oimodeler

Release 0.8.0

A. Meilland

Jul 12, 2023

CONTENTS

1 A Few examples **3**

2 License **9**

3 Acknowledgment **11**

4 Contact **13**

5 Table of Contents **15**

5.1 Overview . . . 15
5.1.1 Software modularity . . . 15
5.1.2 Modules . . . 16
5.1.2.1 oimModel . . . 16
5.1.2.2 oimComponent . . . 16
5.1.2.3 oimParam . . . 16
5.1.2.4 oimData . . . 17
5.1.2.5 oimDataFilter . . . 17
5.1.2.6 oimSimulator . . . 17
5.1.2.7 oimFitter . . . 17
5.1.2.8 oimPlots . . . 17
5.1.2.9 oimUtils . . . 17
5.1.3 Expandability . . . 18
5.2 Installation . . . 18
5.2.1 Dependancies . . . 18
5.2.2 Checking installation . . . 19
5.3 Getting Started . . . 19
5.4 Examples . . . 26
5.4.1 Basic Examples . . . 27
5.4.1.1 Loading oifits data . . . 27
5.4.1.2 Basic models . . . 28
5.4.1.3 Precomputed fits-formated image . . . 35
5.4.1.4 Data/model comparison . . . 41
5.4.1.5 Running a mcmc fit . . . 43
5.4.1.6 Filtering data . . . 47
5.4.1.7 Plotting oifits data . . . 50
5.4.2 Advanced Examples . . . 55
5.4.2.1 Building Complex models . . . 55
5.4.2.2 Precomputed chromatic image-cubes . . . 62
5.4.2.3 Parameters Interpolators . . . 64
5.4.2.4 Fitting a chromatic model . . . 72

5.4.3 Expanding the Software . . . 77
5.4.3.1 Creating New Components . . . 77
5.4.3.2 Creating New Interpolators . . . 92
5.4.4 Performance Tests . . . 97
5.5 API . . . 97

6 Index **99**

The `oimodeler` project aims at developing a modular and easily expandable python-based modelling software for optical interferometry. The project started end of 2021, and the software is currently at an early stage of development.

It allows to manipulate data in the OIFITS format, build complex models from various components, simulate data from the model at the spatial frequencies of your observations, compute chi2, perform model fitting (using mcmc or other fitters), and plot results easily. Components can be defined in the image or Fourier plan using analytical formulas or precomputed images. Components or model parameters can be chromatic and/or time dependent. The software is modular and object oriented, in order to make it easy to expand it by creating new model components or other features from abstract classes.

Warning:

The software is in early development:

- Models: In Fourier and Image plans with chromaticity and time dependence
- Data: Interferometric data only. No photometric or spectroscopic data.
- Data Filters: Filtering wavelength range, and data type (VIS2DATA, VISAMP…)
- Fitters: Implementation of a basic emcee-based fitter with plots for results
- Plots: Basics plots of OIFITS data and uv-plane plot
- Utils: Miscellaneous utilities for OIFITS data (creating and modifying array, getting info..)

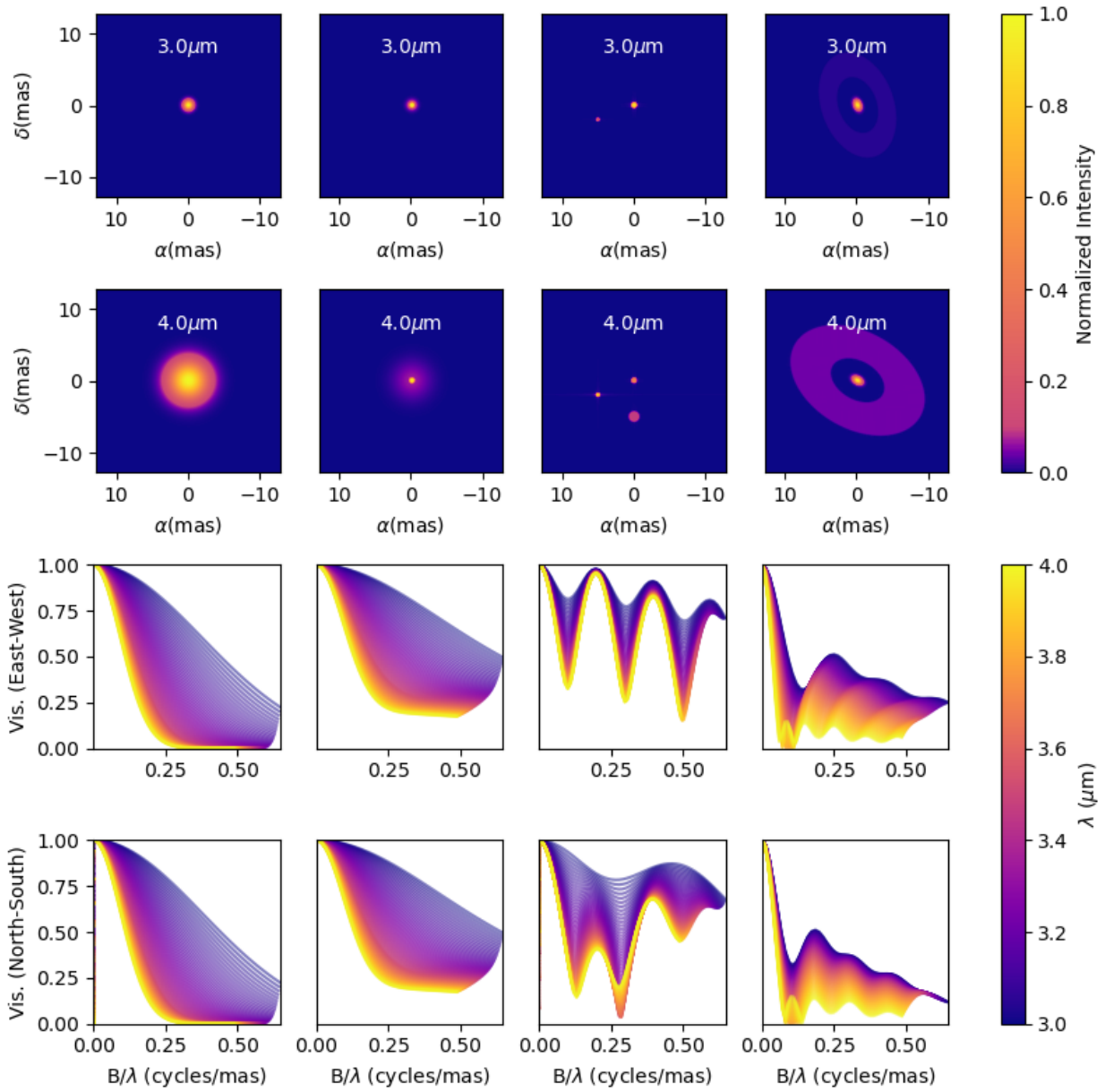
No module is complete and have been fully verified up to now!

CHAPTER

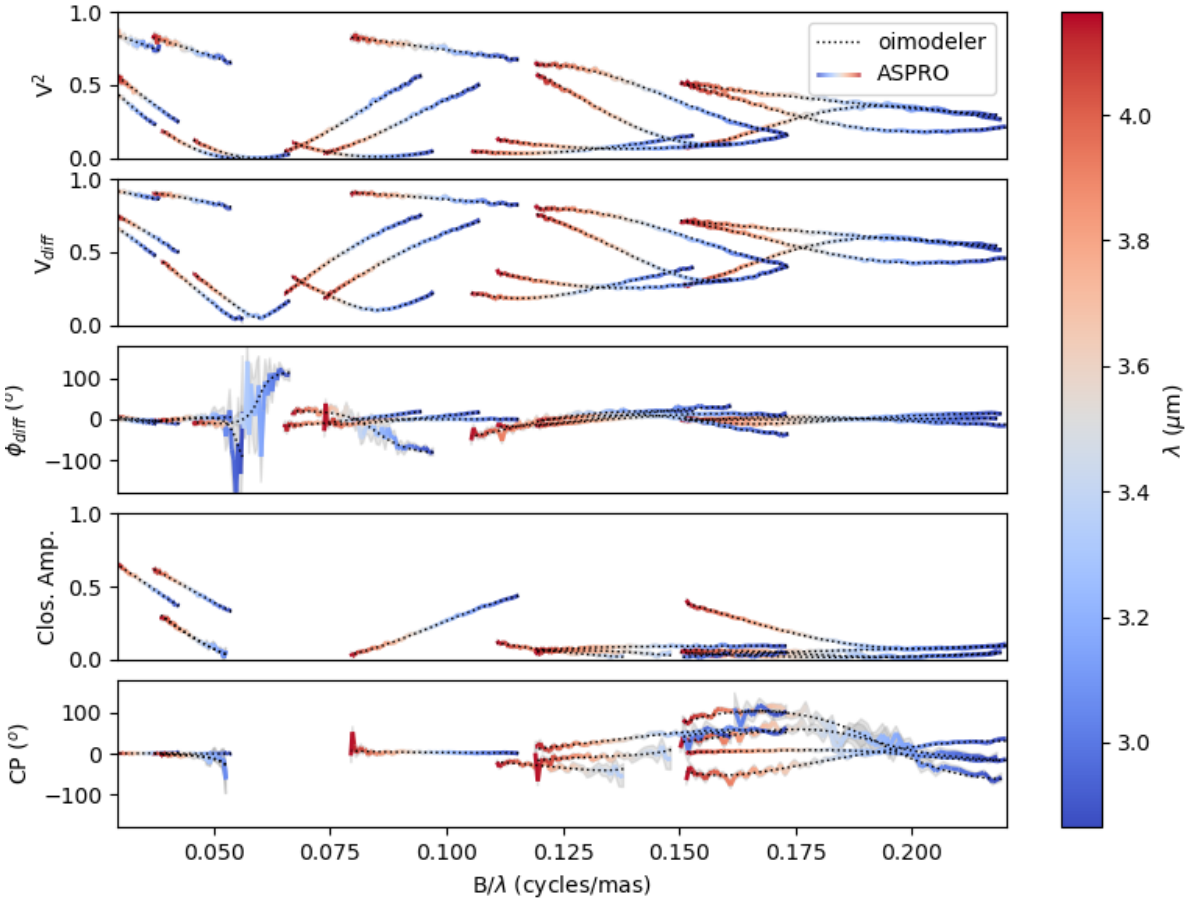
ONE

A FEW EXAMPLES

Here are some plots for the createModelChromatic.py example showing various chromatic-geometric models and the corresponding simulated visibilities.



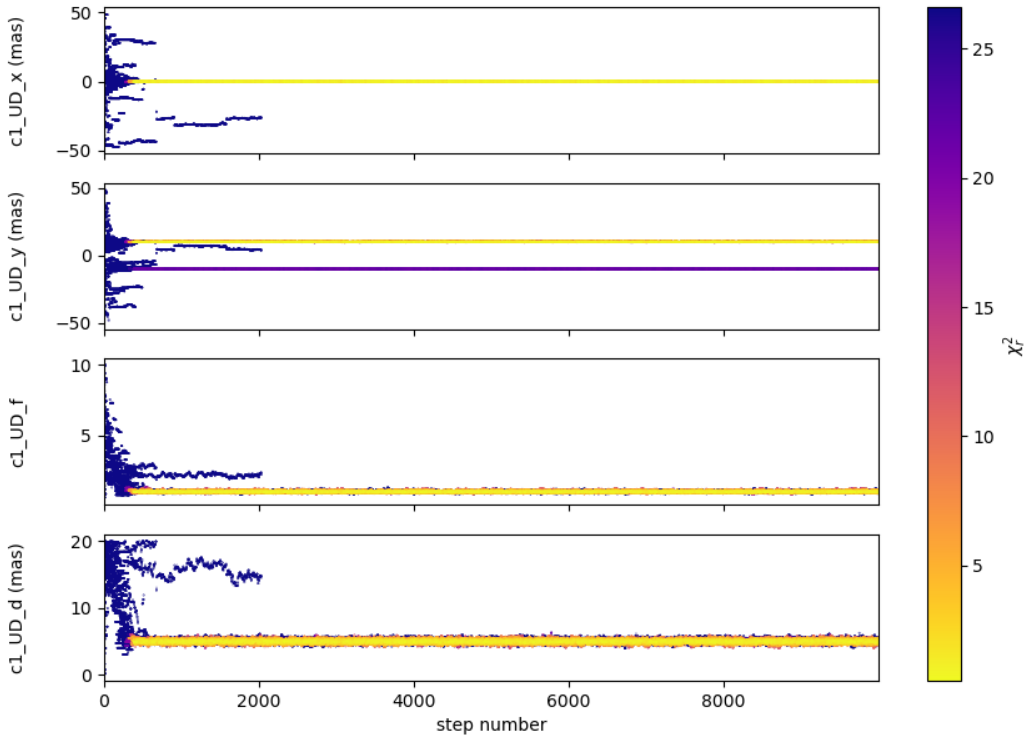
Here is an example from the *createSimulator.py* script showing high-end plots of some MATISSE LM-band data and a model create with `oimodeler`. In that case the data were simulated using the APSRO software from JMMC.

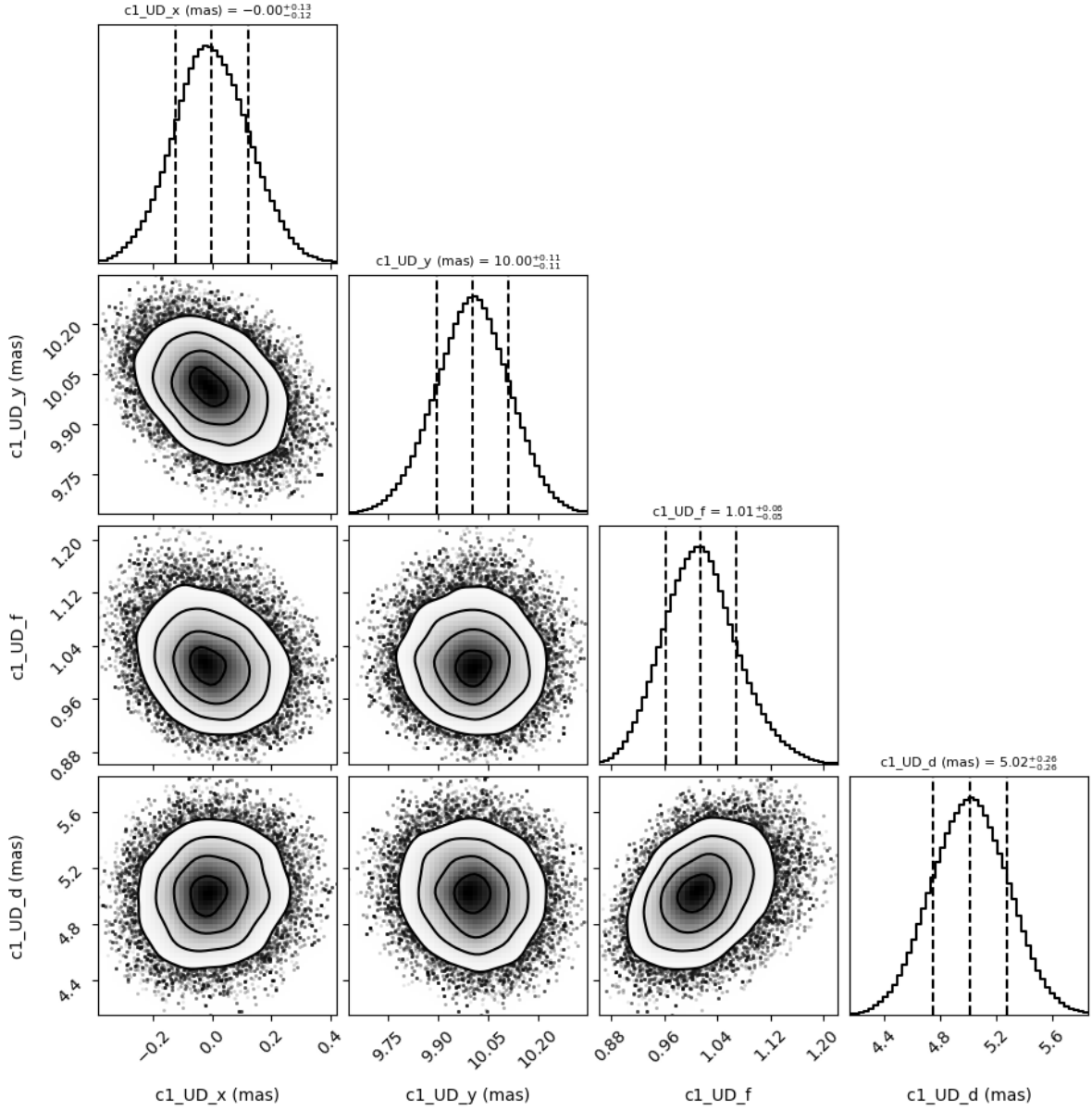


Here is an example from the *simpleFitEmcee.py* script showing:

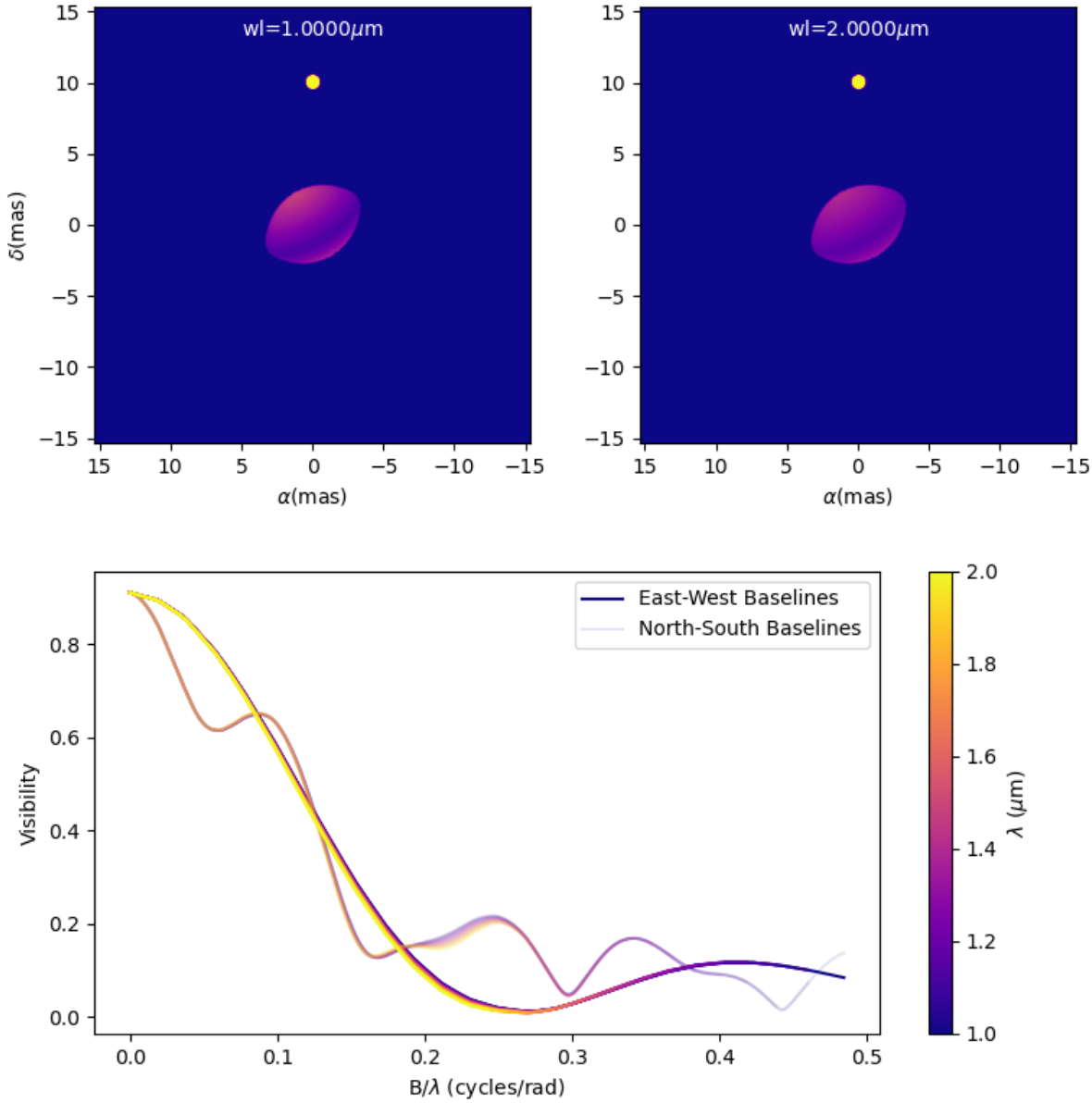
- The parameters values (and corresponding chi2 as a colorscale) for walkers from a emcee run on a binary model
- The famous corners plots for the 4 free parameters: x and y positions of the binary, diameter and flux

Again, the data is simulated with APSRO.


c1_UD_x (mas)
c1_UD_y (mas)
c1_UD_f
c1_UD_d (mas)
step number
χ^2_r



Here is a plot showing a model consisting of a fast rotating star plus a uniform disk. Chromatic images of fast rotator are computed with an external function encapsulated into a `oimodeler` component. The uniform disk is a simple Fourier-based component. The code is in the createCustomComponentImageFastRotator.py


wl=1.0000μm
wl=2.0000μm
δ(mas)
α(mas)
α(mas)
East-West Baselines
North-South Baselines
Visibility
B/λ (cycles/rad)
λ (μm)

CHAPTER

TWO

LICENSE

CHAPTER

THREE

ACKNOWLEDGMENT

The `oimodeler` package is developed with support from the VLTI/MATISSE consortium and the ANR project MASSIF.

CHAPTER

FOUR

CONTACT

If you have any question, comments or want to extend `oimodeler` contact Anthony Meilland. Additionally, you can use the Github issues for bug reports or feature requests or make a pull-request on the repository.

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 Overview

The `oimodeler` package is a project aiming at developing a modular and easily expandable python-based modelling software for optical interferometry.

Note: The project started end of 2021, and the software, although fully functional, is currently in an early stage of development.

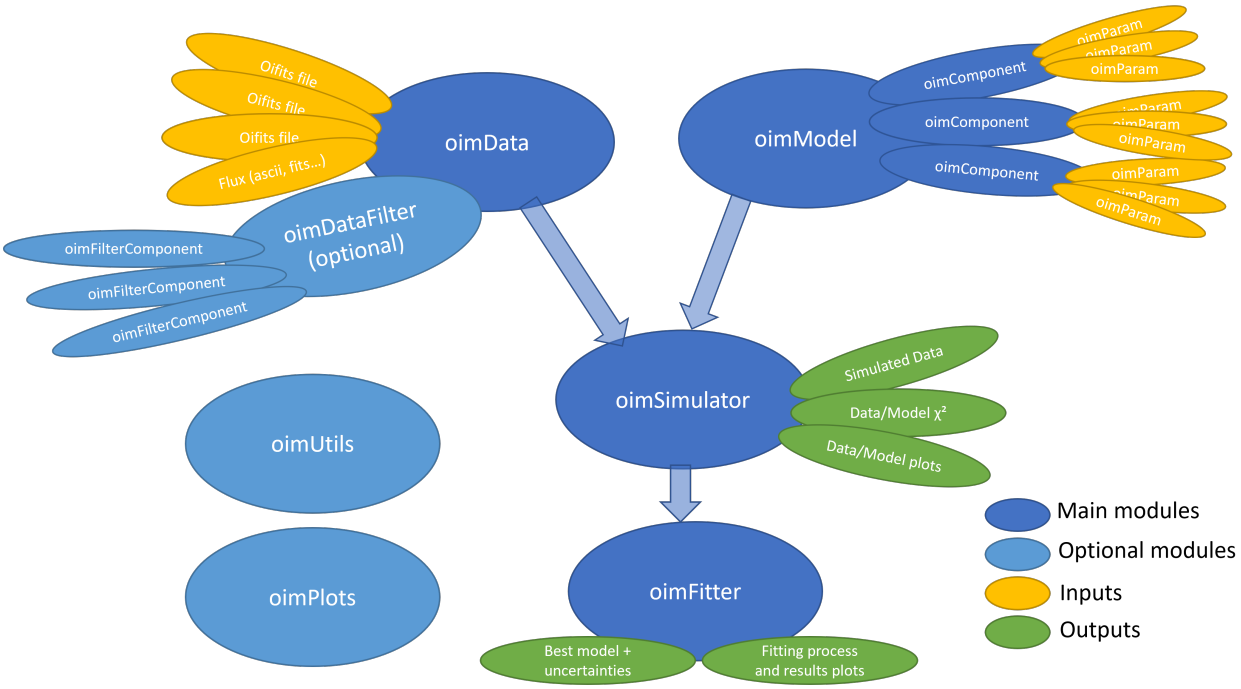
5.1.1 Software modularity

As described in the following and shown in the diagram below, `oimodeler` is a modular software:

- Models can be created with the `oimModel` module and various base components contained in the `oimComponent` module which contain model parameters from the `oimParam` module.
- Interferometric data can be loaded with functionality from the `oimData` module from standard OIFITS files. This module also allows to load flux/spectra in various formats with the `oimData.oimData` class that can optionallly be filtered using the `oimData.oimDataFilter` class with various filters from the `oimDataFilter` module.
- Data simulation/calculation can be done with the `oimSimulator` module that takes `oimData.oimData` and `oimModel.oimModel` classes as input to simulate data from the model at the same spatial/spectral coordinates as the data. The module also allows to compute the model/data chi2.
- The fitting is done with fitters contained in the `oimFitter` module that as its input also takes the `oimData.oimData` and `oimModel.oimModel` classes to perform model fitting.

  Note: Currently only a simple emcee fitter is implemented.

- The `oimPlots` module contains plotting functions for OIFITS data and `oimodeler` objects.
- The `oimUtils` module contains various functions to manipulate OIFITS data.



5.1.2 Modules

5.1.2.1 oimModel

The `oimModel` module is dedicated to the creation of models for optical interferometry. The models are modular and composed of one or many `oimComponent.oimComponent` objects. They can produce complex coherent flux and images and can subsequently be plugged into an `oimSimulator` object and/or any of the fitters contained in the `oimFitter` for data analysis and/or modelling.

5.1.2.2 oimComponent

The `oimComponent` module deals with model components that can be defined analytically in the Fourier plane (e.g., Uniform Disks, 2D-Gaussian distribution, etc.) or image plane (useful if there exists no analytical formula in the Fourier plane). An `oimComponent` can also be used to wrap external code (functions computing images, radial profiles, and hyperspectral cubes), or image-fit files (for instance produced using a radiative transfer model). An `oimComponent` can, in addition, be easily inherited from to create new custom components.

5.1.2.3 oimParam

The `oimParam` module contains basic model parameters. It `oim.Param.oimParam` class is used for the definition of components' parameters (Built from any of the base classes from the `oimComponent` module). There are also parameter linkers (`oimParam.oimParamLinker`) as well as normalizers (`oimParam.oimParamNormalize`), and advanced parameter interpolators (`oimParam.oimParamInterpolator`) that allow to build chromatic and time dependent models.

5.1.2.4 oimData

The `oimData` module allows to encapsulate interferometric (also photometric and spectroscopic) data. The `oimData.oimData` class allows to retrieve the original data as a list of astropy.io.fits.hdulist (via the `oimData.data` attribute), but also provide optimization of the data as vector/structure for faster model fitting.

Warning: Photometric and spectroscopic data not yet implemented!

5.1.2.5 oimDataFilter

The `oimDataFilter` module is dedicated to filtering and modifying data contained in `oimData.oimData` classes. It allows data selection (truncating, removing arrays, and flagging) based on various criteria (wavelengths, SNR . . .), and other data manipulation, such as smoothing and binning of the data.

5.1.2.6 oimSimulator

The `oimSimulator` module is the basic module for model (`oimModel.oimModel`) and data (`oimData.oimData`) comparison. It allows to simulate a new dataset (stored in the `oimSimulator.simulatedData` attribute) with the same quantities and spatial/spectral coordinates of the data and a model. It also allows to compute `chi2` for data and model comparison. The `oimSimulator.oimSimulator` class is fully compatible with the OIFITS2 format and can simulated any kind of data type from an OIFITS file (VIS2DATA, VISAMP in absolute, differential and correlated flux).

5.1.2.7 oimFitter

The `oimFitter` module is dedicated to model fitting. The parent class `oimFitter.oimFitter` is an abstract class that can be inherited from to implement various fitters. Currently. only a simple emcee-based fitter is implemented with `oimFitter.oimEmceeFitter`.

5.1.2.8 oimPlots

The `oimPlots` module contains various plotting tools for OIFITS data and `oimodeler` objects. The `oimPlots.oimAxes` is a subclass of the matplotlib.pyplot.Axes class with methods dedicated to produce plotted OIFITS data from the astropy.io.fits.hdulist format.

5.1.2.9 oimUtils

The `oimUtils` module contains various functions to manipulate OIFITS data such as,

- Retrieving baselines names, length, orientation, getting spatial frequencies
- Creating new OIFITS arrays in astropy.io.fits.hudlist format.

and more.

5.1.3 Expandability

`oimodeler` is written to allow easy implementation of new model component, fitters, data filters, parameter intepolators, data loader (e.g., for non-OIFITS format data), and plots. Feel free to contact Anthony Meilland if you developped custom features and want them to be included in the `oimodeler` distribution or make a pull-request on the Github repository and become a `oimodeler` **contributer**. For bug-reports and feature requests, please use the GitHub issue tracker.

5.2 Installation

The `oimodeler` package can be installed directly using the pip install command:

```
$ pip install git+https://github.com/oimodeler/oimodeler
```

Warning: The examples including dedicated data won't be installed when using pip in this way. They are available on the Github archive for manual download.

Alternatively, you can install the complete `oimodeler` package (including examples and data) by cloning the Github repository:

```
$ git clone https://github.com/oimodeler/oimodeler
$ cd oimodeler/
```

and installing it. As a non-editable install:

```
$ pip install .
```

Or as an editable (development) install:

```
$ pip install -e .
```

5.2.1 Dependancies

The `oimodeler` package requires the following packages:

- numpy
- scipy
- matplotlib
- astropy
- astroquery
- emcee
- corner
- tqdm
- pyFFTW (optional)

And for development some optional packages might be useful:

- pytest

- pytest-cov
- sphinx
- sphinx-autobuild
- sphinx-autodoc-typehints
- sphinx_rtd_theme
- numpydoc

These packages (except the development dependencies) are automatically installed if missing when installing `oimodeler`.

5.2.2 Checking installation

The check if `oimodeler` is properly installed you can run the *Getting Started* or other *Examples* scripts.

5.3 Getting Started

You can download the complete `gettingStarted` script from here.

Note: For this example we will use some OIFITS files located in the examples/testData/ASPRO_MATISSE2 subdirectory of the `oimodeler` Github repository.

If you did not clone the github repository, you will need to manually download the directory containing the examples/.

These data are actually a "fake" dataset simulated with the APSRO software from the JMMC. ASPRO was used to create three MATISSE observations of a binary star with one resolved component. ASPRO allows to us to add realistic noise on the interferometric quantities.

Let's start by importing the `oimodeler` package and specifiying the used paths/directories.

```
from pprint import pprint
from pathlib import Path

import oimodeler as oim


path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "ASPRO_MATISSE2"
save_dir = path / "images"
if not save_dir.exists():
    save_dir.mkdir(parents=True)

files = list(map(str, data_dir.glob("*.fits")))
```

If the `data_dir` is corretly set the `files` variable should be a list of paths to three OIFITS files.

> **Warning:** If you should try to write in a write-protected directory then python will yield an error. To resolve this, you can change the `save_dir` to another directory.
>
> For some examples there is a second `product_dir`, which, in this case, also needs to be changed.

Let's create a simple binary model with one resolved component. It will be built using two components:

- A point source created with the `oimPt` class
- A uniform disk create with the `oimUD` class

The point source has only three parameters, its coordinates x and y (in mas by default) and it flux f. Note that all components parameters are instances of the `oimParam` class. The uniform disk has four parameters: x, y, f, and the disk diameter d which is given in mas by default. If parameters are not set explicitly when the component is created they are set to their default value (e.g., 0 for x, y, and d and 1 for f).

```
ud = oim.oimUD(d=3, f=0.5, x=5, y=-5)
pt = oim.oimPt(f=1)
```

We can print the description of the component easily:

```
pprint(ud)
```

```
... Uniform Disk x=5.00 y=-5.00 f=0.50 d=3.00
```

If we want the details of one of the component parameters, we can access it by its name in the directory `params` of the component. For instance to access the diameter d of the previously created uniform disk:

```
pprint(ud.params['d'])
```

```
... oimParam d = 3 ± 0 mas range=[-inf,inf] free
```

The same is possible for the x coordinate:

```
pprint(ud.params['x'])
```

```
... oimParam x = 5 ± 0 mas range=[-inf,inf] fixed
```

Note that the x parameter is fixed by default (for model fitting) whereas the diameter d is free. The `oimParam` instance also contains the unit (accessible via the `oimParam.unit` attribute as an `astropy.units` object), uncertainties(via `oimParam.error`), and a range for model fitting (via `oimParam.mini` for the lower and `oimParam.maxi` for the upper bound). There are various way of accessing and modifying the value of the parameter or one of its other associated quantities (see the *basic model example* for more details).

For our example, we want to have the coordinates of the uniform disk as free parameters and set them to a range of 50 mas. We will explore a diameter between 0.01 and 20 mas and the flux between 0 and 10. On the other hand, the flux of the point source will be left to a fixed value of one.

```
ud.params['d'].set(min=0.01, max=20)
ud.params['x'].set(min=-50, max=50, free=True)
ud.params['y'].set(min=-50, max=50, free=True)
ud.params['f'].set(min=0., max=10.)
pt.params['f'].free = False
```

Finally, we can build our model consisting of these two components.

```
model = oim.oimModel(ud, pt)
```

We can print all the model's parameters:

```
model.getParameters()
```

```
... {'c1_UD_x': oimParam at 0x1670462cca0 : x=5 ± 0 mas range=[-50,50] free=True,
     'c1_UD_y': oimParam at 0x1670462cac0 : y=-5 ± 0 mas range=[-50,50] free=True,
     'c1_UD_f': oimParam at 0x1670462cd60 : f=0.5 ± 0  range=[0.0,10.0] free=True,
     'c1_UD_d': oimParam at 0x1670462ca90 : d=3 ± 0 mas range=[0.01,20] free=True,
     'c2_Pt_x': oimParam at 0x1670462cc70 : x=0 ± 0 mas range=[-inf,inf] free=False,
     'c2_Pt_y': oimParam at 0x1670462cb80 : y=0 ± 0 mas range=[-inf,inf] free=False,
     'c2_Pt_f': oimParam at 0x167055de490 : f=1 ± 0  range=[-inf,inf] free=False}
```

Or only the free parameters:

```
pprint(model.getFreeParameters())
```

```
... {'c1_UD_x': oimParam at 0x167055ded30 : x=5 ± 0 mas range=[-50,50] free=True,
     'c1_UD_y': oimParam at 0x167055deca0 : y=-5 ± 0 mas range=[-50,50] free=True,
     'c1_UD_f': oimParam at 0x167055dec70 : f=0.5 ± 0  range=[0.0,10.0] free=True,
     'c1_UD_d': oimParam at 0x167055de850 : d=3 ± 0 mas range=[0.01,20] free=True}
```

Let's now compare our data and our model. We will use the class `oimSimulator` that will compute simulated data from our model at the spatial (and optionally, spectral and temporal) frequencies/coordinates from our data.

```
sim = oim.oimSimulator(data=files, model=model)
sim.compute(computeChi2=True, computeSimulatedData=True)
```

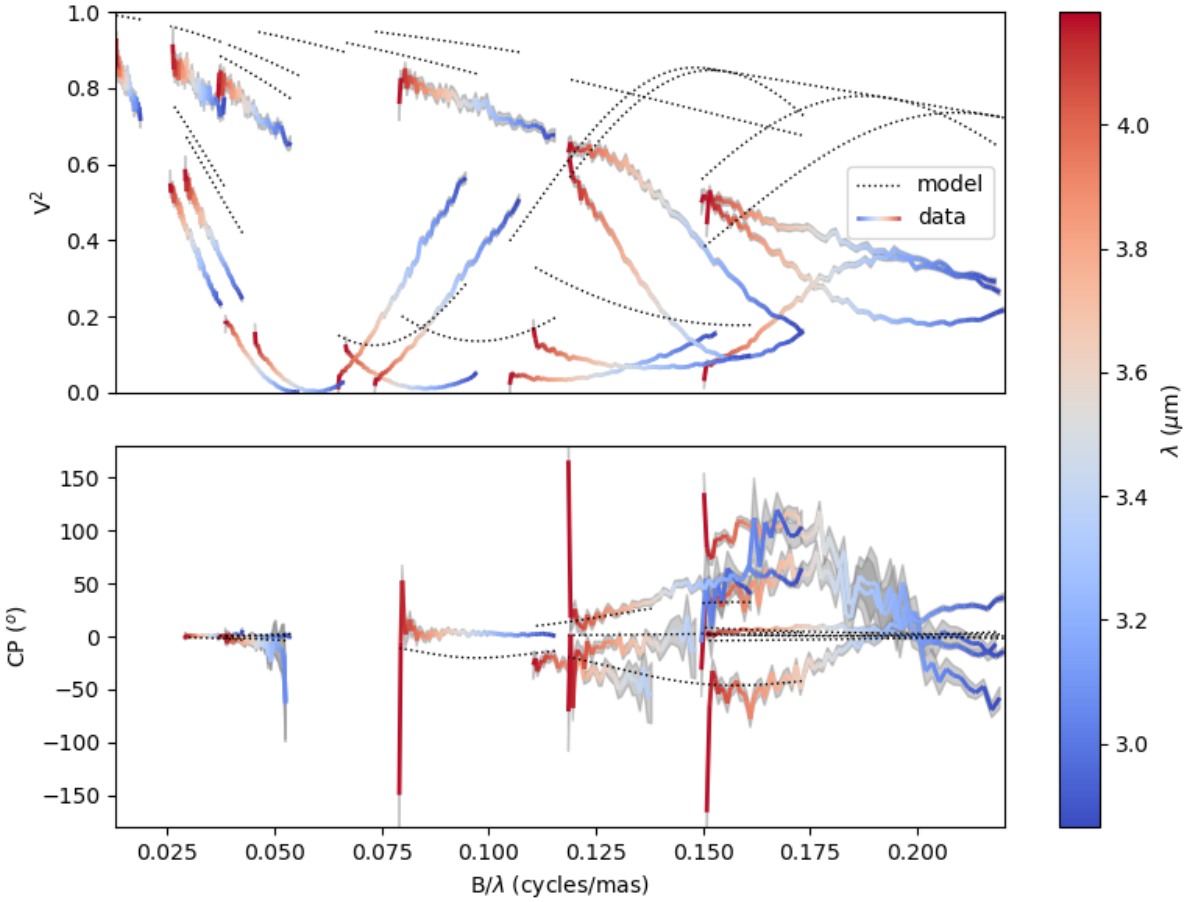
let's print the χ^2_r from our data/model comparison:

```
pprint("Chi2r = {}".format(sim.chi2r))
```

```
... Chi2r = 22510.099167065073
```

Obviously, our model is quite bad. Let's plot a model/data comparison for the square visibility (VIS2DATA) and closure phase (T3PHI):

```
fig0, ax0 = sim.plot(["VIS2DATA", "T3PHI"])
```



The figure `fig0` and axes list `ax0` are returned by the `oimSimulator.plot` method. You can directly save the figure using the `savefig=file_name` keyword.

The `oimSimulator` class doesn't do model-fitting but only data/model comparison. To perform model-fitting we will use the `oimFitterEmcee` class. This class encapsulates the famous emcee implementation of Goodman & Weare's Affine Invariant Markov chain Monte Carlo (MCMC) Ensemble sampler.

Here, we create a simple `emcee` fitter with 10 independent walkers. We can either give the fitter a `oimSimulator` class or some data (as a `oimData` object or list of filenames) and a `oimModel` class.

```
fit = oim.oimFitterEmcee(files, model, nwalkers=10)
```

Before running the fit, we need to prepare our fitter for the mcmc run. We choose to initialize an array of 10 walkers to a uniform random distribution within the range given in the model parameters with `min` and `max`.

```
fit.prepare(init="random")
```

Note: An other possible option for the mcmc fitter initialization is "gaussian". In that case the fitter will initialize the parameters with Gaussian distributions centered on the current `value` of each parameter and with a fwhm equal to its `error` variable.

The initial parameters are stored in the `initialParams` member variable of the fitter.

```
pprint(fit.initialParams)
```

```
... [[30.26628081  26.02405335   7.23061417  19.19829182]
    [ 23.12647935  44.07636861   3.39149131  17.29408761]
    [ -9.311772    47.50156564   9.49185499   4.79198633]
    [-24.05134905 -12.45653228   5.36560382   0.29631924]
    [-28.13992968 -25.25330839   9.64101194   6.21004462]
    [  5.13551292  25.3735599    4.82365667   0.53696176]
    [  3.6240551  -41.03297919   4.79235224   7.12035193]
    [-10.57430397 -40.19561341   6.0687408   11.22285079]
    [ 12.76468252  16.83390367   4.40925502   5.64248841]
    [ 29.12590452  -0.20420277   4.21541399  13.16022251]]
```

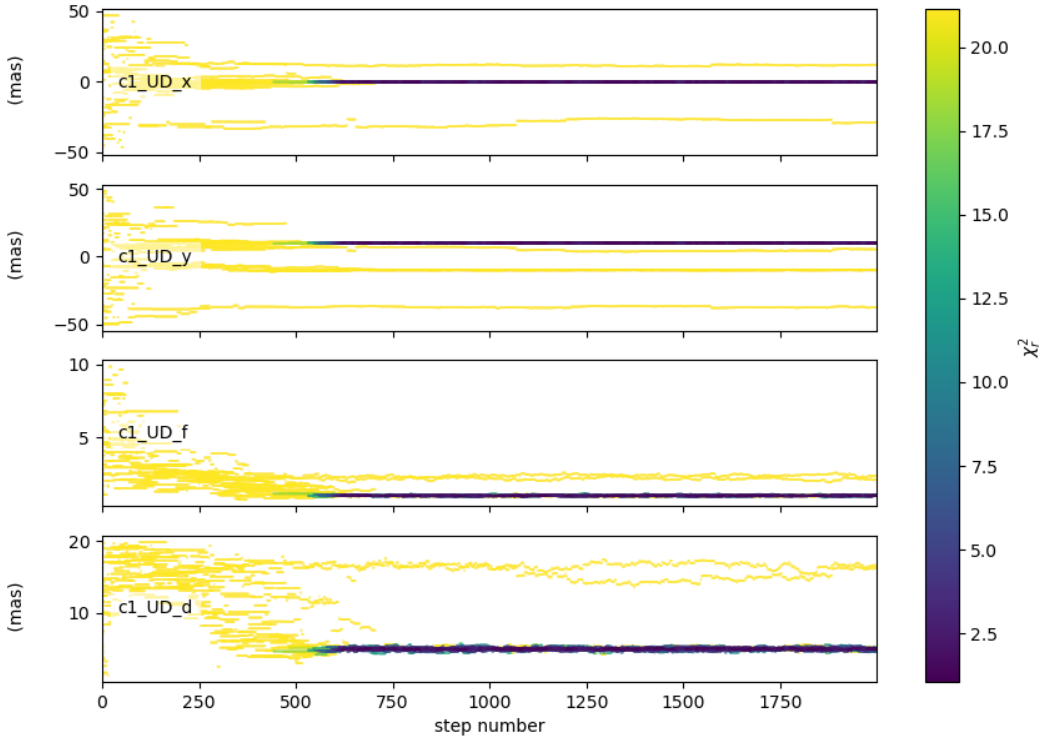
Now we run the fit on 2000 steps. It will compute 20000 models (i.e., `nsteps` x `nwalkers`).

```
fit.run(nsteps=2000, progress=True)
```

```
... 17%|        | 349/2000 [00:10<00:48, 34.29it/s]
```

After the run we can plot the values of the 4 free-parameters for the 10 walkers as a function of the steps of the mcmc run.

```
figWalkers, axeWalkers = fit.walkersPlot()
```



After a few hundred steps most walkers converge to the same position having a good χ^2_r. However, from that figure

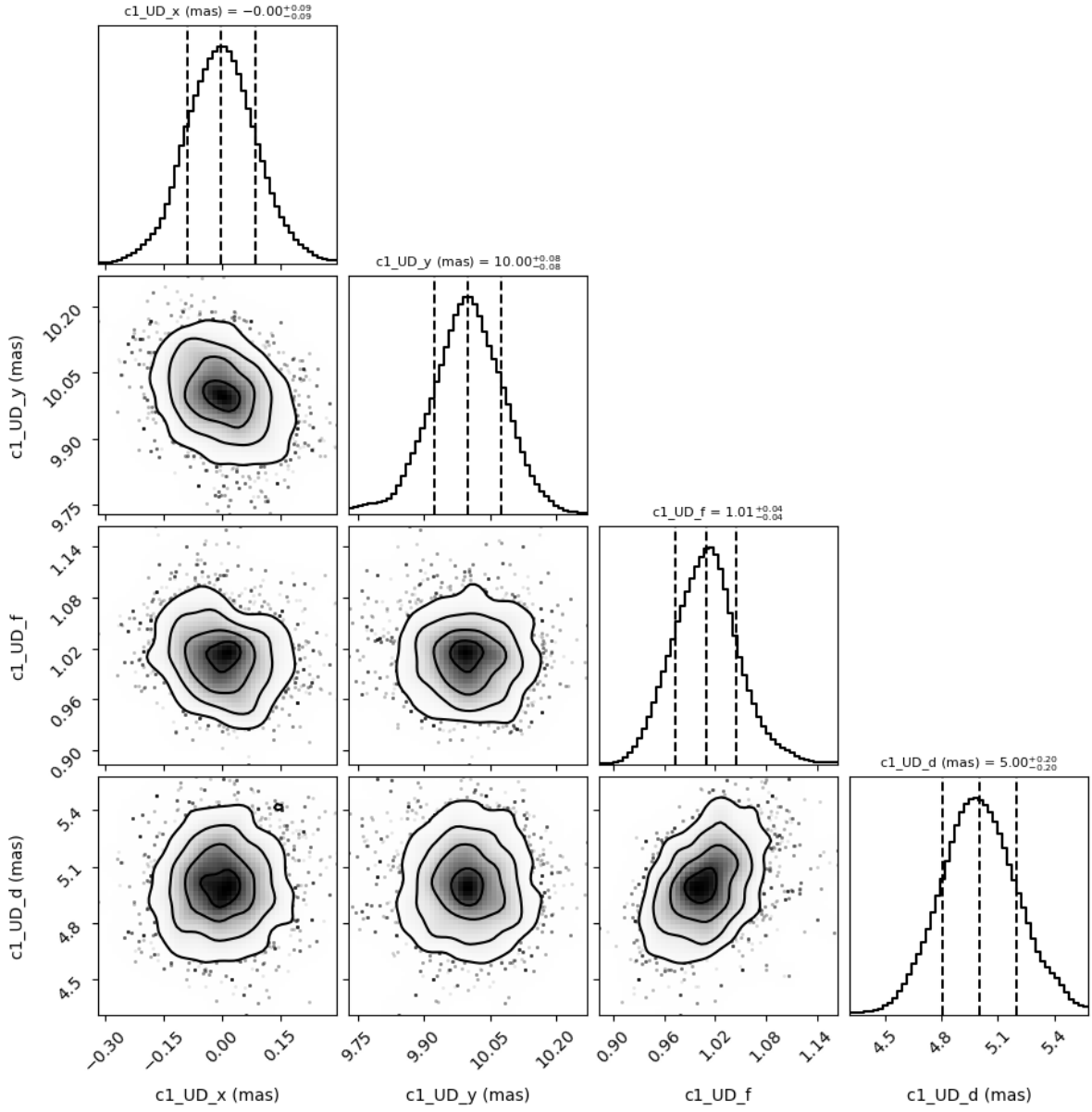
will clearly see that:

- Not all walkers have converged after 2000 steps.
- Some walkers converge to a solution that gives significantly worse χ^2.

In optical interferometry there are often local minima in the χ^2 and it seems that some of our walkers are locked there. In our case, this minima are due to the fact that object is close be symmetrical if not for the fact than one of the component is resolved. Neverless, the χ^2 of the local minimum is about 20 times worse than the one of the global minimum.

We can plot the *famous* corner plot with the 1D and 2D density distributions. For this purpose, the `oimodeler` package uses the corner package. We will discard the 1000 first steps as most of the walkers have converged after that. By default, the corner plot also removes the data with a χ^2 greater than 20 times those of the best model. This option can be changed using the `chi2limfact` keyword in the `oimFitterEmcee.cornerPlot` method.

```
figCorner, axeCorner = fit.cornerPlot(discard=1000)
```



We now can retrieve the result of our fit. The `oimFitterEmcee` fitter can either return the `"best"`, the `"mean"` or the `"median"` model. It also returns uncertainties estimated from the density distribution (see emcee's documentation for more details on the statistics).

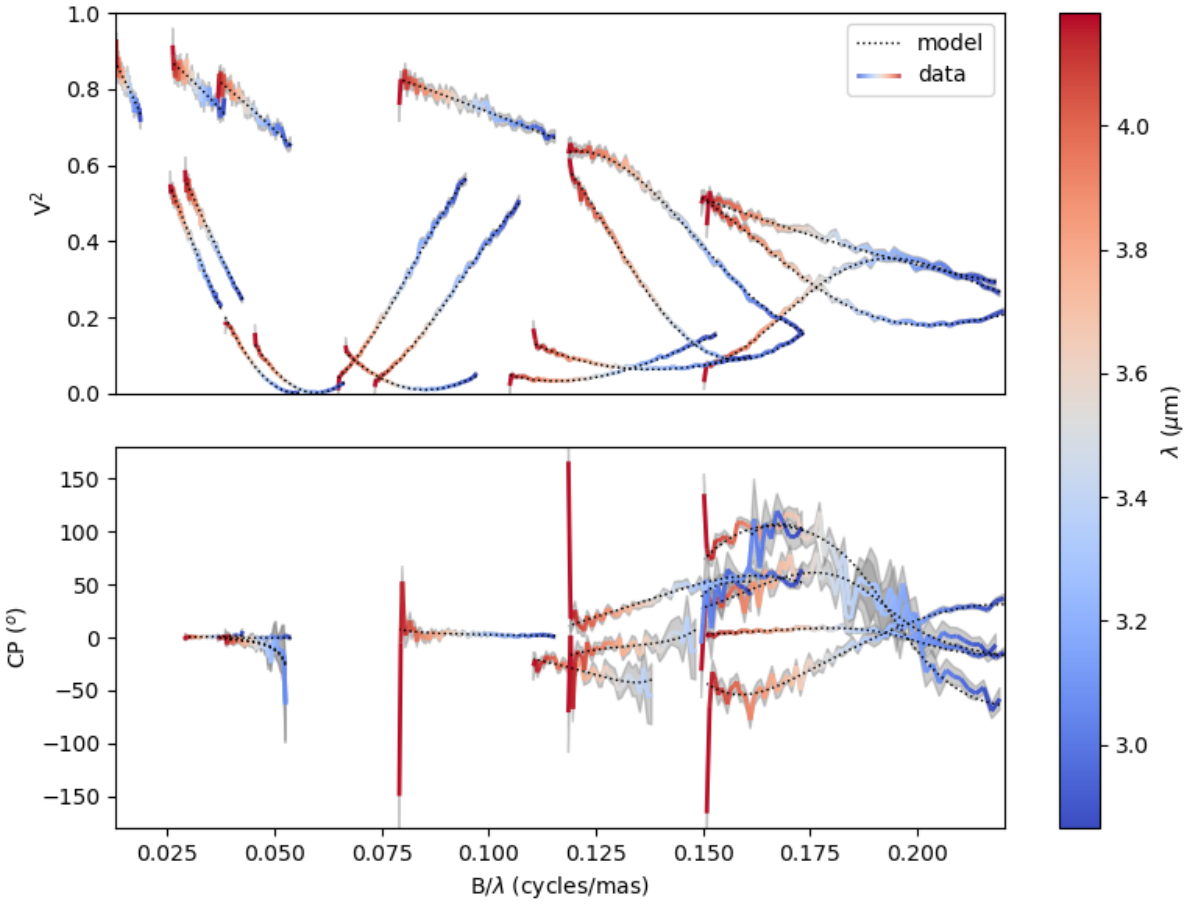
```
median, err_l, err_u, err = fit.getResults(mode='median', discard=1000)
```

To compute the median and mean models we use the `oimFitterEmcee.getResults` method and remove, as in the corner plot, the walkers that didn't converge within the limit set by the `chi2limitfact` keyword (default is 20). Furthermore, we also remove the steps of the burn-in phase with the `discard` keyword.

When procuring the fit's results, the simulated data with these values are also produced simultaneously in the fitter's internal simulator. We can plot the data/model and compute the final χ^2_r.

```
figSim, axSim = fit.simulator.plot(["VIS2DATA", "T3PHI"])
pprint("Chi2r = {}".format(fit.simulator.chi2r))
```

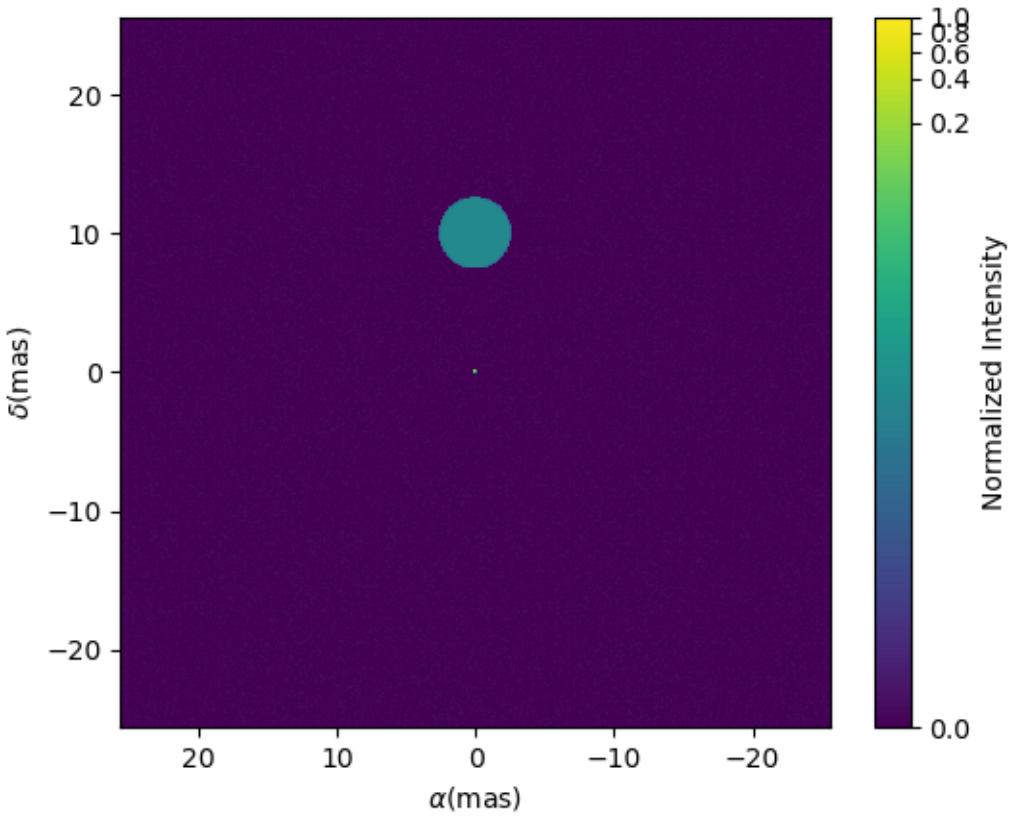
```
... Chi2r = 1.0833528313932081
```



That's better.

Finally, let's plot an image of the model with the best parameters. Here, we generate a `512x512` image with a 0.1 mas pixel size and a 0.1 power-law colorscale:

```
figImg, axImg, im=model.showModel(512, 0.1, normPow=0.1)
```



Here is our nice binary!

That's all for this short introduction.

If you want to go further you can have a look at the *Examples* or *API* sections.

5.4 Examples

All the following examples can be found in the examples subdirectories of the `oimodeler` github repository. Before looking at these examples, you might want to check the *Getting Started* page.

5.4.1 Basic Examples

In this section we presents script showing the basic functionalities of the `oimodeler` software.

5.4.1.1 Loading oifits data

The exampleOimData.py script shows how to create a `oimData` object from a list of OIFITS files and how the data is organized in an `oimData` instance.

```
from pathlib import Path
from pprint import pprint

import matplotlib.pyplot as plt
import numpy as np

import oimodeler as oim

path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "FSCMa_MATISSE"
files = list(map(str, data_dir.glob("*.fits")))

data=oim.oimData(files)
```

The OIFITS data, stored in the `astropy.io.fits.hdulist` format, can be accessed using the `oimData.data` attribute.

```
pprint(data.data)
```

```
... [[astropy.io.fits.hdu.image.PrimaryHDU object at 0x000002657CBD7CA0>, <astropy.io.
↪fits.hdu.table.BinTableHDU object at 0x000002657E546AF0>, <astropy.io.fits.hdu.table.
↪BinTableHDU object at 0x000002657E3EA970>, <astropy.io.fits.hdu.table.BinTableHDU␣
↪object at 0x000002657E3EAAC0>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E406520>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E402EE0>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E406FD0>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E4600D0>],
    [<astropy.io.fits.hdu.image.PrimaryHDU object at 0x000002657E458F70>, <astropy.io.
↪fits.hdu.table.BinTableHDU object at 0x0000026500769BE0>, <astropy.io.fits.hdu.table.
↪BinTableHDU object at 0x000002650080EA60>, <astropy.io.fits.hdu.table.BinTableHDU␣
↪object at 0x00000265007EA430>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x00000265007EAAF0>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002650080EC40>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E4DC820>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E4ECFD0>],
    [<astropy.io.fits.hdu.image.PrimaryHDU object at 0x000002657E4DCCA0>, <astropy.io.
↪fits.hdu.table.BinTableHDU object at 0x0000026500B7EB50>, <astropy.io.fits.hdu.table.
↪BinTableHDU object at 0x000002657E9F79D0>, <astropy.io.fits.hdu.table.BinTableHDU␣
↪object at 0x000002657E5913A0>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E591A60>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E591B20>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E5B7790>, <astropy.io.fits.hdu.table.BinTableHDU object at␣
↪0x000002657E5BAEB0>]]
```

Note: See the OIFITS standard for its conventions and the astropy.oi.fits documentation to learn how it is implemented.

To use the data efficiently in the `oimSimulator` class or any of the fitters contained in the `oimFitter` module they need to be optimized in a simpler vectorial/structure. This step is done automatically when using the simulator or fitter but can be done manually using the following command:

```
data.prepareData()
```

For instance, this create single vectors for the data coordinates:

- The u-axis of the spatial frequencies `data.vect_u`
- The v-axis of the spatial frequencies `data.vect_v`
- The wavelength data `data.vect_wl`

If we now print these we can see their structure:

```
pprint(data.vect_u)
pprint(data.vect_v)
pprint(data.vect_wl)

pprint(data.vect_u.shape)
```

```
... [0. 0. 0. ... 0. 0. 0.]
    [0. 0. 0. ... 0. 0. 0.]
    [4.20059359e-06 4.18150239e-06 4.16233070e-06 ... 2.75303296e-06
    2.72063039e-06 2.68776785e-06]

    (5376,)
```

5.4.1.2 Basic models

The basicModel.py script demonstrates the basic functionalities of the `oimModel` and `oimComponents` objects.

First we import the relevant packages:

```
from pathlib import Path
from pprint import pprint

import matplotlib.pyplot as plt
import numpy as np
import oimodeler as oim
```

A model is a collection of components. All components are derived from the `oimComponent` class. The components may be described in the image plane, by their intensity distribution, or directly in the Fourier plane, for components with known analytical Fourier transforms. In this example we will only focus on the latter type which are all derived from the `oimComponentFourier` class.

In the table below is a list of the current, from the `oimComponentFourier` derived, components.

Class	Description	Parameters
oimPt	Point source	x, y, f
oimBackground	Background	x, y, f
oimUD	Uniform Disk	x, y, f, d
oimEllipse	Uniform Ellipse	x, y, f, d, pa, elong
oimGauss	Gaussian Disk	x, y, f, fwhm
oimEGauss	Elliptical Gaussian Disk	x, y, f, fwhm, pa, elong
oimIRing	Infinitesimal Ring	x, y, f, d
oimEIRing	Elliptical Infinitesimal Ring	x, y, f, d, pa, elong
oimESKIRing	Skewed Infinitesimal Elliptical Ring	x, y, f, d, skw, skwPa, pa, elong
oimRing	Ring defined with din and dout	x, y, f, din, dout
oimERing	Elliptical Ring with din and dout	x, y, f, din, dout, pa, elong
oimESKRing	Skewed Elliptical Ring	x, y, f, din, dout, skw, skwPa, pa, elong
oimRing2	Ring defined with d and dr	x, y, f, d, dr
oimERing2	Elliptical Ring with d and dr	x, y,f, d, dr, pa, elong
oimLinearLDD	Linear Limb Darkened Disk	x, y, f, d, a
oimQuadLDD	Quadratic Limb Darkened Disk	x, y, f, d, a1, a2
oimLorentz	Pseudo-Lorenztian	x, y, fwhm
oimELorentz	Elliptical Pseudo-Lorenztian	x, y, f, fwhm, pa, elong
oimConvolutor	Convolution between 2 components	Parameters from the 2 components

To create models we must first create the components. Let's create a few simple components.

```
pt = oim.oimPt(f=0.1)
ud = oim.oimUD(d=10, f=0.5)
g  = oim.oimGauss(fwhm=5, f=1)
r  = oim.oimIRing(d=5, f=0.5)
```

Here, we have create a point source components, a 10 mas uniform disk, a Gaussian distribution with a 5 mas fwhm and a 5 mas infinitesimal ring.

Note that the model parameters which are not set explicitly during the components creation are set to their default values (i.e., f=1, x=y=0).

We can print the description of the component easily:

```
pprint(ud)
```

```
... Uniform Disk x=0.00 y=0.00 f=0.50 d=10.00
```

Or if you want to print the details of a parameter:

```
pprint(ud.params['d'])
```

```
... oimParam d = 10 ± 0 mas range=[-inf,inf] free
```

Note that the components parameters are instances of the `oimParam` class which hold not only the parameter value stored in the `oimParam.value` attribute, but in addition to it the following attributes:

- `oimParam.error`: the parameters uncertainties (for model fitting).
- `oimParam.unit`: the unit as a `astropy.units` object.
- `oimParam.min`: minimum possible value (for model fitting).

- `oimParam.max`: minimum possible value (for model fitting).
- `oimParam.free`: Describes a free parameter for `True` and a fixed parameter for `False` (for model fitting).
- `oimParam.description`: A string that describes the model parameter.

We can now create our first models using the `oimModel` class.

```
mPt   = oim.oimModel(pt)
mUD   = oim.oimModel(ud)
mG    = oim.oimModel(g)
mR    = oim.oimModel(r)
mUDPt = oim.oimModel(ud, pt)
```

Now, we have four one-component models and one two-component model.

We can get the parameters of our models using the `oimModel.getParameter` method.

```
params = mUDPt.getParameters()
pprint(params)
```

```
... {'c1_UD_x': oimParam at 0x23de5c62fa0 : x=0 ± 0 mas range=[-inf,inf] free=False ,
     'c1_UD_y': oimParam at 0x23de5c62580 : y=0 ± 0 mas range=[-inf,inf] free=False ,
     'c1_UD_f': oimParam at 0x23de5c62400 : f=0.5 ± 0  range=[-inf,inf] free=True ,
     'c1_UD_d': oimParam at 0x23debc1abb0 : d=10 ± 0 mas range=[-inf,inf] free=True ,
     'c2_Pt_x': oimParam at 0x23debc1a8b0 : x=0 ± 0 mas range=[-inf,inf] free=False ,
     'c2_Pt_y': oimParam at 0x23debc1ab80 : y=0 ± 0 mas range=[-inf,inf] free=False ,
     'c2_Pt_f': oimParam at 0x23debc1ac10 : f=0.1 ± 0  range=[-inf,inf] free=True }
```

The method returns a dict of all the model component's parameters. The keys are defined as `x{num of component}_{short Name of component}_{param name}`.

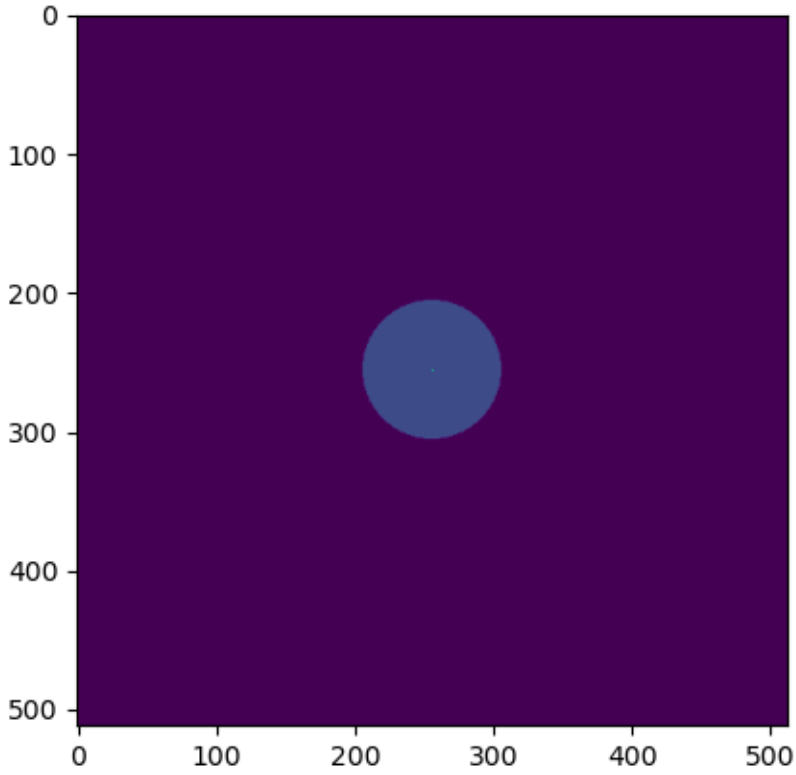
Alternatively, we can get the free parameters using the `getFreeParameters` method:

```
freeParams = mUDPt.getParameters()
pprint(freeParams)
```

```
... {'c1_UD_f': oimParam at 0x23de5c62400 : f=0.5 ± 0  range=[-inf,inf] free=True ,
     'c1_UD_d': oimParam at 0x23debc1abb0 : d=10 ± 0 mas range=[-inf,inf] free=True ,
     'c2_Pt_f': oimParam at 0x23debc1ac10 : f=0.1 ± 0  range=[-inf,inf] free=True }
```

The `oimModel` class can return an image of the model using the `oimModel.getImage` method. It takes two arguments, the image's size in pixels and the pixel size in mas.

```
im = mUDPt.getImage(512, 0.1)
plt.figure()
plt.imshow(im**0.2)
```



We plot the image with a 0.2 power-law to make the uniform disk components visible: Both components have the same total flux but the uniform disk is spread on many more pixels.

The image can also be returned as an `astropy hdu` object (instead of a `numpy array`) setting the `toFits` keyword to `True`. The image will then contained a header with the proper fits image keywords (NAXIS, CDELT, CRVAL, etc.).

```
im = mUDPt.getImage(256, 0.1, toFits=True)
pprint(im)
pprint(im.header)
pprint(im.data.shape)
```

```
... <astropy.io.fits.hdu.image.PrimaryHDU object at 0x000002610B8C22E0>

SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  256
NAXIS2  =                  256
EXTEND  =                    T
CDELT1  = 4.84813681109536E-10
CDELT2  = 4.84813681109536E-10
CRVAL1  =                    0
CRVAL2  =                    0
CRPIX1  =                128.0
```

(continues on next page)

(continued from previous page)

```
CRPIX2  =                128.0
CUNIT1  = 'rad     '
CUNIT2  = 'rad     '
CROTA1  =                    0
CROTA2  =                    0

(256, 256)
```

Note: Currently only **regular** grids in wavelength and time are allowed when exporting to fits-image format. If specified, the *wl* and *t* vectors need to be regularily sampled. The easiest way is to use the numpy.linspace function.

If their sampling is irregular an error will be raised.

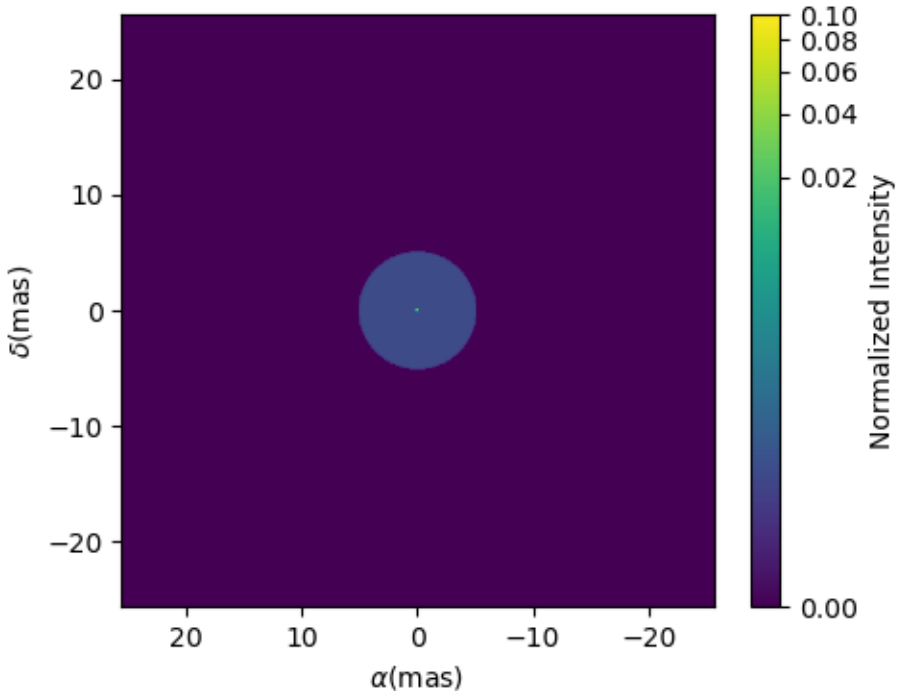
Using the `oimModel.saveImage` method will also return an image in the fits format and save it to the specified fits file.

```
im = mUDPt.saveImage("modelImage.fits", 256, 0.1)
```

Note: The returned image in fits format will be 2D, if time and wavelength are not specified, or if they are numbers, 3D if one of them is an array, and 4D if both are arrays.

Alternatively, we can use the `oimModel.showModel` method which take the same argument as the getImage, but directly create a plot with proper axes and colorbar.

```
figImg, axImg = mUDPt.showModel(512, 0.1, normPow=0.2)
```



In other examples, we use `oimModel` and `oimData` to create data objects and pass them to a `oimSimulator` instance to simulate interferometric quantities from the model at the spatial frequencies from our data. Without the `oimSimulator` class, the `oimModel` can only produce complex coherent flux (i.e., non normalized complex visibility) for a vector of spatial frequecies and wavelengths.

```
wl = 2.1e-6
B = np.linspace(0.0, 300, num=200)
spf = B/wl
```

Here, we have created a vector of 200 spatial frequencies, for baselines ranging from 0 to 300 m at an observing wavelength of 2.1 microns.

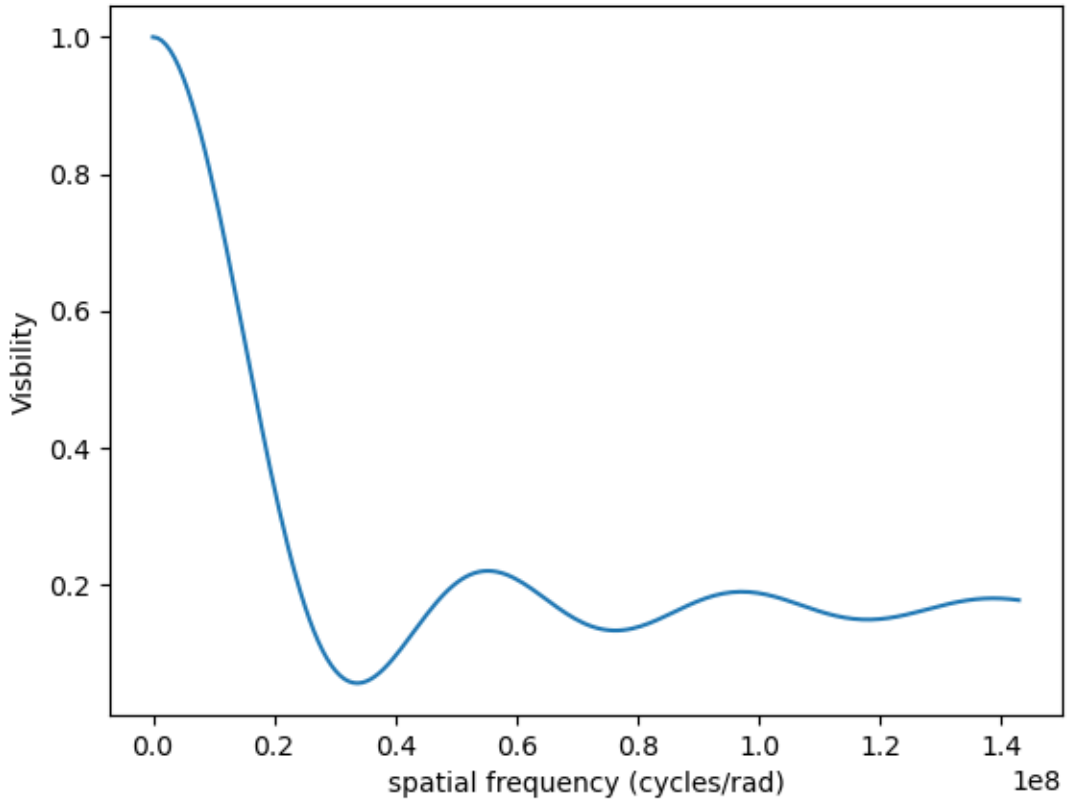
We can now use this vector to get the complex coherent flux (CCF) from our model.

```
ccf = mUDPt.getComplexCoherentFlux(spf, spf*0)
```

The `oimModel.getComplexCoherentFlux` method takes four parameters: The spatial frequencies along the East-West axis, the spatial frequencies along the North-South axis, and optionally, the wavelength and time (mjd). Here, we are dealing with grey and time-independent models so we don't need to specify the wavelength. And, as our models are circular, we don't care about the baseline orientation. That why we set the North-South component of the spatial frequencies to zero.

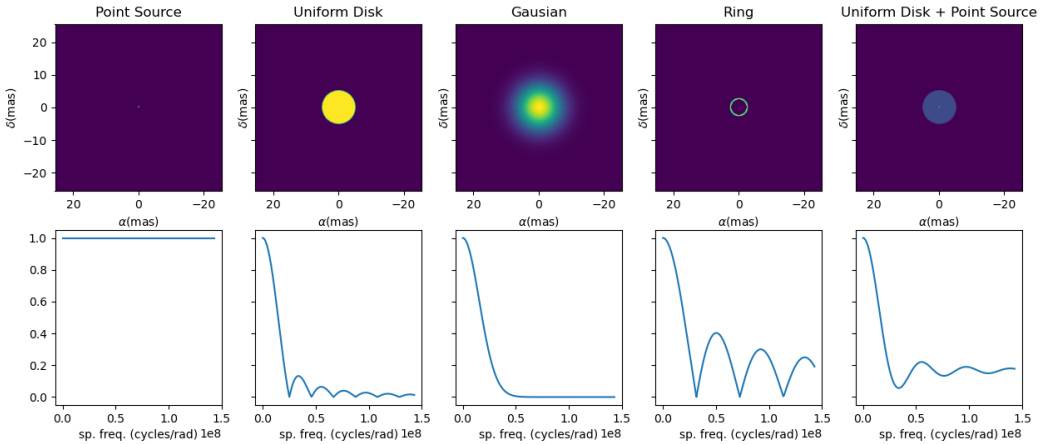
We can now plot the visibility from the CCF as the function of the spatial frequencies:

```
v = np.abs(ccf)
v = v/v.max()
plt.figure()
plt.plot(spf, v)
plt.xlabel("spatial frequency (cycles/rad)")
plt.ylabel("Visbility")
```



Let's finish this example by creating a figure with the image and visibility for all the previously created models.

```
models = [mPt, mUD, mG, mR, mUDPt]
mNames = ["Point Source", "Uniform Disk", "Gausian", "Ring",
          "Uniform Disk + Point Source"]

fig, ax = plt.subplots(2, len(models), figsize=(
    3*len(models), 6), sharex='row', sharey='row')

for i, m in enumerate(models):
    m.showModel(512, 0.1, normPow=0.2, axe=ax[0, i], colorbar=False)
    v = np.abs(m.getComplexCoherentFlux(spf,  spf*0))
    v = v/v.max()
    ax[1, i].plot(spf, v)
    ax[0, i].set_title(mNames[i])
    ax[1, i].set_xlabel("sp. freq. (cycles/rad)")
```



5.4.1.3 Precomputed fits-formated image

In the FitsImageModel.py script, we demonstrate the capability of building models using precomputed image in fits format.

In this example, we will use a semi-physical model for a classical Be star and its circumstellar disk. The model, detailed in Vieira et al. (2015) was taken from the AMHRA service of the JMMC.

Note: AMHRA develops and provides various astrophysical models online, dedicated to the scientific exploitation of high-angular and high-spectral facilities.

Currently available models are:

- Semi-physical gaseous disk of classical Be stars and dusty disk of YSO.
- Red-supergiant and AGB.
- Binary spiral for WR stars.
- Physical limb darkening models.
- Kinematics gaseous disks.
- A grid of supergiant B[e] stars models.

Let's start by importing oimodeler as well as useful packages.

```
from pathlib import Path
from pprint import pprint

import matplotlib.colors as colors
import matplotlib.cm as cm
import numpy as np
import oimodeler as oim
from matplotlib import pyplot as plt
```

The fits-formatted image-cube `BeDisco.fits` that we will use is located in the `examples/basicExamples` directory.

```
path = Path(__file__).parent.parent.parent
file_name = path / "examples" / "BasicExamples" / "BeDISCO.fits"

save_dir = path / "images"
if not save_dir.exists():
    save_dir.mkdir(parents=True)
```

The class for loading fits-images and image-cubes is named `oimComponentFitsImage`. It derives from the `oimComponentImage` (i.e., the partially abstract class for all components defined in the image plane). `oimComponentImage` derives from the fully abstract `oimComponent` (i.e. the parent class of all `oimodeler` components).

Note: To learn more on the image-based models built with the `oimComponent` class check the *Advanced Examples* and the *Expanding the Software* tutorials.

There are two ways to load a fits image into a `oimComponentFitsImage` object. The first one is to open the fits file using the `astropy.io.fits` module of the `astropy` package and then passing it to the `oimComponentFitsImage` class.

```
im = fits.open(file_name)
c = oim.oimComponentFitsImage(im)
```

A simplier way, if the user doesn't need to access directly to the content of `im`, is to pass the filename to the `oimComponentFitsImage` class.

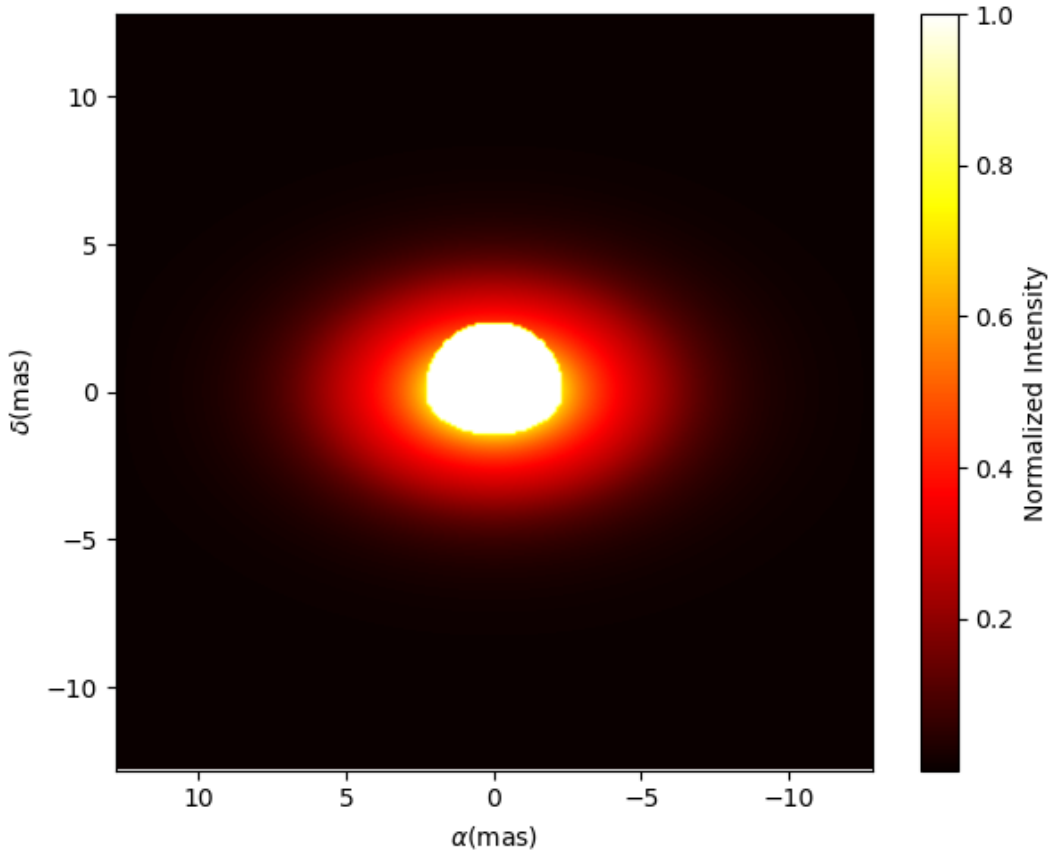
```
c = oim.oimComponentFitsImage(file_name)
```

Finally, we can build our model with this unique component:

```
m = oim.oimModel(c)
```

We can now plot the model image.

```
m.showModel(512, 0.05, legend=True, normalize=True, normPow=1, cmap="hot")
```



Note: Although the image was computed for a specific wavelength (i.e., 1.5 microns), our model is achromatic as we use a single image to generate it. An example with chromatic model buit on a chromatic image-cube is available *here*.

We now create spatial frequencies for a thousand baselines ranging from 0 to 120 m, in the North-South and East-West orientation and at an observing wavlength of 1.5 microns.

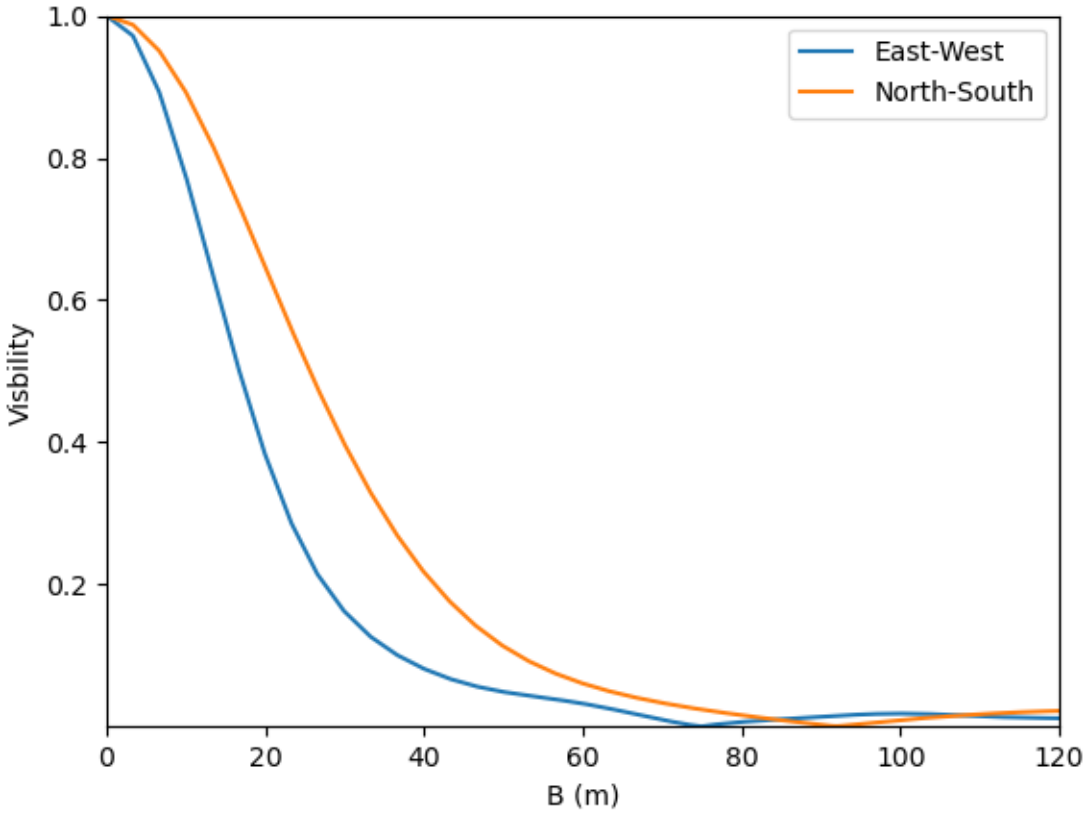
```
wl, nB = 1.5e-6, 1000
B = np.linspace(0, 120, num=nB)

spfx = np.append(B, B*0)/wl # 1st half of B array are baseline in the East-West␣
↪orientation
spfy = np.append(B*0, B)/wl # 2nd half are baseline in the North-South orientation
```

We compute the complex coherent flux and then the absolute visibility

```
ccf = m.getComplexCoherentFlux(spfx, spfy)
v = np.abs(ccf)
v = v/v.max()
```

and, finally, we can plot our results:

```
plt.figure()
plt.plot(B , v[0:nB],label="East-West")
plt.plot(B , v[nB:],label="North-South")
plt.xlabel("B (m)")
plt.ylabel("Visbility")
plt.legend()
plt.margins(0)
```



Let's now have a look at the model's parameters:

```
pprint(m.getParameters())
```

```
... {'c1_Fits_Comp_dim': oimParam at 0x19c6201c820 : dim=128 ± 0  range=[1,inf]␣
↪free=False ,
     'c1_Fits_Comp_f': oimParam at 0x19c6201c760 : f=1 ± 0  range=[0,1] free=True ,
     'c1_Fits_Comp_pa': oimParam at 0x19c00b9bbb0 : pa=0 ± 0 deg range=[-180,180]␣
↪free=True ,
     'c1_Fits_Comp_scale': oimParam at 0x19c6201c9d0 : scale=1 ± 0  range=[-inf,inf]␣
↪free=True ,
     'c1_Fits_Comp_x': oimParam at 0x19c6201c6a0 : x=0 ± 0 mas range=[-inf,inf]␣
↪free=False ,
     'c1_Fits_Comp_y': oimParam at 0x19c6201c640 : y=0 ± 0 mas range=[-inf,inf]␣
↪free=False }
```

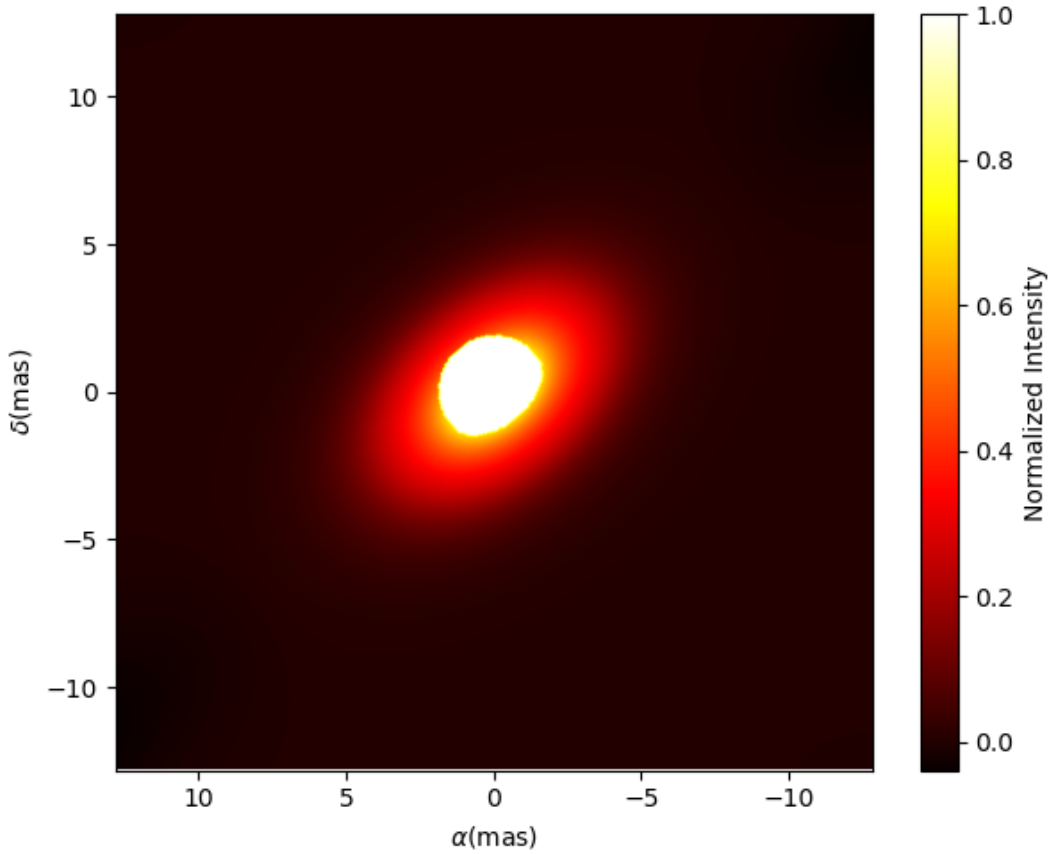
In addition to the x, y, and f parameters, common to all components, the `oimComponentFitsImage` have three additional parameters:

- *dim*: The fixed size of the internal fits image (currently only square images are compatible).
- *pa*: The position of angle of the component (used for rotating the component).
- *scale*: A scaling factor for the component.

The position angle *pa* and the *scale* are both free parameters (as default) and can be used for model fitting.

Let's try to rotate and scale our model and plot the image again.

```
c.params['pa'].value = 45
c.params['scale'].value = 2
m.showModel(256, 0.04, legend=True, normPow=0.4, colorbar=False)
```

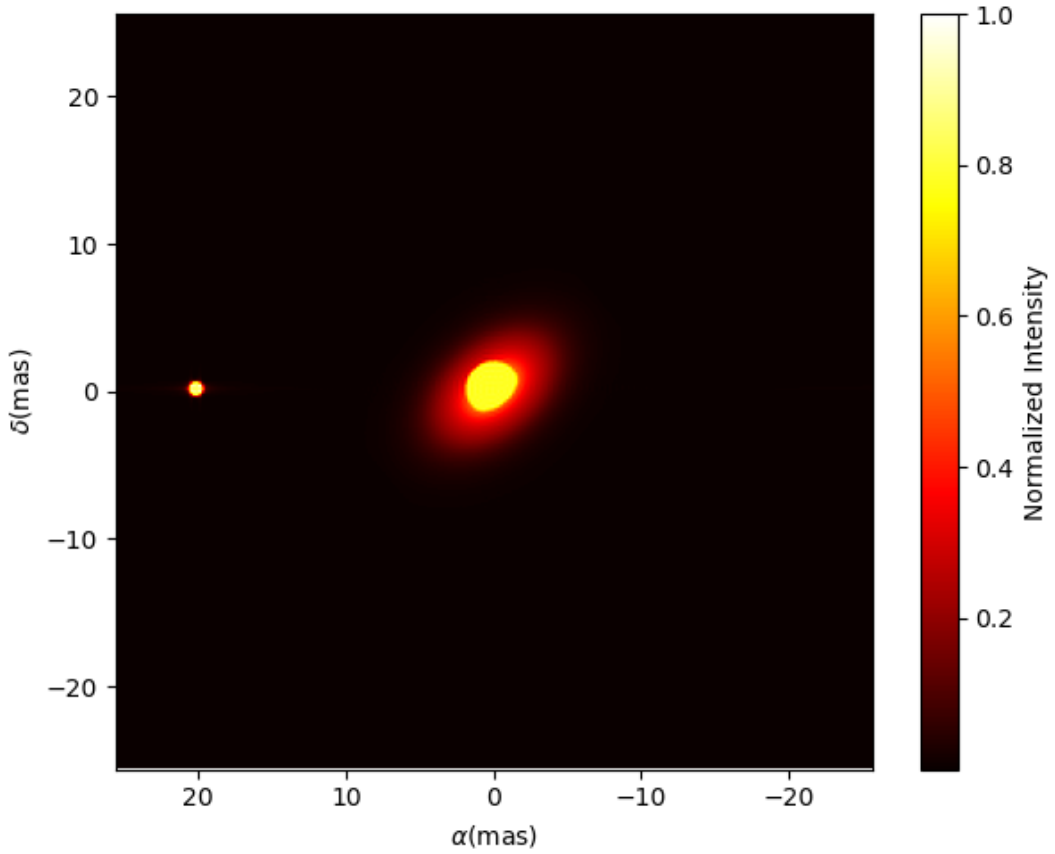


The `oimComponentFitsImage` can be combined with any kind of other component. Let's add a companion (i.e., uniform disk) for our Be star model.

```
c2 = oim.oimUD(x=20, d=1, f=0.03)
m2 = oim.oimModel(c, c2)
```

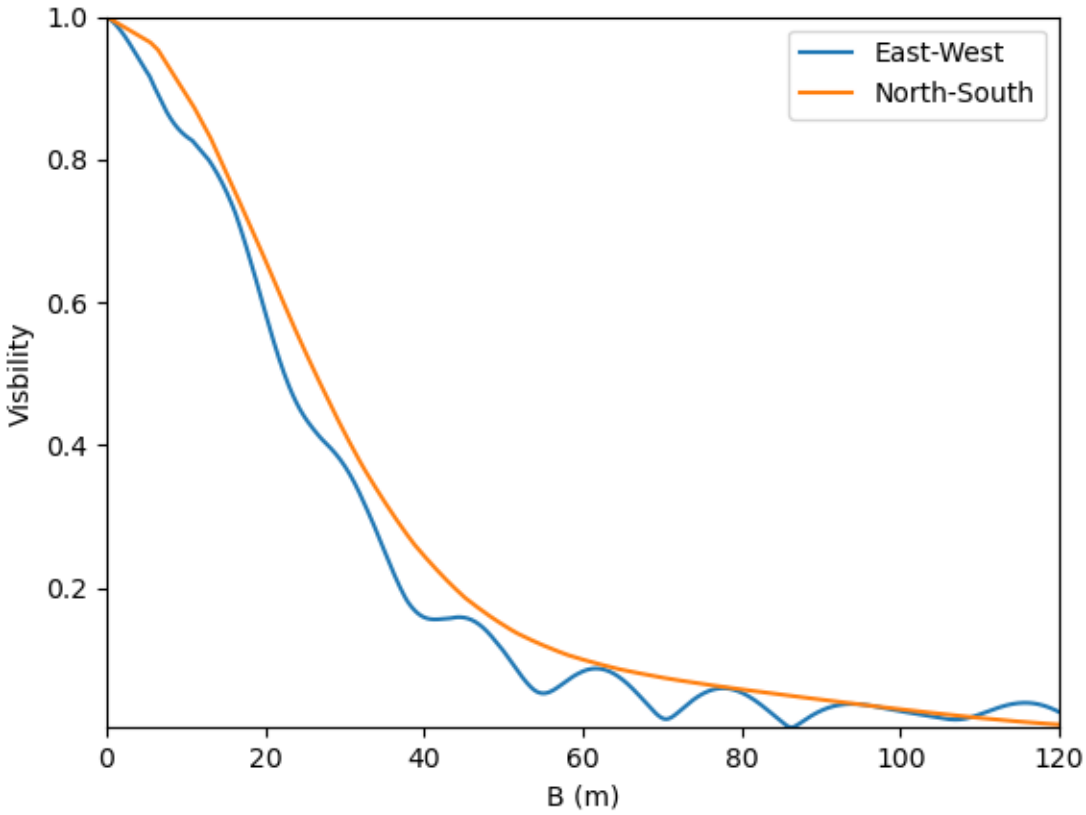
We add a 1 mas companion located at 20 mas West of the central object with a flux of 0.03. We can now plot the image of our new model.

```
m2.showModel(256, 0.2, legend=True, normalize=True, fromFT=True, normPow=1, cmap="hot")
```



To finish this example, let's plot the visibility along North-South and East-West baseline for our binary Be-star model.

```
ccf = m2.getComplexCoherentFlux(spfx, spfy)
v = np.abs(ccf)
v = v/v.max()

plt.figure()
plt.plot(B, v[0:nB], label="East-West")
plt.plot(B, v[nB:], label="North-South")
plt.xlabel("B (m)")
plt.ylabel("Visbility")
plt.legend()
plt.margins(0)
```



5.4.1.4 Data/model comparison

In the exampleOimSimulator.py script, we use the `oimSimulator` class to compare some OIFITS data with a model. We will compute the χ^2_r and plot the comparison between the data an the simulated data from the model.

Let's start by importing the needed modules and setting the variable `files` to the list of the same OIFITS files as in the *Loading oifits data* example.

```
from pathlib import Path
from pprint import pprint

import oimodeler as oim

path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "ASPRO_MATISSE2"
save_dir = path / "images"
if not save_dir.exists():
    save_dir.mkdir(parents=True)

files = list(map(str, data_dir.glob("*.fits")))
```

These OIFITS were simulated with ASPRO as a MATISSE observation of a partly resolved binary star.

We make a model of a binary star where one of the components is resolved. It consists of two components: A uniform

disk and a point source.

```
ud = oim.oimUD(d=3, f=1, x=10, y=20)
pt = oim.oimPt(f=0.5)
model = oim.oimModel([ud, pt])
```

We now create an `oimSimulator` object and feed it with the data and our model.

The data can either be:

- A previously created `oimData`.
- A list of previously opened astropy.io.fits.hdulist.
- A list of paths to the OIFITS files (list of strings).

```
sim = oim.oimSimulator(data=files, model=model)
```

When creating the simulator, it automatically calls the `oimData.prepareData` method of the created `oimData` instance within the simulator instance. This calls the `oimData.prepare` method of `oimData` The function is used to create vectorized coordinates for the data (spatial frequencies in x and y and wavelengths) to be passed to the `oimModel` instance to compute the Complex Coherent Flux (CCF) using the `oimModel.getComplexCoherentFlux` method, and some structures to go back from the `ccf` to the measured interferometric quantities contained in the OIFITS files: VIS2DATA, VISAMP, VISPHI, T3AMP, T3PHI, and FLUXDATA.

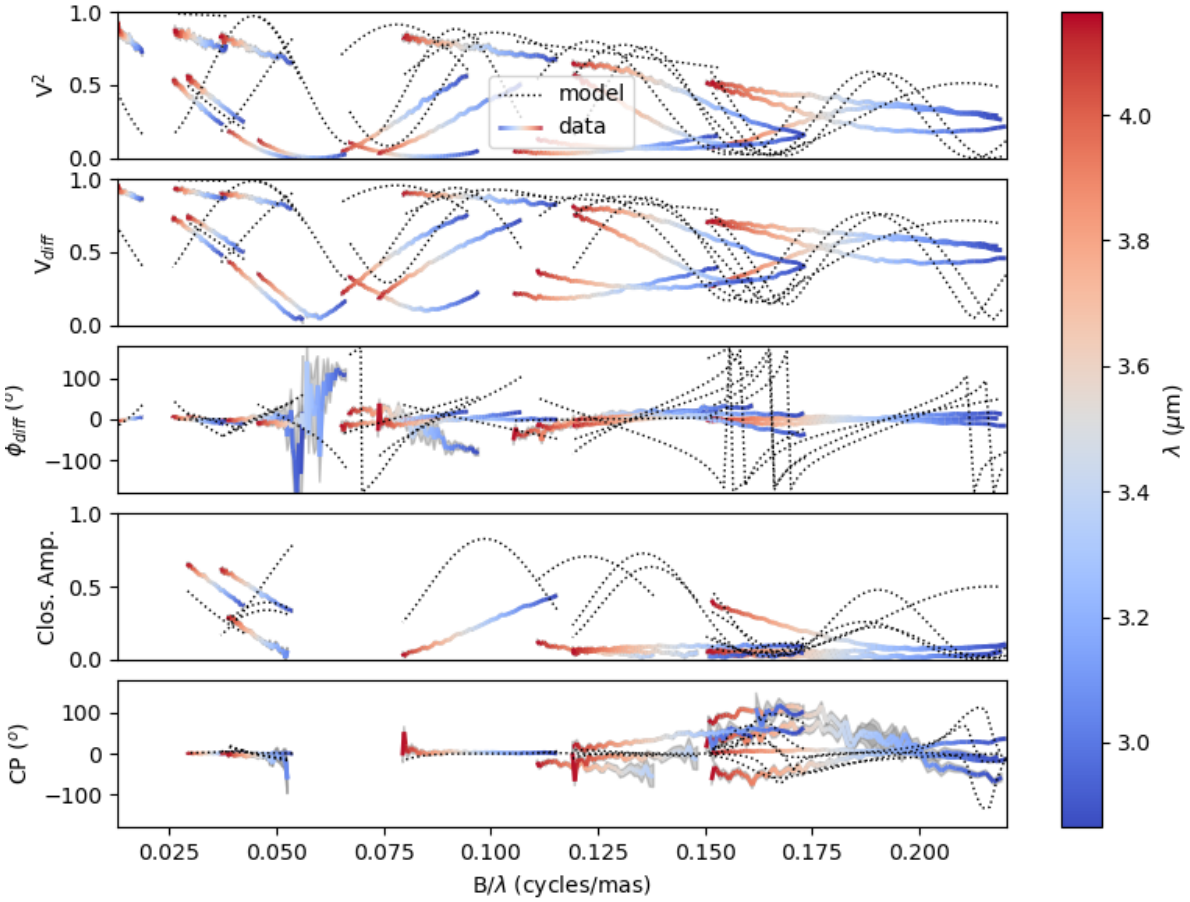
Once the data is prepared, we can call the `oimSimulator.compute` method to compute the χ^2 and the simulated data.

```
sim.compute(computeChi2=True, computeSimulatedData=True)
pprint("Chi2r = {}".format(sim.chi2r))
```

```
... Chi2r = 11356.162973124885
```

Our model isn't fitting the data well. Let's take a closer look and plot the data-model comparison for all interferometric quantities contained in the OIFITS files.

```
fig0, ax0= sim.plot(["VIS2DATA", "VISAMP", "VISPHI", "T3AMP", "T3PHI"])
```



You can now try to fit the model "by hand", or go to the next example where we use a fitter from the `oimFitter` module to automatically find a good fit (and thus well fitting parameters).

5.4.1.5 Running a mcmc fit

In the exampleOimFitterEmcee.py script, we perform a complete emcee run to determine the values of the parameters of the same binary as in the (previous) *Data/model comparison* example.

We start by setting up the script with imports, a data list and a binary model. We don't need to specify values for the biary parameters as they will be fitted.

```
from pathlib import Path
from pprint import pprint

import oimodeler as oim


path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "ASPRO_MATISSE2"
save_dir = path / "images"
if not save_dir.exists():
    save_dir.mkdir(parents=True)
```

(continues on next page)

(continued from previous page)

```
files = list(map(str, data_dir.glob("*.fits")))

ud = oim.oimUD()
pt = oim.oimPt()
model = oim.oimModel([ud,pt])
```

Before starting the run, we need to specify which parameters are free and what their ranges are. By default, all parameters are free, but the components x and y coordinates. For a binary, we need to release them for one of the components. As we only deal with relative fluxes, we can set the flux of one of the components to be fixed to one.

```
ud.params['d'].set(min=0.01, max=20)
ud.params['x'].set(min=-50, max=50, free=True)
ud.params['y'].set(min=-50, max=50, free=True)
ud.params['f'].set(min=0., max=10.)
pt.params['f'].free = False
pprint(model.getFreeParameters())
```

```
... {'c1_UD_x': oimParam at 0x23d940e4850 : x=0 ± 0 mas range=[-50,50] free=True,
     'c1_UD_y': oimParam at 0x23d940e4970 : y=0 ± 0 mas range=[-50,50] free=True,
     'c1_UD_f': oimParam at 0x23d940e4940 : f=0.5 ± 0  range=[0.0,10.0] free=True,
     'c1_UD_d': oimParam at 0x23d940e4910 : d=3 ± 0 mas range=[0.01,20] free=True}
```

We have 4 free-parameters, the position (x, y), the flux f and the diameter d of the uniform disk component.

Now, we can create a fitter with our model and a list of OIFITS files. We use the emcee fitter that has only one parameter, the number of walkers that will explore the parameter space.

```
fit = oim.oimFitterEmcee(files, model, nwalkers=32)
```

Note: If you are not confident with emcee, you should have a look at the documentation here.

We need to initialize the fitter using its `oimFitterEmcee.prepare` method. This is setting the initial values of the walkers. The default method is to set them to random values within the parameters' ranges.

```
fit.prepare(init="random")
pprint(fit.initialParams)
```

```
... [[-37.71319618 -49.22761731   9.3299391   15.51294277]
    [-12.92392301  17.49431852   7.76169304   9.23732472]
    [-31.62470824 -11.05986877   8.71817772   0.34509237]
    [-36.38546264  33.856871     0.81935324   9.04534926]
    [ 45.30227534 -38.50625408   4.89978551  14.93004   ]
    [-38.01416866  -6.24738348   5.26662714  13.16349304]
    [-21.34600438 -14.98116997   1.20948714   8.15527356]
    [-17.14913499  10.40965493   0.37541088  18.81733973]
    [ -9.61039318 -12.02424002   6.81771974  16.22898422]
    [ 49.07320952 -34.48933488   1.75258006  19.96859116]]
```

We can now run the fit. We choose to run 2000 steps as a start and interactively show the progress with a progress bar. The fit should take a few minutes on a standard computer to compute around 64000 models (`nwalkers` x `nsteps`).

```
fit.run(nsteps=2000, progress=True)
```

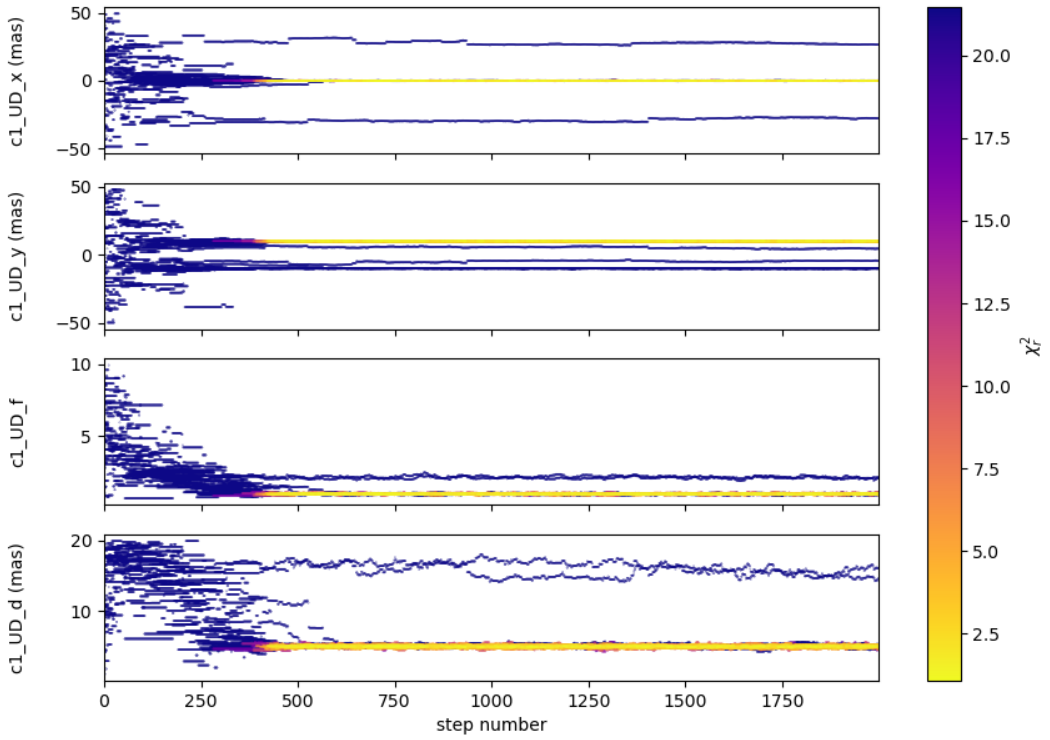
The `oimFitterEmcee` instance stores the emcee sampler as an attribute: `oimFitterEmcee.sampler`. You can, for example, access the chain of walkers and the logrithmic of the probability directly.

```
sampler = fit.sampler
chain   = fit.sampler.chain
lnprob  = fit.sampler.lnprobability
```

We can manipulate these data. The `oimFitterEmcee` implements various methods to retrieve and plot the results of the mcmc run.

The walkers position as the function of the steps can be plotted using the `oimFitterEmcee.walkersPlot` method.

```
figWalkers, axeWalkers = fit.walkersPlot(cmap="plasma_r")
```



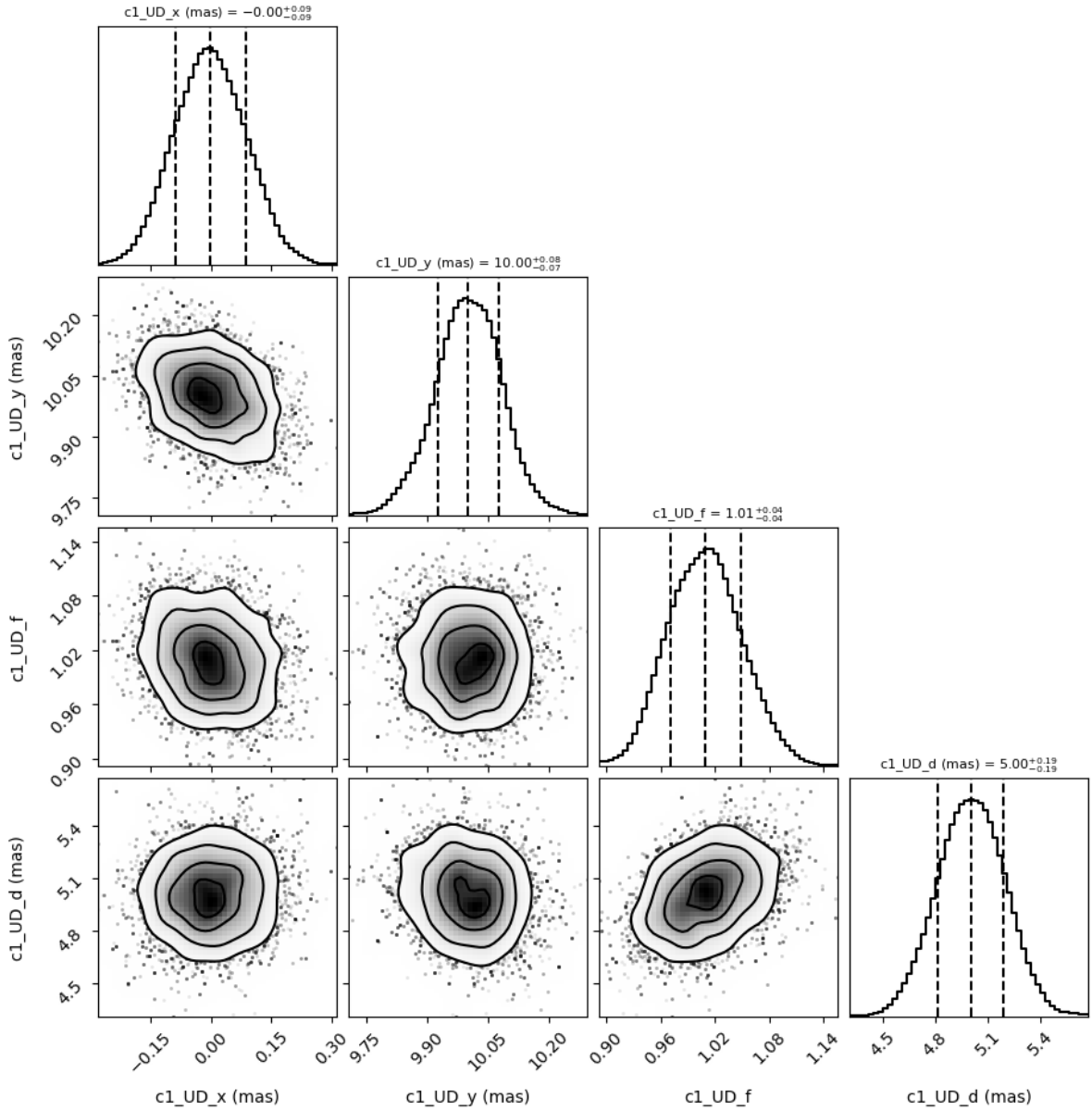
After a few hundred steps most walkers converge to a position with a good χ^2_r. However, from that figure will clearly see that:

- Not all walkers have converged after 2000 steps.
- Some walkers converge to a solution that gives significantly worse χ^2.

In optical interferometry there are often local minimas in the χ^2 and it seems that some of our walkers are locked in such minimas. In our case, this minimum is due to the fact that object is almost symmetrical if not for the fact than one of the component is resolved. Neverless, the χ^2 of the local minimum is about 20 times worse than the one of the global minimum.

We can plot the "famous" corner plot containing a 1D and 2D density distribution. The `oimodeler` package uses the corner library for that purpose. We will discard the first 1000 steps as most of the walkers have converged after that. By default, the corner plot also remove the data with a χ^2 20 times greater than those of the best model. The cutoff can be set with the `chi2limfact` keyword of the `oimFitter.cornerPlot` method.

```
figCorner, axeCorner=fit.cornerPlot(discard=1000)
```



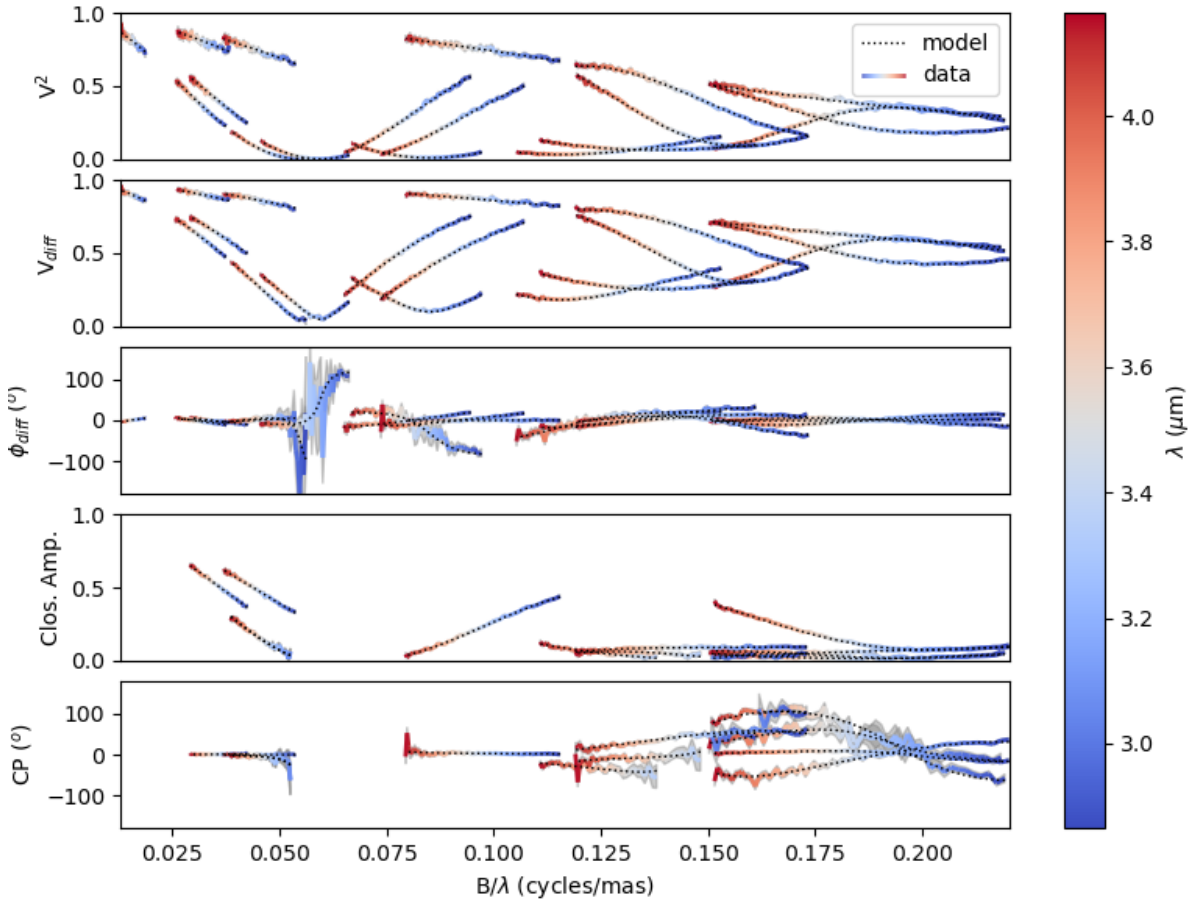
We now can get the result of our fit. The `oimFitterEmcee` fitter can return either the `best`, the `mean` or the `median` model. It return uncertainties estimated from the density distribution (see emcee' for more details).

```
median, err_l, err_u, err = fit.getResults(mode='median', discard=1000)
```

To compute the median and mean model we have to remove, as in the corner plot, the walkers that didn't converge with the `chi2limitfact` keyword (default is 20) and remove the steps of the bruning phase with the `discard` option.

When procuring the results, the simulated data is simultaneously calculated in the fitter's internal simulator. We can again plot the data/model and compute the final χ^2_r:

```
figSim, axSim=fit.simulator.plot(["VIS2DATA", "VISAMP", "VISPHI", "T3AMP", "T3PHI"])
pprint("Chi2r = {}".format(fit.simulator.chi2r))
```



5.4.1.6 Filtering data

Filtering can be applied to the `oimData` class using the `oimDataFilter` class. It is basically a stack of filters derived from the `oimDataFilterComponent` abstract class. The example presented here comes from the exampleOimDataFilter.py script.

As done before the required packages and create a list of the OIFITS files.

```
from pathlib import Path

import matplotlib.pyplot as plt
import oimodeler as oim

path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "FSCMa_MATISSE"
save_dir = path / "images"
if not save_dir.exists():
```

(continues on next page)

(continued from previous page)

```
    save_dir.mkdir(parents=True)

files = list(map(str, data_dir.glob("*.fits")))
```

We create an `oimData` object which will contain the OIFITS data.

```
data = oim.oimData(files)
```

We now create a simple filter to cut the data to a specific wavelength range with the `oimWavelengthRangeFilter` class.

```
f1 = oim.oimWavelengthRangeFilter(targets="all", wlRange=[3.0e-6, 4e-6])
```

The `oimWavelengthRangeFilter` has two keywords:

- `targets`: Which is common to all filter components: It specifies the targeted files within the data structure to which the filter applies.
  - Possible values are: `"all"` for all files (which we use in this example).
  - A single file specify by its index.
  - Or a list of indexes.
- `wlRange`: The wavelength range to cut as a two elements list (min wavelength and max wavelength), or a list of multiple two-elements lists if you want to cut multiple wavelengths ranges simultaneously. In our example you have selected wavelength between 3 and 4 microns. Wavelengths outside this range will be removed from the data.

Now we can create a filter stack with this single filter and apply it to our data.

```
filters = oim.oimDataFilter([f1])
data.setFilter(filters)
```

By default the filter will be automatically activated as soon as a filter is set using the `oimData.setFilter` method of the `oimData` class. This means that querying the `oimData.data` attribute will return the filtered data, and that when using the `oimData` class within an `oimSimulator` or an `oimFitter`, the filtered data will be used instead of the unfiltered data.

Note: The unfiltered data can always be accessed using the `oimData._data` attribute and, in a similar manner, also the filtered data (that may be `None` if no filters have been applied) using the private attribute `oimData._filteredData`.

To switch off a filter we can either call the `oimData.setFilter` method without any arguments (this will remove the filter completely),

```
data.setFilters()
```

or set the `useFilter` attribute to `False`.

```
data.useFilter = False
```

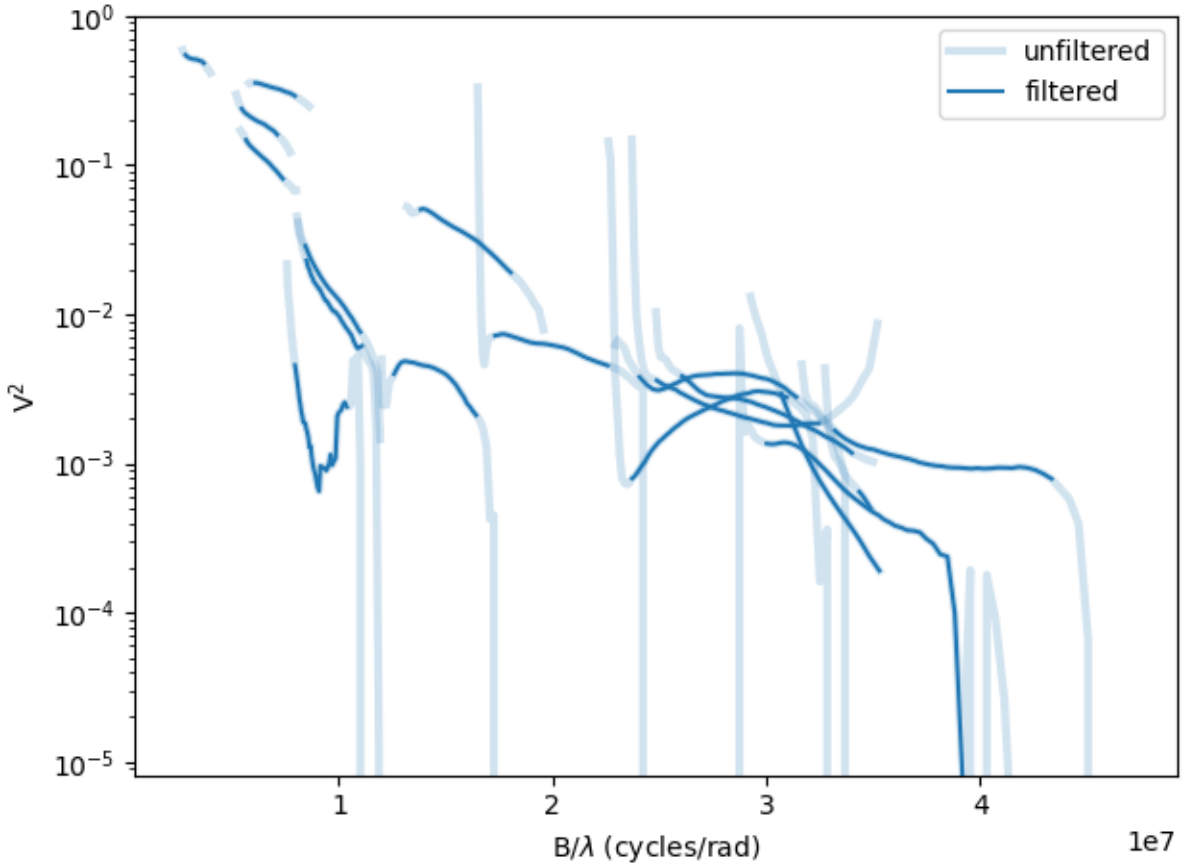
Let's plot the unfiltered and filtered data using the `oimPlot` method.

```
fig = plt.figure()
ax = plt.subplot(projection='oimAxes')
```

(continues on next page)

(continued from previous page)

```
data.useFilter = False
ax.oiplot(data, "SPAFREQ", "VIS2DATA", color="tab:blue", lw=3, alpha=0.2, label=
↪"unfiltered")

data.useFilter = True
ax.oiplot(data, "SPAFREQ", "VIS2DATA", color="tab:blue", label="filtered")

ax.set_yscale('log')
ax.legend()
ax.autolim()
```



Other filters for data selection are:

- `oimRemoveArrayFilter`: Removes array(s) (such as OI_VIS, OI_T3, etc.) from the data.
- `oimDataTypeFilter`: Removes data type(s) (such as VISAMP, VISPHI, T3AMP, etc.) from the data.

Note: Actually, `oimDataTypeFilter` doesn't remove the columns with the data type from any array as these columns are complusory in the the OIFITS format definition. Instead, it is setting all the values of the column to zero which `oimodeler` will recognize as empty for data simulation and model fitting.

```
f2 = oim.oimRemoveArrayFilter(targets="all", arr=["OI_VIS", "OI_FLUX"])
f3 = oim.oimDataTypeFilter(targets="all", dataType=["T3AMP"," T3PHI"])
data.setFilter(oim.oimDataFilter([f1, f2, f3]))
```

Here, we create a new filter stack with the previous wavelength filter *f1*, a filter *f2* for removing the array OI_VIS and OI_FLUX from the data, and a filter *f3* removing the columns T3AMP and T3PHI. Basically, we only have the VIS2DATA left in our OIFITS structure.

Note: Removing T3AMP and T3PHI from the OI_T3 is equivalent for model-fitting to removing the array OI_T3.

5.4.1.7 Plotting oifits data

Beyond the specific plots shown in the previous example, the `oimPlot` module allows to plot most of the OIFITS data in a very simple way. The example presented here comes from the exampleOimPlot.py script.

Let's start by setting up the project with imports, path, and some data.

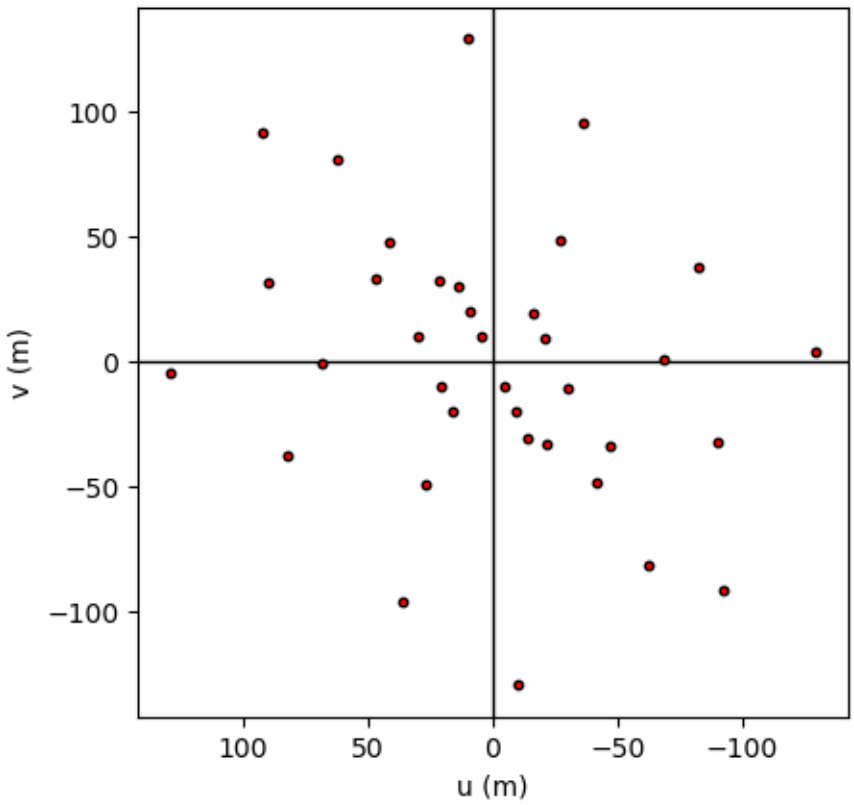
```
from pathlib import Path

import matplotlib.pyplot as plt
import oimodeler as oim


path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "ASPRO_MATISSE2"
save_dir = path / "images"
if not save_dir.exists():
    save_dir.mkdir(parents=True)

files = list(map(str, data_dir.glob("*.fits")))
```

The `oimodeler` package comes with the `oimAxes` class that is a subclass of the standard matplotlib.pytplot.Axes class (the base class for all matplotlib plots). To use it, you simply need to specify it as a projection (actually this calls the subclass) when creating an axe or axes.

```
fig1 = plt.figure()
ax1 = plt.subplot(projection='oimAxes')
```

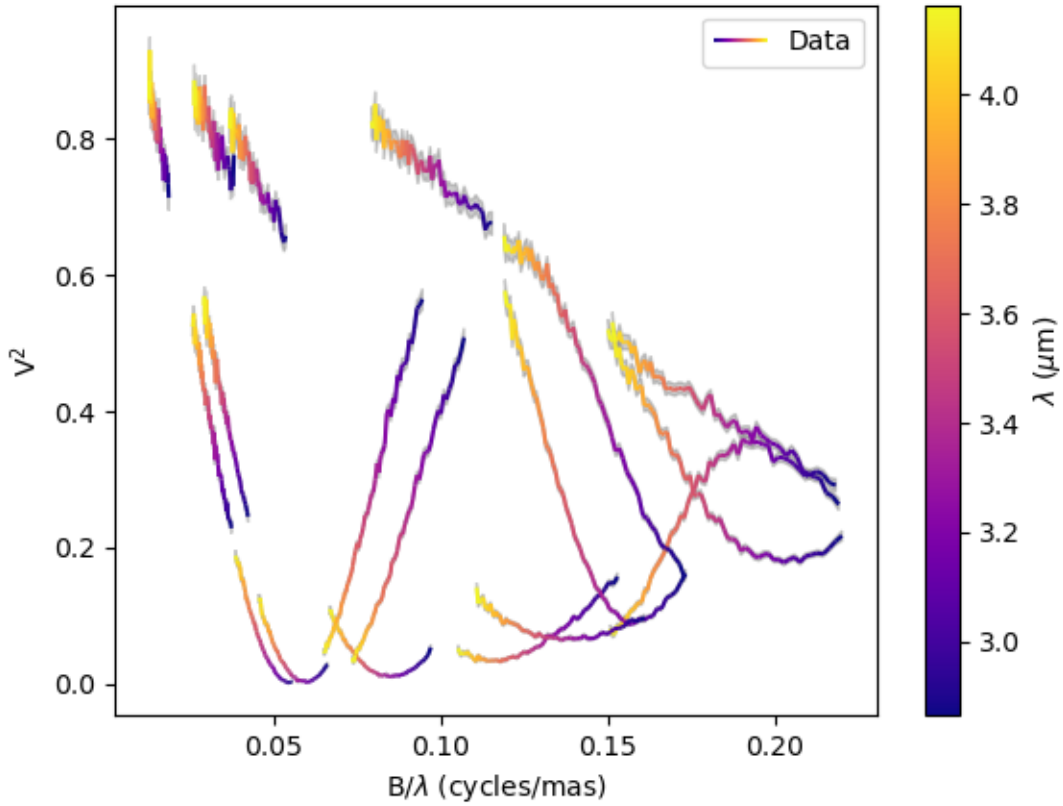
First, we can plot the classic uv coverage using the `oimAxes.uvplot` method by passing the list of OIFITS files (filename or opened) or an instance of a `oimData` class.

```
ax1.uvplot(data)
```



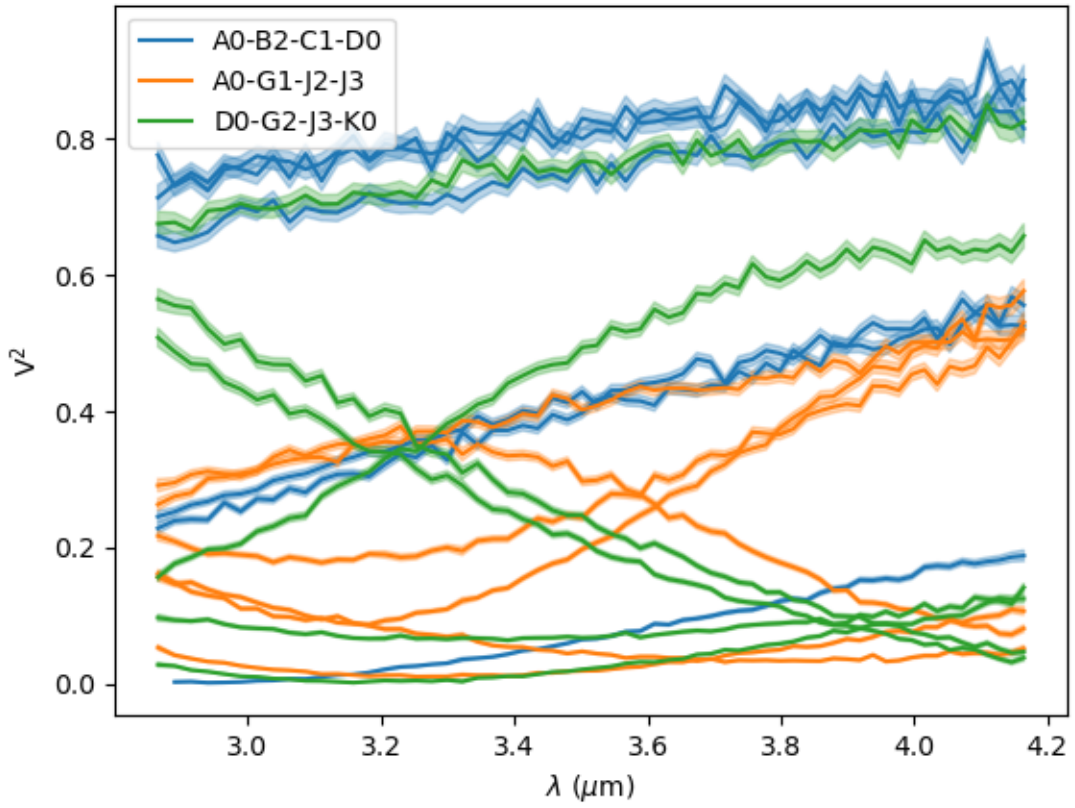
We can use the `oiplot` method of the `oimAxes` to plot any quantity inside an OIFITS file as a function of another one. For instance, let's plot the squared visibilities as a function of the spatial frequencies with the wavelength (in microns) as a colorscale.

```
fig2 = plt.figure()
ax2 = plt.subplot(projection='oimAxes')
lamcol=ax2.oiplot(data, "SPAFREQ", "VIS2DATA", xunit="cycles/mas", label="Data",
                  cname="EFF_WAVE", cunitmultiplier=1e6, errorbar=True)

plt.colorbar(lamcol, ax=ax2, label="$\\lambda$ ($\mu$m)")
ax2.legend()
```



We can also plot the square visibility as the function of the wavelength while colouring the curves by the interferometer configurations (i.e., the list of all baselines within one file). Note that we can pass parameters to the error plots with the `kwargs_error` keyword.

```
fig3 = plt.figure()
ax3 = plt.subplot(projection='oimAxes')
ax3.oiplot(data, "EFF_WAVE", "VIS2DATA", xunitmultiplier=1e6, color="byConfiguration",
           errorbar=True, kwargs_error={"alpha": 0.3})
ax3.legend()
```



Note: Special values of the color option are `"byFile"`, `"byConfiguration"`, `"byArrname"`, or `"byBaseline"`. Other values will be interpreted as a standard matplotlib colorname. When using one of these values, the corresponding labels are added to the plots. Using the `oimAxes.legend` method will automatically add the proper names.

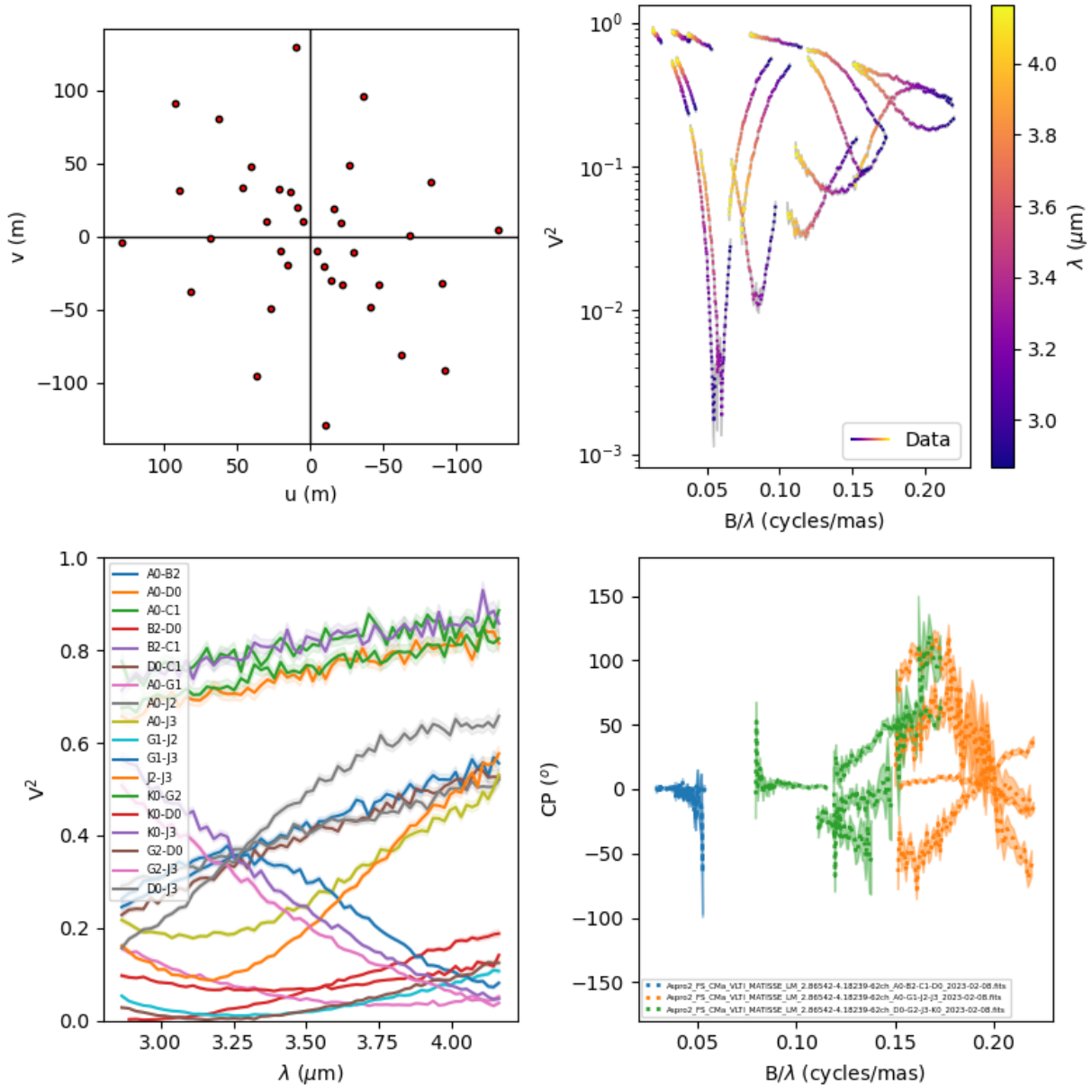
Finally, we can create a `2x2` figure with multiple plots. The projection keyword has to be set for all `oimAxes` using the `subplot_kw` keyword in the matplotlib.pyplot.subplots method.

```
fig4, ax4 = plt.subplots(2, 2, subplot_kw=dict(
    projection='oimAxes'), figsize=(8, 8))

ax4[0, 0].uvplot(data)

lamcol = ax4[0, 1].oiplot(data, "SPAFREQ", "VIS2DATA", xunit="cycles/mas", label="Data",
                          cname="EFF_WAVE", cunitmultiplier=1e6, ls=":", errorbar=True)

fig4.colorbar(lamcol, ax=ax4[0, 1], label="$\\lambda$ ($\mu$m)")
ax4[0, 1].legend()
ax4[1, 0].oiplot(data, "EFF_WAVE", "VIS2DATA", xunitmultiplier=1e6, color="byBaseline",
                 errorbar=True, kwargs_error={"alpha": 0.1})

ax4[1, 0].legend(fontsize=6)
ax4[1, 1].oiplot(data, "SPAFREQ", "T3PHI", xunit="cycles/mas", errorbar=True,
```

(continues on next page)

(continued from previous page)

```
                lw=2, ls=":", color="byFile")

ax4[1, 1].legend(fontsize=4)
ax4[0, 1].set_yscale('log')
ax4[1, 0].autolim()
ax4[1, 1].autolim()
```



5.4.2 Advanced Examples

5.4.2.1 Building Complex models

In the example complexModel.py we create and play with more complex Fourier-based models which includes:

- Flatenning of some components
- Linked parameters between components
- Chromaticity of some parameters

First, we import the useful packages and create a set of spatial frequencies and wavelengths to be used to generate visibilities.

```
from pathlib import Path
from pprint import pprint

import numpy as np
import oimodeler as oim

fromFT = False
nB = 500  # number of baselines
nwl = 100  # number of walvengths

wl = np.linspace(3e-6, 4e-6, num=nwl)
B = np.linspace(1, 400, num=nB)
Bs = np.tile(B, (nwl, 1)).flatten()
wls = np.transpose(np.tile(wl, (nB, 1))).flatten()
spf = Bs/wls
spf0 = spf*0
```

Unlike in the previous example with the grey data, we create a 2D-array for the spatial frequencies of `nB` baselines by `nwl` wavelengths. The wavlength vector is tiled itself to have the same length as the spatial frequency vector. Finally, we flatten the vector to be passed to the `getComplexCoherentFlux` method.

Let's create our first chromatic components. Linearly chromatic parameter can added to grey Fourier-based model by using the `oimInterp` macro with the parameter `"wl"` when creating a new component.

```
g = oim.oimGauss(fwhm=oim.oimInterp("wl" wl=[3e-6, 4e-6], values=[2, 8]))
```

We have created a Gaussian component with a `fwhm` growing from 2 mas at 3 microns to 8 mas at 4 microns.

Note: Parameter interpolators are described in details in the *following example*.

We can access to the interpolated value of the parameters using the `__call__` operator of the `oimParam` class with values passed for the wavelengths to be interpolated:

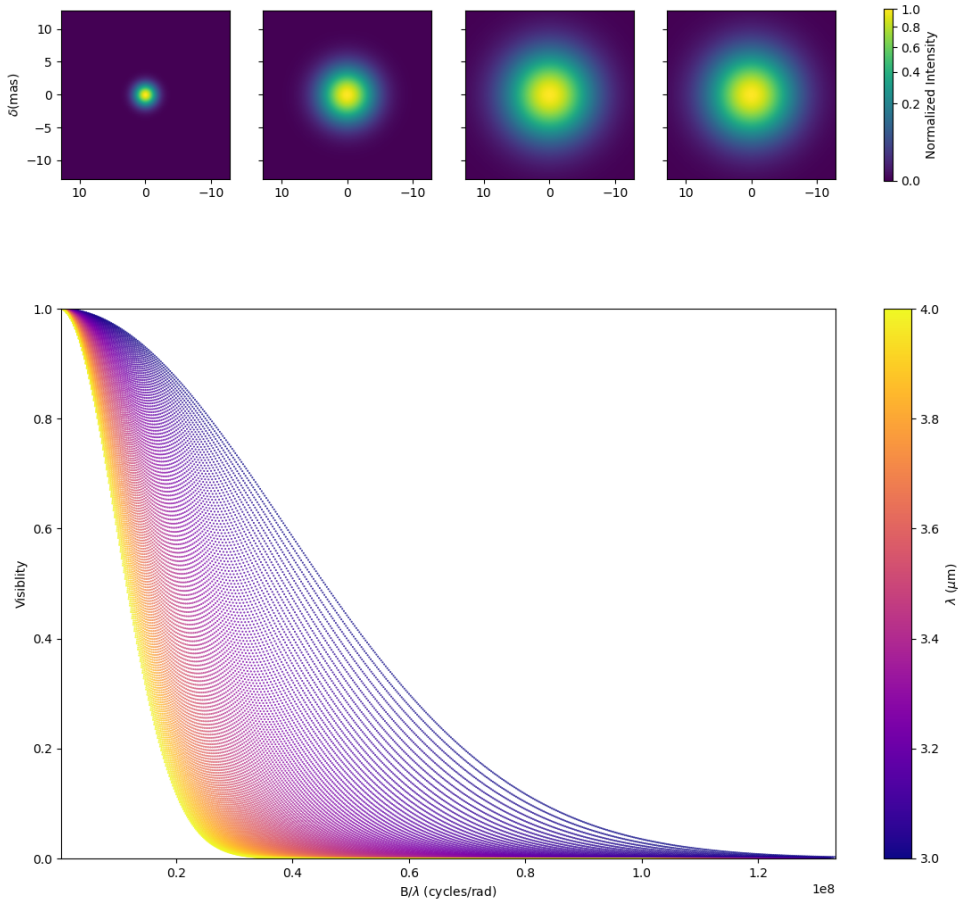
```
pprint(g.params['fwhm']([3e-6, 3.5e-6, 4e-6, 4.5e-6]))
```

```
... [2. 5. 8. 8.]
```

The values are interpolated within the wavelength range [3e-6, 4e-6] and fixed beyond this range.

Let's build a simple model with this component and plot the images at few wavelengths and the visibilities for the baselines we created before.

```
vis = np.abs(mg.getComplexCoherentFlux(
    spf, spf*0, wls)).reshape(len(wl), len(B))
vis /= np.outer(np.max(vis, axis=1), np.ones(nB))

figGv, axGv = plt.subplots(1, 1, figsize=(14, 8))
sc = axGv.scatter(spf, vis, c=wls*1e6, s=0.2, cmap="plasma")
figGv.colorbar(sc, ax=axGv, label="$\\lambda$ ($\\mu$m)")
axGv.set_xlabel("B/$\\lambda$ (cycles/rad)")
axGv.set_ylabel("Visiblity")
axGv.margins(0, 0)
```



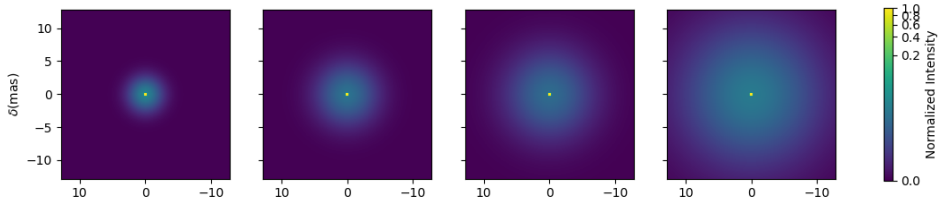
Now let's add a second component: An uniform disk with a chromatic flux.

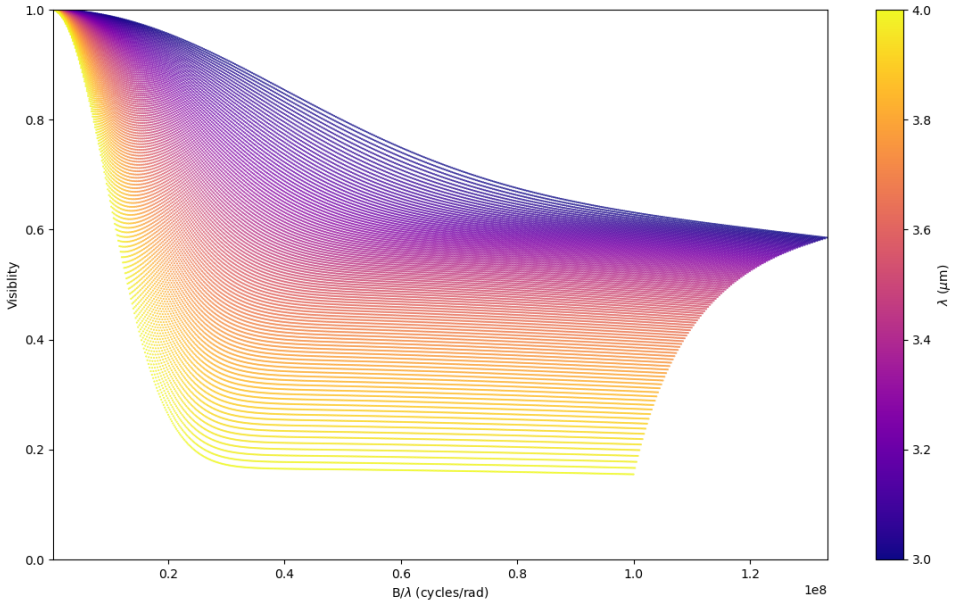
```
ud = oim.oimUD(d=0.5, f=oim.oimInterp("wl", wl=[3e-6, 4e-6], values=[2, 0.2]))
m2 = oim.oimModel([ud, g])
fig2im, ax2im, im2 = m2.showModel(256, 0.1, wl=[3e-6, 3.25e-6, 3.5e-6, 4e-6],
                                  swapAxes=True, normPow=0.2, figsize=(3.5, 2.5),
                                  fromFT=fromFT, normalize=True,
                                  savefig=save_dir / "complexModel_UDAndGauss.png")

vis = np.abs(m2.getComplexCoherentFlux(
    spf, spf*0, wls)).reshape(len(wl), len(B))
```

(continues on next page)

(continued from previous page)

```
vis /= np.outer(np.max(vis, axis=1), np.ones(nB))

fig2v, ax2v = plt.subplots(1, 1, figsize=(14, 8))
sc = ax2v.scatter(spf, vis, c=wls*1e6, s=0.2, cmap="plasma")
fig2v.colorbar(sc, ax=ax2v, label="$\\lambda$ ($\\mu$m)")
ax2v.set_xlabel("B/$\\lambda$ (cycles/rad)")
ax2v.set_ylabel("Visiblity")
ax2v.margins(0, 0)
ax2v.set_ylim(0, 1)
```





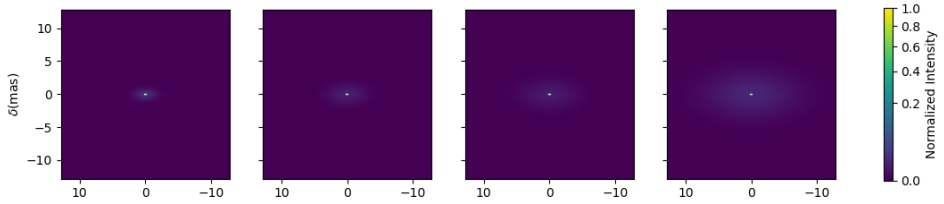
Now let's create a similar model but with elongated components. We will replace the uniform disk by an ellipse and the Gaussian by an elongated Gaussian.

```
eg = oim.oimEGauss(fwhm=oim.oimInterp(
    "wl", wl=[3e-6, 4e-6], values=[2, 8]), elong=2, pa=90)
el = oim.oimEllipse(d=0.5, f=oim.oimInterp(
    "wl", wl=[3e-6, 4e-6], values=[2, 0.2]), elong=2, pa=90)

m3 = oim.oimModel([el, eg])
fig3im, ax3im, im3 = m3.showModel(256, 0.1, wl=[3e-6, 3.25e-6, 3.5e-6, 4e-6],
                                  figsize=(3.5, 2.5), normPow=0.5, fromFT=fromFT,␣
↪normalize=True,
```

(continues on next page)

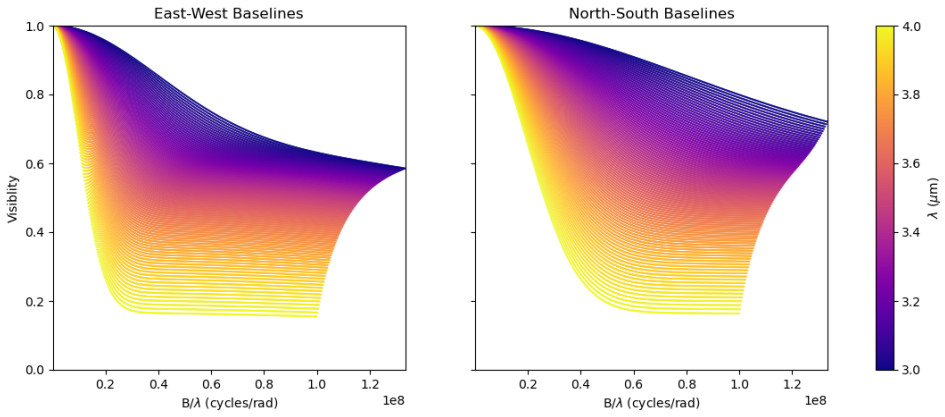
(continued from previous page)

```
                                        savefig=save_dir / "complexModel_Elong.png")
```



Now that our model is no more circular, we need to take care of the baselines orientations. Let's plot both North-South and East-West baselines.

```
fig3v, ax3v = plt.subplots(1, 2, figsize=(14, 5), sharex=True, sharey=True)

# East-West
vis = np.abs(m3.getComplexCoherentFlux(
    spf, spf*0, wls)).reshape(len(wl), len(B))
vis /= np.outer(np.max(vis, axis=1), np.ones(nB))
ax3v[0].scatter(spf, vis, c=wls*1e6, s=0.2, cmap="plasma")
ax3v[0].set_title("East-West Baselines")
ax3v[0].margins(0, 0)
ax3v[0].set_ylim(0, 1)
ax3v[0].set_xlabel("B/$\\lambda$ (cycles/rad)")
ax3v[0].set_ylabel("Visiblity")

# North-South
vis = np.abs(m3.getComplexCoherentFlux(
    spf*0, spf, wls)).reshape(len(wl), len(B))
vis /= np.outer(np.max(vis, axis=1), np.ones(nB))
sc = ax3v[1].scatter(spf, vis, c=wls*1e6, s=0.2, cmap="plasma")
ax3v[1].set_title("North-South Baselines")
ax3v[1].set_xlabel("B/$\\lambda$ (cycles/rad)")
fig3v.colorbar(sc, ax=ax3v.ravel().tolist(), label="$\\lambda$ ($\\mu$m)")
```



Let's have a look at our last model's free parameters.

```
pprint(m3.getFreeParameters())
```

```
... {'c1_eUD_f_interp1': oimParam at 0x23d9e7194f0 : f=2 ± 0  range=[-inf,inf]␣
↪free=True ,
    'c1_eUD_f_interp2': oimParam at 0x23d9e719520 : f=0.2 ± 0  range=[-inf,inf]␣
↪free=True ,
    'c1_eUD_elong': oimParam at 0x23d9e7192e0 : elong=2 ± 0  range=[-inf,inf]␣
↪free=True ,
    'c1_eUD_pa': oimParam at 0x23d9e719490 : pa=90 ± 0 deg range=[-inf,inf] free=True ,
    'c1_eUD_d': oimParam at 0x23d9e7193a0 : d=0.5 ± 0 mas range=[-inf,inf] free=True ,
    'c2_EG_f': oimParam at 0x23d9e7191c0 : f=1 ± 0  range=[-inf,inf] free=True ,
    'c2_EG_elong': oimParam at 0x23d9e7191f0 : elong=2 ± 0  range=[-inf,inf] free=True␣
↪,
    'c2_EG_pa': oimParam at 0x23d9e719220 : pa=90 ± 0 deg range=[-inf,inf] free=True ,
    'c2_EG_fwhm_interp1': oimParam at 0x23d9e7192b0 : fwhm=2 ± 0 mas range=[-inf,inf]␣
↪free=True ,
    'c2_EG_fwhm_interp2': oimParam at 0x23d9e719340 : fwhm=8 ± 0 mas range=[-inf,inf]␣
↪free=True }
```

We see here that for the Ellipse (`C1_eUD`) the f parameter has been replaced by two independent parameters called `c1_eUD_f_interp1` and `c1_eUD_f_interp2`. They represent the value of the flux at 3 and 4 microns. We could have added more reference wavelengths in our model and would have ended with more parameters. The same happens for the elongated Gaussian (`C2_EG`) fwhm.

Currently our model has 10 free parameters. In certain cases we might want to link or share two or more parameters. In our case, we might consider that the two components have the same `pa` and `elong`. This can be done easily. To share a parameter you can just replace one parameter by another.

```
eg.params['elong'] = el.params['elong']
eg.params['pa'] = el.params['pa']

pprint(m3.getFreeParameters())
```

```
... {'c1_eUD_f_interp1': oimParam at 0x23d9e7194f0 : f=2 ± 0  range=[-inf,inf]␣
↪free=True ,
    'c1_eUD_f_interp2': oimParam at 0x23d9e719520 : f=0.2 ± 0  range=[-inf,inf]␣
↪free=True ,
    'c1_eUD_elong': oimParam at 0x23d9e7192e0 : elong=2 ± 0  range=[-inf,inf]␣
↪free=True ,
    'c1_eUD_pa': oimParam at 0x23d9e719490 : pa=90 ± 0 deg range=[-inf,inf] free=True ,
    'c1_eUD_d': oimParam at 0x23d9e7193a0 : d=0.5 ± 0 mas range=[-inf,inf] free=True ,
    'c2_EG_f': oimParam at 0x23d9e7191c0 : f=1 ± 0  range=[-inf,inf] free=True ,
    'c2_EG_fwhm_interp1': oimParam at 0x23d9e7192b0 : fwhm=2 ± 0 mas range=[-inf,inf]␣
↪free=True ,
    'c2_EG_fwhm_interp2': oimParam at 0x23d9e719340 : fwhm=8 ± 0 mas range=[-inf,inf]␣
↪free=True }
```

That way we have reduced our number of free parameters to 8. If you change the, for instance, the `params['elong']` or `el.params['elong']` values it will change both parameters are they are actually the same instance of the `oimParam` class.

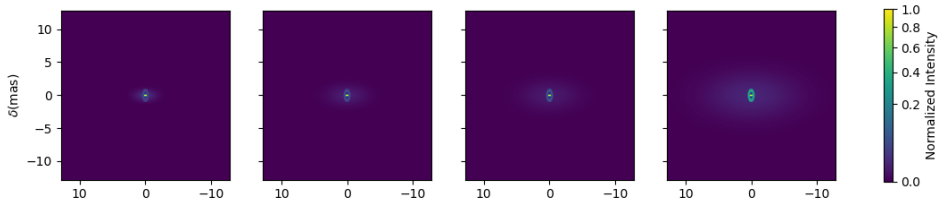
Let's create a new model which include a elongated ring perpendicular to the Gaussian and Ellipse `pa` and with a inner and outer radii equals to 2 and 4 times the ellipse diameter, respectively.

```
er = oim.oimERing()
er.params['elong'] = eg.params['elong']
```

(continues on next page)

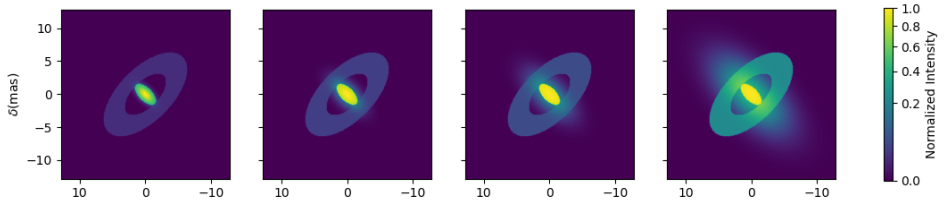
(continued from previous page)

```
er.params['pa'] = oim.oimParamLinker(eg.params["pa"], "add", 90)
er.params['din'] = oim.oimParamLinker(el.params["d"], "mult", 2)
er.params['dout'] = oim.oimParamLinker(el.params["d"], "mult", 4)

m4 = oim.oimModel([el, eg, er])

fig4im, ax4im, im4 = m4.showModel(256, 0.1, wl=[3e-6, 3.25e-6, 3.5e-6, 4e-6],
                                  figsize=(3.5, 2.5), normPow=0.5, fromFT=fromFT,
                                  normalize=True,
                                  savefig=save_dir / "complexModel_link.png")
```



Although quite complex this models only have 9 free parameters. If we change the ellipse diameter and its position angle, the components will scale (except the Gaussian that fwhm is independent) and rotate.

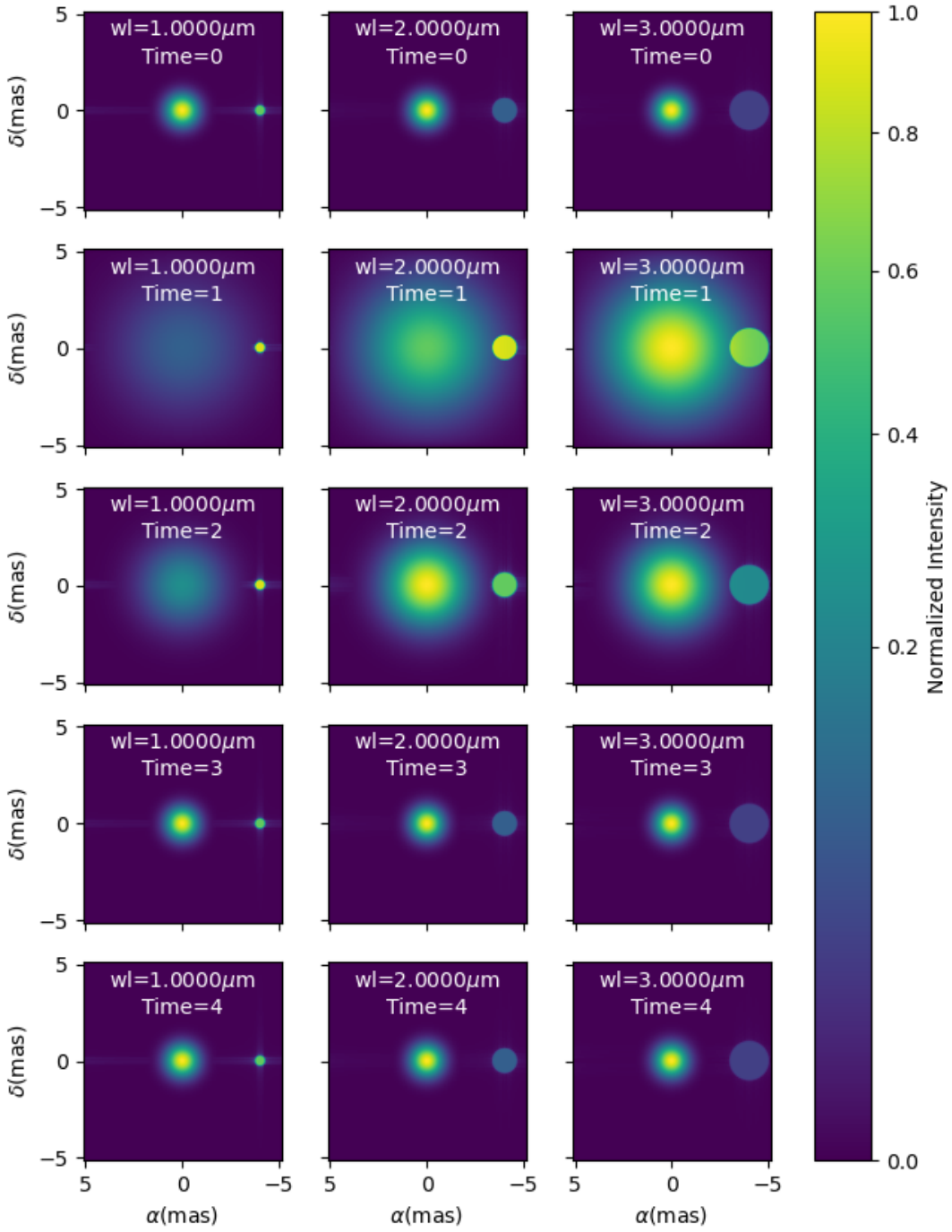
```
el.params['d'].value = 4
el.params['pa'].value = 45

m4.showModel(256, 0.1, wl=[3e-6, 3.25e-6, 3.5e-6, 4e-6], normPow=0.5, figsize=(3.5, 2.5))
```



You can also add time dependent parameters to your model using `oimInterpTime <oimodeler.oimParam.oimInterp()` class which works similarly to the `oimInterpWl` class.

Here, we create a two-components model with a time dependent Gaussian fwhm and a wavelength dependent uniform disk diameter.

```
gd1 = oim.oimGauss(fwhm=oim.oimInterp('time', mjd=[0, 1, 3], values=[1, 4, 1]))
ud1 = oim.oimUD(d=oim.oimInterp("wl", wl=[1e-6, 3e-6], values=[0.5, 2]), x=-4, y=0, f=0.
↪1)

m5 = oim.oimModel(gd1, ud1)

wls=np.array([1,2,3])*1e-6
times=[0,1,2,3,4]

fig5im, ax5im, im5 = m5.showModel(256, 0.04, wl=wls, t=times, legend=True, figsize=(2.5,␣
↪2))
```


wl=1.0000μm
Time=0
wl=2.0000μm
Time=0
wl=3.0000μm
Time=0
wl=1.0000μm
Time=1
wl=2.0000μm
Time=1
wl=3.0000μm
Time=1
wl=1.0000μm
Time=2
wl=2.0000μm
Time=2
wl=3.0000μm
Time=2
wl=1.0000μm
Time=3
wl=2.0000μm
Time=3
wl=3.0000μm
Time=3
wl=1.0000μm
Time=4
wl=2.0000μm
Time=4
wl=3.0000μm
Time=4
δ(mas)
α(mas)
5
0
−5
1.0
0.8
0.6
0.4
0.2
0.0
Normalized Intensity

5.4.2.2 Precomputed chromatic image-cubes

In the FitsImageCubeModels.py.py script, we demonstrate the capability of building models using pre-computed chromatic image-cubes in the fits format.

In this example we will use a chromatic image-cube computed around the $Br\,\gamma$ emission line for a classical Be Star circumstellar disk. The model, detailed in Meilland et al. (2012) was taken form the AMHRA service of the JMMC.

Note: AMHRA develops and provides various online astrophysical models dedicated to the scientific exploitation of high angular and high spectral facilities. Currently available models are : semi-physical gaseous disk of classical Be stars and dusty disk of YSO, Red-supergiant and AGB, binary spiral for WR stars, physical limb-darkening models, kinematics gaseous disks, and a grid of supergiant B[e] stars models.

Let's start by importing `oimodeler` as well as useful packages.

```
from pathlib import Path
from pprint import pprint

import matplotlib.colors as colors
import matplotlib.cm as cm
import numpy as np
import oimodeler as oim
from matplotlib import pyplot as plt
```

The fits-formatted image-cube we will use, *KinematicsBeDiskModel.fits*, is located in the *./examples/AdvancedExamples* directory.

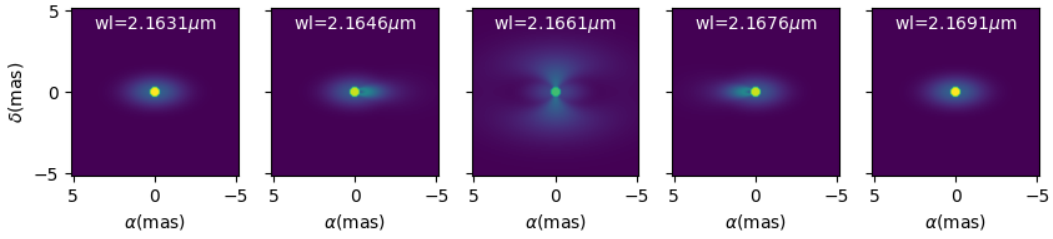
```
path = Path(__file__).parent.parent.parent
file_name = path / "examples" / "AdvancedExamples" / "KinematicsBeDiskModel.fits"
```

We build our model using a single component of the type `oimComponentFitsImage` which allows to load fits images or image-cubes.

```
c = oim.oimComponentFitsImage(file_name)
m = oim.oimModel(c)
```

We can now plot images of the model through the $Br\gamma$ emission line (21661 μ m).

```
wl0, dwl, nwl = 2.1661e-6, 60e-10, 5
wl = np.linspace(wl0-dwl/2, wl0+dwl/2, num=nwl)
m.showModel(256, 0.04, wl=wl, legend=True, normPow=0.4, colorbar=False,
            figsize=(2, 2.5),
            savefig=save_dir / "FitsImageCube_BeDiskKinematicsModel_images.png")
```

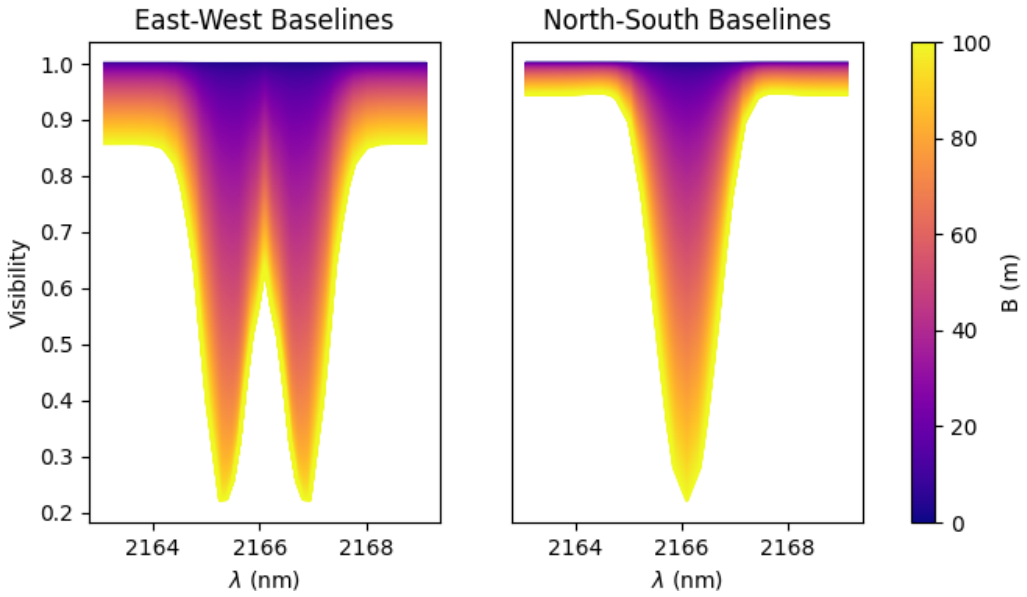


We now compute the visibility for a series of North-South and East-West baselines ranging between 0 and 100m and with the wavelength ranging through the emission line.

```
nB = 1000
nwl = 51
wl = np.linspace(wl0-dwl/2, wl0+dwl/2, num=nwl)

B = np.linspace(0, 100, num=nB//2)

# 1st half of B array are baseline in the East-West orientation
Bx = np.append(B, B*0)
By = np.append(B*0, B)  # 2nd half are baseline in the North-South orientation

Bx_arr = np.tile(Bx[None, :], (nwl, 1)).flatten()
By_arr = np.tile(By[None, :], (nwl,  1)).flatten()
wl_arr = np.tile(wl[:, None], (1, nB)).flatten()

spfx_arr = Bx_arr/wl_arr
spfy_arr = By_arr/wl_arr

vc = m.getComplexCoherentFlux(spfx_arr, spfy_arr, wl_arr)
v = np.abs(vc.reshape(nwl, nB))
v = v/np.tile(v[:, 0][:, None], (1, nB))
```

Finally, we plot the results as a function of the wavelength and with a colorscale in terms of the baseline length.

```
fig, ax = plt.subplots(1, 2, figsize=(8, 4))
titles = ["East-West Baselines", "North-South Baselines"]

for iB in range(nB):
    cB = (iB % (nB//2))/(nB//2-1)
    ax[2*iB//nB].plot(wl*1e9, v[:, iB],
                      color=plt.cm.plasma(cB))

for i in range(2):
    ax[i].set_title(titles[i])
    ax[i].set_xlabel(r"$\lambda$ (nm)")
ax[0].set_ylabel("Visibility")
ax[1].get_yaxis().set_visible(False)

norm = colors.Normalize(vmin=np.min(B), vmax=np.max(B))
sm = cm.ScalarMappable(cmap=plt.cm.plasma, norm=norm)
fig.colorbar(sm, ax=ax, label="B (m)")
```



As expected, for a rotating disk (see Meilland et al. (2012) for more details), the visibility for the baselines along the major-axis show a W-shaped profile through the line, whereas the visibliity along the minor-axis of the disk show a V-shaped profile.

5.4.2.3 Parameters Interpolators

In the previous example, we have introduction parameters interpolators that allow to create chromatic and/or time-dependent models. Here we present in more details these interpolators. This example can be found in the paramInterpolators.py script.

The following table summarize the available interpolators and their parameters. Most of them will be presented in this example.

Class name	oimInterp macro	Description	Parameters
oimParamInterpolatorWl	"wl"	Interp between key wl	wl, values
oimParamInterpolatorTime	"time"	Interp between key time	mjd, values
oimParamGaussianWl	"GaussWl"	Gaussian in wl	val0, value, x0, fwhm
oimParamGaussianTime	"GaussTime"	Gaussian in time	val0, value, x0, fwhm
oimParamMultipleGaussianWl	"mGaussWl"	Multiple Gauss. in wl	val0 and value, x0, fwhm
oimParamMultipleGaussianTime	"mGaussTime"	Multiple Gauss. in time	val0 and value, x0, fwhm
oimParamCosineTime	"cosTime"	Asym. Cosine in Time	T0, P, values (optional x0)
oimParamPolynomialWl	"polyWl"	Polynomial in wl	coeffs
oimParamPolynomialTime	"polyTime"	Polynomial in time	coeffs

We start by importing the standard packages.

```
from pathlib import Path
from pprint import pprint

import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cm as cm
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import oimodeler as oim
```

In order to simplify plotting the various interpolators we define a plotting function that can works for either a chromatic or a time-dependent model. With some baseline length, wavelength, time vectors passed and some model and interpolated parameter, the function will plot the interpolated parameters as a function of the wavelength or time, and the corresponding visibilities.

```
def plotParamAndVis(B, wl, t, model, param, ax=None, colorbar=True):
    nB = B.size

    if t is None:
        n = wl.size
        x = wl*1e6
        y = param(wl, 0)
        xlabel = r"$\lambda$ ($\mu$m)"
    else:
        n = t.size
        x = t-60000
        y = param(0, t)
        xlabel = "MJD - 60000 (days)"

    Bx_arr = np.tile(B[None, :], (n, 1)).flatten()
    By_arr = Bx_arr*0

    if t is None:
        t_arr = None
        wl_arr = np.tile(wl[:, None], (1, nB)).flatten()
        spfx_arr = Bx_arr/wl_arr
        spfy_arr = By_arr/wl_arr
    else:
        t_arr = np.tile(t[:, None], (1, nB)).flatten()
        spfx_arr = Bx_arr/wl
        spfy_arr = By_arr/wl
        wl_arr = None

    v = np.abs(model.getComplexCoherentFlux(
        spfx_arr, spfy_arr, wl=wl_arr, t=t_arr).reshape(n, nB))

    if ax is None:
        fig, ax = plt.subplots(2, 1)
    else:
        fig = ax.flatten()[0].get_figure()

    ax[0].plot(x, y, color="r")

    ax[0].set_ylabel("{} (mas)".format(param.name))
    ax[0].get_xaxis().set_visible(False)

    for iB in range(1, nB):
        ax[1].plot(x, v[:, iB]/v[:, 0], color=plt.cm.plasma(iB/(nB-1)))
```

(continues on next page)

(continued from previous page)

```
    ax[1].set_xlabel(xlabel)
    ax[1].set_ylabel("Visibility")

    if colorbar == True:
        norm = colors.Normalize(vmin=np.min(B[1:]), vmax=np.max(B))
        sm = cm.ScalarMappable(cmap=plt.cm.plasma, norm=norm)
        fig.colorbar(sm, ax=ax, label="Baseline Length (m)")

    return fig, ax, v
```

We will need a baseline length vector (here 200 baselines between 0 and 60m) and we will build for each model either a length 1000 wavelength or time vector.

```
nB = 200
B = np.linspace(0, 60, num=nB)

nwl = 1000
nt = 1000
```

Now, let's start with our first interpolator: A Gaussian in wavelength (also available for time). It can be used to model spectral features like atomic lines or molecular bands in emission or absorption.

It has 4 parameters :

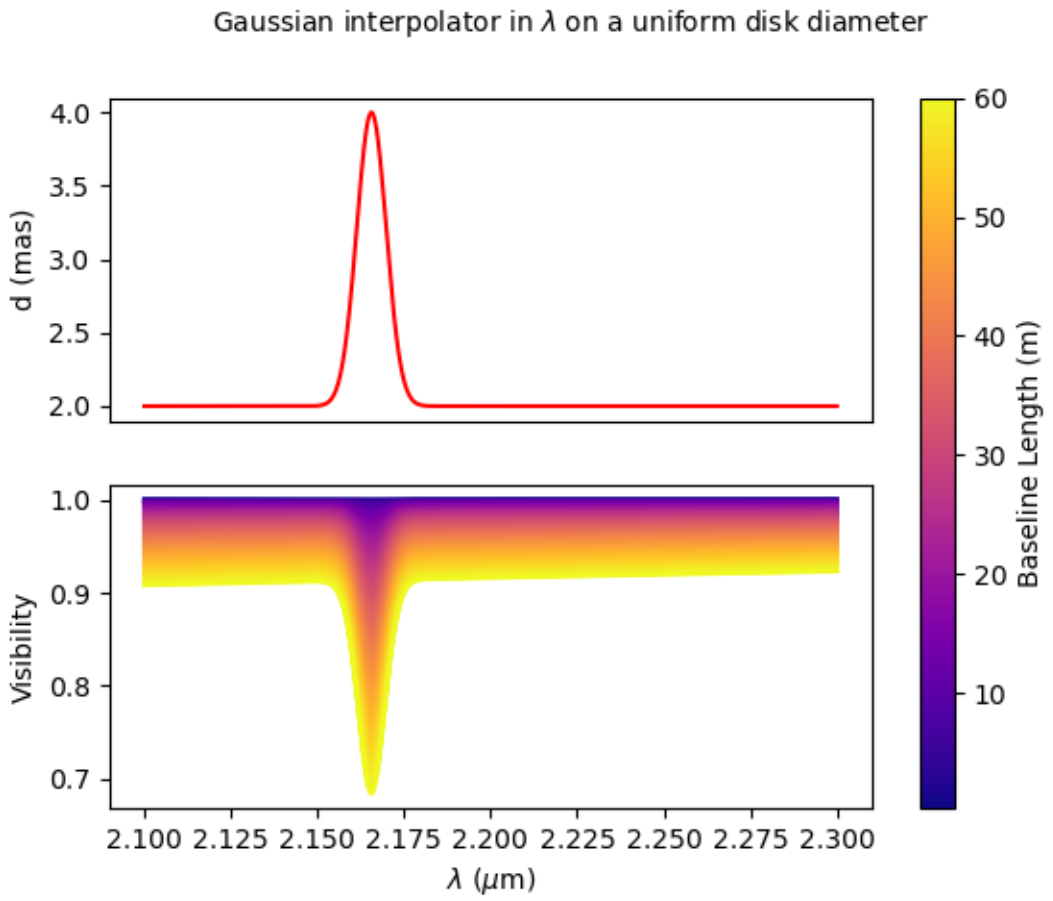
- A central wavelength `x0`
- A value outside the Gaussian (or offset) : `val0`
- A value at the maximum of the Gaussian : `value`
- A full width at half maximum : `fwhm`

To create such an interpolator, we use the class `oimInterp` class and specify `GaussWl` as the type of interpolator. In our example below we create a Uniform Disk model with a diameter interpolated between 2 mas (outside the Gaussian range) and 4 mas at the top of the Gaussian. The central wavelength is set to 2.1656 microns (Brackett Gamma hydrogen line) and the fwhm to 10nm.

```
c1 = oim.oimUD(d=oim.oimInterp('GaussWl', val0=2, value=4, x0=2.1656e-6, fwhm=1e-8))
m1 = oim.oimModel(c1)
```

Finally, we can define the wavelength range and use our custom plotting function.

```
wl = np.linspace(2.1e-6, 2.3e-6, num=nwl)
fig, ax, im = plotParamAndVis(B, wl, None, m1, c1.params['d'])
fig.suptitle("Gaussian interpolator in $\lambda$ on a uniform disk diameter",␣
↪fontsize=10)
```



The parameters of the interpolator can be accessed using the `params` attribute of the `oimParamInterpolator`:

```
pprint(c1.params['d'].params)
```

```
... [oimParam at 0x2610e25e220 : x0=2.1656e-06 ± 0 m range=[0,inf] free=True ,
     oimParam at 0x2610e25e250 : fwhm=1e-08 ± 0 m range=[0,inf] free=True ,
     oimParam at 0x2610e25e280 : d=2 ± 0 mas range=[-inf,inf] free=True ,
     oimParam at 0x2610e25e2b0 : d=4 ± 0 mas range=[-inf,inf] free=True ]
```

Each one can also be accessed using their name as an attribute:

```
pprint(c1.params['d'].x0)
```

```
... oimParam x0 = 2.1656e-06 ± 0 m range=[0,inf] free
```

These parameters will behave like normal free or fixed parameters when performing model fitting. We can get the full list of parameters from our model using the `getParameter` method.

```
pprint(m1.getParameters())
```

```
... {'c1_UD_x': oimParam at 0x2610e25e100 : x=0 ± 0 mas range=[-inf,inf] free=False ,
     'c1_UD_y': oimParam at 0x2610e25e130 : y=0 ± 0 mas range=[-inf,inf] free=False ,
     'c1_UD_f': oimParam at 0x2610e25e160 : f=1 ± 0  range=[-inf,inf] free=True ,
```

(continues on next page)

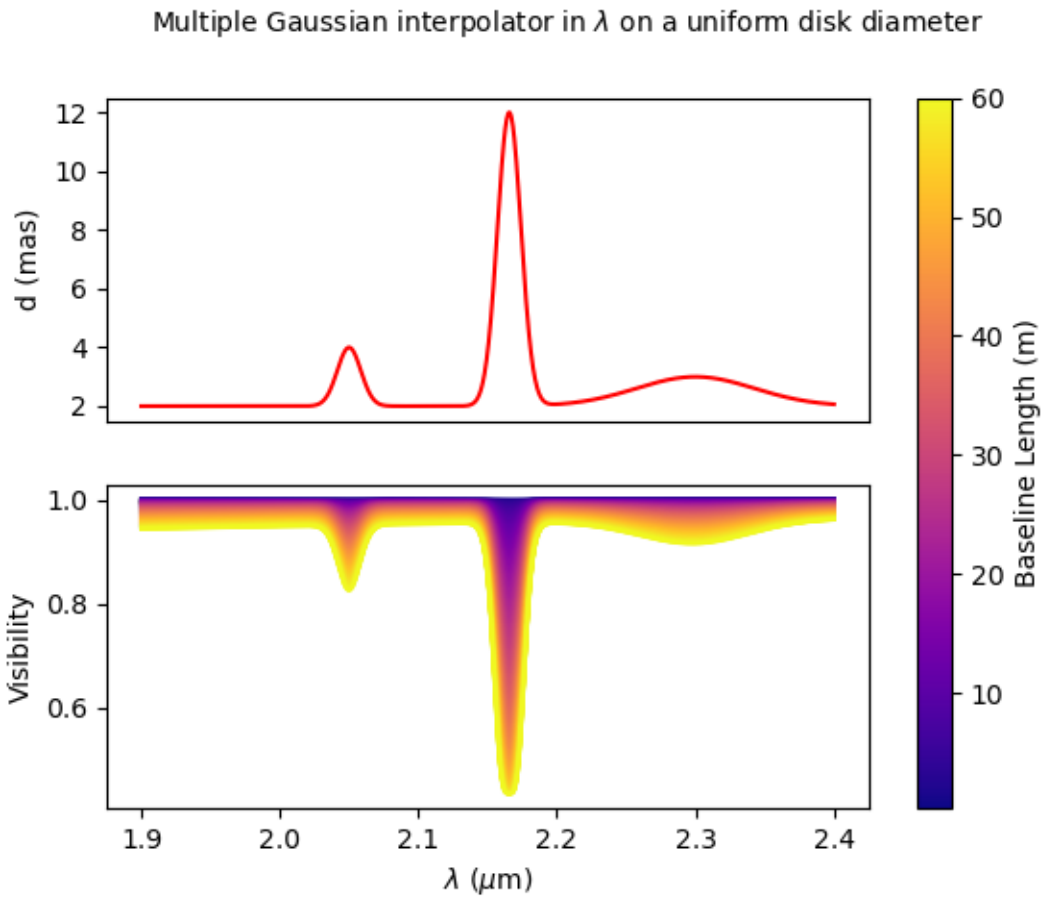
(continued from previous page)

```
    'c1_UD_d_interp1': oimParam at 0x2610e25e220 : x0=2.1656e-06 ± 0 m range=[0,inf]␣
↪free=True ,
    'c1_UD_d_interp2': oimParam at 0x2610e25e250 : fwhm=1e-08 ± 0 m range=[0,inf]␣
↪free=True ,
    'c1_UD_d_interp3': oimParam at 0x2610e25e280 : d=2 ± 0 mas range=[-inf,inf]␣
↪free=True ,
    'c1_UD_d_interp4': oimParam at 0x2610e25e2b0 : d=4 ± 0 mas range=[-inf,inf]␣
↪free=True }
```

In the dictionary returned by the `getParameters` method, the four interpolator parameters are called `c1_UD_d_interpX`.

The second interpolator presented here is the multiple Gaussian in wavelength (also available for time). It is a generalisation of the first interpolator but with multiple values for `x0`, `fwhm` and `values`.

```
c2 = oim.oimUD(f=0.5, d=oim.oimInterp("mGaussWl", val0=2, values=[4, 0, 0],
                                      x0=[2.05e-6, 2.1656e-6, 2.3e-6],
                                      fwhm=[2e-8, 2e-8, 1e-7]))
pt = oim.oimPt(f=0.5)
m2 = oim.oimModel(c2, pt)

c2.params['d'].values[1] = oim.oimParamLinker(
    c2.params['d'].values[0], "mult", 3)
c2.params['d'].values[2] = oim.oimParamLinker(
    c2.params['d'].values[0], "add", -1)

wl = np.linspace(1.9e-6, 2.4e-6, num=nwl)

fig, ax, im = plotParamAndVis(B, wl, None, m2, c2.params['d'])
fig.suptitle(
    "Multiple Gaussian interpolator in $\lambda$ on a uniform disk diameter",␣
↪fontsize=10)
```



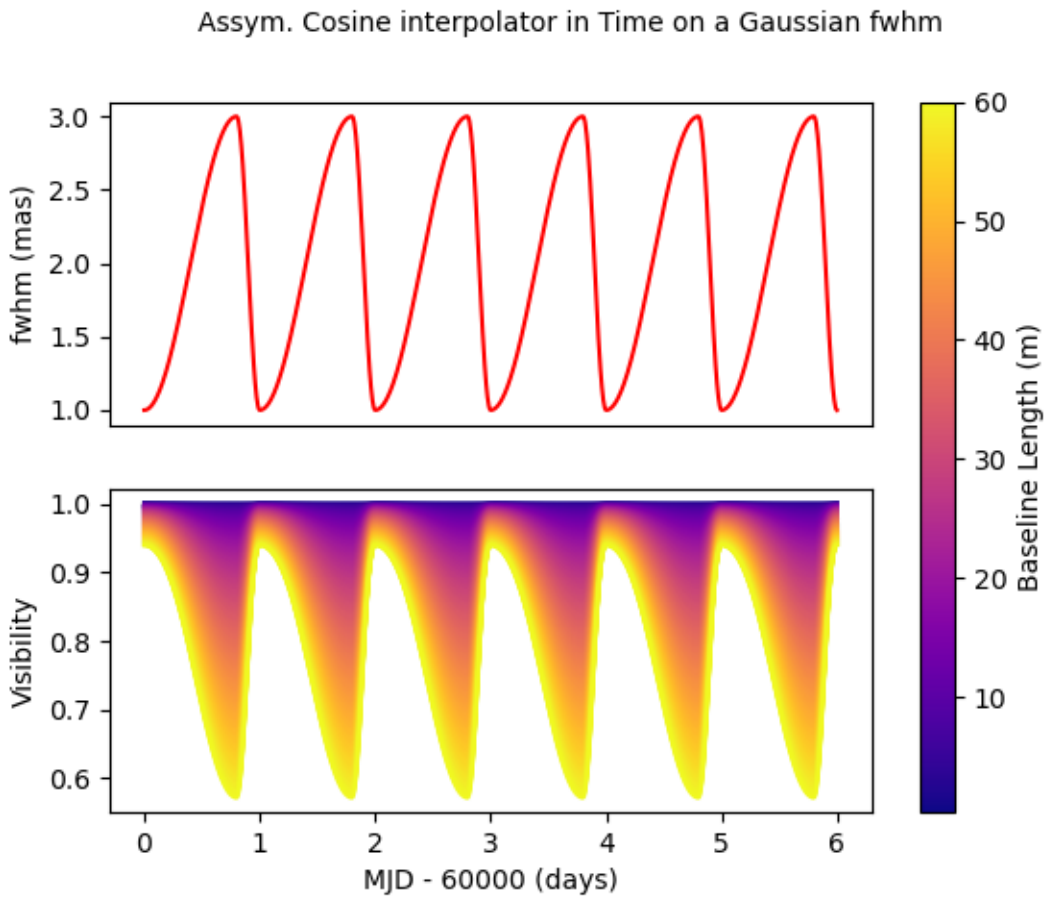
Here, to reduce the number of free parameters of the model with have linked the second and third `values` of the interpolator to the first one.

Let's look at our third interpolator: An asymmetric cosine interpolator in time. As it is cyclic it might be used to simulated a cyclic variation, for example a pulsating star.

It has 5 parameters :

- The Epoch (mjd) of the minimum value: `T0`.
- The period of the variation in days `P`.
- The mini and maximum values of the parameter as a two-elements array : `value`.
- Optionally, the asymmetry : `x0` (x0=0.5 means no assymetry, x0=0 or 1 maximum asymmetry).

```
c3 = oim.oimGauss(fwhm=oim.oimInterp(
    "cosTime", T0=60000, P=1, values=[1, 3], x0=0.8))
m3 = oim.oimModel(c3)

t = np.linspace(60000, 60006, num=nt)
wl = 2.2e-6

fig, ax, im = plotParamAndVis(B, wl, t, m3, c3.params['fwhm'])
fig.suptitle(
    "Assym. Cosine interpolator in Time on a Gaussian fwhm", fontsize=10)
```



Now, let's have a look at the classic wavelength interpolator (also available for time). jIt has two parameters:

- A list of reference wavelengths: `wl`.
- A list of values at the reference wavelengths: `values`.

Values will be interpolated in the range, using either linear (default), quadratic, or cubic interpolation set by the keyword `kind`. Outside the range of defined wavelengths the values will be either fixed (default) or extrapolated depending on the value of the `extrapolate` keyword.

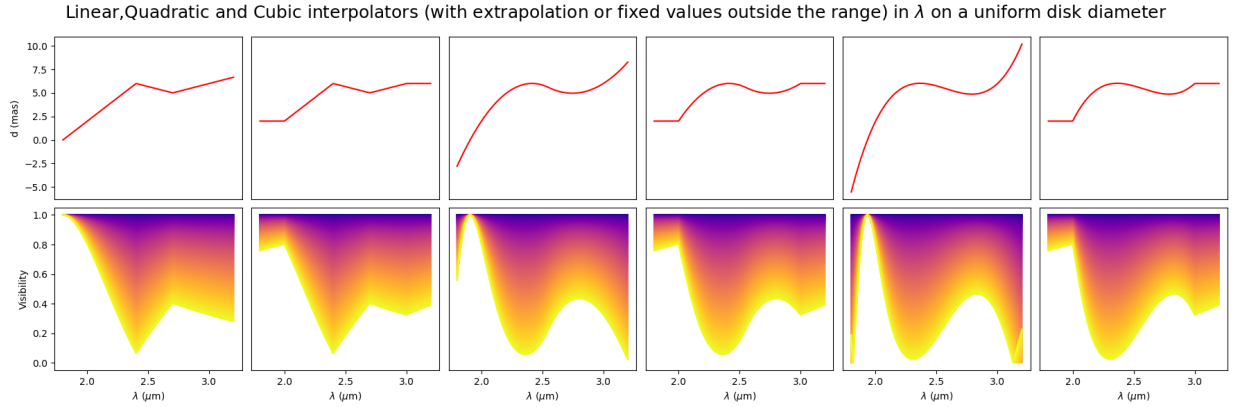
Here, we present examples with the three kind of interpolation and with or without extrapolation.

```
c4 = oim.oimIRing(d=oim.oimInterp("wl", wl=[2e-6, 2.4e-6, 2.7e-6, 3e-6], values=[2, 6, 5,
↪ 6],
                                  kind="linear", extrapolate=True))
m4 = oim.oimModel(c4)
wl = np.linspace(1.8e-6, 3.2e-6, num=nwl)
fig, ax = plt.subplots(2, 6, figsize=(18, 6), sharex=True, sharey="row")

plotParamAndVis(B, wl, None, m4, c4.params['d'], ax=ax[:, 0], colorbar=False)
c4.params['d'].extrapolate = False
plotParamAndVis(B, wl, None, m4, c4.params['d'], ax=ax[:, 1], colorbar=False)

c4.params['d'].extrapolate = True
c4.params['d'].kind = "quadratic"
plotParamAndVis(B, wl, None, m4, c4.params['d'], ax=ax[:, 2], colorbar=False)
```

(continues on next page)

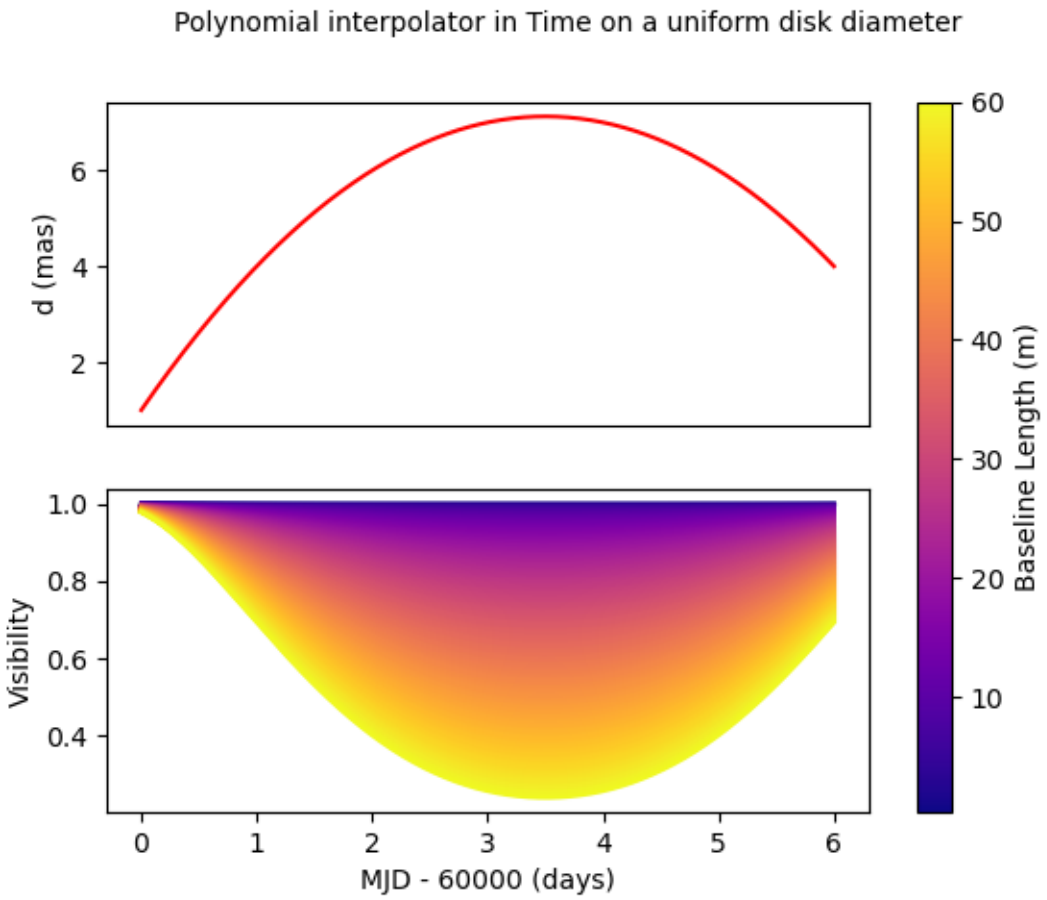
(continued from previous page)

```
c4.params['d'].extrapolate = False
plotParamAndVis(B, wl, None, m4, c4.params['d'], ax=ax[:, 3], colorbar=False)

c4.params['d'].extrapolate = True
c4.params['d'].kind = "cubic"
plotParamAndVis(B, wl, None, m4, c4.params['d'], ax=ax[:, 4], colorbar=False)
c4.params['d'].extrapolate = False
plotParamAndVis(B, wl, None, m4, c4.params['d'], ax=ax[:, 5], colorbar=False)

plt.subplots_adjust(left=0.05, bottom=0.1, right=0.99, top=0.9,
                    wspace=0.05, hspace=0.05)
for i in range(1, 6):
    ax[0, i].get_yaxis().set_visible(False)
    ax[1, i].get_yaxis().set_visible(False)

fig.suptitle("Linear, Quadratic and Cubic interpolators (with extrapolation"
             r" or fixed values outside the range) in $\lambda$ on a uniform"
             " disk diameter", fontsize=18)
```



Finally, we can also use a polynominal interpolator in time (also available for wavelength). Its free parameters are the coefficients of the polynomial. The parameter **x0** allows to shift the reference time (in mjd) from 0 to an arbitrary date.

```
c5 = oim.oimUD(d=oim.oimInterp('polyTime', coeffs=[1, 3.5, -0.5], x0=60000))
m5 = oim.oimModel(c5)

wl = 2.2e-6
t = np.linspace(60000, 60006, num=nt)

fig, ax, im = plotParamAndVis(B, wl, t, m5, c5.params['d'])
fig.suptitle(
    "Polynomial interpolator in Time on a uniform disk diameter", fontsize=10)
```



As for other part of the oimodeler software, **oimParamInterpolator** was designed so that users can easily create their own interoplators using inheritage. See the *Creating New Interpolators* example.

5.4.2.4 Fitting a chromatic model

In the example chromaticModelFit.py we will show how to perform model-fitting with a simple chromatic model.

We will use some ASPRO-simulated data that were computed using a chromatic image-cubes exported from the same `oimodeler` model used for model fitting.

Let's first start by importing packages and setting the path to the data directory.

```
from pathlib import Path
from pprint import pprint

import numpy as np
import oimodeler as oim


path = Path(__file__).parent.parent.parent
data_dir = path / "examples" / "testData" / "ASPRO_CHROMATIC_SKWDISK"
save_dir = path / "images"
product_dir = path / "examples" / "testData" / "IMAGES"
if not save_dir.exists():
```

(continues on next page)

(continued from previous page)

```
    save_dir.mkdir(parents=True)
if not product_dir.exists():
    product_dir.mkdir(parents=True)
```

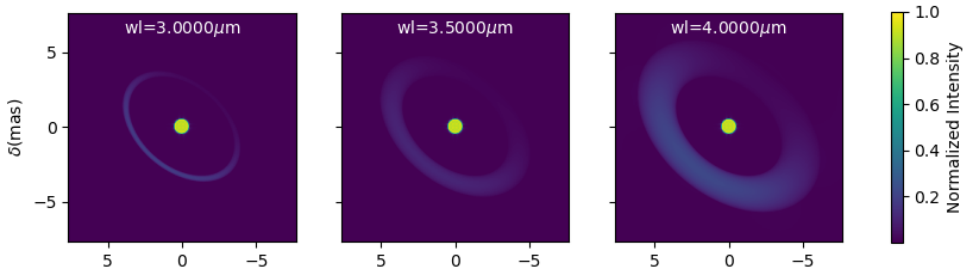
We will build a model mimicing a star (uniform disk) and the inner rim of a dusty disk (Skewed ring). The flux ratio between the two components will depend on the wavelength as well as the outer radius of the skewed ring.

```
star = oim.oimUD(d=1, f=oim.oimInterp(
    "wl", wl=[3e-6, 4e-6], values=[0.5, 0.1]))
ring = oim.oimESKRing(din=8, dout=oim.oimInterp(
    "wl", wl=[3e-6, 4e-6], values=[9, 14]), elong=1.5, skw=0.8, pa=50)
ring.params['f'] = oim.oimParamNorm(star.params['f'])
ring.params["skwPa"] = oim.oimParamLinker(ring.params["pa"], "add", 90)
model = oim.oimModel(star, ring)
```

We used the `oimInterp` class with the `"wl"` option to build a linear interpolator for the parameter `f` of the uniform disk with two reference wavlengths at 3 and 4 microns with a value of the flux of 0.5 and 0.1, respectively. We also link the flux of the skewed ring so that the total flux is normalized. The ring outer radius `dout` is also interpolated between 9 mas at 3 microns and 14 at 4 microns. Finally, we set the ring skweness position angle `skwPa` to be perpendicular to the ring major-axis (set with the `pa` parameter).

We can have a look at our model at three wavelengths 3, 3.5 and 4 microns.

```
model.showModel(256, 0.06, wl=np.linspace(3., 4, num=3)*1e-6, legend=True,
                normalize=True, normPow=1, fromFT=True,
                savefig=save_dir / "chromaticModelFitImageInit.png")
```



The simulated data that we will use where created with fits-formated image-cube computed with this image using the `getImage` method with the `toFits=True` option. Here we simulate 50 wavelengths for the cube as ASPRO doesn't interpolate between wavelengths of imported image-cube yet.

```
img = model.getImage(256, 0.06, toFits=True, fromFT=True,
                     wl=np.linspace(3, 4, num=50)*1e-6)
img.writeto(product_dir / "skwDisk.fits", overwrite=True)
```

Using this model and the ASPRO software, we have simulated 3 MATISSE observations: One with each of the available standard configuration of the ATs telescopes at VLTI: **small**, **medium** and **large**. The three observations were exported as a single OIFITS file.

Let's load it into a `oimData` object, and apply an filter from the `oimDataFilter` module that will keep only VISDATA2 and T3PHI more the model-fitting process.

```
files = list(map(str, data_dir.glob("*.fits")))
data = oim.oimData(files)
```

(continues on next page)

(continued from previous page)

```
f1 = oim.oimRemoveArrayFilter(targets="all", arr=["OI_VIS", "OI_FLUX"])
f2 = oim.oimDataTypeFilter(targets="all", dataType=["T3AMP"])
data.setFilter(oim.oimDataFilter([f1, f2]))
```

Let's have a look at our model's free parameters:

```
params = model.getFreeParameters()
pprint(params)
```

```
... {'c1_UD_d': oimParam at 0x2a0edc241c0 : d=1 ± 0 mas range=[-inf,inf] free=True ,
     'c1_UD_f_interp1': oimParam at 0x2a0edc242b0 : f=0.5 ± 0  range=[-inf,inf]␣
↪free=True ,
     'c1_UD_f_interp2': oimParam at 0x2a0edc242e0 : f=0.1 ± 0  range=[-inf,inf]␣
↪free=True ,
     'c2_SKER_din': oimParam at 0x2a0edc24220 : din=8 ± 0 mas range=[-inf,inf]␣
↪free=True ,
     'c2_SKER_dout_interp1': oimParam at 0x2a0edc24e80 : dout=9 ± 0 mas range=[-inf,
↪inf] free=True ,
     'c2_SKER_dout_interp2': oimParam at 0x2a0edc24e50 : dout=14 ± 0 mas range=[-inf,
↪inf] free=True ,
     'c2_SKER_elong': oimParam at 0x2a0edc24310 : elong=1.5 ± 0  range=[-inf,inf]␣
↪free=True ,
     'c2_SKER_pa': oimParam at 0x2a0edc24400 : pa=50 ± 0 deg range=[-inf,inf] free=True␣
↪,
     'c2_SKER_skw': oimParam at 0x2a0edc24460 : skw=0.8 ± 0  range=[-inf,inf] free=True␣
↪}
```

Here, we see that the flux of the uniform disk and the outer radius of the skewed ring have both been replaced by two parameters representing thir respective values at the reference wavelengths: `c1_UD_f_interp1` is the flux at 3 microns and `c1_UD_f_interp2` the flux at 4 microns.

Before running the fit we need to set the parameter space for all free parameters:

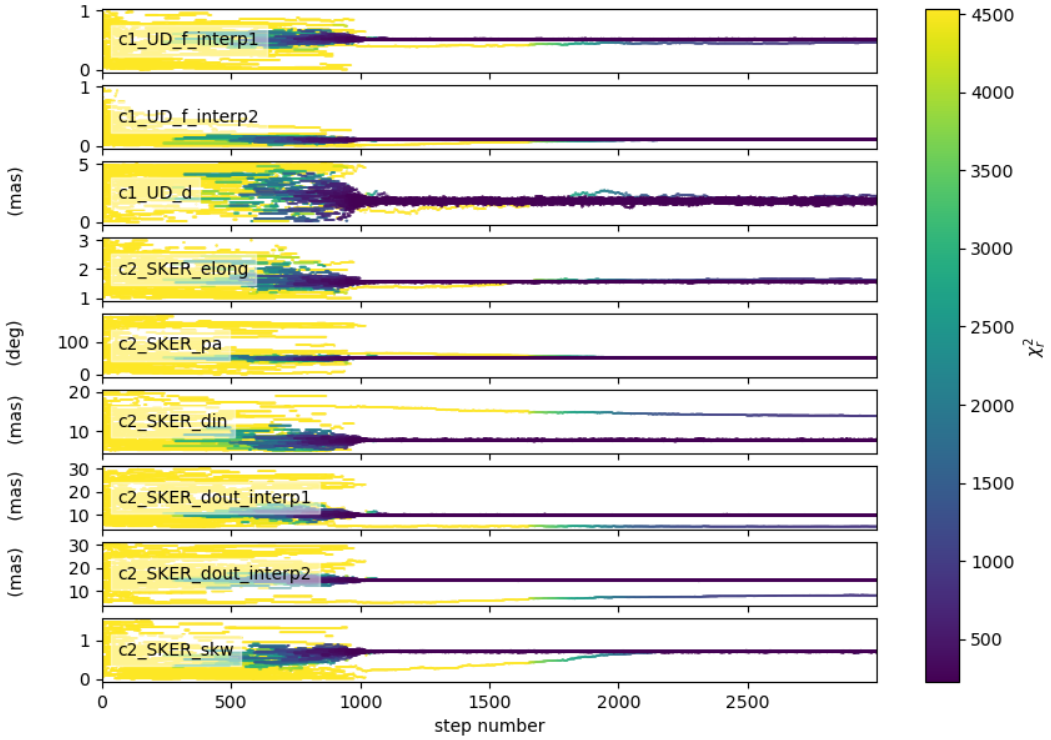
```
params['c1_UD_f_interp1'].set(min=0.0, max=1)
params['c1_UD_f_interp2'].set(min=-0.0, max=1)
params['c1_UD_d'].set(min=0, max=5, free=True)
params['c2_SKER_pa'].set(min=0., max=180)
params['c2_SKER_elong'].set(min=1, max=3)
params['c2_SKER_din'].set(min=5, max=20.)
params['c2_SKER_skw'].set(min=0, max=1.5)
params['c2_SKER_dout_interp1'].set(min=5., max=30.)
params['c2_SKER_dout_interp2'].set(min=5., max=30.)
```

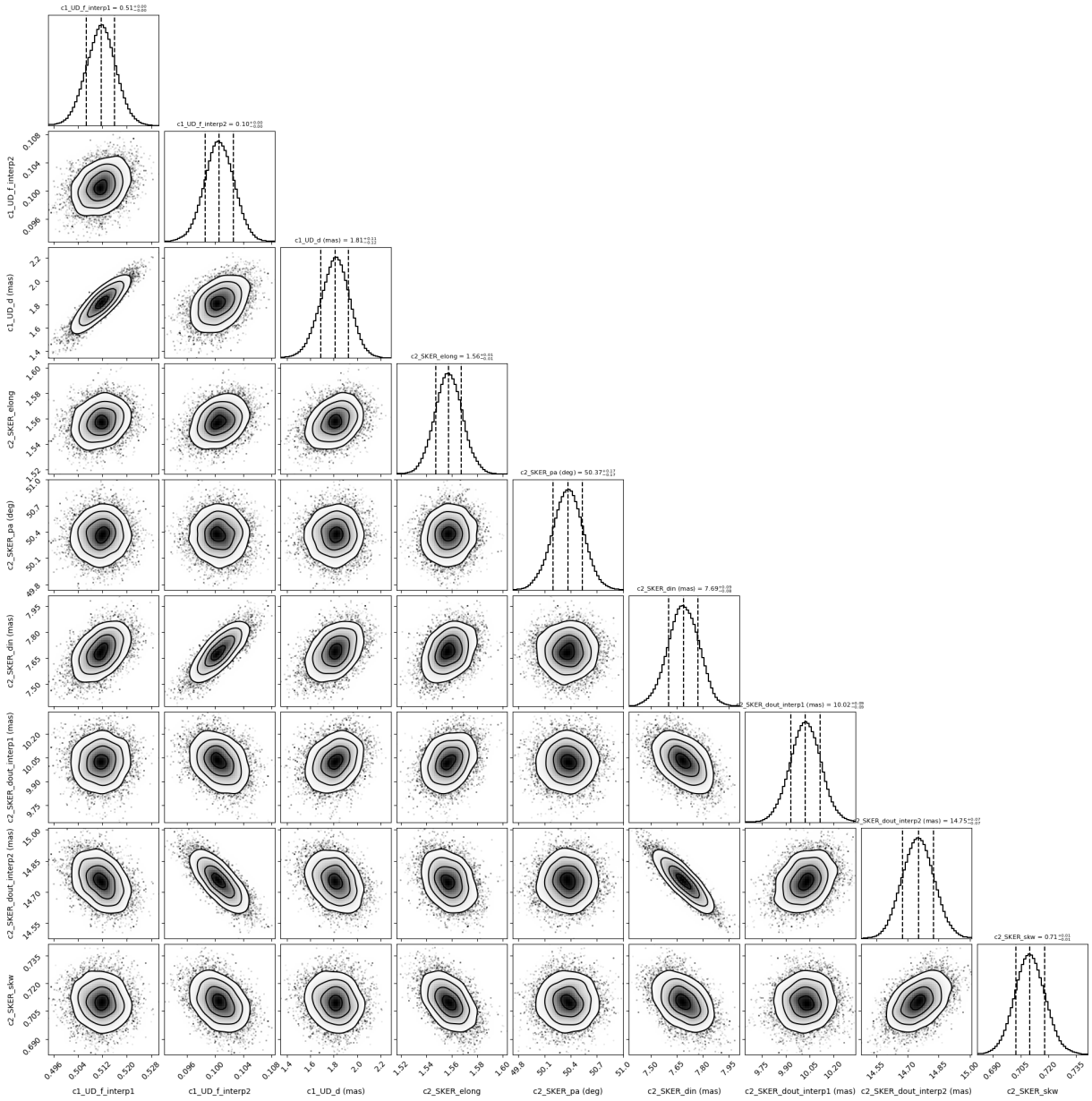
Now we can perform the model-fitting using the *emcee <https://emcee.readthedocs.io/en/stable/>* -based fitter with 30 walkers, 2000 steps and starting at a random position within the parameter space.

```
fit = oim.oimFitterEmcee(data, model, nwalkers=50)
fit.prepare(init="random")
fit.run(nsteps=3000, progress=True)
```

Plotting the walkers and the corner plot (discarding the first half of the steps of the run).

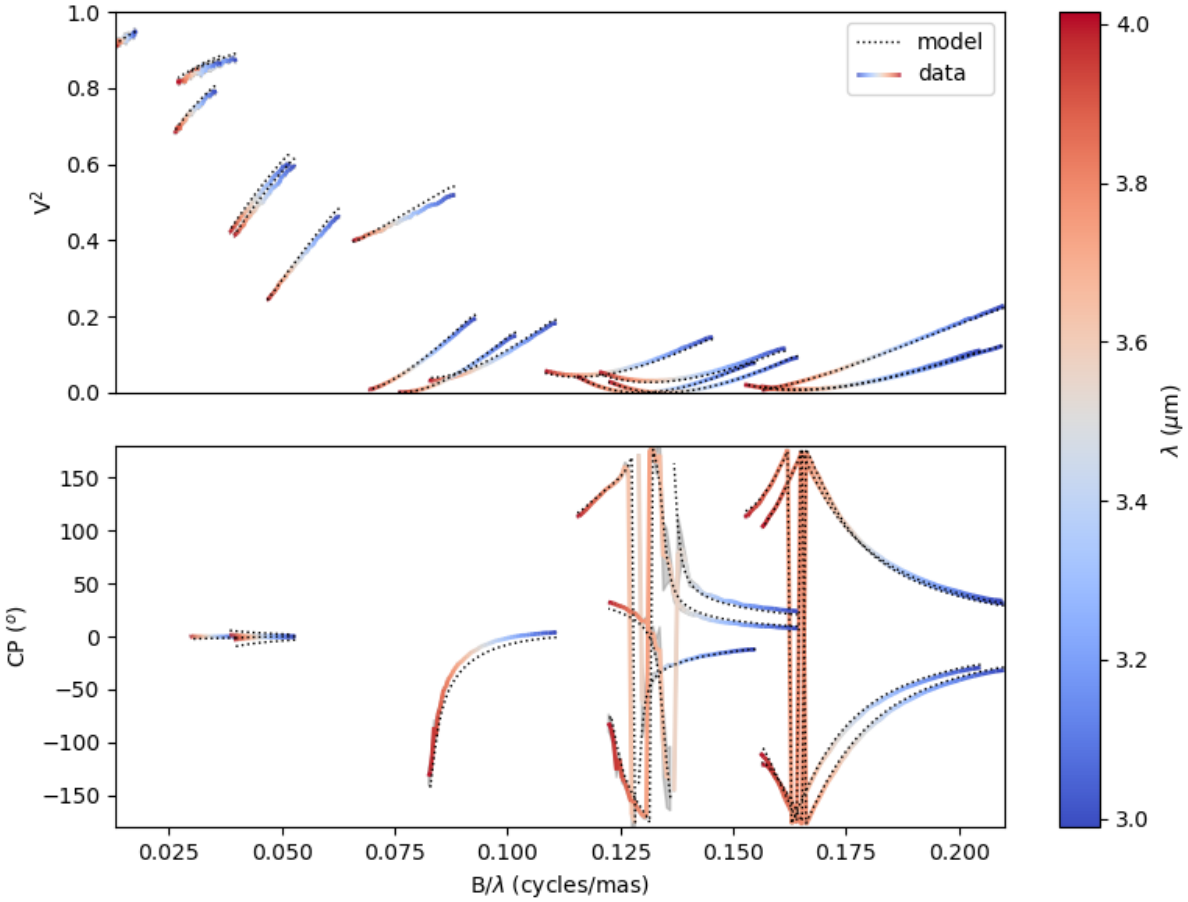
```
figWalkers, axeWalkers = fit.walkersPlot()
figCorner, axeCorner = fit.cornerPlot(discard=1000)
```





Finally, getting the best parameters and the uncertainties and plotting the fit data.

```
median, err_l, err_u, err = fit.getResults(mode='best', discard=1000)
fit.simulator.plot(["VIS2DATA", "T3PHI"])
```



5.4.3 Expanding the Software

In this section we present examples that show how to expand the functionality of the `oimodeler` software by creating customs objects: `oimComponents`, `oimFilters`, `oimFitters`, and custom plotting functions or utils.

5.4.3.1 Creating New Components

Box (Fourier plan formula)

In the createCustomComponentFourier.py example we show how to implement a new model component using a formula in the Fourier plane. The component will inherit from the `oimComponentFourier` class. The Fourier formula should be implemented as the `oimComponentFourier._visFunction` method and, optionally, the formula in the image plan can be implemented using the `oimComponentFourier._imageFunction` method.

For this example we will show how to implement a basic rectangular box component. We start by importing the required packages:

```
from pathlib import Path

import astropy.units as u
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import oimodeler as oim
```

Our new component will be named `oimBox`, and it will have two parameters, *dx* and *dy* the size of the box in the x and y directions. Let's start to implement the oimBox class and its `__init__` method.

```
class oimBox(oim.oimComponentFourier):
    name = "2D Box"
    shortname = "BOX"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.params["dx"] = oim.oimParam(
            name="dx", value=1, description="Size in x", unit=u.mas)
        self.params["dy"] = oim.oimParam(
            name="dy", value=1, description="Size in y", unit=u.mas)
        self._eval(**kwargs)
```

The class inherits from `oimComponentFourier`. The `__init__` method is called with the `**kwargs` to allow the passing of keyword arguments. To inherit from the parent class, we first call its initialization method with `super().__init__`. Then, we define the two new parameters, *dx* and *dy*, which are instances of the `oimParam` class. Finally, we need to call the `oimComponent._eval` method that allows the parameters to be processed.

Now that the new class is created, we need to implement its `oimComponent._visFunction` method, with the Fourier transform formula of our component. This method is called when using the `oimComponent.getComplexCoherentFlux` method.

Note that the component parameters should be called with (*wl*, *t*), to allow parameter chromaticity and time dependence. The parameters have a unit. This unit should also be used to allow the use of other units (via unit conversion) when creating instances of the component.

In our case, the complex visibilty of a rectangle is quite easy to write. It is a simple 2D-sinc function. Note that the x and y sizes are converted from the given unit (usually mas) to rad.

```
def _visFunction(self, ucoord, vcoord, rho, wl, t):
    x = self.params["dx"](wl, t)*self.params["dx"].unit.to(u.rad)*ucoord
    y = self.params["dy"](wl, t)*self.params["dy"].unit.to(u.rad)*vcoord
    return np.sinc(x)*np.sinc(y)
```

We also need to implement the image that will be created when using the `oimComponent.getImage` method. If not implemented, the model will use the Fourier based formula to compute the image. It will also be the case if the keyword `fromFT` is set to `True`, when calling the `getImage` method. However, it is always interesting to implement the image method, at least for debugging purposes, to check that the image computed with the image formula and using the `fromFT` option gives compatible results. We will see that a bit later in an example.

For our box, we can implement the image method with logical operations

```
def _imageFunction(self, xx, yy, wl, t):
    return ((np.abs(xx) <= self.params["dx"](wl, t)/2) &
            (np.abs(yy) <= self.params["dy"](wl, t)/2)).astype(float)
```

The full code of the `oimBox` component is quite short.

```
class oimBox(oim.oimComponentFourier):
    name = "2D Box"
```

(continues on next page)

(continued from previous page)

```
    shortname = "BOX"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.params["dx"] = oim.oimParam(
            name="dx", value=1, description="Size in x", unit=u.mas)
        self.params["dy"] = oim.oimParam(
            name="dy", value=1, description="Size in y", unit=u.mas)
        self._eval(**kwargs)

    def _visFunction(self, ucoord, vcoord, rho, wl, t):
        x = self.params["dx"](wl, t)*self.params["dx"].unit.to(u.rad)*ucoord
        y = self.params["dy"](wl, t)*self.params["dy"].unit.to(u.rad)*vcoord
        return np.sinc(x)*np.sinc(y)

    def _imageFunction(self, xx, yy, wl, t):
        return ((np.abs(xx) <= self.params["dx"](wl, t)/2) &
                (np.abs(yy) <= self.params["dy"](wl, t)/2)).astype(float)
```

We can now use it as we do with any other `oimodeler` component. Let's build our first model with it.

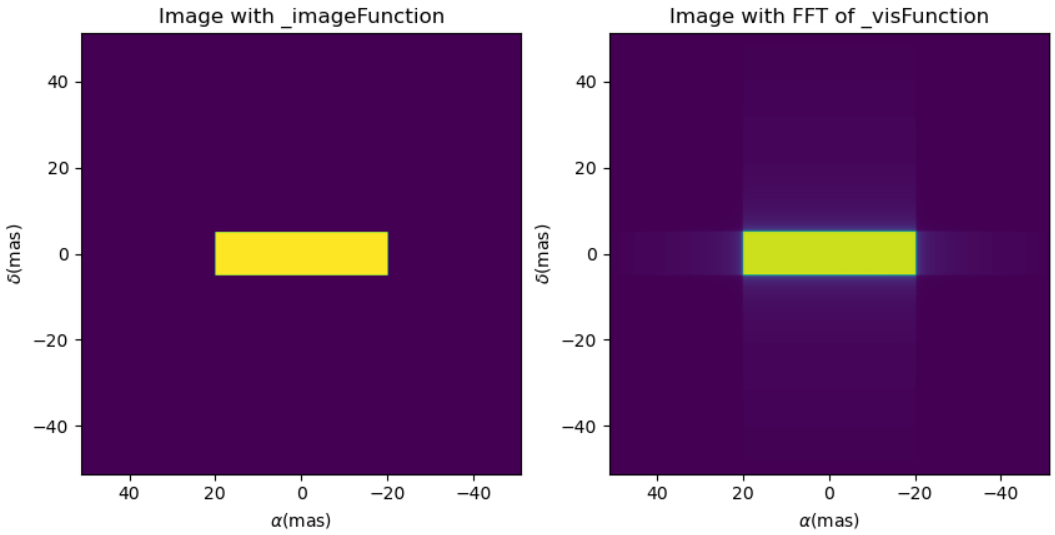
```
b1 = oimBox(dx=40, dy=10)
m1 = oim.oimModel([b1])
```

Now we can create images of our model:

- In the image plane with the `_imageFunction`.
- In the Fourier plane with the `_visFunction` (with the FFT).

Both can be plotted with the `oimModel.showModel` method. To create the image from the FFT of the visibilty function, we just need to set the `fromFT` keyword to `True`.

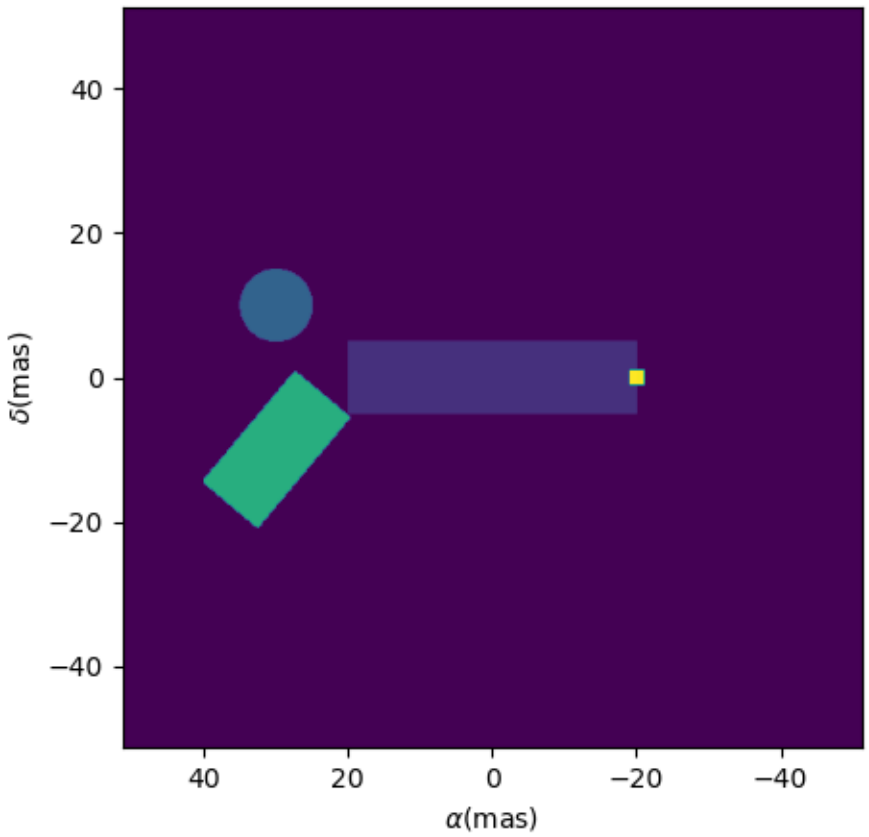
```
fig, ax = plt.subplots(1, 2, figsize=(10,5))
m1.showModel(512, 0.2, axe=ax[0], colorbar=False)
m1.showModel(512, 0.2, axe=ax[1], fromFT=True, colorbar=False)
ax[0].set_title("Image with _imageFunction")
ax[1].set_title("Image with FFT of _visFunction")
```



Of course, as our `oimBox` inherits from the `oimComponent` class, it has three addtional parameters available: Its position described by x and y, and the flux f. All components can also be rotated using the position angle *pa* parameter. Note, that if `elliptic=True` is not set at the component creation as a class variable, the postion angle *pa* parameters (and the *elong* parameter) are not added to the model.

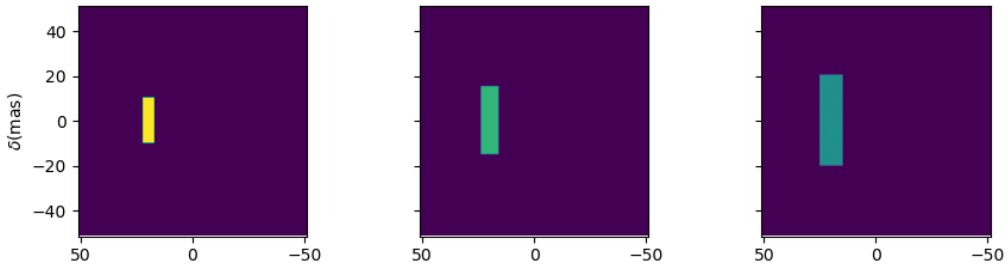
Let's create a complex model with boxes and uniform disk.

```
b2 = oimBox(dx=2, dy=2, x=-20, y=0, f=0.5)
b3 = oimBox(dx=10, dy=20, x=30, y=10, pa=-40, f=10)
c = oim.oimUD(d=10, x=30, y=10)
m2 = oim.oimModel([b1, b2, b3, c])
m2.showModel(512, 0.2, colorbar=False, figsize=(5, 5))
```



We could also create a chromatic box component using the `oimInterpWl` class or link parameters with the `oimParamLinker` class.

```
b4 = oimBox(dx=oim.oimInterpWl([2e-6, 2.4e-6], [5, 10]), dy=2, x=20, y=0, f=0.5)
b4.params['dy'] = oim.oimParamLinker(b4.params['dx'], 'mult', 4)

m3 = oim.oimModel([b4])
m3.showModel(512, 0.2, wl=[2e-6, 2.2e-6, 2.4e-6], colorbar=False, swapAxes=True)
```



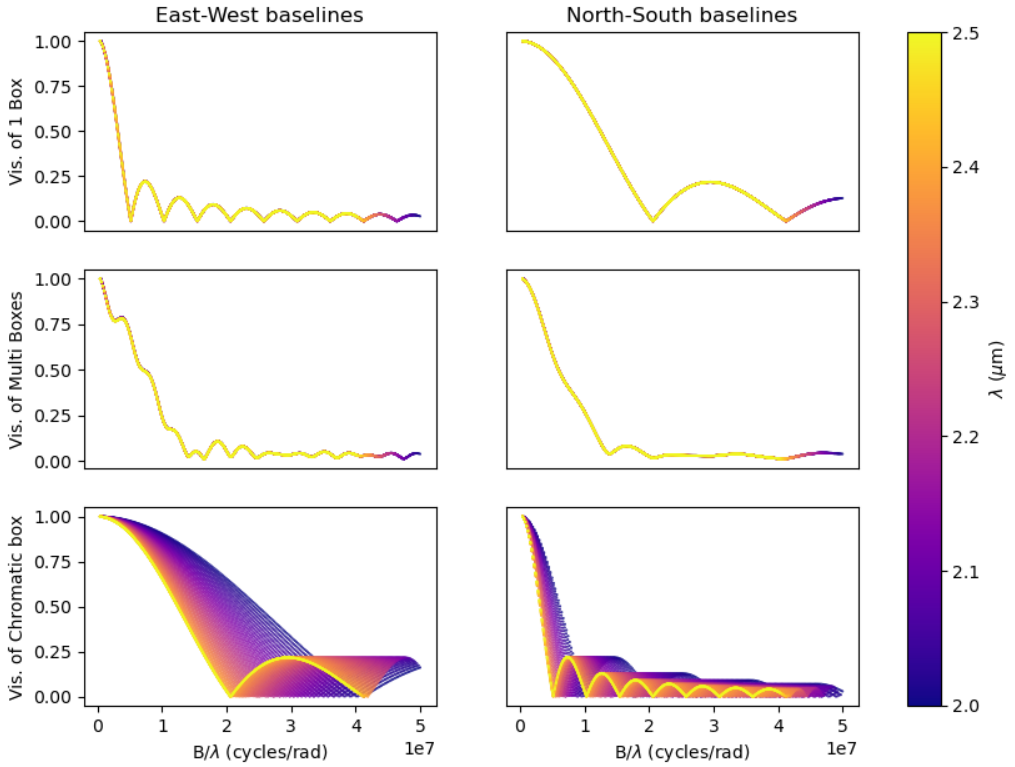
Let's finish this example by plotting the visibility of such models for a set of East-West and North-South baselines and wavelengths in the K-band.

```
nB = 200  # number of baselines
```

(continues on next page)

(continued from previous page)

```
nwl = 50  # number of walvengths

# Create some spatial frequencies
wl = np.linspace(2e-6, 2.5e-6, num=nwl)
B = np.linspace(1, 100, num=nB)
Bs = np.tile(B, (nwl, 1)).flatten()
wls = np.transpose(np.tile(wl, (nB, 1))).flatten()
spf = Bs/wls
spf0 = spf*0

fig, ax=plt.subplots(3, 2, figsize=(10, 7))

models=[m1, m2, m3]
names =["1 Box", "Multi Boxes","Chromatic box"]

for i, m in enumerate(models):
    visWest = np.abs(m.getComplexCoherentFlux(spf, spf0, wls)).reshape(nwl, nB)
    visWest /= np.outer(np.max(visWest, axis=1), np.ones(nB))
    visNorth = np.abs(m.getComplexCoherentFlux(
        spf0, spf, wls)).reshape(nwl, nB)
    visNorth /= np.outer(np.max(visNorth, axis=1), np.ones(nB))

    cb = ax[i, 0].scatter(spf, visWest, c=wls*1e6, s=0.2, cmap="plasma")
    ax[i, 1].scatter(spf, visNorth, c=wls*1e6, s=0.2, cmap="plasma")
    ax[i, 0].set_ylabel(f"Vis. of {names[i]}")

    if i != 2:
        ax[i, 0].get_xaxis().set_visible(False)
        ax[i, 1].get_xaxis().set_visible(False)

    ax[i, 1].get_yaxis().set_visible(False)

ax[2,0].set_xlabel("B/$\\lambda$ (cycles/rad)")
ax[2,1].set_xlabel("B/$\\lambda$ (cycles/rad)")
ax[0,0].set_title("East-West baselines")
ax[0,1].set_title("North-South baselines")
```



Of course, only the third model is chromatic.

Fast Rotator (External model)

In the createCustomComponentImageFastRotator.py example, we will create a new component derived from the `oimImageComponent`, using an external function that return a chromatic image cube.

The model is a simple implementation of a fast rotating star flattened by rotation (Roche Model) including gravity darkening ($T_{eff} \propto g_{eff}^{\beta}$). The emission is a simple blackbody.

First, let's import a few packages used in this example:

```
from pathlib import Path

import matplotlib.colors as colors
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
import oimodeler as oim
from astropy import units as units
```

Here is the code of the `fastRotator` external function that we want to encapsulate into a `oimComponent` to be used in `oimodeler`.

```
def fastRotator(dim0, size, incl, rot, Tpole, lam, beta=0.25):
    h = 6.63e-34
```

(continues on next page)

(continued from previous page)

```
    c = 3e8
    kb = 1.38e-23

    a = 2./3*(rot)**0.4+1e-9
    K = np.sin(1./3.)*np.pi

    K1 = h*c/kb
    nlam = np.size(lam)
    incl = np.deg2rad(incl)

    x0 = np.linspace(-size, size, num=dim0)
    idx = np.where(np.abs(x0) <= 1.5)
    x = np.take(x0, idx)
    dim = np.size(x)
    unit = np.ones(dim)
    x = np.outer(x, unit)
    x = np.einsum('ij, k->ijk', x, unit)

    y = np.swapaxes(x, 0, 1)
    z = np.swapaxes(x, 0, 2)

    yp = y*np.cos(incl)+z*np.sin(incl)
    zp = y*np.sin(incl)-z*np.cos(incl)

    r = np.sqrt(x**2+yp**2+zp**2)
    theta = np.arccos(zp/r)

    x0 = (1.5*a)**1.5*np.sin(1e-99)
    r0 = a*np.sin(1/3.)*np.arcsin(x0)/(1.0/3.*x0)

    x2 = (1.5*a)**1.5*np.sin(theta)
    rin = a*np.sin(1/3.)*np.arcsin(x2)/(1.0/3.*x2)

    rhoin = rin*np.sin(theta)/a/K
    dr = (rin/r0-r) >= 0
    Teff = Tpole*(np.abs(1-rhoin*a)**beta)

    if nlam == 1:
        flx = 1./(np.exp(K1/(lam*Teff))-1)

        im = np.zeros([dim, dim])

        for iz in range(dim):
            im = im*(im != 0)+(im == 0) * \
                dr[:, :, iz]*flx[:, :, iz]  # *limb[:,:,iz]

        im = np.rot90(im)

        tot = np.sum(im)
        im = im/tot
        im0 = np.zeros([dim0, dim0])
```

(continues on next page)

(continued from previous page)

```
        im0[dim0//2-dim//2:dim0//2+dim//2, dim0//2-dim//2:dim0//2+dim//2] = im
    else:
        unit = np.zeros(nlam)+1
        dr = np.einsum('ijk, l->ijkl', dr, unit)
        flx = 1./(np.exp(K1/np.einsum('ijk, l->ijkl', Teff, lam))-1)
        im = np.zeros([dim, dim, nlam])

        for iz in range(dim):
            im = im*(im != 0)+dr[:, :, iz, :]*flx[:, :, iz, :]*(im == 0)

        im = np.rot90(im)
        tot = np.sum(im, axis=(0, 1))
        for ilam in range(nlam):
            im[:, :, ilam] = im[:, :, ilam]/tot[ilam]

        im0 = np.zeros([dim0, dim0, nlam])
        im0[dim0//2-dim//2:dim0//2+dim//2, dim0//2-dim//2:dim0//2+dim//2, :] = im
        return im0
```

Now, we will define the new class for the fast rotator model. It will be derived from the `oimComponentImage` class as the model is defined in the image plane. We first write the `__init__` method of the new class. It needs to includes all the model parameters.

```
class oimFastRotator(oim.oimComponentImage):
    name = "Fast Rotator"
    shortname = "FRot"

    def __init__(self, **kwargs):
        super(). __init__(**kwargs)

        self.params["incl"] = oim.oimParam(
            name="incl", value=0, description="Inclination angle", unit=units.deg)
        self.params["rot"] = oim.oimParam(
            name="rot", value=0, description="Rotation Rate", unit=units.one)
        self.params["Tpole"] = oim.oimParam(
            name="Tpole", value=20000, description="Polar Temperature", unit=units.K)
        self.params["dpole"] = oim.oimParam(
            name="dplot", value=1, description="Polar diameter", unit=units.mas)
        self.params["beta"] = oim.oimParam(
            name="beta", value=0.25, description="Gravity Darkening Exponent",␣
↪unit=units.one)

        self._t = np.array([0])
        self._wl = np.linspace(0.5e-6, 15e-6, num=10)
        self._eval(**kwargs)
```

Note: Unlike for models defined in the Fourier plane, you need to define the internal wavelength `self._wl` and time `self._t` grids with their respective class attributes.

Here, we set the time to a fixed value so that the model will be time independent. The wavelength dependence of the model is set to a vector of 10 reference wavelengths between 0.5 and 15 microns. This will be used to compute

reference images and linear interpolation in wavelength will be used on the Fourier transforms of the images.

Together with the parameter *dim* (dimension of the image in x and y), the `self._wl` and the `self._t` set the length dimensions of the internal image hypercube (4-dimensional: *x*, *y*, *wl*, and *t*).

Now we can implement the call to the `fastRotator` function. As it is an external function that computes its own spatial and spectral grid we need to implement it in the `oimComponentImage._internalImage` method.

```
def _internalImage(self):
    dim = self.params["dim"].value
    incl = self.params["incl"].value
    rot = self.params["rot"].value
    Tpole = self.params["Tpole"].value
    dpole = self.params["dpole"].value
    beta = self.params["beta"].value

    im = fastRotator(dim, 1.5, incl, rot, Tpole, self._wl, beta=beta)
    im = np.tile(np.moveaxis(im, -1, 0)[None, :, :, :], (1, 1, 1, 1))
    self._pixSize = 1.5*dpole/dim*units.mas.to(units.rad)
    return im
```

Here we need to reshape the result of the `fastRotator` function to the proper shape for an internal image of the `oimImageComponent` class. The `FastRotator` returns a 3D image-cube (*x*, *y*, *wl*). We move its axis and reshape it to a 4D image-hypercube (*t*, *wl*, *x*, *y*).

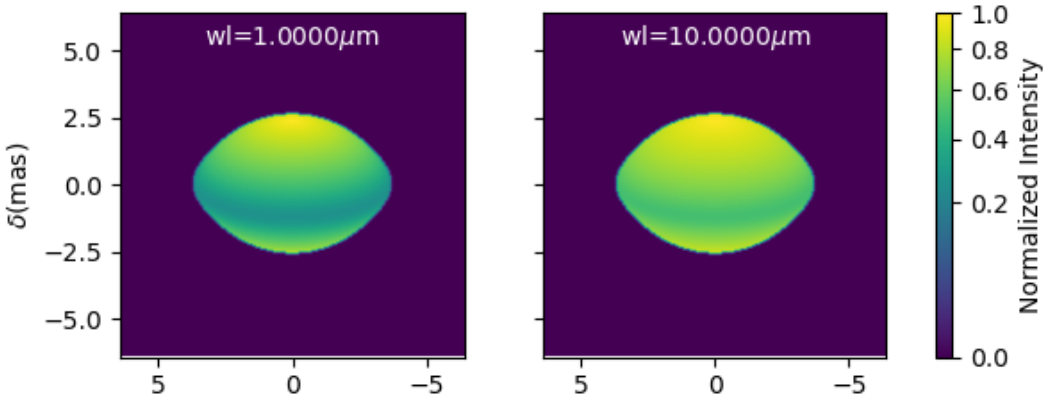
Finally, we need to set the pixel size (in rad) using the `self._pixSize` private attribute. For our example, we compute a `fastRotator` on a grid of 1.5 polar diameter (because the equatorial diameter goes up to 1.5 polar diameter for a critically rotating star). The pixel size formula depends on the *dpole* and *dim* parameters.

Let's build our first model with this brand new component.

```
c = oimFastRotator(dpole=5, dim=128, incl=-70, rot=0.99, Tpole=20000, beta=0.25)
m = oim.oimModel(c)
```

We can now plot the model images at various wavelengths as we do for any other `oimModel`.

```
m.showModel(512, 0.025, wl=[1e-6, 10e-6 ], legend=True, normalize=True)
```



Let's create a some spatial frequencies, with some chromaticity. For that we create baselines in the East-West and North-South orientations.

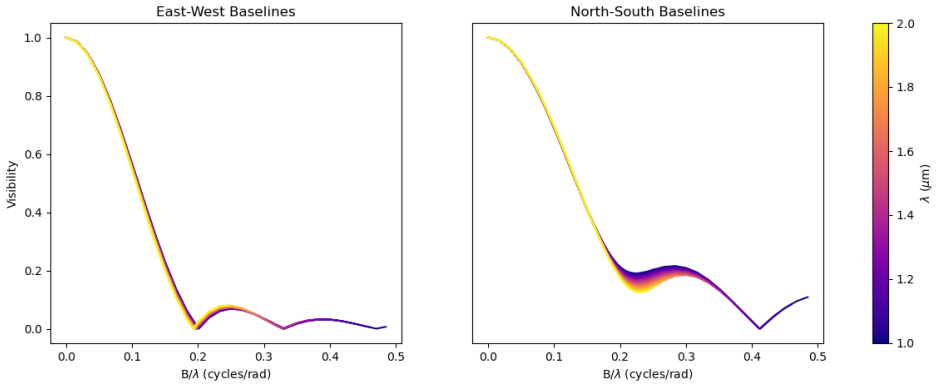
```
nB = 1000
nwl = 20
wl = np.linspace(1e-6, 2e-6, num=nwl)

B = np.linspace(0, 100, num=nB//2)

# 1st half of B array are baseline in the East-West orientation
Bx = np.append(B, B*0)
By = np.append(B*0, B)  # 2nd half are baseline in the North-South orientation

Bx_arr = np.tile(Bx[None, :], (nwl, 1)).flatten()
By_arr = np.tile(By[None, :], (nwl,  1)).flatten()
wl_arr = np.tile(wl[:, None], (1, nB)).flatten()

spfx_arr = Bx_arr/wl_arr
spfy_arr = By_arr/wl_arr
```

We now compute the complex coherent flux and then extract the visiblity from it. Note that the model is already normalized to one so that we don't need to divide the complex coherent flux by the zero frequency.

```
vc = m.getComplexCoherentFlux(spfx_arr, spfy_arr, wl_arr)
v = np.abs(vc.reshape(nwl, nB))
```

Finally, we plot the East-West and North-South visiblity with a colorscale for the wavelength.

```
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
titles = ["East-West Baselines", "North-South Baselines"]
for iwl in range(nwl):
    cwl = iwl/(nwl-1)
    ax[0].plot(B/wl[iwl]/units.rad.to(units.mas), v[iwl, :nB//2],
               color=plt.cm.plasma(cwl))
    ax[1].plot(B/wl[iwl]/units.rad.to(units.mas), v[iwl, nB//2:],
               color=plt.cm.plasma(cwl))

for i in range(2):
    ax[i].set_title(titles[i])
    ax[i].set_xlabel("B/$\lambda$ (cycles/rad)")
ax[0].set_ylabel("Visibility")
ax[1].get_yaxis().set_visible(False)

norm = colors.Normalize(vmin=np.min(wl)*1e6, vmax=np.max(wl)*1e6)
sm = cm.ScalarMappable(cmap=plt.cm.plasma, norm=norm)
fig.colorbar(sm, ax=ax, label="$\\lambda$ ($\\mu$m)")
```



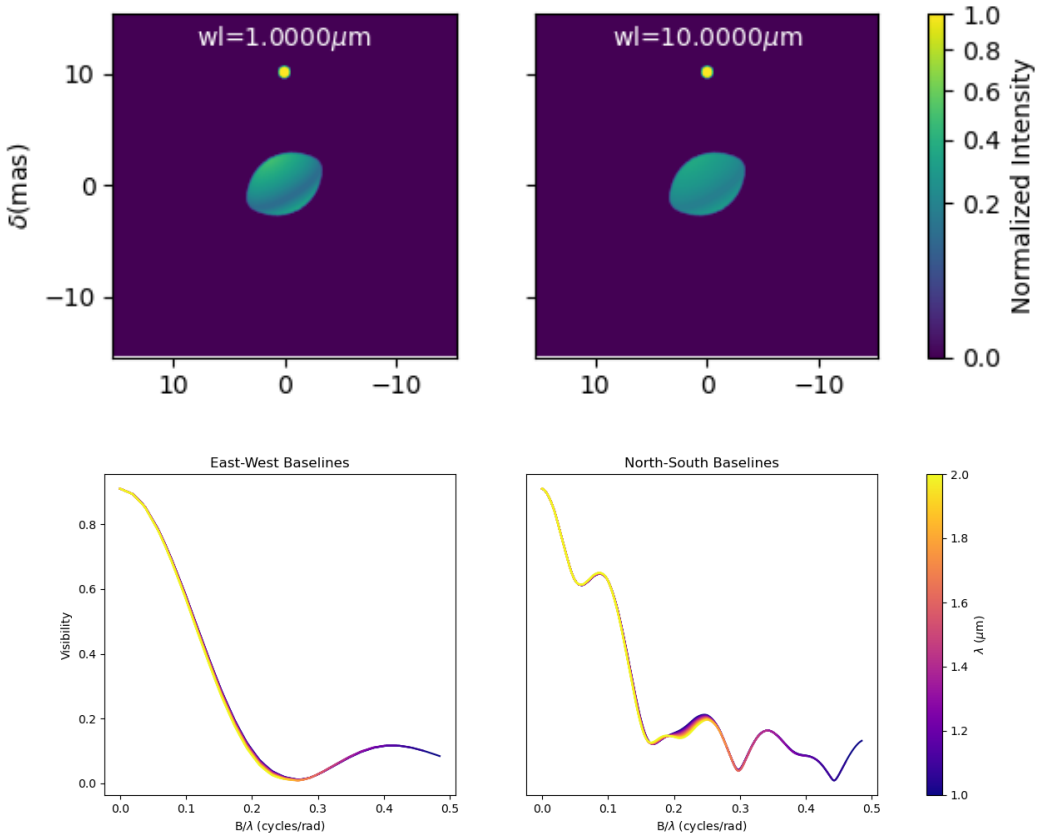
This new `oimfastRotator` component can be rotated and used together with other `oimComponent` classes to build more complex models.

Here, we add a uniform disk component `oimUD`:

```
c.params['f'].value = 0.9
c.params['pa'].value = 30
ud = oim.oimUD(d=1, f=0.1, y=10)
m2 = oim.oimModel(c, ud)
```

And finally, we produce the same plots as before for this new complex model.

```
m2.showModel(512, 0.06, wl=[1e-6, 10e-6], legend=True, normalize=True, normPow=0.5,
             savefig=save_dir / "customCompImageFastRotator2.png")
vc = m2.getComplexCoherentFlux(spfx_arr, spfy_arr, wl_arr)
v = np.abs(vc.reshape(nwl, nB))

fig, ax = plt.subplots(1, 2, figsize=(15, 5))
titles = ["East-West Baselines", "North-South Baselines"]
for iwl in range(nwl):
    cwl = iwl/(nwl-1)
    ax[0].plot(B/wl[iwl]/units.rad.to(units.mas), v[iwl, :nB//2],
               color=plt.cm.plasma(cwl))
    ax[1].plot(B/wl[iwl]/units.rad.to(units.mas), v[iwl, nB//2:],
               color=plt.cm.plasma(cwl))

for i in range(2):
    ax[i].set_title(titles[i])
    ax[i].set_xlabel("B/$\lambda$ (cycles/rad)")
ax[0].set_ylabel("Visibility")
ax[1].get_yaxis().set_visible(False)

norm = colors.Normalize(vmin=np.min(wl)*1e6, vmax=np.max(wl)*1e6)
sm = cm.ScalarMappable(cmap=plt.cm.plasma, norm=norm)
fig.colorbar(sm, ax=ax, label="$\\lambda$ ($\\mu$m)")
```



Spiral (Image plan formula)

In the createCustomComponentImageSpiral.py example we will create a new component derived from the `oimImageComponent` class, which describes a logarithmic spiral.

But first let's import a few packages used in this example:

```
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import oimodeler as oim
from astropy import units as units
```

Now we will define the new class for the spiral model. Again, it will be derived from the `oimComponentImage` class as the model is defined in the image plane. We first write the `__init__` method of the new class. It needs to includes all the model's parameters.

```
class oimSpiral(oim.oimComponentImage):
    name = "Spiral component"
    shorname = "Sp"
    elliptic = True

    def __init__(self, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(**kwargs)
        self.params["fwhm"] = oim.oimParam(**oim._standardParameters["fwhm"])
        self.params["P"] = oim.oimParam(name="P",
                                        value=1, description="Period in mas", unit=units.
↪mas)
        self.params["width"] = oim.oimParam(name="width",
                                            value=0.01, description="Width as filling␣
↪factor", unit=units.one)

        self._pixSize = 0.05*units.mas.to(units.rad)
        self._t = np.array([0])  # constant value <=> static model
        self._wl = np.array([0])  # constant value <=> achromatic model
        self._eval(**kwargs)
```

Here we chose to fix the pixel size in the `__init__` method. As we don't intend to have chromaticity, we fixed the internal time and wavelength arrays.

Unlike in the previous example, as we don't use an externally computed image, so we can implement the `oimComponentImage._imageFunction` of the class instead of the `oimComponentImage._internaImage` one.

The main difference is that the `oimComponentImage._imageFunction` directly provides the 4D-grid in time, wavelength and x and y.

```
def _imageFunction(self, xx, yy, wl, t):
    # As xx and yy are transformed coordinates, r and phi takes into account
    # the ellipticity and orientation using the pa and elong keywords
    r = np.sqrt(xx**2+yy**2)
    phi = np.arctan2(yy, xx)

    p = self.params["P"](wl, t)
    sig = self.params["fwhm"](wl, t)/2.35
    w = self.params["width"](wl, t)

    im = 1 + np.cos(-phi-2*np.pi*np.log(r/p+1))
    im = (im < 2*w)*np.exp(-r**2/(2*sig**2))
    return im
```

Note: As *xx* and *yy* are transformed coordinates, *r* and *phi* takes into account the ellipticity and orientation using the *pa* and *elong* keywords.

We create a model consisting of two components: The newly defined `oimSpiral` class and a uniform disk (`oimUD`).

```
ud = oim.oimUD(d=2, f=0.2)
c = oimSpiral(dim=256, fwhm=5, P=0.1, width=0.2, pa=30, elong=2, x=10, f=0.8)
m = oim.oimModel(c, ud)
```

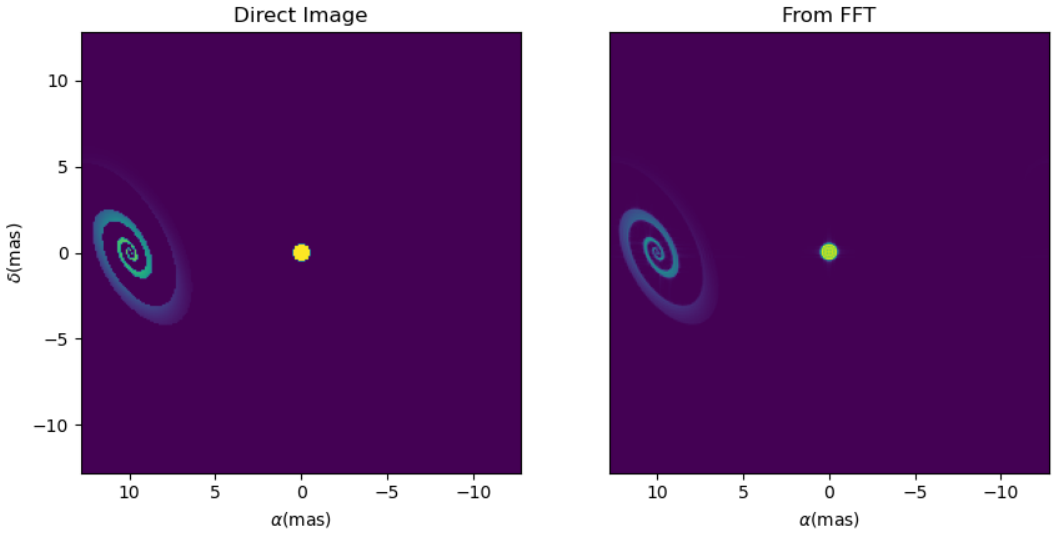
Then, we plot the image of the model (using the direct image formula and going back and forth to and from the Fourier plane).

```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
m.showModel(256, 0.1, swapAxes=True, fromFT=False,
            normPow=1, axe=ax[0], colorbar=False)
```

(continues on next page)

(continued from previous page)

```
m.showModel(256, 0.1, swapAxes=True, fromFT=True,
            normPow=1, axe=ax[1], colorbar=False)
ax[1].get_yaxis().set_visible(False)
ax[0].set_title("Direct Image")
ax[1].set_title("From FFT")
```



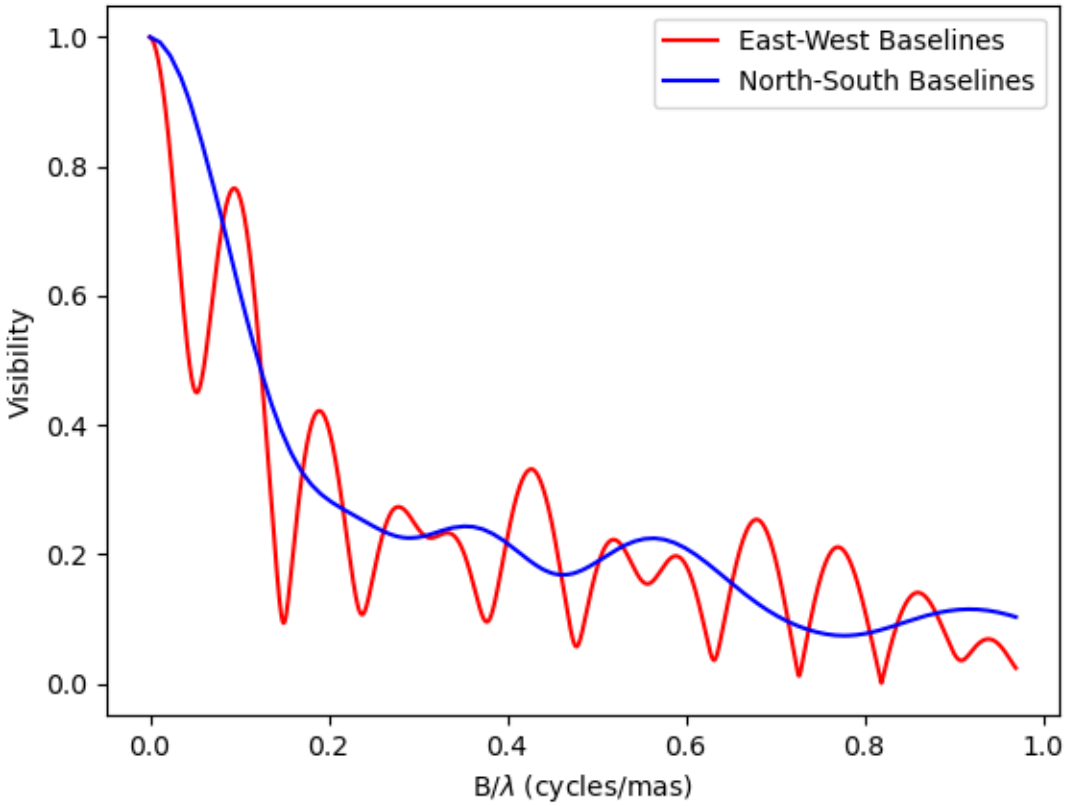
And finally, the visibility from the models for a fixed wavelength and a series of baselines in two perpendicular orientations.

```
nB = 5000
nwl = 1
wl = 0.5e-6

B = np.linspace(0, 100, num=nB//2)
Bx = np.append(B, B*0)
By = np.append(B*0, B)

spfx = Bx/wl
spfy = By/wl

vc = m.getComplexCoherentFlux(spfx, spfy)
v = np.abs(vc/vc[0])

fig, ax = plt.subplots(1, 1)
label = ["East-West Baselines",]

ax.plot(B/wl/units.rad.to(units.mas),
        v[:nB//2], color="r", label="East-West Baselines")
ax.plot(B/wl/units.rad.to(units.mas),
        v[nB//2:], color="b", label="North-South Baselines")

ax.set_xlabel("B/$\lambda$ (cycles/mas)")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("Visibility")
ax.legend()
```



Exp. Ring (Radial profile)

Note: Examples will be added when the `oimComponentRadialProfile` is implemented.

5.4.3.2 Creating New Interpolators

In the createCustomParamInterpolator.py example we will create a new parameter interpolator derived from the `oimParaminterpolator` class. The new class will allow chromatic interpolation with a vector of evenly spaced values in a range of wavelengths.

First we load some useful package and also set the random seed to a fixed value as we will use it to initalize our vector.

```
from pathlib import Path
from pprint import pprint

import matplotlib.colors as colors
```

(continues on next page)

(continued from previous page)

```
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
import oimodeler as oim
from scipy.interpolate import interp1d

np.random.seed(1)
```

As for the components, we derive our interpolator from a base class, this time `oimParamInterpolator`. We need to implement the, for this class unique `oimParamInterpolator._init` method that will be called by the `__init__` method of the base class. This method should contain information on the interpolator parameters.

```
class oimParamLinearRangeWl(oim.oimParamInterpolator):
    def _init(self, param, wl0=2e-6, dwl=1e-9, values=[], kind="linear", **kwargs):

        self.kind = kind

        n = len(values)
        self.wl0 = (oim.oimParam(**oim._standardParameters["wl"]))
        self.wl0.name = "wl0"
        self.wl0.description = "Initial wl of the range"
        self.wl0.value = wl0
        self.wl0.free = False

        self.dwl = (oim.oimParam(**oim._standardParameters["wl"]))
        self.dwl.name = "dwl"
        self.dwl.description = "wl step in range"
        self.dwl.value = dwl
        self.dwl.free = False

        self.values = []

        for i in range(n):
            self.values.append(oim.oimParam(name=param.name, value=values[i],
                                            mini=param.min, maxi=param.max,
                                            description=param.description,
                                            unit=param.unit, free=param.free,
                                            error=param.error))
```

The first argument of the class, `param` is the `oimParam` on which the new interpolator will be built.

The next arguments are the interpolator parameters, here :

- The initial wavelength of the range `wl0`
- The wavelength step in the range of interpolation : `dwl`
- The values at the reference wavelength : `values`
- The method for interpolation (from scipy interp1d) `kind`

The `**kwargs` is added for backward-compatibility.

The parameters `wl0`, `dwl` are created from the `_standardParameters["wl"]` dictionary (contained in the `oimParam` module) for the wavelength. Their name, descriptions, and value are updated, and they are set as fixed parameter by default (`free=False`).

The values vector of parameters is created from the input parameter `param`. For each parameter in the vector the value is set to the proper one given as input parameter.

The second method to implement is the `oimParamInterpolator._interpFunction` which is the core function of the interpolation. It has two input parameters: The wavelength *wl* and the time *t* for which the parameter shoud be interpolated. As our interoplator is not time dependent, we can ignore *t*.

```
def _interpFunction(self, wl, t):
    vals = np.array([vi.value for vi in self.values])
    nwl = vals.size
    wl0 = np.linspace(self.wl0.value, self.wl0.value +
                      self.dwl.value*nwl, num=nwl)
    return interp1d(wl0, vals, kind=self.kind, fill_value="extrapolate")(wl)
```

In this method we:

- Create a numpy array from the values of the `self.values` vector from the `oimParam` class.
- A second numpy array for the regular grid of walvengths from the `self.wl0` and `self.dwl` parameters.
- Interpolate the values at *wl* using the scipy interp1d function.
- Return the resulting interpolated values of the parameter.

For model-fitting purposes, we also need to tell `oimodeler` what are the parameters of our interpolator. This is done by implementing the `oimParamInterpolator._getParams` method. This method is called by a property `params` of the base class `oimParamInterpolator`.

```
def _getParams(self):
    params = []
    params.extend(self.values)
    params.append(self.wl0)
    params.append(self.dwl)
    return params
```

This method simply returns the list of the interpolator parameters. Here, the list of the reference values `self.values`, the initial wavelength `self.wl0` and the wavelength step `self.dwl`. We omit the `kind` parameter as we consider it more as an option than a real parameter.

Finally, if we want to use our interpolator using the `oimInterp` macro, we need to reference it in the `_interpolator` dictionary contained in the `oimParam` module.

```
oim._interpolator["rangeWl"] = oimParamLinearRangeWl
```

Now, we can use our new interpolator to build a component and a model. Let's build a chromatic uniform disk with 10 reference wavelengths between 2 and 2.5 microns. For the example, we will fill the `values` vector with random diameters from 4 to 7 mas.

```
nref = 10
c = oim.oimUD(d=oim.oimInterp('rangeWl', wl0=2e-6, kind="cubic",
                              dwl=5e-8, values=np.random.rand(nref)*3+4))
m = oim.oimModel(c)
```

We can print the parameters of our model:

```
pprint(m.getParameters())
```

```
... {'c1_UD_x': oimParam at 0x17829999e80 : x=0 ± 0 mas range=[-inf,inf] free=False ,
     'c1_UD_y': oimParam at 0x17829999fd0 : y=0 ± 0 mas range=[-inf,inf] free=False ,
     'c1_UD_f': oimParam at 0x17829999f40 : f=1 ± 0  range=[-inf,inf] free=True ,
     'c1_UD_d_interp1': oimParam at 0x178253c9250 : d=5.251066014107722 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp2': oimParam at 0x178253c9280 : d=6.160973480326474 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp3': oimParam at 0x178253c92b0 : d=4.000343124452034 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp4': oimParam at 0x178253c92e0 : d=4.9069977178955195 ± 0 mas␣
↪range=[-inf,inf] free=True ,
     'c1_UD_d_interp5': oimParam at 0x178253c9310 : d=4.4402676724513395 ± 0 mas␣
↪range=[-inf,inf] free=True ,
     'c1_UD_d_interp6': oimParam at 0x178253c9340 : d=4.277015784306394 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp7': oimParam at 0x178253c9370 : d=4.558780634133012 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp8': oimParam at 0x178253c93a0 : d=5.036682181129143 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp9': oimParam at 0x178253c93d0 : d=5.19030242269201 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp10': oimParam at 0x178253c9400 : d=5.616450202010071 ± 0 mas␣
↪range=[-inf,inf] free=True ,
     'c1_UD_d_interp11': oimParam at 0x178253c9220 : wl0=2e-06 ± 0 m range=[0,inf]␣
↪free=False ,
     'c1_UD_d_interp12': oimParam at 0x178253b5df0 : dwl=5e-08 ± 0 m range=[0,inf]␣
↪free=False }
```

The interpolator replaced the single `oimParam` for the diameter *c1_UD_d* by 12 `oimParam`: 10 for the reference values of the diameter (filled by random in our initialization), one for the initial wavelength `wl0` and another for the wavèlength step `dwl`.

We can also get the free parameters:

```
pprint(m.getFreeParameters())
```

```
... {'c1_UD_f': oimParam at 0x17829999f40 : f=1 ± 0  range=[-inf,inf] free=True ,
     'c1_UD_d_interp1': oimParam at 0x178253c9250 : d=5.251066014107722 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp2': oimParam at 0x178253c9280 : d=6.160973480326474 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp3': oimParam at 0x178253c92b0 : d=4.000343124452034 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp4': oimParam at 0x178253c92e0 : d=4.9069977178955195 ± 0 mas␣
↪range=[-inf,inf] free=True ,
     'c1_UD_d_interp5': oimParam at 0x178253c9310 : d=4.4402676724513395 ± 0 mas␣
↪range=[-inf,inf] free=True ,
     'c1_UD_d_interp6': oimParam at 0x178253c9340 : d=4.277015784306394 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp7': oimParam at 0x178253c9370 : d=4.558780634133012 ± 0 mas range=[-
↪inf,inf] free=True ,
     'c1_UD_d_interp8': oimParam at 0x178253c93a0 : d=5.036682181129143 ± 0 mas range=[-
↪inf,inf] free=True ,
```

(continues on next page)

(continued from previous page)

```
    'c1_UD_d_interp9': oimParam at 0x178253c93d0 : d=5.19030242269201 ± 0 mas range=[-
↪inf,inf] free=True ,
    'c1_UD_d_interp10': oimParam at 0x178253c9400 : d=5.616450202010071 ± 0 mas␣
↪range=[-inf,inf] free=True }
```

Here the `x` and `y` parameters are removed as they are fixed by default, as well as `wl0` and `dwl`.

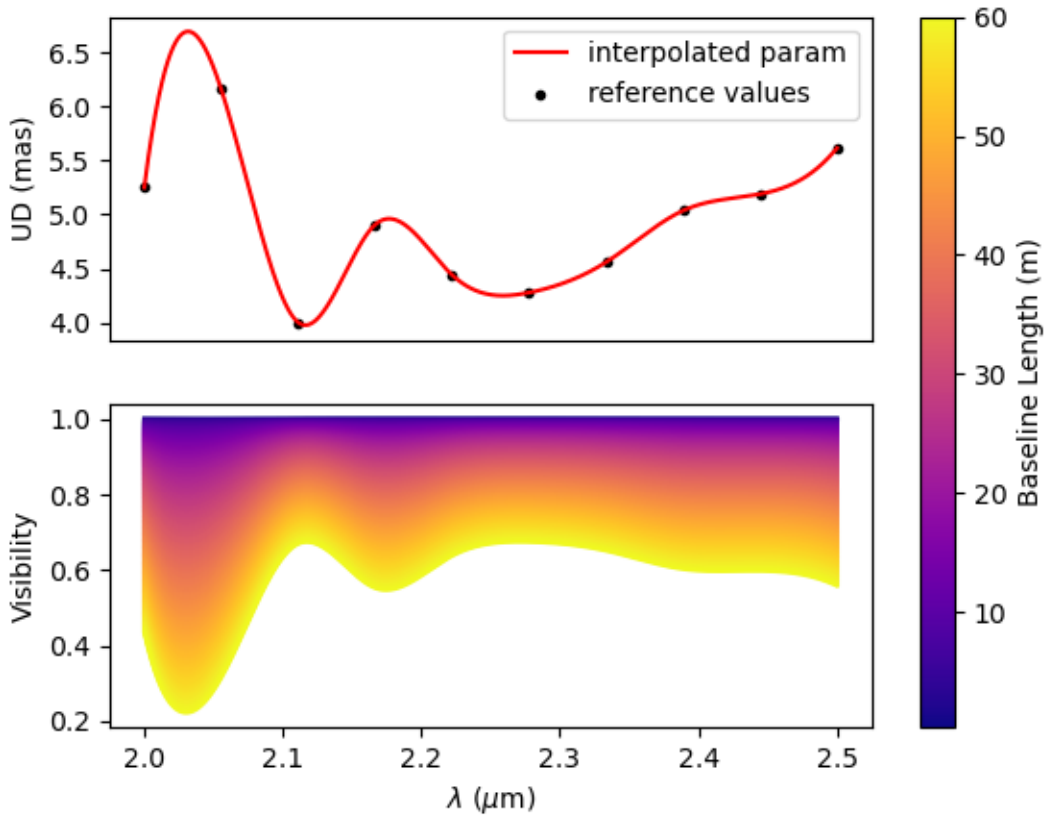
Let's plot the interpolated values of the parameters in the 2-2.5 micron range with 1000 values as well as the corresponding visibility for 200 East-West baselines ranging from 0 to 60m.

First, we create the wavelength vector and the spatial frequencies and wavelength arrays.

```
nB = 200
B = np.linspace(0, 60, num=nB)
nwl = 1000
wl = np.linspace(2.0e-6, 2.5e-6, num=nwl)
Bx_arr = np.tile(B[None, :], (nwl, 1)).flatten()
wl_arr = np.tile(wl[:, None], (1, nB)).flatten()
spfx_arr = Bx_arr/wl_arr
spfy_arr = spfx_arr*0
```

Finally, we compute the visibilty using the `oimModel.getComplexCoherentFlux` method and plot everything together.

```
v = np.abs(m.getComplexCoherentFlux(spfx_arr, spfy_arr, wl_arr).reshape(nwl, nB))

fig, ax = plt.subplots(2, 1)
ax[0].plot(wl*1e6, c.params['d'](wl, 0), color="r", label="interpolated param")
ax[0].scatter(wl0*1e6, vals, marker=".", color="k", label="reference values")
ax[0].set_ylabel("UD (mas)")
ax[0].get_xaxis().set_visible(False)
ax[0].legend()

for iB in range(1,nB):
    ax[1].plot(wl*1e6, v[:, iB]/v[:, 0], color=plt.cm.plasma(iB/(nB-1)))

ax[1].set_xlabel("$\lambda$ ($\mu$m)")
ax[1].set_ylabel("Visibility")

norm = colors.Normalize(vmin=np.min(B[1:]), vmax=np.max(B))
sm = cm.ScalarMappable(cmap=plt.cm.plasma, norm=norm)
fig.colorbar(sm, ax=ax, label="Baseline Length (m)")
```



5.4.4 Performance Tests

5.5 API

The oimodeler library contains the following modules:

Module	Description
oimParam	Model parameters and parameter interpolators
oimModel	Main class for model creation
oimComponent	Base classes for all model components
oimBasicFourierComponents	Basic components defined in the Fourier plan
oimCustomComponents	Custom components added by the community
oimFTBackends	Backends for Fast Fourier Transform
oimData	Main class for manipulating oifits data
oimFluxData	Photometric and spectrascopic data wrappers
oimDataFilter	Data filtering and modifying
oimSimulator	Data/Model simulation and comparison
oimFitter	Classes for Model fitting
oimPlots	Plotting functions and classes
oimUtils	Various utilities for optical interferometry
oimOptions	Global options of the oimodeler software

CHAPTER

SIX

INDEX

- genindex
- modindex