.85868421 -0.85868421]]
```

Similarly, you can get and plot all the mesh element centers associated to knots number 160, 655 and 1000:

```
ind = np.array([160,655,1000])
Cents = M2bis.Cents[:,M2bis.Knots_Centsind[:,ind].flatten()]
print Cents
ax = M2bis.plot_Knots(Ind=ind, Elt='BMKC')
plt.show()

# [[ 1.83922727  1.07454545  1.70192308  1.87418182  1.13548182  1.72115385   1.83922727  1.07454545
#  [-0.66452632 -0.05        0.13140693 -0.66452632 -0.05        0.13140693  -0.59428947 -0.03
```

The Mesh3D object class is currently being built... to be finished.

Now that we have access to a mesh, we can build basis functions on it. The basis functions available in **ToFu_Mesh** are all B-splines, as illustrated below.

3.3 The BaseFunc1D, BaseFunc2D and BaseFunc3D object classes

The use of B-spline allows for more flexibility and more accuracy than the standard pixels (which are B-splines of degree 0). Indeed, most of the tomographic algorithms using series expansion in physical space assess the regularity of the solution by computing the integral of a norm of one of its derivatives. While the use of pixels forces you to use discrete approximations of the derivative operators, the use of B-splines of sufficient degree allows to use an exact formulation of the derivative operators.

The attributes of a BaseFunc1D objects are the following:

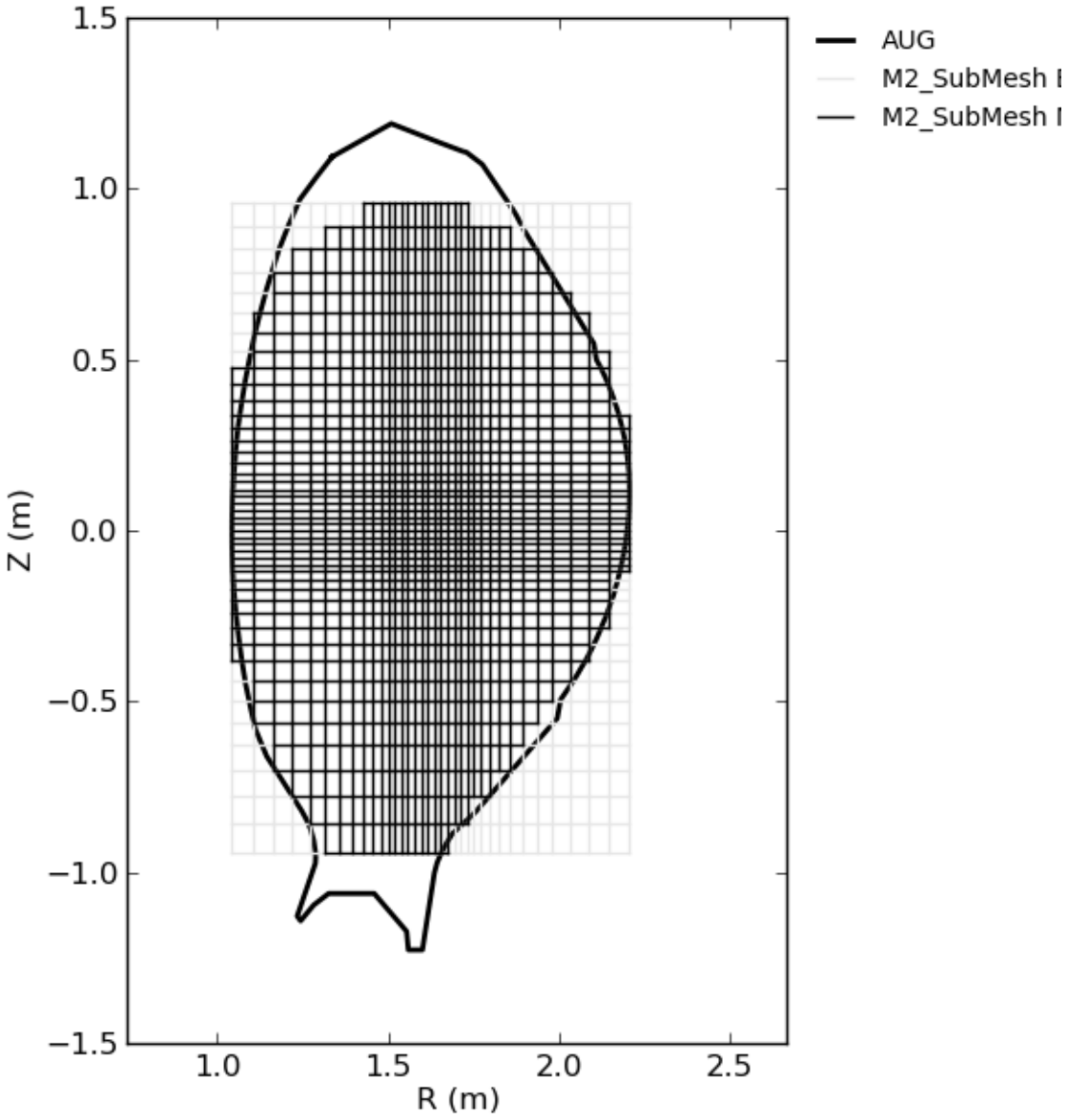

Figure 3.4: Submesh of the 2D mesh with customized resolution in chosen regions with selected elements only (using an input polygon)

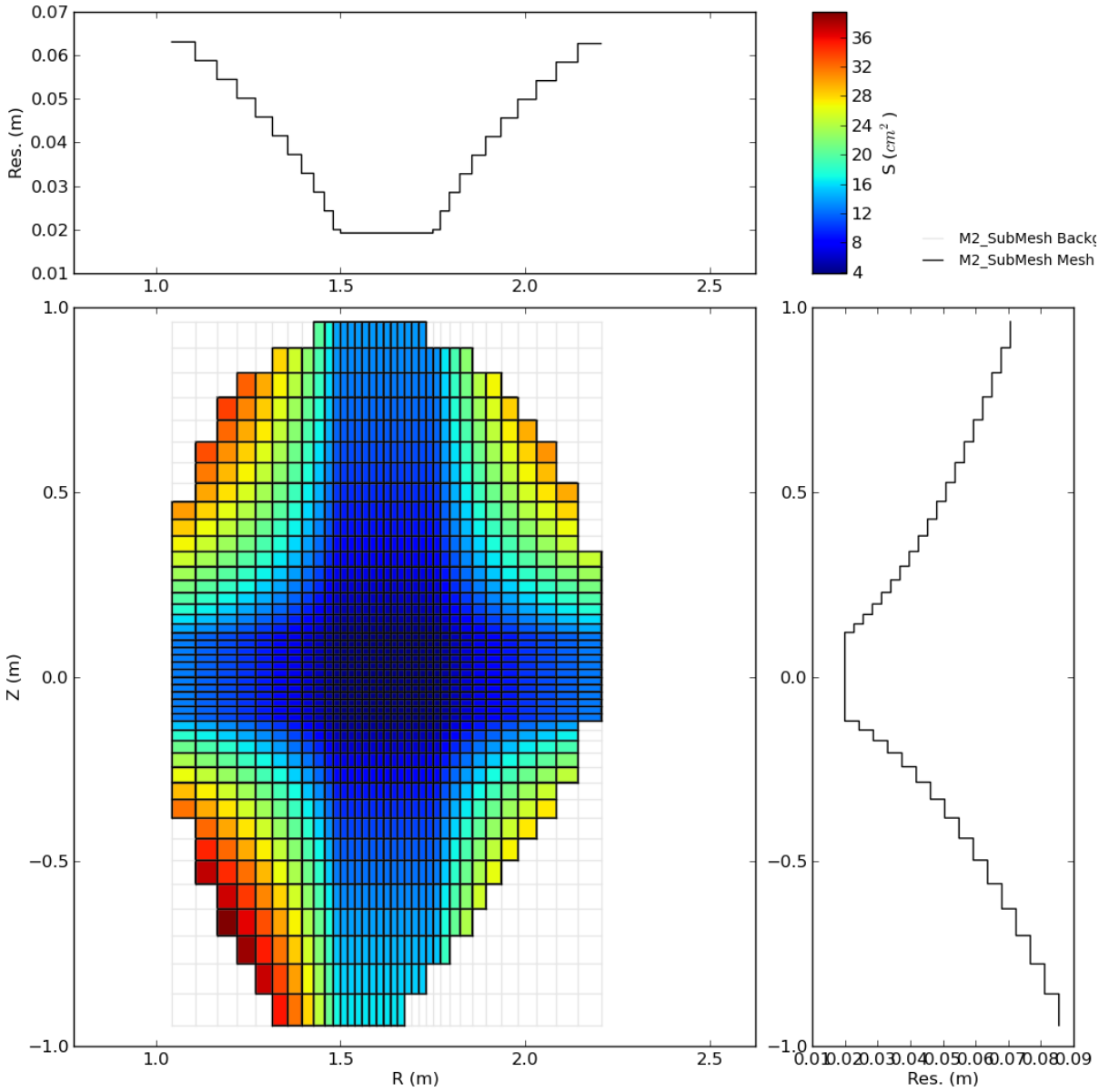

Figure 3.5: Local spatial resolution of the created 2D mesh (both linear and surface)

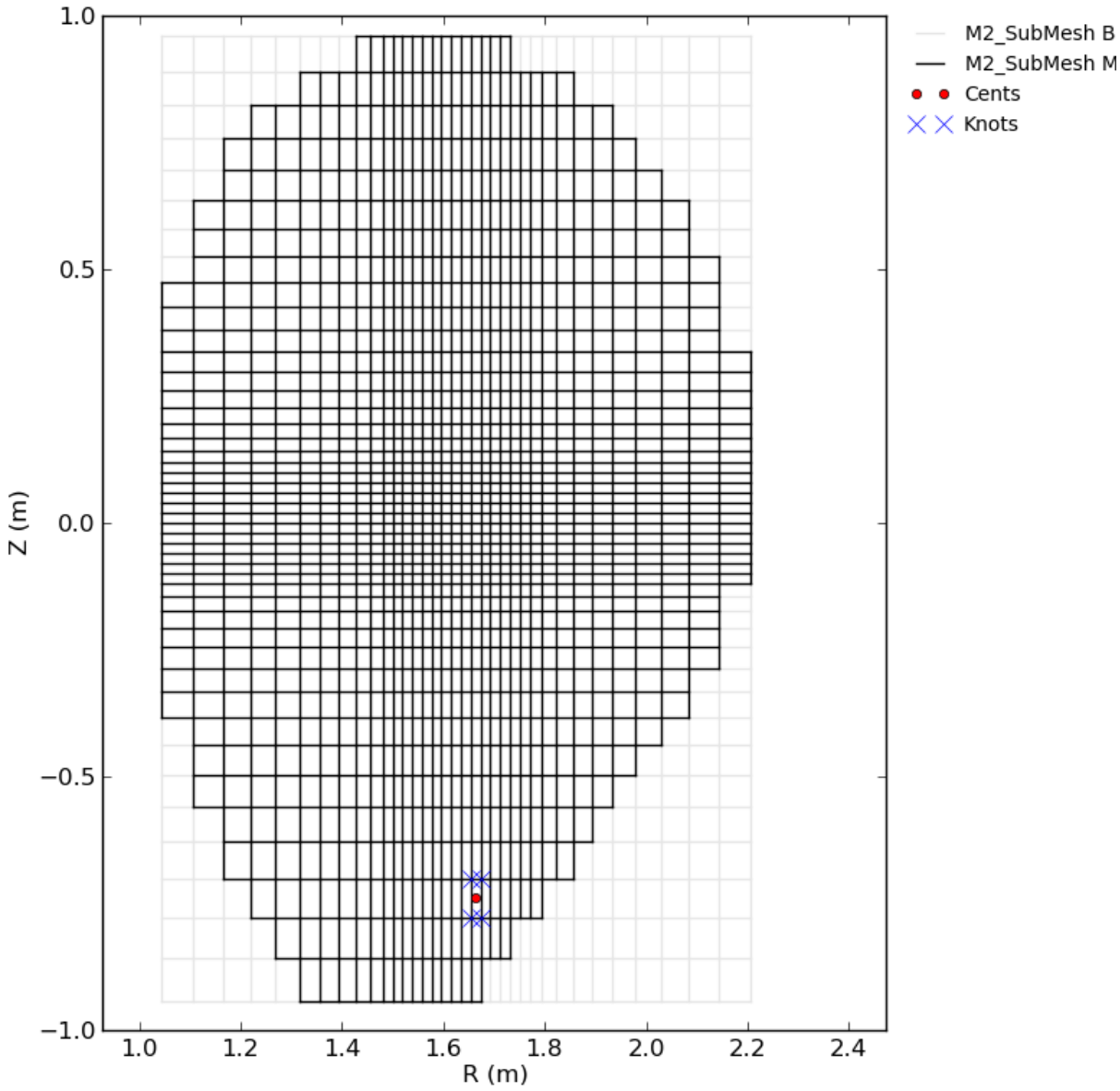

Figure 3.6: Selected mesh element and its associated Knots

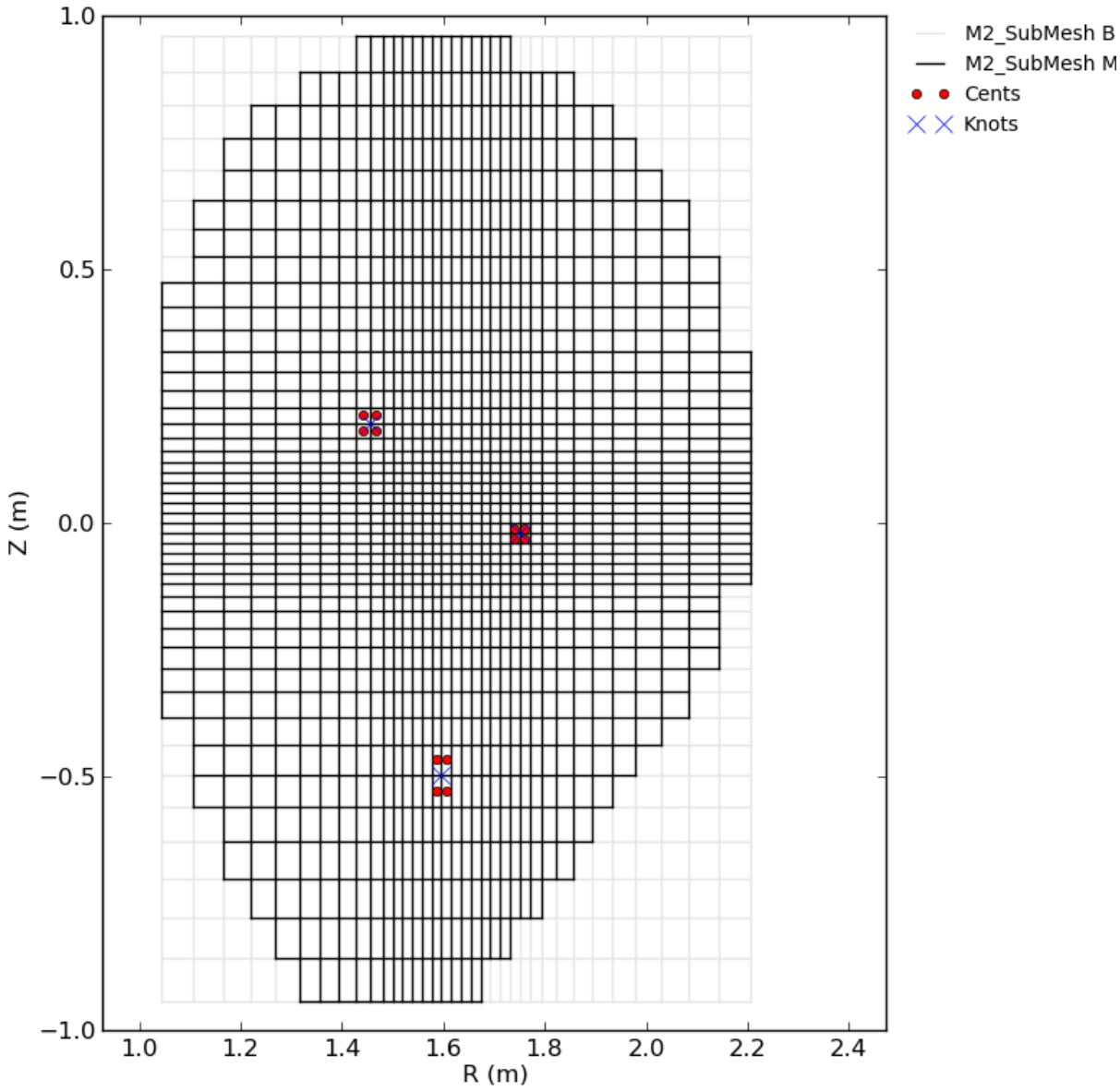

Figure 3.7: Selected mesh knots and their associated mesh elements

Table 3.4: The attributes of a BaseFunc1D object

Attribute	Description
self.ID	The ID class of the object
self.Mesh	The Mesh1D object on which the basis functions are built
self.LFunc, self.NFunc, self.Deg, self.Bound	The list of basis functions and the number of basis functions (self.NFunc=len(self.LFunc)), the degree of the basis functions, and the boundary condition (only 0 implemented so far, all points have 1 multiplicity)
self.Func_Centsind, self.Func_Knotsind, self.Func_PMax	An array giving the correspondence index between each basis function and all its associated mesh centers (and there are methods to go the other way around), its associated mesh knots, and the position of the maximum of each basis function (either oa mesh center or a knot depending on its degree).

Other quantities, indices or functions of interest are not stored as attributes, but instead accessible through methods, as will be illustrated in the following:

One of the most common issues in SXR tomography on Tokamaks is the boundary constraint that one must enforce at the plasma edge to force the SXR emissivity field to smoothlty decrease to zero in order to avoid artefacts on the tomographic reconstructions. With pixels, this usually has to be done by adding artificial detectors that 'see' the edge pixels only and are associated to a 'measured' value of zero (and the regularisation process does the rest). With B-splines of degree 2 for example, this constraint can be built-in the basis functions and enforced without having to add any artificial constraint, provided the underlying mesh is created accordingly, as illustrated in the following example, where BaseFunc1D object of degree 2 is created and a method is used to fit its coefficients to an input gaussian-like function:

```
BF1 = TFM.BaseFunc1D('BF1',M1,2)
FF = lambda xx: np.exp(-(xx-1.5)**2/0.2**2) + 0.4*np.exp(-(xx-1.65)**2/0.01**2)
Coefs, res = BF1.get_Coefs(ff=FF)
ax = BF1.plot(Coefs=Coefs, Elt='TL')
ax.plot(np.linspace(1.,2.,500), FF(np.linspace(1.,2.,500)), c='r', lw=2, label='Ref function')
#ax.figure.savefig(RP+"/../doc/source/figures_doc/Fig_Tutor_ToFuMesh_BF1.png",frameon=None,bbox_inche
```

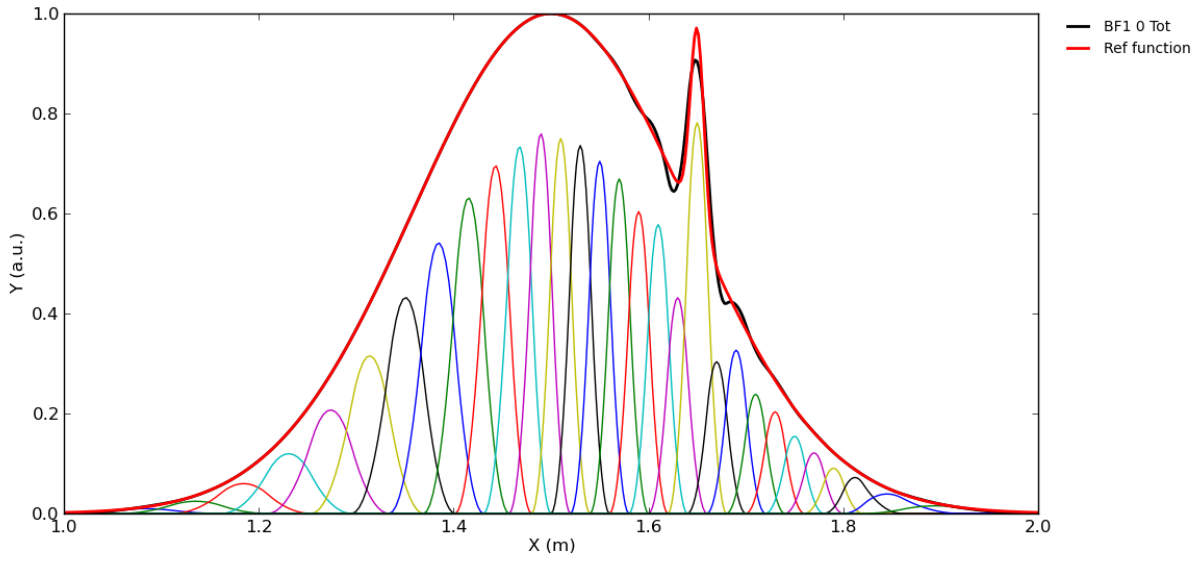

Figure 3.8: 1D B-splines of a BaseFunc1D object, with arbitrary coefficients to create a gaussian-like profile

By construction, and because we have only used points with multiplicity equal to one so far, the profile can only

decrease smoothly to zero near the edge.

The BaseFunc1D object also comes with methods to compute and plot local values its derivatives, or of some operators of interest that rely on derivatives. In particular, the following example shows the plots of the first derivative, the second derivative and a quantity called the Fisher Information that is the first derivative squared and divided by the function value. As usual, the 'Elt' kwdarg is used to specify whether we want only the total function ('T') or the detail of the list of all the underlying B-splines ('L', which is not possible for non-linear operators):

```
ax = BF1.plot(Coefs=Coefs, Deriv='D2', Elt='T',Totdict={'c':'k','lw':2})
ax = BF1.plot(ax=ax, Coefs=Coefs, Deriv='D1N2', Elt='T',Totdict={'c':'b','lw':2})
ax.figure.savefig(RP+"/../doc/source/figures_doc/Fig_Tutor_ToFuMesh_BF1_Deriv.png",frameon=None,bbox_
```

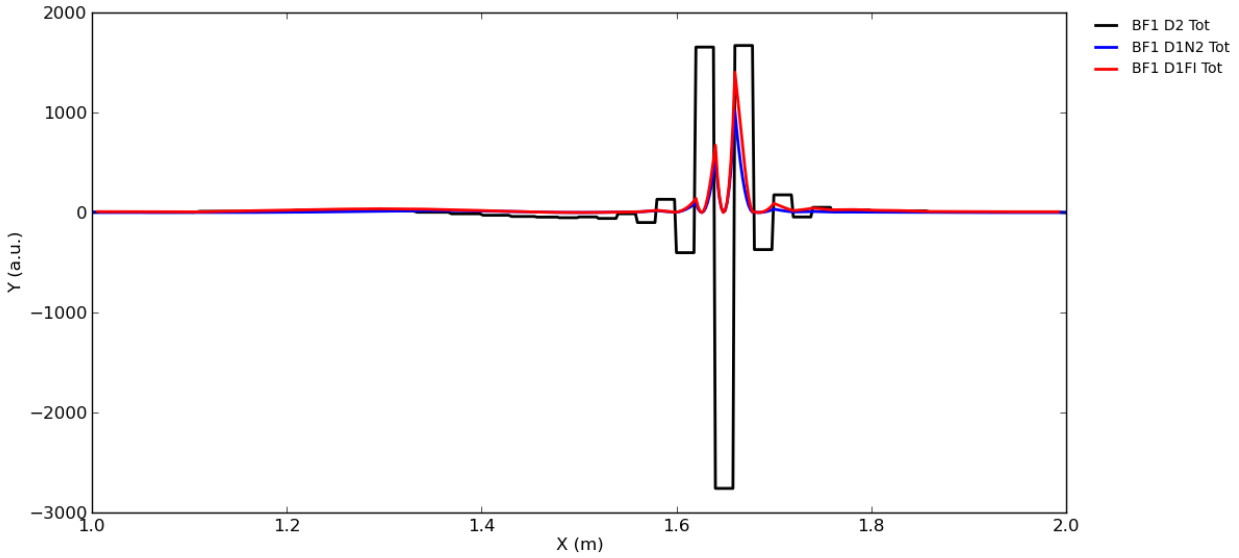

Figure 3.9: Some quantities of interest, based on derivative operators, for the chosen BaseFunc1D object

This was done using the 'Deriv' kwdarg, which can take several values, as shown in the table below:

Table 3.5: The available values of the 'Deriv' keyword argument for a BaseFunc1D object

Value	Description
0, 1, 2, 3 or 'D0', 'D1', 'D2', 'D3'	Respectively the B-splines themselves (0-th order derivative), the first, second and third order derivative
'D0N2', 'D1N2', 'D2N2', 'D3N2'	The squared norm of the 0th, 1st, 2nd and 3rd order derivatives
'D1FI'	The Fisher Information, which is the squared norm of the 1st order derivative, divided by the function value

Keep in mind that we are only using exact derivatives here, so the current version of **ToFu_Mesh** does not provide discretised operators and you have to make sure that you only compute derivatives for B-splines of sufficiently high degree.

Finally, the BaseFunc1D object also comes with methods to compute the value of the integral of the previous operators on the support of the B-spline. When it is possible, another method also returns the matrix that can be used to compute this integral using a vector of coefficients for the B-splines, along with a flag 'm' that indicates how the matrix should be used:

```
# Getting integral operators and values
A, m = BF1.get_IntOp(Deriv='D0')
Int = BF1.get_IntVal(Coefs=Coefs, Deriv='D0')
print A.shape, m

print Int
# (30,) 0
```

When m==0, it means that A is a vector (Int=A*Coefs), and when m==1, it means A is matrix and the integral requires a square operation (Int=Coefs*A*Coefs). The following integrals are implemented:

Table 3.6: The available values of the 'Deriv' keyword argument for integral computation

Value	Description
0 or 'D0'	The integrals of the B-splines themselves (0-th order derivative, integrals of higher order derivatives are all zero)
'D0N2', 'D1N2', 'D2N2', 'D3N2'	The integrals of the squared norm of the 0th, 1st, 2nd and 3rd order derivatives (only 0-th order derivative implemented so far, for Deg=0,1, but not for Deg=2,3)
'D1FI'	The integrated Fisher Information, not implemented so far

Finally, you can also plot a series of selected basis functions and there associated mesh elements (useful for detailed analysis and for debugging). Note tht you can also provide a 'Coefs' vector if you do not wish to use the default Coefs=1. value for representation.

```
ax = BF1.plot_Ind(Ind=[0,5,8], Elt='LCK')
plt.show()
```

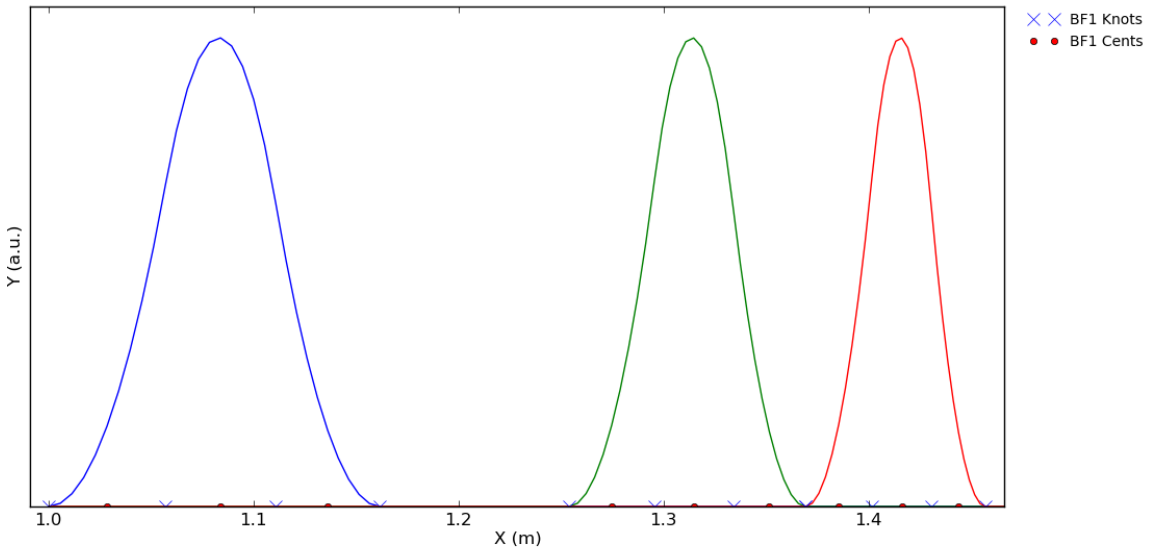

Figure 3.10: Some selected basis functions and their associated mesh centers and knots

All these functionalities are also found in the BaseFunc2D object, which additionally provides specific attributes and methods:

Table 3.7: The attributes of a BaseFunc2D object

Attribute	Description
self.ID	The ID class of the object
self.Mesh	The Mesh2D object on which the basis functions are built
self.LFunc, self.NFunc, self.Deg, self.Bound	The list of basis functions and the number of basis functions (self.NFunc=len(self.LFunc)), the degree of the basis functions, and the boundary condition (only 0 implemented so far, all points have 1 multiplicity)
self.Func_Centsind, self.Func_Knotsind, self.Func_PMax	An array giving the correspondence index between each basis function and all its associated mesh centers (and there are methods to go the other way around), its associated mesh knots, and the position of the maximum of each basis function (either oa mesh center or a knot depending on its degree).
self.FuncInterFunc	An array containing indices of all neighbouring basis functions of each basis function (neighbouring in the sense that the intersection of their respective supports is non-zero)

Due to its 2D nature, the BaseFunc2D object class is also equiped with methods to get the support (self.get_SuppRZ) and quadrature points (self.get_quadPoints) of each basis function.

Like the BaseFunc1D object, it provides a method for a least square fit of an input function. In the following example, the coefficients are determined using this method and then fed to various plotting methods used to visalise the function itself or some of its derivatives:

```
BF2 = TFM.BaseFunc2D('BF2',M2bis,1)
"""
PathFile = RP + '/Inputs/AUG_Tor.txt'
PolyRef = np.loadtxt(PathFile, dtype='float', comments='#', delimiter=None, converters=None, skiprows
Tor2 = TFG.Tor('AUG',PolyRef)
def Emiss(Points):
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Z = Points[2,:]
    Val = np.exp(-(R-1.68)**2/0.20**2 - (Z-0.05)**2/0.35**2) - 0.50*np.exp(-(R-1.65)**2/0.08**2 - (Z
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return Val

ax1, ax2 = BF2.plot_fit(ff=Emiss)
"""
Coefs, res = 1.,0#BF2.get_Coefs(ff=Emiss)
"""
f, axarr = plt.subplots(2,4, sharex=True, facecolor="w" ,figsize=(20,13))
ax = BF2.plot(ax=axarr[0,0], Coefs=Coefs,Deriv='D1', DVect=TFD.BF2_DVect_Def)
ax.axis("equal"), ax.set_title("D1-Z")
ax = BF2.plot(ax=axarr[1,0], Coefs=Coefs,Deriv='D1', DVect=TFD.BF2_DVect_Defbis)
ax.axis("equal"), ax.set_title("D1-R")
ax = BF2.plot(ax=axarr[0,1], Coefs=Coefs,Deriv='D1N2')
ax.axis("equal"), ax.set_title("D1N2")
ax = BF2.plot(ax=axarr[1,1], Coefs=Coefs,Deriv='D1FI')
ax.axis("equal"), ax.set_title("D1FI")
ax = BF2.plot(ax=axarr[0,2], Coefs=Coefs,Deriv='D2Lapl')
ax.axis("equal"), ax.set_title("D2Lapl")
ax = BF2.plot(ax=axarr[1,2], Coefs=Coefs,Deriv='D2LaplN2')
ax.axis("equal"), ax.set_title("D2LaplN2")
ax = BF2.plot(ax=axarr[0,3], Coefs=Coefs,Deriv='D2Gauss')
ax.axis("equal"), ax.set_title("D2Gauss")
ax = BF2.plot(ax=axarr[1,3], Coefs=Coefs,Deriv='D2Mean')
ax.axis("equal"), ax.set_title("D2Mean")
plt.show()
```

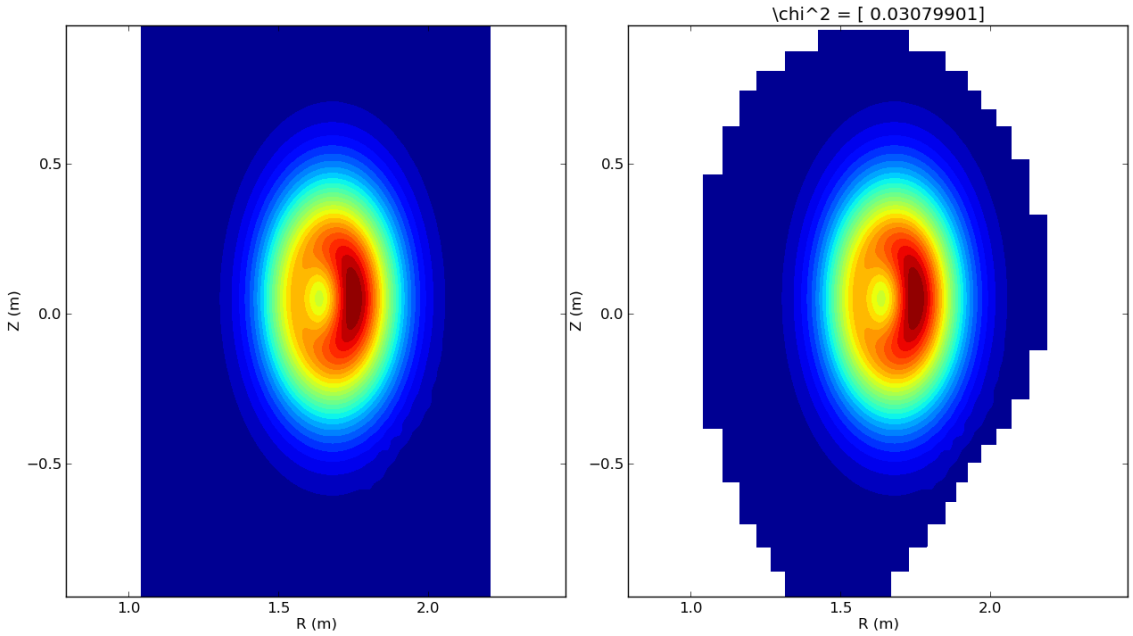

Figure 3.11: Input 2D emissivity model and fitted BaseFunc2D

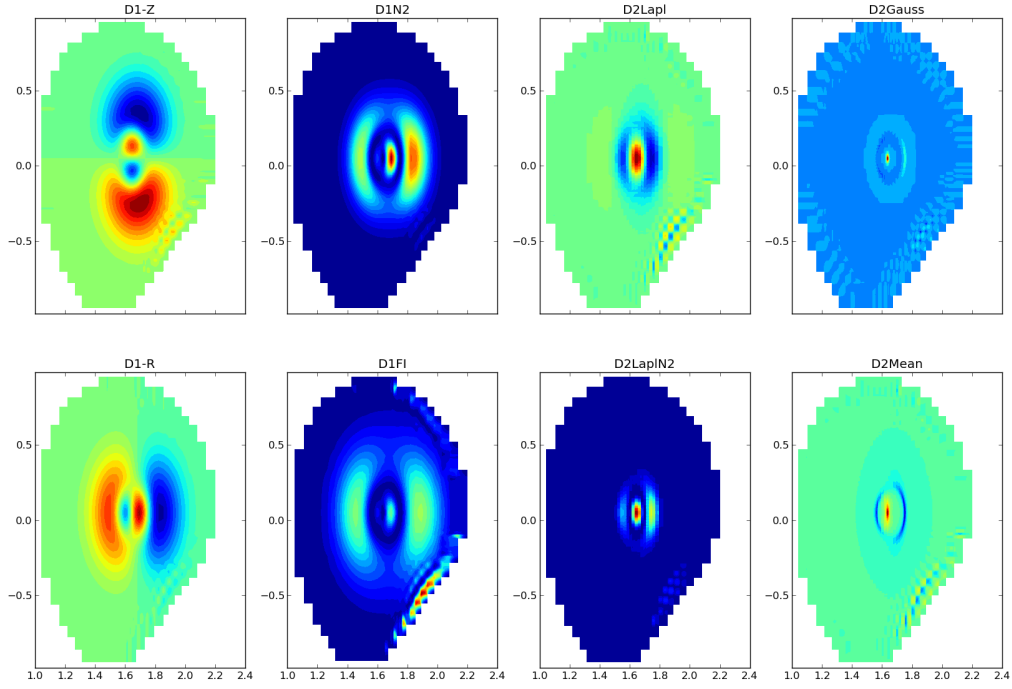

Figure 3.12: Series of derivatives or local quantities of interest of the fitted BaseFunc2D object

Like for the BaseFunc1D object, and in order to facilitate detailed analysis and possibly debugging, you can also plot the key points, support and value of some selected basis functions of your choice:

```
ax = BF2.plot_Ind(Ind=[200,201,202, 300,301,302, 622,623,624,625,626, 950], Elt='L', EltM='M', Coefs=
ax = BF2.plot_Ind(Ind=[200,201,202, 300,301,302, 622,623,624,625,626, 950], Elt='SP', EltM='MCK', Coe
plt.show()
```

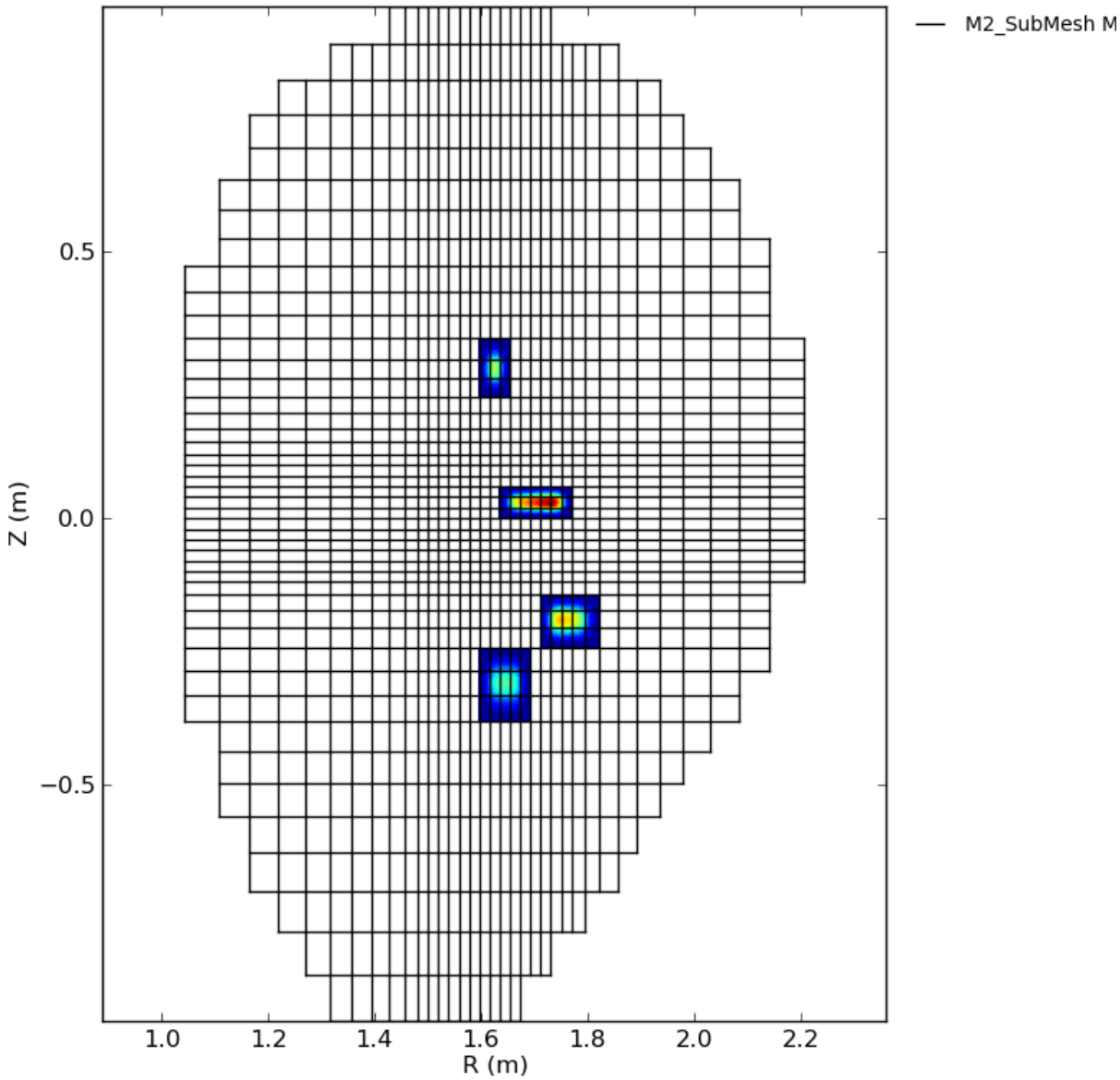

Figure 3.13: Local values of the selected local basis functions, with the underlying mesh

Finally, you can access values and operators of interest regarding some integrated quantities like the squared norm of the gradient, the squared laplacian (to be finished)...

```
print BF2.Mesh.Surf
print "Int radiation : ", BF2.get_IntVal(Deriv='D0', Coefs=1.)
print "Int sq. gradient : ", BF2.get_IntVal(Deriv='D1N2', Coefs=Coefs)
```

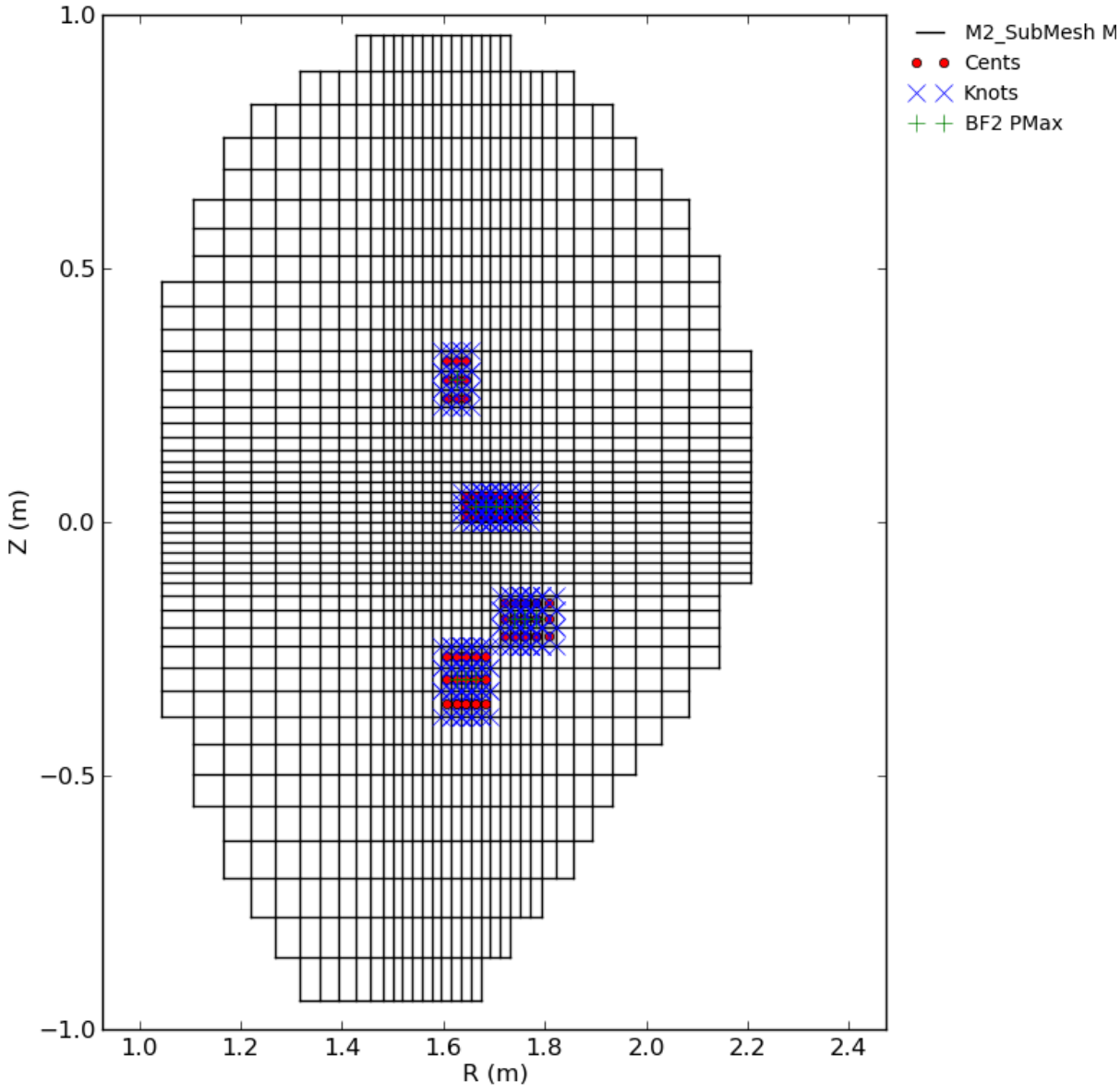

Figure 3.14: Support and PMax of the selected local basis functions, with the underlying mesh and centers and knots associated to the selected local basis functions

```
# 1.69963173663
# Surf :

# Vol :
# Surf :
# Vol :
```

The following table lists the operators which are available in **ToFu_Mesh**, depending on the value of the kwdarg 'Deriv':

Table 3.8: The available values of the 'Deriv' keyword argument for integral computation

Value	Description
0 or 'D0'	The integrals of the B-splines themselves (integrals of higher order derivatives are all zero)
'D0N2', 'D1N2', 'D2N2', 'D3N2'	The integrals of the squared norm of the 0th, 1st, 2nd and 3rd order derivatives (only 0-th order derivative implemented so far, for Deg=0,1, but not for Deg=2,3)
'D1FI'	The integrated Fisher Information, not implemented so far

CHAPTER

FOUR

TOFU_MATCOMP

(This project is not finalised yet, work in progress...)

ToFu_MatComp, is a ToFu module aimed at computing the geometry matrix associated to a diagnostic geometry from **ToFu_Geom** and a set of basis functions from **ToFu_Mesh**. From the first, it requires either a GDetect object or a GLOS object (keep in mind that a GDetect object automatically includes its associated GLOS object), or more simply a list of Detect objects or LOS objects (in case you don't want to define their group equivalent for any particular reason). From **ToFu_Mesh**, it requires a BaseFunc2D or BaseFunc3D object.

The outupt (i.e.: the computed geometry matrix) can be retrieved directly as a numpy array, or as a **ToFu_MatComp** object, which includes the array as an attribute and also provides useful methods to quickly explore its main characteristics, as illustrated in the following.

Hence, **ToFu_MatComp** provides the following object classes :

Table 4.1: The object classes in ToFu_Geom

Name	Description	Inputs needed
GMat2D	A geometry matrix computed from a BaseFunc2D object (i.e.: a 2D set of basis functions, assuming the toroidal angle is an ignorable coordinate).	A GDetect object or a list of Detect objects, and a BaseFunc2D object.
GMat3D	A geometry matrix computed from a BaseFunc3D object (i.e.: a 3D set of basis functions, not implemented yet...).	To do...

The following will give a more detailed description of each object and its attributes and methods through a tutorial at the end of which you should be able to compute your own geometry matrix and access its main characteristics.

4.1 Getting started with ToFu_MatComp

To use **ToFu_MatComp**, you first need to import it as well as **ToFu_PathFile**. Of course, **matplotlib** and **numpy** will also be useful.

```
import numpy as np
import matplotlib.pyplot as plt
import ToFu_PathFile as TFPF
import ToFu_Defaults as TFD
import ToFu_Geom as TFG
import ToFu_MatComp as TFMC
import os
import cPickle as pck # for saving objects
RP = TFPF.Find_Rootpath()
```

The os module is used for exploring directories and the cPickle module for saving and loading objects. We first need to load a BaseFunc2D object (created using **ToFu_Mesh** and saved), as well as a **GDetect** object (created with **ToFu_Geom** and saved):

```
GD = TFPF.open_object(RP+'/Objects/TFG_GDetect_AUG_SXR_Test_F_2_D20141128_T195755.pck')
BF0 = TFPF.open_object(RP+'/Objects/TFM_BaseFunc2D_AUG_SXR_Rough1_D0_D20141202_T230455.pck')
BF1 = TFPF.open_object(RP+'/Objects/TFM_BaseFunc2D_AUG_SXR_Rough1_D1_D20141202_T230455.pck')
BF2 = TFPF.open_object(RP+'/Objects/TFM_BaseFunc2D_AUG_SXR_Rough1_D2_D20141202_T230455.pck')
```

In the following, we will illustrate the capacities of **ToFu_MatComp** with the F camera of the SXR diagnostic of ASDEX Upgrade and a relatively coarse 2D mesh with resolution around 2 cm near in the central region and around 6 cm near the edge, on which degree 0 bivariate B-splines have been imposed (we will illustrate later the use of 1st and 2nd order bivariate B-splines).

Now we simply need to build the associated geometry matrix. Computation will be done in two steps : first, an index matrix will be computed (a numpy array of boolean) that indicates for each detector which mesh elements it can see (by checking whether they are it is projected viewing cone), this first step typically takes 1-5 min for each detector and helps a lot making the second step faster. The second step consists in proper computing of the integrated contribution of each basis function for each detector. This is obviously longer and typically takes 2-6 min per detector (instead of at least 10 times more without the first step). Finally, both for comparison purposes and for those users who want to use a pure LOS approach, another geometry matrix is computed with a pure LOS approximation, which is obviously much faster and typically takes 0.01-1 s per detector (as always, it depends on the mesh resolution and basis function degree).

```
GM0 = TFMC.GMat2D('AUG_SXR_F2_Rough1_D0', BF0, GD, Mode='simps')
GM1 = TFMC.GMat2D('AUG_SXR_F2_Rough1_D1', BF1, GD, Mode='simps')
GM2 = TFMC.GMat2D('AUG_SXR_F2_Rough1_D2', BF2, GD, Mode='simps')
```

Now that we have a proper GMat2D object, let us use its built-in methods to explore its properties.

First of all, we can plot the total contribution (from all the basis functions) to each detector simply by plotting the sum of the geometry matrix, and comparinmg it to the sum of the LOS-approximated geometry matrix. You can do this manually or use the dedicated built-in method, which also shows the sum in the other dimension (i.e.: the total contribution of each basis function to all detectors):

```
ax1, ax2 = GM0.plot_sum(TLOS=True)
plt.show()
```

We can see that there seems to be little difference between the full 3D and the LOS approximated matrices, but let us go a little further into the details by visualising the values of the gometry matrix for a particular chosen detector, and compare it to its LOS-approximated equivalent:

```
axP, axM, axBF = GM0.plot_OneDetect_PolProj(8, TLOS=True)
axM.set_xlim(400,500)
plt.show()
```

Similarly, we can go the other way around and visualise the values of the geometry matrix for any chosen basis function (and thus see how it contributes to various detectors):

```
axP, axD, axDred = GM0.plot_OneBF_PolProj(450, TLOS=True)
plt.show()
```

We can see significant differences when we consider the details of a specific line (or column) of the geometry matrix, which is important because it provides the set of equations that link the basis functions to the measurements. If you want to perform an inversion, you should pay particular attention to this set of equations as the tomography problem hinges on Fredholm integral equations of the first kind, making it an ill-posed problem particularly sensitive to errors both in the measurements and in the equations.

In summary, Despite similar sum (i.e. total contribution to each detector), we observe that with the LOS approximation

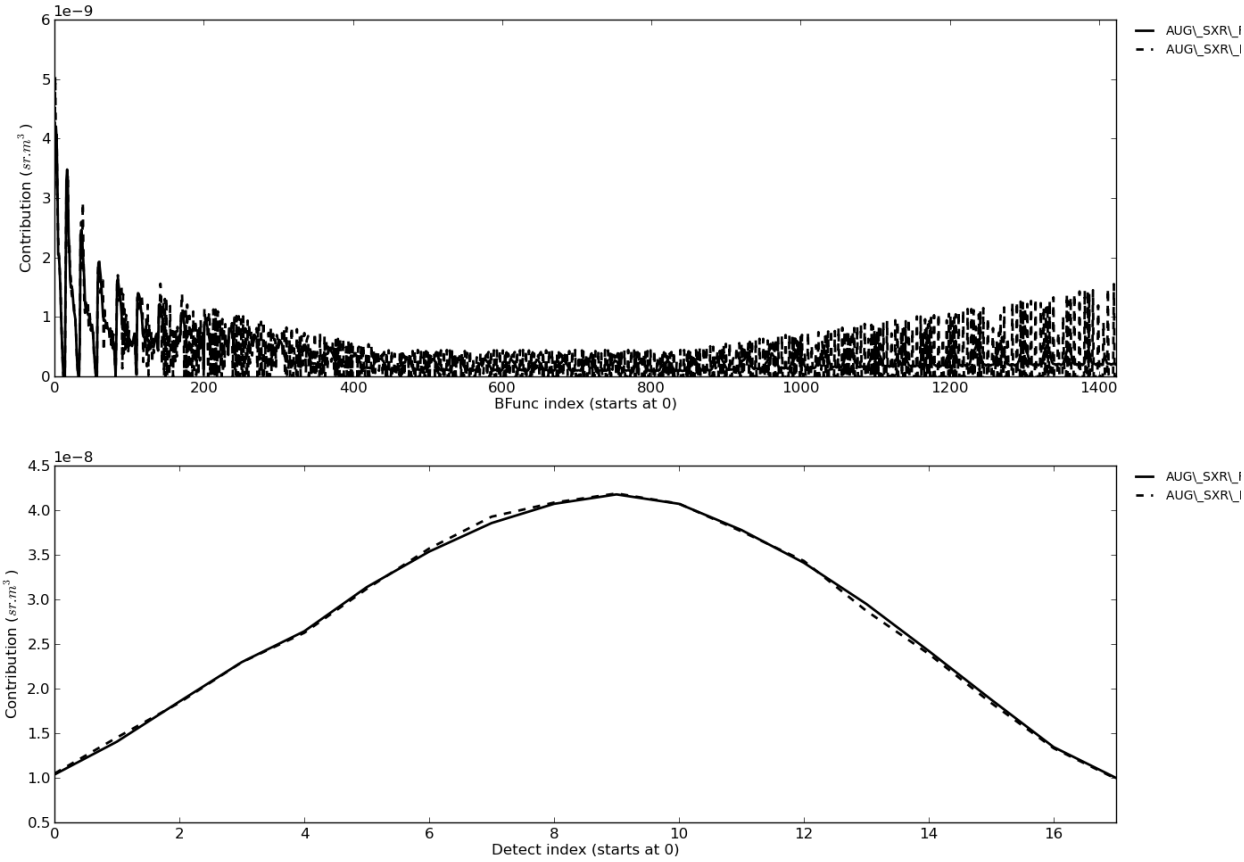

Figure 4.1: Total contribution of each basis function (top) and total contribution to each detector (bottom) for a 0th order set of B-splines and the F camera of ASDEX Upgrade, with both LOS and 3D computations

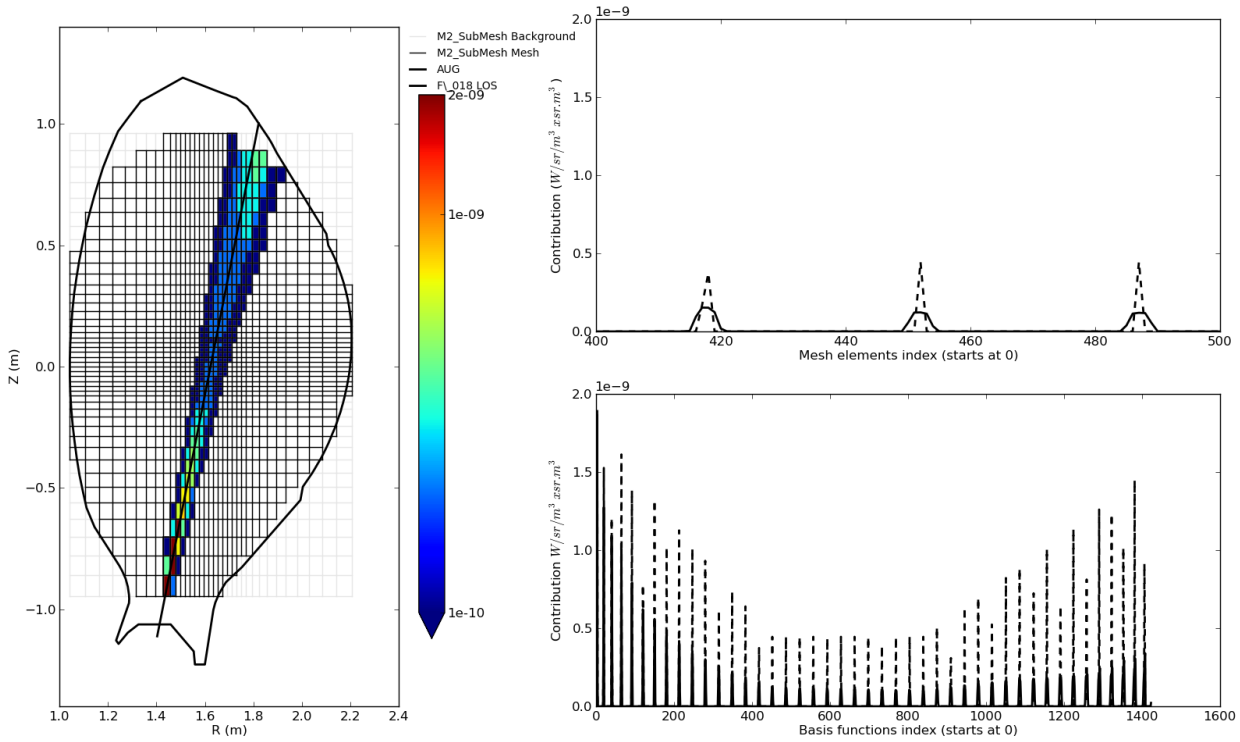

Figure 4.2: Total contribution of each 0th order basis function to detector F_016 of ASDEX Upgrade, decomposed on mesh elements (top) and basis functions (bottom), with both LOS and 3D computations

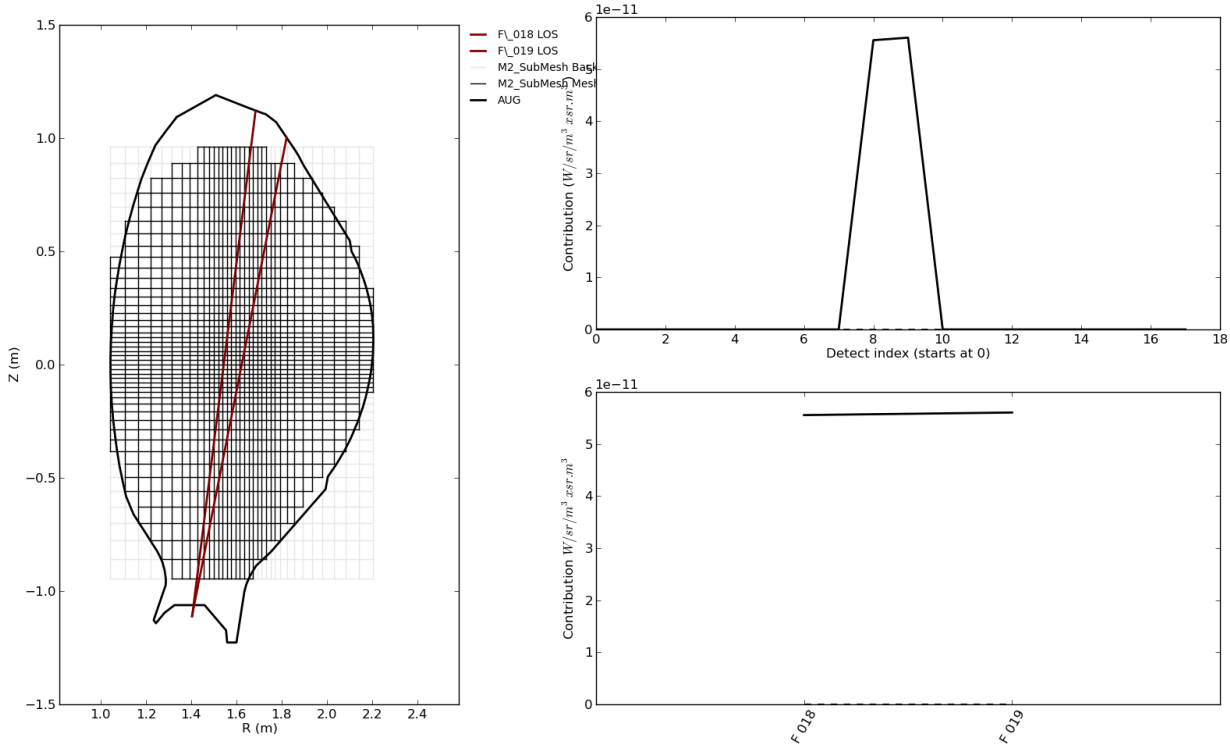

Figure 4.3: Total contribution of a particular 0th order basis function to each detector of camera F of ASDEX Upgrade, with both LOS and 3D computations, the chosen pixel is ognored by the LOS approximation, while in reality it is seen by two detectors.

the number of pixels that contribute to the signal is smaller but that their contribution is generally over-estimated as compared to the full 3D computation. If we consider each line of the geometry matrix, this line represents the equation associated to a particular detector measurement f_i:

$$f_i = M_{i,1}b_1 + M_{i,2}b_2 + ... + M_{i,N}b_N$$

Our observation then means that both computations give the same sum of terms on the right hand side, but that the LOS approximation tends to give higher values but for a fewer number of terms, thus affecting the spread of the weights on the different terms. This is an important limitation of the LOS approximation when it used to compute a geometry matrix using pixels as basis functions.

Now let us consider the same matrix but computed with 1st and 2nd order bivariate B-splines:

```
ax1, ax2 = GM1.plot_sum(TLOS=True)
axP, axM, axBF = GM1.plot_OneDetect_PolProj(8, TLOS=True)
axM.set_xlim(400,500)
axP, axD, axDred = GM1.plot_OneBF_PolProj(450, TLOS=True)
plt.show()
```

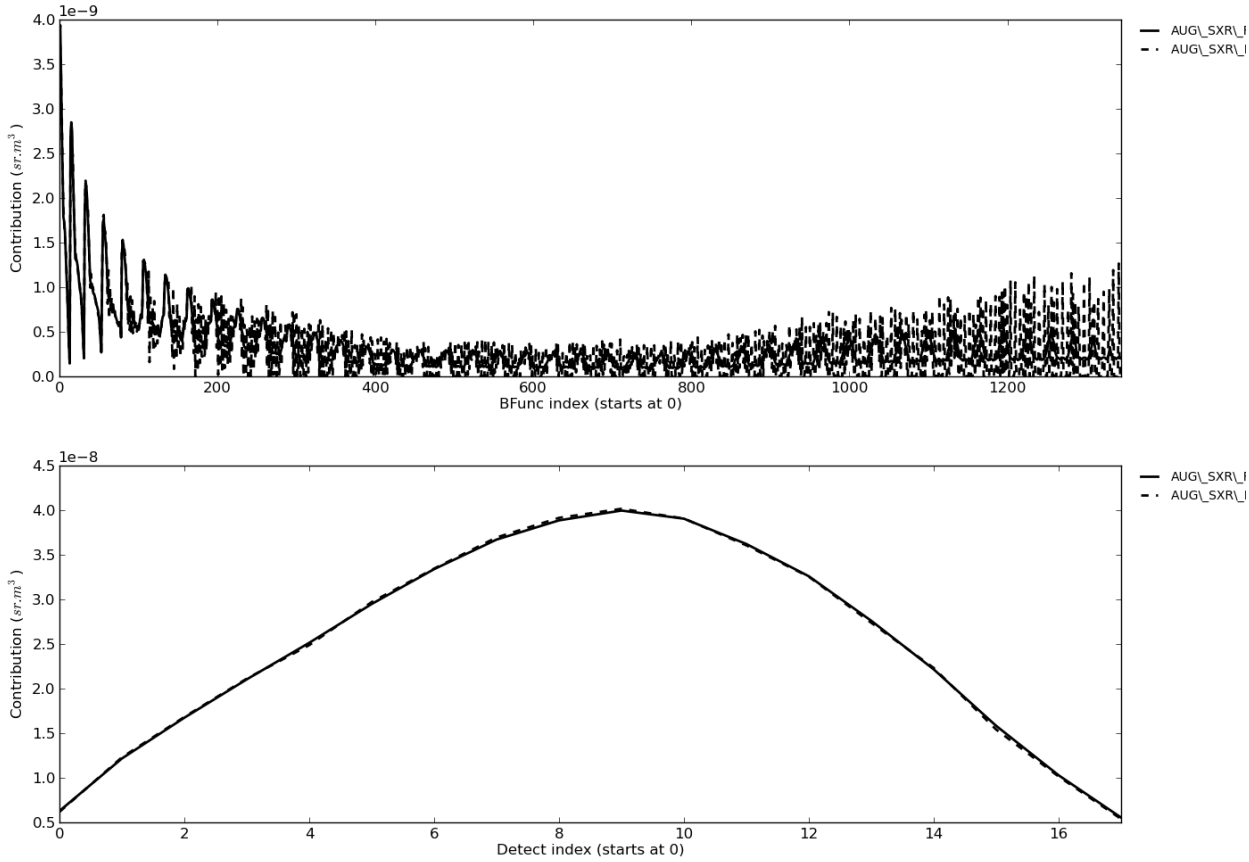

Figure 4.4: Total contribution of each basis function (top) and total contribution to each detector (bottom) for a 1st order set of B-splines and the F camera of ASDEX Upgrade, with both LOS and 3D computations

```
ax1, ax2 = GM2.plot_sum(TLOS=True)
axP, axM, axBF = GM2.plot_OneDetect_PolProj(8, TLOS=True)
axM.set_xlim(400,500)
axP, axD, axDred = GM2.plot_OneBF_PolProj(450, TLOS=True)
plt.show()
```

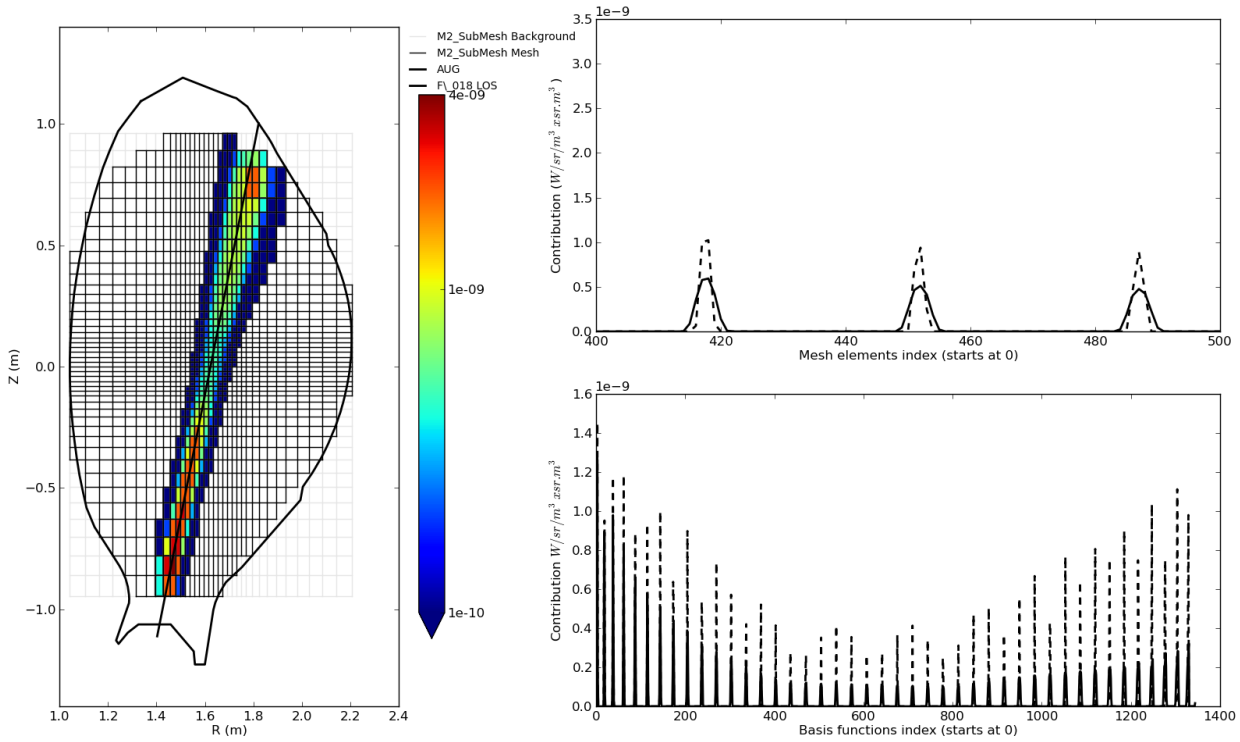

Figure 4.5: Total contribution of each 1st order basis function to detector F_016 of ASDEX Upgrade, decomposed on mesh elements (top) and basis functions (bottom), with both LOS and 3D computations

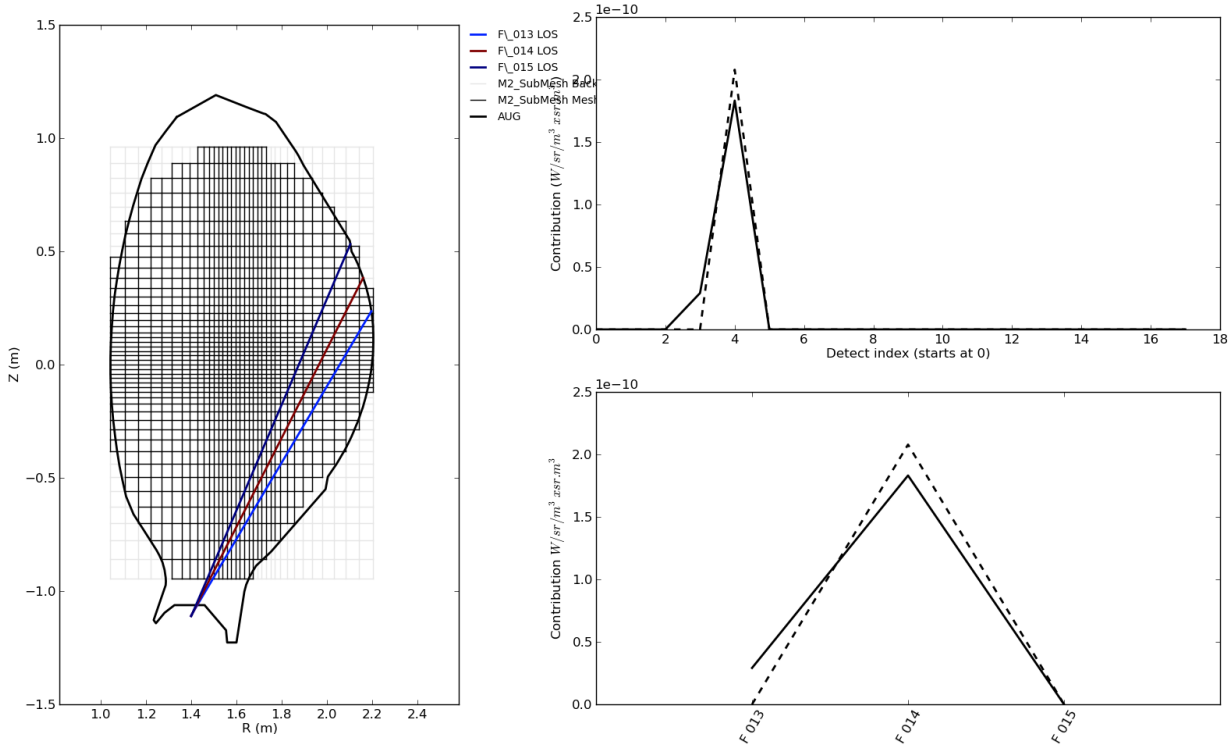

Figure 4.6: Total contribution of a particular 1st order basis function to each detector of camera F of ASDEX Upgrade, with both LOS and 3D computations, the chosen pixel is ignored by the LOS approximation, while in reality it is seen by two detectors.

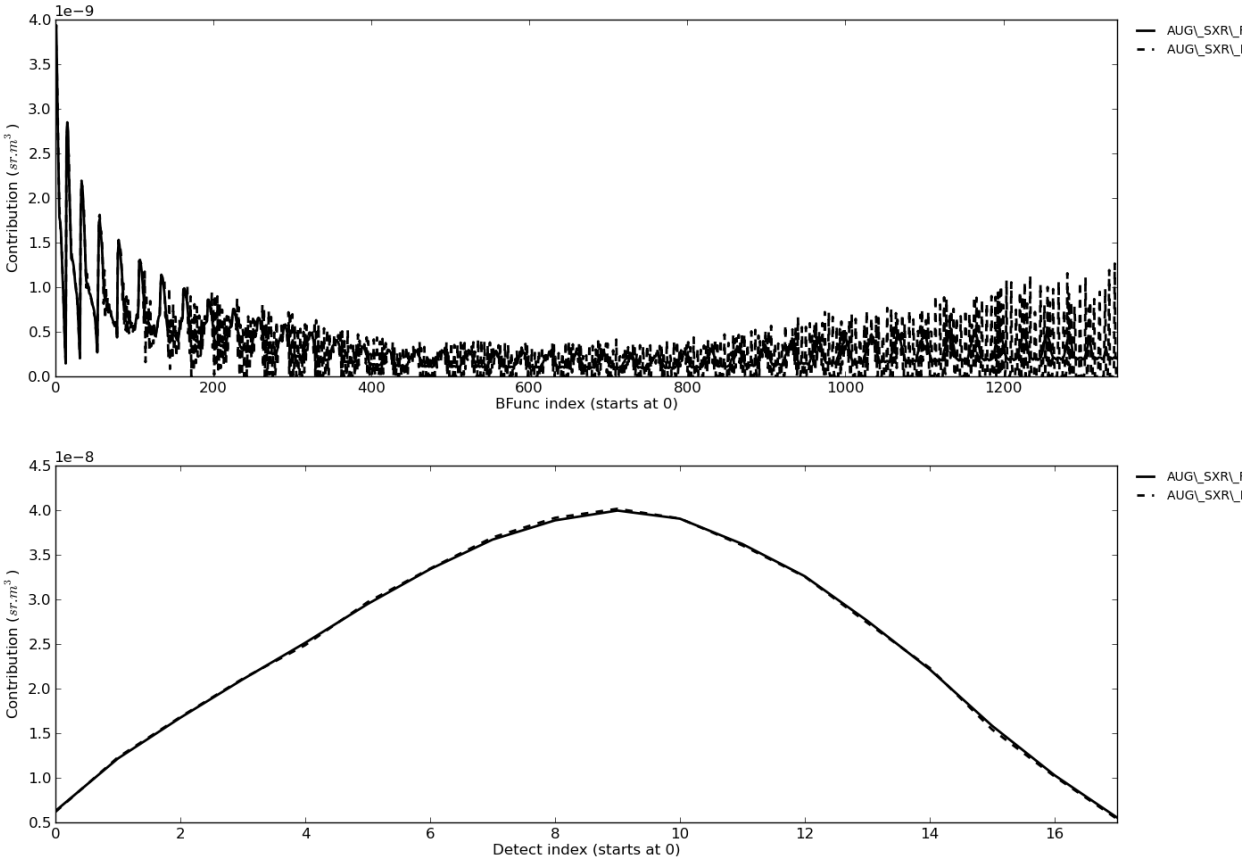

Figure 4.7: Total contribution of each basis function (top) and total contribution to each detector (bottom) for a 2nd order set of B-splines and the F camera of ASDEX Upgrade, with both LOS and 3D computations

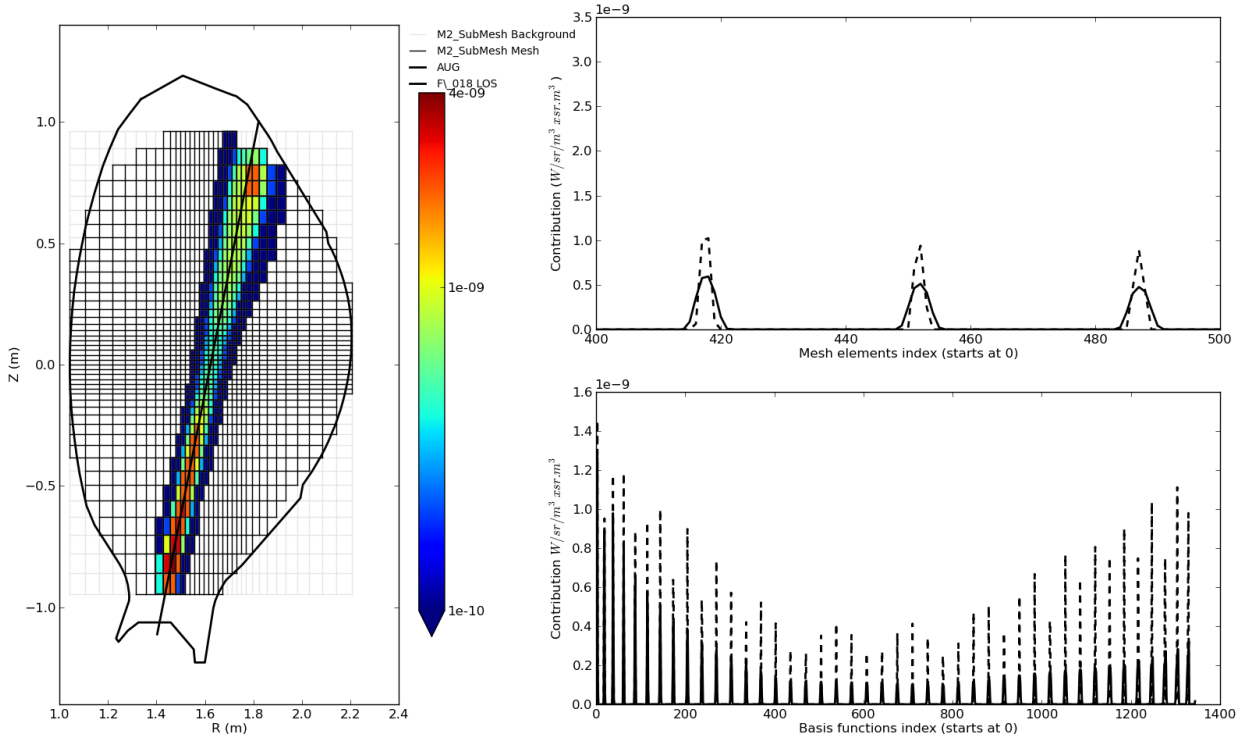

Figure 4.8: Total contribution of each 2nd order basis function to detector F_016 of ASDEX Upgrade, decomposed on mesh elements (top) and basis functions (bottom), with both LOS and 3D computations

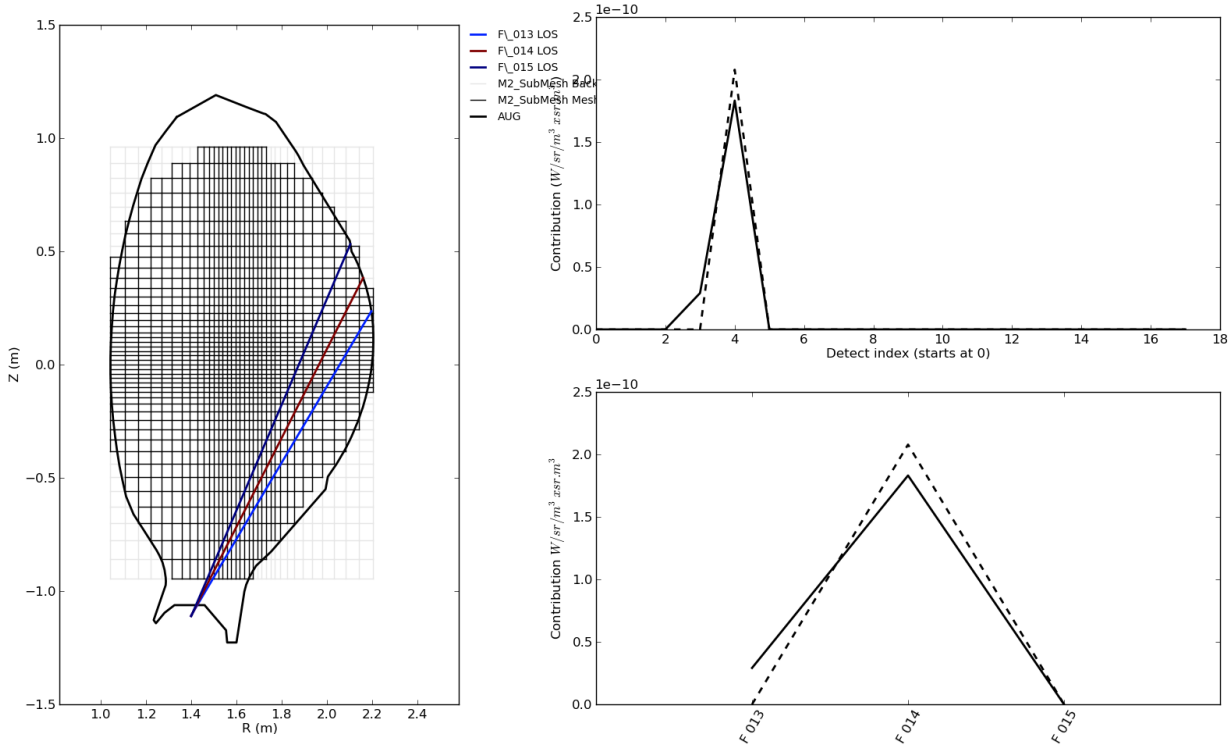

Figure 4.9: Total contribution of a particular 2nd order basis function to each detector of camera F of ASDEX Upgrade, with both LOS and 3D computations, the chosen pixel is ignored by the LOS approximation, while in reality it is seen by two detectors.

We see that the overlapping of higher-order basis functions ensures a more balanced distribution of the weights computed with a LOS approximation. This, and the fact that the basis functions are more regular, makes higher order basis functions a valuable improvement for tomographic inversions using a geometry matrix computed with a LOS approximatiopn. Obviously a full 3D computation remains even more accurate.

Now that the geometry is computed (with whatever method or basis functions), it can be used as in two ways : either as the set of equation necessary for solving the tomographic inversions (see **ToFu_Inv**), or as a pre-computed intermediate for forward-modelling or synthetic diagnostic (i.e.: reconstructing the measurements assuming an input emissivity field). This method only requires that the chosen basis functions are relevant for the input emissivity (i.e.: don't use a GMat2D object if the emissivity is not toroidally constant, or if the emissivity is anisotropic). Once you are sure that you have a relevant set of basis functions with their associated geometry matrix, just fit the basis functions to the input emissivity (this will give you the coefficients of each basis function) and use the geometry matrix to get the associated measurements, as illustrated below:

```
Tor2 = GD.Tor
def Emiss1(Points):
    R  = np.sqrt(Points[0,:]**2+Points[1,:]**2)
    Z = Points[2,:]
    Val = np.exp(-(R-1.68)**2/0.20**2 - (Z-0.05)**2/0.35**2) - 0.50*np.exp(-(R-1.65)**2/0.08**2 - (Z-
    ind = Tor2.isinside(np.array([R,Z]))
    Val[~ind] = 0.
    return 1000.*Val

Coefs0 = BF0.get_Coefs(ff=Emiss1)
Coefs1 = BF1.get_Coefs(ff=Emiss1)
Coefs2 = BF2.get_Coefs(ff=Emiss1)

ax1, ax2, ax3, ax4 = GM0.plot_Sig(Coefs=Coefs0, TLOS=True)
ax1, ax2, ax3, ax4 = GM1.plot_Sig(Coefs=Coefs1, TLOS=True)
ax1, ax2, ax3, ax4 = GM2.plot_Sig(Coefs=Coefs2, TLOS=True)
plt.show()
```

This method is faster than the direct, brute-force computation introduced in **ToFu_Geom**, but is limited by the relevance of the basis functions with respect to the input emissivity. We can see that the LOS approximation generally gives better results (in a synthetic diagnostic approach) when used with higher-order basis functions (as explained earlier). The difference is visible between 0th and 1st order basis functions (but not so much between 1st and 2nd order basis functions).

Furthermore, another general tendency appears : the LOS approximation tends to underestimate the signal for the lines on the High Field Side (HFS) and to overestimate it for the LOS on the Low Field Side (LFS), with respect to the region of maximum emissivity. This is consistent with the fact that the toroidicity induces a general shift towards the LFS. Hence the geometrically optimal LOS (from the center of mass of the detector to the center of mass of the intersection of of all its appertures) is optimal in cartesian coordinates but not in cylindrical coordinates. A different LOS (chosen taking into account the toroidicity, for example by computing the center of mass the viewing cone in (R,theta) coordinates) would probably help solve this issues and would allow you to use a pure LOS approximation with better validity (to do in ToFu_Geom...).

Again, the above numerical results are just helpful to understand what's going on, but keep in mind that the degree of accuracy of the LOS approximation not only depends on the geometry, but also on the input emissivity that you are using (i.e.: large gradients / curvature, toroidal changes, anistropy, localised particular features...).

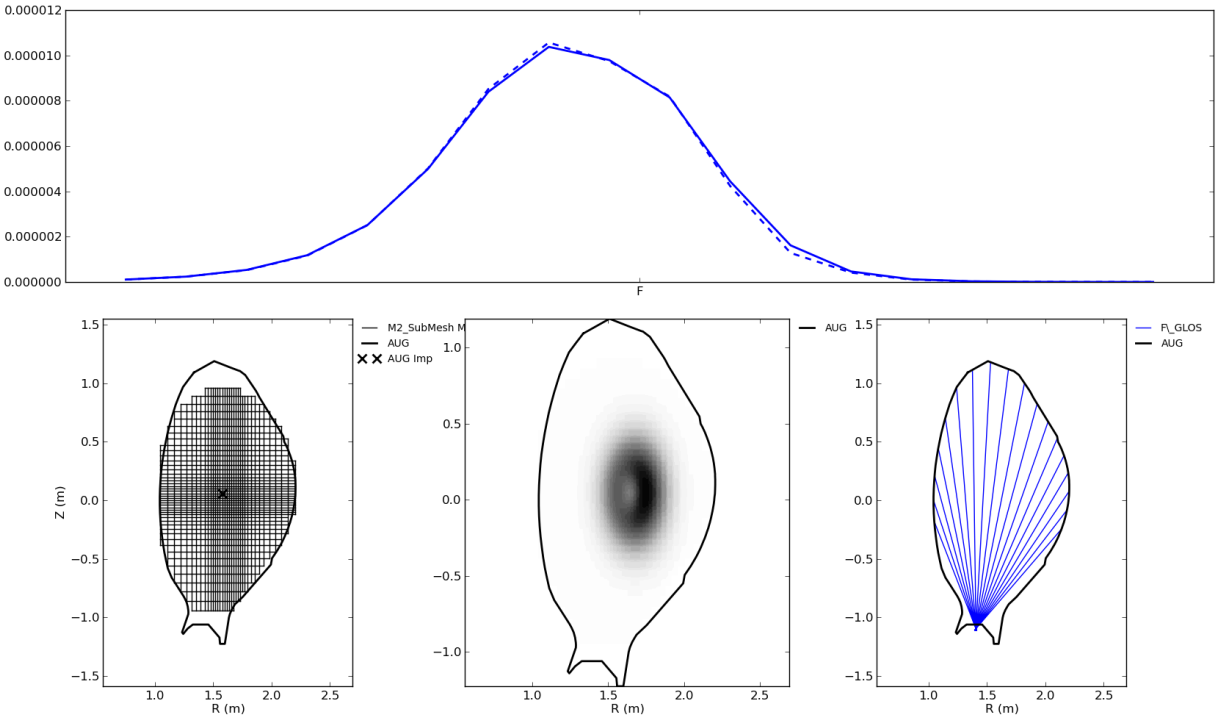

Figure 4.10: Synthetic diagnostic using decomposition of an input emissivity on a set of 0th order B-splines, geometry matrix computed with both 3D and LOS approach

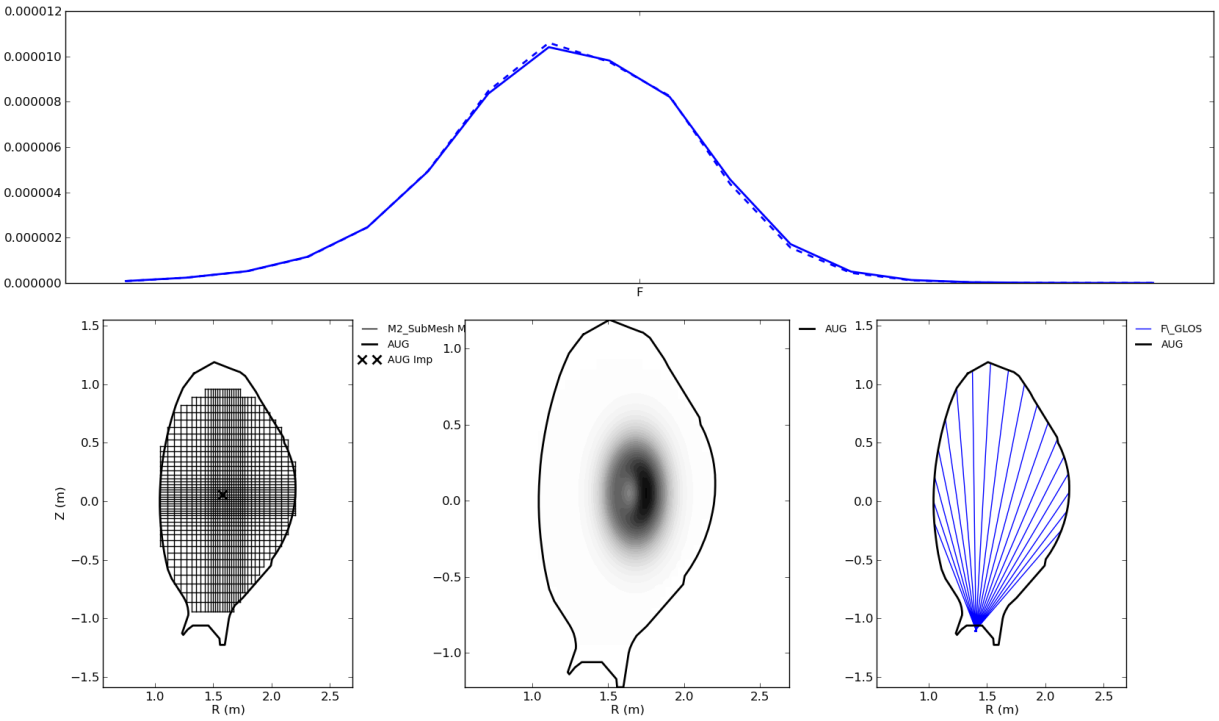

Figure 4.11: Synthetic diagnostic using decomposition of an input emissivity on a set of 1st order B-splines, geometry matrix computed with both 3D and LOS approach

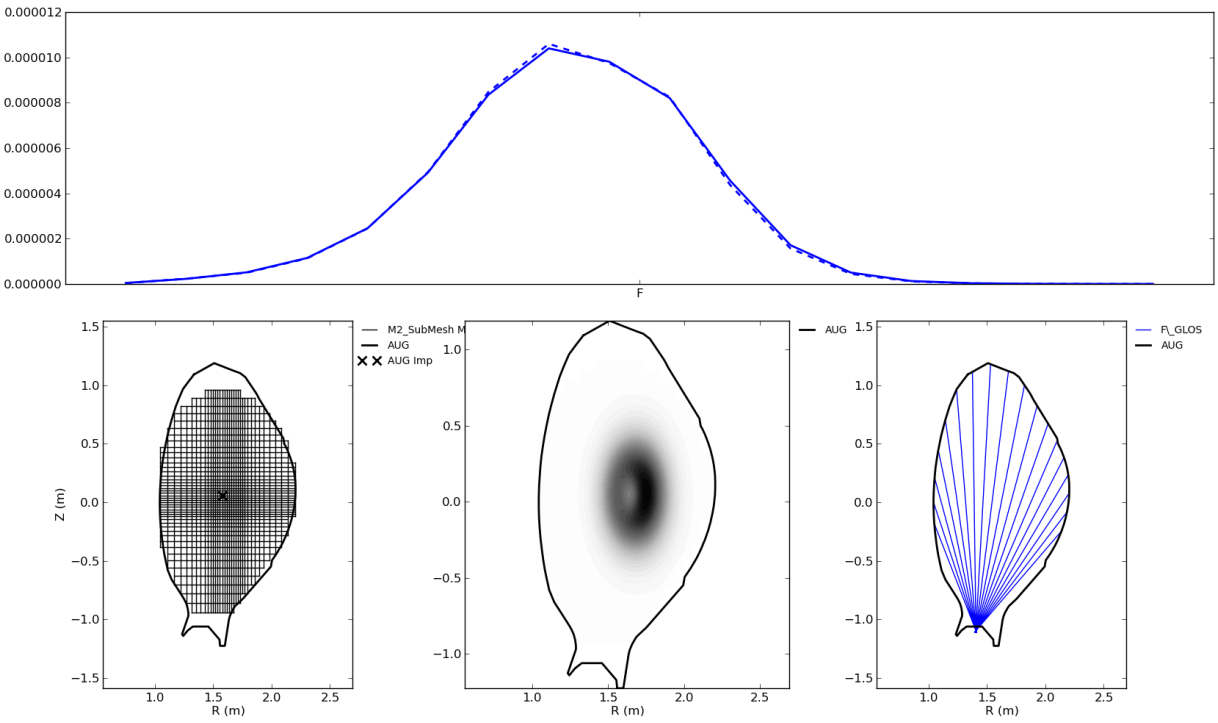

Figure 4.12: Synthetic diagnostic using decomposition of an input emissivity on a set of 2nd order B-splines, geometry matrix computed with both 3D and LOS approach

4.2 Limits to the LOS approximation for the geometry matrix computation

As we saw, the fact that the final solution is probably compatible with the LOS approximation (i.e. it has 'reasonable' curvature across most viewing cones) does not mean that the LOS approximation is valid with the basis functions used to compute it. Indeed, the LOS approximation can basically be written as follows:

$$f_i = \int_{LOS} \iint_S \int_{4\pi} \epsilon^\eta \delta_i d^2\Omega d^2 S dl = E_i \int_{LOS} \langle \epsilon^\eta \rangle \, dl$$

Where E_i is the etendue of detector i and:

$$\langle \epsilon^\eta \rangle = \frac{1}{E_i} \iint_S \int_{4\pi} \epsilon^\eta \delta_i d^2\Omega d^2 S$$
$$E_i = \iint_S \int_{4\pi} \delta_i d^2\Omega d^2 S$$

The corresponding LOS-approximated signal would be:

$$\hat{f}_i = E_i \int_{LOS} \epsilon^\eta dl$$

Hence, the LOS approximation is valid if we can safely assume that the local value of the emissivity on the LOS is a approximation of its value averaged on the surface perpendicular to the LOS:

$$\hat{f}_i \approx f_i \Leftrightarrow \int_{LOS} \langle \epsilon^\eta \rangle \, dl \approx \int_{LOS} \epsilon^\eta dl$$

Which can be fulfilled if (but not exclusively if):

$$\forall \, l \in LOS \, , \, \epsilon^\eta (l) \approx \frac{1}{E_i} \iint_S \int_{4\pi} \epsilon^\eta \delta_i d^2\Omega d^2 S \, (l) = \langle \epsilon^\eta \rangle \, (l)$$

When applied to a physical (i.e. real) emissivity field, this assumption may hold because the emissivity field usually considered varies sufficiently slowly in the direction perpendicular to the LOS (in the limits of the viewing cone). However, when computing the geometry matrix, this equation is not applied to a 'physiscal' emissivity field, but to individual basis functions. In particular, if the nature of these basis functions allows for steep variations across the LOS and in the limits of the viewing cone, then the LOS approximation may need to be questionned.

A typical case is pixels (i.e.: 0th order bivariate B-splines), particularly when they have a size too small compared to the local beam width (ref : my thesis + ingesson). In such cases, the above integral, taken for a single pixel (which is what is evaluated in the geometry matrix) can in no way be approximated by the value in the pixel (i.e.: on the LOS).

This is a very common mistake : even if the LOS approximation is valid for the final solution, it does not mean it is valid for the basis functions that you are using ! And, ironicaly, using it for computing the geometry matrix anyway will lead to a final solution that will be less regular than it should be because the LOS approximation tends to overestimate the contribution of some pixels and underestimate the contribution of others. Paradoxically, not using the LOS approximation for the geometry matrix is both more physical and leads to solutions which are more likely to be compatible with this LOS approximation ! Again, this depends on the basis functions you are using (nature and size).

If you still want to use pixels and the (pure) LOS approximation together, a rule of thumb to limit the bias is to use pixels of a size comparable to the beam width in the region where the signal is maximal (but there will still be situations in which the approximation will not hold, see : my thesis). Another quite common solution is to make your own 'homemade' routine to compensate for the beam width (for example with anti-aliasing, with adaptative LOS or with hybrid 1D-2D-3D solutions). Most people have their own tricks to compensate in a way or another.

Other solutions are either to use the LOS approximation but with different basis functions such as 1st or 2nd order bivariate B-splines (both because their support overlap and because they are more regular), or 'simply' not to use the LOS approximation (but an accurate full 3D computation requires a lot of painful work). Since both these solutions are fully implemented in **ToFu**, you can start rejoicing and using it :-)

CHAPTER

FIVE

TOFU_INV

(This project is not finalised yet, work in progress...)

ToFu_Mesh, is a ToFu module aimed at handling spatial discretisation of a 3D scalar field in a vacuum chamber (typically the isotropic emissivity of a plasma). Such discretisation is done using B-splines of any order relying on a user-defined rectangular mesh (possibily with variable grid size). It is particularly useful for tomographic inversions and fast synthetic diagnostics.

It is designed to be used jointly with the other **ToFu** modules, in particular with **ToFu_Geom** and **ToFu_MatComp**. It is a ToFu-specific discretisation library which remains quite simple and straightforward. However, its capacities are limited to rectangular mesh and it may ultimately be percieved as a much less powerful version of **PIGASUS/CAID**. Users who wish to use **ToFu** only for tomographic inversions may find **ToFu_Mesh** sufficient for thir needs, others, who wish to use a synthetic diagnostic approach, and/or to use **ToFu_Mesh** jointly with plasma physics codes (MHD...) may prefer using **PIGASUSCAID** for spatial discreatisation.

Hence, **ToFu_Mesh** mainly provides two object classes : one representing the mesh, and the other one (which uses the latter) representing the basis functions used for discretisation:

Table 5.1: The object classes in ToFu_Geom

Name	Description	Inputs needed
ID	An identity object that is used by all **ToFu** objects to store specific identity information (name, file name if the object is to be saved, names of other objects necessary for the object creation, date of creation, signal name, signal group, version...)	By default only a name (a character string) is necessary, A default file name is constructed (including the object class and date of creation), but every attribute can be modified and extra attribute can be added to suit the specific need of the the data acquisition system of each fusion experiment or the naming conventions of each laboratory.
Mesh1D, Mesh2D, Mesh3D	1D, 2D and 3D mesh objects, storing the knots and centers, as well as the correspondence between knots and centers in both ways. The higher dimension mesh objects are defined using lower dimension mesh objects. The Mesh 2D object includes an enveloppe polygon. They all include plotting methods and methods to select a subset of the total mesh. The Mesh 3D object is not finished.	A numpy array of knots, which can be defined using some of the functions detailed below (for easy creation of linearly spaced knots with chosen resolution).
BaseFun1D, BaseFunc2D, BaseFunc3D	1D, 2D and 3D families of B-splines, relying on Mesh1D, Mesh2D, Mesh3D objects, with chosen degree and multiplicity for each dimension. Includes methods for plotting, for determining the support and knots and centers associated to each basis function, as well as for computing 1st, 2nd or 3rd order derivatives (as functions), and local value (summation of all basis functions or their derivatives at a given point and for given weights). Includes methods for computing integrals of derivative operators...	A Mesh object of the adapted dimension, and a degree value.

The following will give a more detailed description of each object and its attributes and methods through a tutorial at the end of which you should be able to create your own mesh and basis functions and access its main characteristics.

5.1 Getting started with ToFu_Mesh

Once you have downloaded the whole **ToFu** package (and made sur you also have scipy, numpy and matplotlib, as well as a free polygon-handling library called Polygon which can be downloaded at), just start a python interpreter and import **ToFu_Geom** (we will always import **ToFu** modules 'as' a short name to keep track of the functionalities of each module). To handle the local path of your computer, we will also import the small module called **ToFu_PathFile**, and **matplotlib** and **numpy** will also be useful:

The os module is used for exploring directories and the cPickle module for saving and loading objects.

5.2 The Tor object class

To define the volume of the vacuum chamber, you need to know the (R,Z) coordinates of its reference polygon (in a poloidal cross-section). You should provide it as a (2,N) numpy array where N is the number of points defining the polygon. To give the Tor object its own identity you should at least choose a name (i.e.: a character string). For more elaborate identification, you can define an ID object and give as an input instead of a simple name. You can also provide the position of a "center" of the poloidal cross-section (in 2D (R,Z) coordinates as a (2,1) numpy array) that

will be used to compute the coordinates in transformation space any LOS using this Tor object (and the sinogram of any scalar emissivity field using this Tor object). If not provided, the center of mass of the reference polygon is used as a default "center".

In the following, we will use the geometry of ASDEX Upgrade as a example. We first have to give a reference polygon ('PolyRef' below) as a (2,N) numpy array in (R,Z) coordinates.

Alternatively, you can store PolyRef in a file and save this file locally, or use one of the default tokamak geometry stored on the **ToFu** database where Tor input polygons are stored in 2 lines .txt files (space-separated values of the R coordinates on the first line, and corresponding Z coordinates on the second line). Here, we use the default ASDEX Upgrade reference polygon stored in AUG_Tor.txt.

We now have created two Tor objects, and **ToFu_Geom** has computed a series of geometrical characteristics that will be useful later (or that simply provide general information). TO BE FINISHED !!!!!!!!!!!!!!!

$$\nabla^2 u = \sin(x)$$

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*