Pyimfit Documentation

Release 0.12.0

Peter Erwin

Mar 13, 2023

OVERVIEW AND SAMPLE USAGE:

CHAPTER 1

Installation of PyImfit

Note that since PyImfit is only meant to work with Python 3 (specifically, version 3.5 or later on Linux and version 3.6 or later on macOS), I reference `pip3` instead of just `pip` in the example installation commands below. If your version of `pip` automatically installs into Python 3, then you don't need to explicitly specify `pip3`.

1.1 Standard Installation: macOS

A precompiled binary version ("wheel") of PyImfit for macOS can be installed from PyPI via `pip`:

```
$ pip3 install pyimfit
```

PyImfit requires the following Python libraries/packages (which will automatically be installed by `pip` if they are not already present):

- Numpy
- Scipy

Astropy is also useful for reading in FITS files as numpy arrays (and is required by the unit tests).

1.2 Standard Installation: Linux

PyImfit can also be installed on Linux using `pip`. Since this involves building from source, you will need to have a working C++-11-compatible compiler (e.g., GCC version 4.8.1 or later); this is probably true for any reasonably modern Linux installation. (**Note:** a 64-bit Linux system is required.)

```
$ pip3 install pyimfit   [or "pip3 install --user pyimfit", if installing for your␣
↪own use]
```

If the installation fails with a message containing something like "fatal error: Python.h: No such file or directory", then you may be missing headers files and static libraries for Python development; see this Stackexchange question for guidance on how to do that.

CHAPTER 2

Sample Usage

The following assumes an interactive Python session (such as an iPython session or Jupyter notebook).

Read an image from a FITS file, read a model description from an Imfit-format text file, and then fit the model to the data:

```
from astropy.io import fits
import pyimfit


# 1. A simple fit to an image (no PSF or mask)

imageFile = "<path-to-FITS-file-directory>/ic3478rss_256.fits"
imfitConfigFile = "<path-to-config-file-directory>/config_exponential_ic3478_256.dat"

# read in image data
image_data = fits.getdata(imageFile)

# construct model (ModelDescription object) from config file
model_desc = pyimfit.ModelDescription.load(configFile)

# create an Imfit object, using the previously loaded model configuration
imfit_fitter = pyimfit.Imfit(model_desc)

# load the image data and image characteristics and do a standard fit
# (using default chi^2 statistics and Levenberg-Marquardt solver)
imfit_fitter.fit(image_data, gain=4.725, read_noise=4.3, original_sky=130.14)

# check the fit and print the resulting best-fit parameter values
if imfit_fitter.fitConverged is True:
    print("Fit converged: chi^2 = {0}, reduced chi^2 = {1}".format(imfit_fitter.
↪fitStatistic,
        imfit_fitter.reducedFitStatistic))
    print("Best-fit parameter values:")
    print(imfit_fitter.getRawParameters())
```

(continues on next page)

(continued from previous page)

```
# 2. Same basic model and data, but now with PSF convolution and a mask

# Load PSF image from FITS file, then create Imfit fitter with model + PSF
psfImageFile = "<path-to-FITS-file-directory>/psf_moffat_35.fits"
psf_image_data = fits.getdata(psfImageFile)

imfit_fitter2 = pyimfit.Imfit(model_desc, psf=psf_image_data)

# load the image data and characteristics, and also a mask image, but don't run the␣
↪fit yet
maskImageFile = "<path-to-FITS-file-directory>/mask.fits"
mask_image_data = fits.getdata(maskImageFile)

imfit_fitter2.loadData(image_data, mask=mask_image_data, gain=4.725, read_noise=4.3,␣
↪original_sky=130.14)

# do the fit, using Nelder-Mead simplex (instead of default Levenberg-Marquardt) as␣
↪the solver
imfit_fitter2.doFit(solver="NM")
```

You can also programmatically construct a model within Python (rather than having to read it from a text file):

```
# define a function for making a simple bulge+disk model, where both components
# share the same central coordinate (SimpleModelDescription class)
def galaxy_model(x0, y0, PA_bulge, ell_bulge, n, I_e, r_e, PA_disk, ell_disk, I_0, h):
    model = pyimfit.SimpleModelDescription()
    # define the limits on X0 and Y0 as +/-10 pixels relative to initial values
    model.x0.setValue(x0, [x0 - 10, x0 + 10])
    model.y0.setValue(y0, [y0 - 10, y0 + 10])

    bulge = pyimfit.make_imfit_function('Sersic', label='bulge')
    bulge.PA.setValue(PA_bulge, [0, 180])
    bulge.ell.setValue(ell_bulge, [0, 1])
    bulge.I_e.setValue(I_e, [0, 10*I_e])
    bulge.r_e.setValue(r_e, [0, 10*r_e])
    bulge.n.setValue(n, [0.5, 5])

    disk = pyimfit.make_imfit_function('Exponential', label='disk')
    disk.PA.setValue(PA_disk, [0, 180])
    disk.ell.setValue(ell_disk, [0, 1])
    disk.I_0.setValue(I_0, [0, 10*I_0])
    disk.h.setValue(h, [0, 10*h])

    model.addFunction(bulge)
    model.addFunction(disk)

    return model


model_desc = galaxy_model(x0=33, y0=33, PA_bulge=90.0, ell_bulge=0.2, n=4, I_e=1,
                        r_e=25, pa_disk=90.0, ell_disk=0.5, I_0=1, h=25)

imfit_fitter = pyimfit.Imfit(model_desc)

# etc.
```

Another way to construct the model is by defining it using a set of nested Python dicts, and passing the parent dict to the `ModelObject.dict_to_ModelDescription` function:

```python
# define a function for making a simple bulge+disk model, where both components
# share the same central coordinate; this version uses dicts internally

def galaxy_model(x0, y0, PA_bulge, ell_bulge, n, I_e, r_e, PA_disk, ell_disk, I_0, h):
    # dict describing the bulge (first define the parameter dict, with initial values
    # and lower & upper limits for each parameter)
    p_bulge = {'PA': [PA_bulge, 0, 180], 'ell_bulge': [ell, 0, 1], 'n': [n, 0.5, 5],
                'I_e': [I_e, 0.0, 10*I_e], 'r_e': [r_e, 0.0, 10*r_e]}
    bulge_dict = {'name': "Sersic", 'label': "bulge", 'parameters': p_bulge}
    # do the same thing for the disk component
    p_disk = {'PA': [PA_disk, 0, 180], 'ell_disk': [ell, 0, 1], 'I_0': [I_0, 0, 10*I_
↪0],
                'h': [h, 0.0, 10*h]}
    disk_dict = {'name': "Exponential", 'label': "disk", 'parameters': p_disk}

    # make dict for the function set that combines the bulge and disk components
    # with a single shared center, and then a dict for the whole model
    funcset_dict = {'X0': [x0, x0 - 10, x0 + 10], 'Y0': [y0, y0 - 10, y0 + 10],
                    'function_list': [bulge_dict, disk_dict]}
    model_dict = {'function_sets': [funcset_dict]}

    model = pyimfit.ModelDescription.dict_to_ModelDescription(model_dict)
    return model


model_desc = galaxy_model(x0=33, y0=33, PA_bulge=90.0, ell_bulge=0.2, n=4, I_e=1,
                        r_e=25, pa_disk=90.0, ell_disk=0.5, I_0=1, h=25)

imfit_fitter = pyimfit.Imfit(model_desc)

# etc.
```

You can get a list of PyImfit's image functions ("Sersic", "Exponential", etc.) from the package-level variable `pyimfit.imageFunctionList`, and you can get a list of the parameter names for each image function from `pyimfit.imageFunctionDict`. Full descriptions of the individual image functions and their parameters can be found in the Imfit manual (PDF)

CHAPTER 3

Overview of PyImfit

PyImfit is a Python wrapper around the (C++-based) image-fitting program Imfit (Github site).

Terminology:

Imfit (boldface) refers to the C++ program (and the C++-based library that PyImfit is built upon).

PyImfit is the general name for this Python package; `pyimfit` is the official Python name (e.g., `import pyimfit`).

Finally, `Imfit` refers to the `pyimfit.Imfit` class, which does most of the work.

3.1 For Those Already Familiar with Imfit

If you've already used the command-line version of **Imfit**, here are the essential things to know:

- PyImfit operates on 2D NumPy arrays instead of FITS files; to use a FITS file, read it into Python via, e.g., astropy.io.fits.
- Models (and initial parameter values and parameter limits for a fit) are specified via the ModelDescription class. The utility function `parse_config_file` will read a standard **Imfit** configuration file and return an instance of that class with the model specification. You can also build up a `ModelDescription` instance by programmatically specifying components from within Python, or via a dict-based description.
- Fitting is done by instantiating an `Imfit` object with a `ModelDescription` object as input, then adding a 2D NumPy array as the data to be fit (along with, optionally, mask and error images, the image A/D gain value, etc.) with the `loadData` method, and then calling the `doFit` method (along with the minimization algorithm to use). Or just call the `fit` method and supply the data image, etc., as part of its input.
- Once the fit is finished, information about the fit (final $\chi 2$ value, best-fit paremeter values, etc.) and the best-fitting model image can be obtained by querying properties and methods of the `Imfit` object.

See Sample Usage for a simple example of how this works.

3.2 The Basics

There are three basic things you can do with PyImfit:

1. Generate model images
2. Fit models to a pre-existing (data) image
3. Generate $\chi 2$ or likelihood values from a comparison of model and data (e.g., for use with other fitting software, MCMC analysis, etc.)

In **Imfit** (and PyImfit), a "model" consists of one or more *image functions* from a library built into **Imfit**, sharing one or more common locations within an image and added together to form a summed model image. Each image function can generate a 2D image; the final model image is the sum of its component image functions. Optionally, the summed model image can then be convolved with a user-supplied Point-Spread Function (PSF) image to simulate the effects of seeing and telescope optics. (For greater accuracy, subsections of the image can be oversampled on a finer pixel scale and convolved with a correspondingly oversampled PSF image or images; these subsection are then downsampled back to the native image pixel scale.)

3.2.1 Specify the model (and its parameters)

The model is specified by an instance of the `ModelDescription` class.

For the command-line program, this is done via a "configuration" text file, which has a specific format described in the Imfit manual (PDF), or in this page of the online docs.

If you have a configuration file, you can load it via the convenience function `parse_config_file`

```
model_desc = pyimfit.parse_config_file(configFilePath)
```

where `configFilePath` is a string specifying the path to the configuration file.

You can also construct a `ModelDescription` instance programmatically from within Python; see below for a very simple example, or Sample Usage for a slightly more complicated example. Finally, you can create a `ModelDescription` instance by calling the class function `ModelDescription.dict_to_ModelDescription` with a dict-based description of the model; see below for an example.

(You can get a list of the available image functions – "Sersic", "Exponential", etc. – from the package-level variable `pyimfit.imageFunctionList`, and you can get a list of the parameter names for each function from `pyimfit.imageFunctionDict`. These functions are described in detail in the Imfit manual (PDF).)

Once you have a `ModelDescription` object describing the model, you can create an instance of the `Imfit` class based on the model; optionally, if you want the model to be convolved with a PSF, you can also supply the PSF image (in the form of a 2D NumPy array):

```
imfitter = pyimfit.Imfit(model_desc)
imfitter = pyimfit.Imfit(model_desc, psfImage)
```

Creating a model; setting parameter values and limits

Each image-function parameter within a model can have a "current" value (e.g., the initial guess for the fitting process, the result from the fit, etc.) and either: a set of lower and upper limits for possible values **or** the string "fixed", which means the parameter value should be kept constant during fits.

Note: Unless otherwise specified, all size values are in pixels, and all intensity/surface-brightness values are in counts/pixel. (Photometric zero points are not needed except for the optional case of computing model magnitudes; see below.)

A very simple example of programmatically constructing a model:

```
model_desc = pyimfit.SimpleModelDescription()
# define the limits on the central-coordinate X0 and Y0 as +/-10 pixels relative to␣
↪initial values
# (note that Imfit treats image coordinates using the IRAF/Fortran numbering scheme:␣
↪the lower-left
# pixel in the image has coordinates (x,y) = (1,1))
model_desc.x0.setValue(105, [95,115])
model_desc.y0.setValue(62, [52,72])

# create an Exponential image function, then define the parameter initial values and␣
↪limits
disk = pyimfit.make_imfit_function("Exponential", label="disk")
# set initial values, lower and upper limits for central surface brightness I_0,␣
↪scale length h;
# specify that ellipticity is to remain fixed
disk.I_0.setValue(100.0, [0.0, 500.0])
disk.h.setValue(25, [10,50])
disk.PA.setValue(40, [0, 180])
disk.ell.setValue(0.5, fixed=True)

model_desc.addFunction(disk)

print(model_desc)
X0      105.0       95.0,115.0
Y0      62.0        52.0,72.0
FUNCTION Exponential    # LABEL disk
PA      40.0        0.0,100.0
ell     0.5     fixed
I_0     100.0       0.0,500.0
h       25.0        10.0,50.0
```

Constructing the same model using Python dicts:

```
# for each function, set up a dict mapping parameter names to lists of values and␣
↪(optional) limits;
# (e.g., the 'PA' parameter for the Exponential function has an initial value of 40␣
↪and lower and upper
# limits of 0 and 100, while the 'ell' parameter has an initial value of 0.5 and will␣
↪be held fixed
# during the fit);
# then make a dict for that function
exponentialParamsDict = {'PA': [40, 0,100], 'ell': [0.5, "fixed"], 'I_0': [100.0, 0.0,
↪500.0], 'h': [25, 10,50]}
exponentialDict = {'name': "Exponential", 'label': "disk", 'parameters':␣
↪exponentialParamsDict}

# make one or more function-set dicts
functionSetDict = {'X0': [105, 95,115], 'Y0': [62, 52,72], 'function_list':␣
↪[exponentialDict]}

# finally, make the dict describing the model and instantiate a ModelDescription␣
↪object from it
modelDict = {'function_sets': [functionSetDict]}
model_desc = pyimfit.ModelDescription.dict_to_ModelDescription(modelDict)

print(model_desc)
```

(continues on next page)

(continued from previous page)

```
X0      105.0        95.0,115.0
Y0      62.0         52.0,72.0
FUNCTION Exponential    # LABEL disk
PA      40.0         0.0,100.0
ell     0.5     fixed
I_0     100.0        0.0,500.0
h       25.0         10.0,50.0
```

3.2.2 Fit a model to the data

Specify the data (and optional mask) image

The data image must be a 2D NumPy array (internally, it will be converted to double-precision floating point with native byte order, if it isn't already).

You pass in the data image to the previously generated `Imfit` object (`imfitter`) using the latter's `loadData` method:

```
imfitter.loadData(data_im)
```

You can also specify a mask image, which should be a NumPy integer or float array where values = 0 indicate *good* pixels, and values > 0 indicate bad pixels that should not be used in the fit. Alternatively, if the data array is a NumPy MaskedArray, then *its* mask will be used. (If the data array is a MaskedArray *and* you supply a separate mask image, then the final mask will be the composition of the data array's mask and the mask image.)

```
imfitter.loadData(data_im, mask=mask_im)
```

Image-description parameters, statistical models and fit statistics

When calling the `loadData` method, you can tell the `Imfit` object about the statistical model you want to use: what the assumed uncertainties are for the data values, and what "fit statistic" is to be minimized during the fitting process.

- $\chi 2$ with data-based errors (default): the default is a standard $\chi 2$ approach using per-pixel Gaussian errors, with the assumption that the errors (sigma values) can be approximated by the square root of the data values.
- $\chi 2$ with model-based errors: Alternately, you can specify *model-based* errors, where the sigma values are the square root of the *model* values (these are automatically recomputed for every iteration of the fitting process).
- $\chi 2$ with user-supplied errors: You can also supply a noise/error array which is the same size as the data array and holds per-pixel sigma or variance values precomputed in some fashion (e.g., from an image-reduction pipeline).
- Poisson-based ("Poisson Maximum-Likelihood-Ratio" = "PMLR"): Finally, you can specify that individual pixel errors come from the model assuming a true Poisson process (rather than the Gaussian approximation to Poisson statistics that's used in the $\chi 2$ approaches). This is particularly appropriate when individual pixel values of the data are low.

You can also tell the `Imfit` object useful things about the data values: what A/D gain conversion was applied, any Gaussian read noise, any constant background value that was previously subtracted from the data image, etc. (You do not need to do this if you are supplying your own noise/errror array.)

Whatever you chose, you can specify this as part of the call to `loadData`, e.g.

```
# default chi^2, assuming an A/D gain of 4.5 e-/ADU and Gaussian read noise with␣
↪sigma^2 = 0.7 e-
imfitter.loadData(data_im, gain=4.5, read_noise=0.7)

# chi^2 with model-based errors
imfitter.loadData(data_im, gain=4.5, read_noise=0.7, use_model_for_errors=True)

# chi^2 with a NumPy variance array `variances_im` (gain and read noise are not␣
↪needed)
imfitter.loadData(data_im, error=variances_im, error_type="variance")

# Poisson Maximum-Likelihood-Ratio statistics (read noise is not used in this mode)
imfitter.loadData(data_im, gain=4.5, use_poisson_mlr=True)
```

Performing the Fit

To actually perform the fit, you call the `doFit` method on the `Imfit` object. You can specify which of the three different minimization algorithms you want to use with the `solver` keyword; the default is "LM" for the Levenberg-Marquardt minimizer.

- "LM" = Levenberg-Marquardt (the default): this is a fast, gradient-descent-based minimizer.
- "NM" = Nelder-Mead Simplex: slower, possibly less likely to be trapped in local minimum of the fit landscape.
- "DE" = Differential Evolution: genetic-algorithm-based; very slow; probably least likely to be trapped in local minima. (This method ignores the initial parameter guesses, instead choosing random values selected from within the lower and upper parameter bounds.)

E.g.,

```
# default Levenberg-Marquardt fit
result = imfitter.doFit()

# fit using Nelder-Mead simplex
result = imfitter.doFit(solver='NM')
```

Feedback from the fit: By default, the `Imfit` object is silent during the fitting process. If you want to see feedback, you can set the `verbose` keyword of the `doFit()` method: `verbose=1` will print out periodic updates of the current fit statistic (e.g., $\chi2$; `verbose=2` will also print the current best-fit parameter values of the model each time it prints the current fit statistic.

WARNING: Currently, there is no way to interrupt a fit once it has started! (Other than killing the underlying Python process, that is. This may change in the future.)

Shortcut: Load data and do the fit in one step

A shortcut is to call the `fit` method on the `Imfit` object. This lets you supply the data image (along with the optional mask), specify the statistical model ($\chi2$, etc.) and (optionally) the minimization algorithm and verbosity, and start the fit all in one go

```
result = imfitter.fit(data_im, gain=4.5, use_poisson_mlr=True, solver="NM", verbose=1)
```

Inspecting the results of a fit

The Imfit object returns an instance of the `FitResult` class, which is closely based on the `OptimizeResult` class of `scipy.optimize` and is basically a Python dict with attribute access

There are three or four basic things you might want to look at in the `FitResult` object when the fit finishes. You can get these things from the `FitResult` object that's returned from the `doFit()` method, or by querying the Imfit object; the examples below show each possibility.

1. See if the fit actually converged (either `True` or `False`):

```
result.fitConverged
imfitter.fitConverged
```

2. See the value of the final fit statistic, and related values:

```
result.fitStat   # final chi^2 or PMLR value
result.reducedFitStat   # reduced version of same
result.aic   # corresponding Akaike Information Criterion value
result.bic   # corresponding Bayesian Information Criterion value

imfitter.fitStatistic
imfitter.reducedFitStatistic
imfitter.AIC
imfitter.BIC
```

3.A. Get the best-fit parameter values in the form of a 1D NumPy array:

```
bestfit_parameters = result.params
bestfit_parameters = imfitter.getRawParameters()
```

3.B. Get the 1-sigma uncertainties on the best-fit parameter values in the form of a 1D NumPy array. Note that these are only produced if the default Levenberg-Marquardt solver was used, and are fairly crude estimates that should be used with caution. A somewhat better approach might be to do bootstrap resampling, or even use a Markov Chain Monte Carlo code such as "emcee".

```
bestfit_parameters_errs = results.paramErrs
bestfit_parameters_errs = imfit_fitter.getParameterErrors()
```

Other things you might be interested in:

1. Get the best-fitting model image (a 2D NumPy array)

```
bestfit_model_im = imfitter.getModelImage()
```

2. Get fluxes and magnitudes for the best-fitting model – note that what is returned is a tuple of the total flux/magnitude and a NumPy array of the fluxes/magnitudes for the individual components of the model (in the order they are listed in the model):

```
# get the total flux (counts or whatever the pixel values are) and the
# individual-component fluxes
(totalFlux, componentFluxes) = imfitter.getModelFluxes()

# get total and individual-component magnitudes, if you know the zero point
# for your image (25.72 in this example)
(totalMag, componentMagnitudes) = imfitter.getModelMagnitudes(zeroPoint=25.72)
```

Of course, you might also want to inspect the residuals of the fit; since your data image and the output best-fit model image are both NumPy arrays, this is simple enough:

```
residual_im = data_im - bestfit_model_im
```

Getting the model description

There are two ways to get a copy of the current model description (which will include the current best-fit parameter values if a successful fit was performed, though it will *not* include parameter error estimates). The first returns a `ModelDescription` object; the second returns a dict containing information about the model (which may be simpler to inspect). The dict format can then be used with `pyimfit.ModelDescription.dict_to_ModelDescription()` to generate a new ModelObject instance.

```
model_desc = imfitter.getModelDescription()

model_dict = imfitter.getModelAsDict()
```

3.2.3 Generate a model image (without fitting)

Sometimes you may want to generate model images without fitting any data. In this case, you can call the `getModelImage` method on the `Imfit` object without running the fit.

```
model_im = imfitter.getModelImage(shape=image_shape)
```

where `image_shape` is a 2-element integer tuple defining the image shape in the usual NumPy fashion (i.e., an image with n_rows and n_colums has shape=(n_columns,n_rows)).

If the `Imfit` object (`imfitter`) already has a data image assigned to it, then the output image will have the same dimensions as the data image, and you do not need to specify the shape.

Note that by default this will generate a model image using the current parameter values of the model (the initial values, if no fit has been done, or the best-fit values if a fit *has* been done). You can specify that a *different* set of parameter values (in the form of a 1-D NumPy array of the correct length) should be used to compute the model via the `newParameters` keyword:

```
model_im = imfitter.getModelImage(newParameters=parameter_array)
```

CHAPTER 4

Defining Models

PyImfit uses models which are instances of the ModelDescription class (or subclasses thereof).

A "model" is defined as a collection of "image functions", grouped into one or more "function sets". Each function set (a.k.a. "function block") is a collection of one or more image functions with the same central coordinates (X0,Y0) within the image. (The SimpleModelDescription class is a subclass which holds just one function set.)

A ModelDescription object can be instantiated using a pre-existing Imfit configuration file; it can also be constructed programmatically within Python.

4.1 Image Functions

A list of the available image functions can be found in the module-level variable `pyimfit.imageFunctionList`, or by calling the function `pyimfit.get_function_list()`, and a dict containing lists of the parameter names for individual image functions can be found in `pyimfit.imageFunctionDict` (this dict can also be obtained by calling the function `pyimfit.get_function_dict()`). E.g.,

```
In [1]: pyimfit.imageFunctionDict['Exponential']
Out[1]: ['PA', 'ell', 'I_0', 'h']
```

Detailed descriptions of the individual image functions can be found in Chapter 6 of the Imfit manual (PDF), and background information on most can be found in Section 6 of Erwin (2015). (Note that the latter reference won't include the more recent functions.)

The following is a brief list of the available image functions; see the Imfit manual for more details.

- 2D image functions: Most of these have a position-angle parameter ("PA") which defines their orientation on the image (measured in degrees counter-clockwise from the +x image axis). Many also have an ellipticity parameter ("ell") defining their shape. The most common type of 2D image function has elliptical isophotes with a particular radial surface-brightness profile (e.g., BrokenExponential, Core-Sersic, Exponential, etc.).
  - **BrokenExponential** – Elliptical isophotes with a radial surface-brightness profile following a broken-exponential function. Geometric parameters: PA, ell

- **BrokenExponential2D** – Isophotes for a perfectly edge-on disk, similar to “EdgeOnDisk” (below) but with a radial broken-exponential profile. Geometric parameters: PA
- **Core-Sersic** – Elliptical isophotes with a Core-Sérsic (REFS) radial surface-brightness profile. Geometric parameters: PA, ell
- **EdgeOnDisk** – The analytic form of an edge-on exponential disk (van der Kruit & Searle 1981), using the Bessel-function solution of van der Kruit & Searle (1981) for the radial profile and the generalized sech function of van der Kruit (1988) for the vertical profile. Geometric parameters: PA
- **EdgeOnRing** – A simplistic model for an edge-on ring, using an off-center Gaussian for the radial profile and another Gaussian (with different sigma) for the vertical profile. Geometric parameters: PA
- **EdgeOnRing2Side** – As for “EdgeOnRing”, but with the radial profile similar to that of “GaussianRing2Side” (asymmetric Gaussian). Geometric parameters: PA
- **Exponential** – Elliptical isophotes with a radial surface-brightness profile following an exponential function. Geometric parameters: PA, ell
- **Exponential_GenEllipse** – As for the “Exponential” function, but with isophotes having generalized ellipse shapes (boxy to disky). Geometric parameters: PA, ell, c0 (boxy/disk isophote-shape parameter)
- **FerrersBar2D** – Isophotes (generalized elliptical shapes) for a 2D version of the Ferrers ellipsoid. Geometric parameters: PA, c0 (boxy/disk isophote-shape parameter)
- **FlatBar** – An elongated structure with a broken-exponential profile along its major axis, suitable for the outer parts of (some) bars in massive disk galaxies; see Erwin et al. (2021) for more details and examples of use. Geometric parameters: PA, ell, deltaPA_max
- **FlatSky** – Produces a constant background for the entire image.
- **Gaussian** – Elliptical isophotes with a radial surface-brightness profile following a Gaussian function. Geometric parameters: PA, ell
- **GaussianRing** – An elliptical ring with a radial Gaussian profile (peaking at the user-specified semi-major axis). Geometric parameters: PA, ell
- **GaussianRing2Side** – As for “GaussianRing”, except that the ring profile is an asymmetric Gaussian, with different widths on the inner and outer sides. Geometric parameters: PA, ell
- **GaussianRingAz** – As for “GaussianRing”, except that the surface brightness in the ring varies as a function of azimuth. Geometric parameters: PA, ell
- **ModifiedKing** – Elliptical isophotes with a radial surface-brightness profile following the “modified King” function (Elson 1999; Peng et al. 2010), which is a generalization of the original King (1962) profile. Geometric parameters: PA, ell
- **ModifiedKing2** – Identical to “ModifiedKing”, except that the tidal/truncation radius parameter is replaced by a concentration parameter. Geometric parameters: PA, ell
- **Moffat** – Elliptical isophotes with a radial surface-brightness profile following the Moffat profile. Geometric parameters: PA, ell
- **PointSource** – This produces an interpolated, scaled copy of the user-suppled PSF image.
- **Sersic** – Elliptical isophotes with a radial surface-brightness profile following the Sérsic function. Geometric parameters: PA, ell
- **Sersic_GenEllipse** – As for the “Sersic” function, but with isophotes having generalized ellipse shapes (boxy to disky). Geometric parameters: PA, ell, c0 (boxy/disk isophote-shape parameter)
- **TiltedSkyPlane** – Produces a background for the entire image in the form of ain inclined plane.

- 3D image functions (luminosity-density functions): These generate a 2D image via line-of-sight integration through a 3D luminosity-density model, seen at arbitrary inclination.
  - **ExponentialDisk3D** – A disk model where the luminosity density follows a radial exponential profile and a vertical generalized sech (van der Kruit 1988) profile.
  - **BrokenExponentialDisk3D** – As for "ExponentialDisk3D", but with the radial profile specified by a broken-exponential function.
  - **FerrersBar3D** – The classic Ferrers (1877) triaxial ellipsoid.
  - **GaussianRing3D** – A 3D ring, where the luminosity density follows a radial Gaussian profile and a vertical exponential profile.

4.2 More Information

See Chapters 5 and 6 of the Imfit manual (PDF)

CHAPTER 5

PSF Convolution

As part of the modeling process, PyImfit can convolve the generated model image with a Point-Spread Function (PSF) image to simulate the effects of telescope optics, atmospheric seeing, etc.

The simplest approach is to use a single PSF image with the same intrinsic resolution (i.e., with pixels that have the same intrinsic size as the data and model images) to convolve the entire image.

However, it is also possible to convolve one or more subsections of the image with higher-resolution (oversampled) PSF images. In this case, the specified subsections are first computed as higher-resolution model images, convolved with the higher-resolution PSF image, and then downsampled to the data image resolution and copied into the appropriate location in the final model image.

5.1 Requirements for PSF images

PSF images should be square, 2D NumPy floating-point arrays, ideally with the center of the PSF kernel in the center of the image – so square images with an odd number of pixels are the best approach. They can come from any source: FITS images of stars, FITS images from telescope modeling software, NumPy arrays generated in Python, etc.

5.2 Basic PSF Convolution

To make use of PSF convolution in a model, the PSF image should be supplied when you instantiate the Imfit object that will be used for fitting. E.g.,

```
imfit_fitter = pyimfit.Imfit(model_description, psf=psf_image)
```

By default, the PSF image will be automatically normalized so that the sum of its pixel values = 1, so you do not need to normalize it first. If you *don't* want the PSF image normalized (as is the case for, e.g., some interferomteric PSFs), then set the `psfNormalization` keyword to False:

```
imfit_fitter = pyimfit.Imfit(model_description, psf=psf_image, psfNormalization=False)
```

5.3 Convolving with Oversampled PSFs

PyImfit allows you to designate one or more subsections of an image to be modeled and convolved with a PSF in an "oversampled" mode, using a PSF image that is oversampled with respect to the data image.

Thus, if you specify a 10x10-pixel subsection of the image and supply a PSF image that is oversampled by a factor of 5, that part of the model will be computed in a 50x50-pixel grid (plus appropriate padding around the edges), convolved with the oversampled PSF image, and finally 5x5-downsampled and copied into the specified 10x10-pixel subsection of the model image.

To specify an oversampling region, you create a PsfOversampling object; this is easiest to do with the `MakePsfOversampler()` function. You then place the PsfOversampling object(s) into a list and add the list to the Imfit object as part of the call to the `loadData` method.

For example: assuming that `oversampledPsf_image` is a NumPy array for a PSF that is oversampled by a factor of 5 (i.e., each data pixel corresponds to a 5x5 array of pixels in the oversampled PSF image) and you want the oversampled region within the data image to span (x,y) = [35:45,50:60]:

```
psfOsamp = pyimfit.MakePsfOversampler(oversampledPsf_image, 5, (35,45, 50,60))
osampleList = [psfOsamp]
imfit_fitter.loadData(data_imate, psf_oversampling_list=osampleList, ...)
```

Important: The image section is specified using 1-based (Fortran/IRAF) indexing, where the lower-left pixel of the image has coordinates (x,y) = (1,1), and it *includes* the endpoints. Thus, `pyimfit.MakePsfOversampler(osampPsfImage, 5, (35,45, 50,60))` will in Python/NumPy terms apply to the image region [34:45,49:60].

CHAPTER 6

Fitting an Image (Choosing Fit Statistics and Solvers)

In the example code below, we assume that an instance of the fitting.Imfit class has already been created and supplied with the necessary ModelDescription.

For example,

```
imfit_fitter = pyimfit.Imfit(someModelDescription)
```

6.1 Finding the best-fit solution by minimizing a fit statistic

Fitting a model to an image in PyImfit involves calculating a model image and then comparing it pixel-by-pixel with a data image to derive a summary "fit statistic" (based on some total likelihood value). The goal is to *minimize* the fit statistic (which corresponds to maximizing the likelihood). This is done by iteratively adjusting the parameters of the model and recomputing the fit statistic until convergence is achieved; the algorithm which oversees this process is called a "minimizer" or "solver".

6.2 Fit statistics (chi^2 and all that)

Which fit statistic to use depends in part on your data source, and so is determined as part of the `loadData` method of the Imfit class.

1. Chi^2 Statistics

   A. Sigmas estimated from the data values

   B. Sigmas estimated from the model values

   C. Sigmas from a pre-existing error/uncertainty/variance image

To specify model-based chi^2 as the fit statistic

```
imfit_fitter.loadData(imageData, ..., use_model_for_errors=True)
```

To supply your own error/uncertainty map

```
imfit_fitter.loadData(imageData, error=errorImageData, ...)
imfit_fitter.loadData(imageData, error=errorImageData, error_type="variance", ...)
```

2. Pure Poisson Statistics

To specify Poisson-MLR as the fit statistic

```
imfit_fitter.loadData(imageData, ..., use_poisson_mlr=True)
```

3. Possible specification of image A/D gain, exposure time, etc.

For any case *except* using a pre-existing error map, you may need to supply information about how the values in the data image can be converted to *detected* counts (e.g., detected photoelectrons), since the underlying statistical models assume the latter. For example, if the per-pixel values were converted to ADUs via an A/D gain, you should supply the gain value (in electrons/ADU); if the values are counts/second, you should also supply the total intgration time. If there was a significant read noise term, this should also be described. The relevant keywords for the `loadData` and `fit` methods are: `gain` (A/D gain in electrons/ADU), `read_noise` (Gaussian read noise in electrons), `exp_time` (seconds, *if* the data values are ADU/sec), and `n_combined` (number of combined exposures). An example:

```
imfit_fitter.loadData(imageData, ..., gain=3.1, exp_time=800, read_noise=7.5)
```

Note: If you are using Poisson-MLR as the fit statistic, then `read_noise` should not be used (the Poisson MLR statistical model cannot handle a Gaussian read-noise term).

4. Possible pre-subtracted background level

In some cases, it may be convenient to work with data images where the sky background has been removed. The fitting process needs to know about this, since otherwise there will be problems with data pixels having values near or below zero. You can specify a *constant* background level that has already been subtracted from the image, using the `original_sky` keyword for the `loadData` and `fit` methods; the value should be in the same units as the data pixels (e.g., ADU, ADU/sec, etc.). An example:

```
imfit_fitter.loadData(imageData, ..., original_sky=244.9)
```

6.3 Minimizers/Solvers

To actually find the best fit, you tell the Imfit object to find the minimum fit-statistic value, using a particular minimization algorithm.

PyImfit has three minimizers (a.k.a. solvers):

- Levenberg-Marquardt: This is the default, and is by far the fastest; it has the drawback of being the most prone to being trapped in local minima in the fit-statistic landscape. It requires initial guesses for each parameter value.

  ```
  imfit_fitter.DoFit()
  imfit_fitter.DoFit(solver="LM")
  ```

- Nelder-Mead Simplex: This is a slower algorithm, generally held to be less likely to be trapped in local fit-statistic minima. Like Levenberg-Marquardt, it requires initial guesses for each parameter value.

```
imfit_fitter.DoFit(solver="NM")
```

- Differential Evolution: This is a genetic-algorithms approach, and is the slowest of all the algorithms. Unlike the other two solvers, it requires lower and upper parameter *limits* for all non-fixed parameters. Initial guesses for the non-fixed parameter values are *ignored*.

```
imfit_fitter.DoFit(solver="DE")
```

Roughly speaking, the Nelder-Mead simplex minimizer is about an order of magnitude slower than Levenberg-Marquardt, and Differential Evolution is itself about an order of magnitude slower than Nelder-Mead simplex.

6.4 More Information

For more information:

- Section 4 of Erwin (2015) (ADS link)
- Chapter 9 of the Imfit manual (PDF)

CHAPTER 7

Bootstrap Resampling for Parameter Uncertainties

When you call the doFit method on an Imfit object with the default Levenberg-Marquardt solver, the solution automatically involves estimation of uncertainties on the best-fit parameters, in the form of 1-sigma Gaussian errors. These are generally not very accurate (they assume the fit-statistic landscape is quadratic) and do not include possible correlations between uncertainties in different parameters.

7.1 Bootstrap Resampling

A somewhat better (albeit more time-consuming) approach is to estimate the parameter uncertainties – including possible correlations between the values of different parameters – via bootstrap resampling.

You can do this by calling the runBootstrap method on an Imfit object – ideally *after* you've found a best-fit solution. E.g., assuming you've properly set up an Imfit object ("imfitter") and supplied it with the data image, etc.:

```
imfitter.doFit()
bootstrapResults = imfitter.runBootstrap(nIterations)
```

where nIterations is an integer specifying the total number of iterations (ideally a minimum of several hundred).

This returns a 2D NumPy array which has one row per iteration, with each column contain the collected values for a single parameter. The ordering of the parameter columns is the same as in the model.

You can get a list of numbered parameter names by using the getColumnNames keyword:

```
parameterNames, bootstrapResults = imfitter.runBootstrap(nIterations,␣
↪getColumnNames=True)
```

parameterNames is a list of strings of the form ['X0_1', 'Y0_1', 'n_1', etc.]. The 'X0','Y0' parameters are numbered by function set; all other parameters are numbered by function (so the parameters of the very first function in the model have '_1' appended to their names, all parameters from the second will have _2 appended, and so forth). This is make it possible to distinguish different parameters when multiple versions of the same function, or just multiple functions that have the same parameter names, are used in the model. (This numbering scheme is exactly the same as used by the command-line program imfit in its bootstrap-output files.)

7.1.1 Analyzing the Results

One thing you can do, if the model is not *too* complicated, is make a scatterplot matrix (a.k.a. corner plot) of the parameters. The Python package corner.py can be used for this; here's a quick-and-dirty example:

```
import corner

corner.corner(bootstrapResults, labels=parameterNames)
```

A sample result is shown below (see the Imfit tutorial for more about the data and model used to generate this figure).

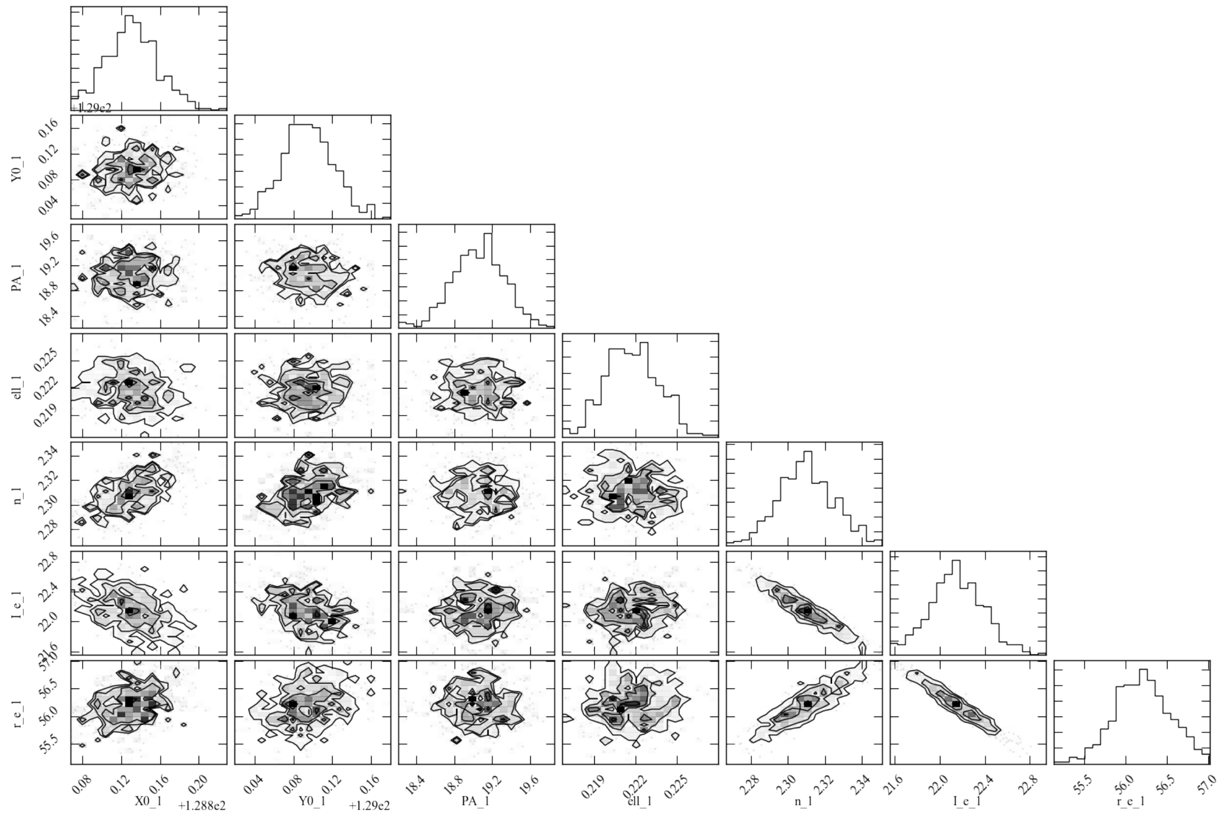


Figure 1: Scatterplot matrix of parameter values from 500 rounds of bootstrap resampling fits to an SDSS *r*-band image of the galaxy IC 3478 (single-Sérsic model, no PSF convolution). Note the clear correlations between the Sérsic model parameters (*n*, *r_e*, *I_e*).

See here for an example of using bootstrap output to estimate uncertainties on derived quantities, such as bulge/total values.

7.2 Using MCMC

Estimates of parameter uncertainties and correlations can also be obtained via Markov-Chain Monte Carlo (MCMC) approaches. Although the MCMC option of **Imfit** (`imfit-mcmc`) is not part of PyImfit, you *can* use instances of the Imfit class with Python-based MCMC codes, such as emcee; see here for an example.

CHAPTER 8

Example of using PyImfit with Markov-Chain Monte Carlo code "emcee"

This is a Jupyter notebook demonstrating how to use PyImfit with the MCMC code emcee.

If you are seeing this as part of the readthedocs.org HTML documentation, you can retrieve the original .ipynb file here.

Some initial setup for nice-looking plots:

```
%pylab inline

matplotlib.rcParams['figure.figsize'] = (8,6)
matplotlib.rcParams['xtick.labelsize'] = 16
matplotlib.rcParams['ytick.labelsize'] = 16
matplotlib.rcParams['axes.labelsize'] = 20
```

```
Populating the interactive namespace from numpy and matplotlib


/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/IPython/
↪core/magics/pylab.py:160: UserWarning: pylab import has clobbered these variables: [
↪'mean']
`%matplotlib` prevents importing * from pylab and numpy
  "\n`%matplotlib` prevents importing * from pylab and numpy"
```

8.1 Create image-fitting model using PyImfit

Load the pymfit package; also load astropy.io.fits so we can read FITS files:

```
import pyimfit
from astropy.io import fits
```

Load data image (in this case, a small cutout of an SDSS image showing a faint star):

```
imageFile = "./pyimfit_emcee_files/faintstar.fits"
image_faintstar = fits.getdata(imageFile)
```

Create a ModelDescription instance based on an imfit configuration file (which specifies a single elliptical Gaussian model):

```
configFile = "./pyimfit_emcee_files/config_imfit_faintstar.dat"
model_desc = pyimfit.ModelDescription.load(configFile)
```

Alternately, you can create the ModelDescription programmatically from within Python:

```
# create a SimpleModelDescription instance (one function block); specify the x0,y0␣
↪center for the function block.
model_desc = pyimfit.SimpleModelDescription()
# define the X0,Y0 initial guess and limits
model_desc.x0.setValue(5.0, [3.0,8.0])
model_desc.y0.setValue(6.0, [3.0,8.0])
# create a Gaussian image function for the star and set its parameters' initial␣
↪guesses and limits
star_function = pyimfit.make_imfit_function("Gaussian")
star_function.PA.setValue(155, [140,170])
star_function.ell.setValue(0.1, [0,0.3])
star_function.I_0.setValue(250, [220,320])
star_function.sigma.setValue(1.0, [0.1,1.5])
# now add the image function to the model
model_desc.addFunction(star_function)
```

Create an Imfit instance containing the model, and add the image data and image-description info:

```
imfit_fitter = pyimfit.Imfit(model_desc)
imfit_fitter.loadData(image_faintstar, gain=4.72, read_noise=1.15, original_sky=124.
↪94)
```

Fit the model to the data (using the default Levenberg-Marquardt solver) and extract the best-fitting parameter values (X0, Y0, PA, ellipticity, I_0, sigma):

```
results = imfit_fitter.doFit(getSummary=True)
p_bestfit = results.params

print("Best-fitting parameter values:")
for i in range(len(p_bestfit) - 1):
    print("{0:g}".format(p_bestfit[i]), end=", ")
print("{0:g}\n".format(p_bestfit[-1]))
```

```
Best-fitting parameter values:
5.64339, 6.18794, 155.354, 0.0950157, 268.92, 1.00772
```

8.2 Define log-probability functions for use with emcee

Emcee requires a function which calculates and returns the log of the posterior probability (using the likelihood and the prior probability).

We'll create a general function for this which takes as input the current model parameters, an Imfit instance which can compute the fit statistic for those parameters (= $-2 \times$ log likelihood) and a user-supplied function for computing the prior; this will return the sum of the log likelihood and the log of the prior:

```
def lnPosterior_for_emcee( params, imfitter, lnPrior_func ):
    """
    Returns log of posterior probability (which is calculated as the
    product of the specified prior and the likelihood computed by the
    Imfit object using the specified parameter values).

    Parameters
    ----------
    params : 1D numpy ndarray of float
        vector of current parameter values

    imfitter : pyimfit.Imfit instance

    lnPrior_func : function or other callable
        Should compute and return log of prior probability
        signature = lnPrior_func(parameter_vector, Imfit_instance)

    Returns
    -------
    logPosterior : float
    """
    lnPrior = lnPrior_func(params, imfitter)
    if not np.isfinite(lnPrior):
        return -np.inf
    # note that Imfit.computeFitStatistic returns -2 log(likelihood)
    lnLikelihood = -0.5 * imfitter.computeFitStatistic(params)
    return lnPrior + lnLikelihood
```

Now, we'll create a prior-probability function.

For simplicity, we'll use the case of constant priors within parameter limits, with the parameter limits obtained from a user-supplied Imfit instance. (But you can make the prior-probability function as complicated as you like.)

```
def lnPrior_limits( params, imfitter ):
    """
    Defines prior-probability distributions as flat within specified limits.
    If any parameter is outside the limits, we return -np.inf; otherwise, we
    return ln(1.0) = 0 (not strictly speaking a correct probability, but it
    works for this case).

    Parameters
    ----------
    params : 1D numpy ndarray of float

    imfitter : pyimfit.Imfit instance

    Returns
    -------
    logPrior : float
    """
    parameterLimits = imfitter.getParameterLimits()
    if None in parameterLimits:
        raise ValueError("All parameters must have lower and upper limits.")
    nParams = len(params)
    for i in range(nParams):
        if params[i] < parameterLimits[i][0] or params[i] > parameterLimits[i][1]:
            return -np.inf
    return 0.0
```

8.3 Set up and run Markov-Chain Monte Carlo using emcee

Import emcee, and also corner (so we can make a nice plot of the results):

```
import emcee
import corner
```

Specify the number of dimensions (= number of parameters in the model) and a large number of walkers, then instantiate a standard emcee sampler, using our previously defined posterior function (the Imfit instance containing the data and model and the simple prior function are provided as extra arguments):

```
ndims, nwalkers = 6, 100

sampler = emcee.EnsembleSampler(nwalkers, ndims, lnPosterior_for_emcee, args=(imfit_
↪fitter, lnPrior_limits))
```

Define some initial starting values – 0.1% Gaussian perturbations around the previously determined best-fit parameters:

```
initial_pos = [p_bestfit * (1 + 0.001*np.random.randn(ndims)) for i in␣
↪range(nwalkers)]
```

Run the sampler for 500 steps (reset it first, in case we're running this again, to ensure we start anew):

```
sampler.reset()
final_state = sampler.run_mcmc(initial_pos, 500)
```

Plot values from all the walkers versus step number to get an idea of where convergence might happend (here, we just plot the ellipticity and I_0 values):

```
def PlotAllWalkers( sample_chain, parameterIndex, yAxisLabel ):
    nWalkers = sample_chain.shape[0]
    for i in range(nWalkers):
        plot(sample_chain[i,:,parameterIndex], color='0.5')
    xlabel('Step number')
    ylabel(yAxisLabel)

PlotAllWalkers(sampler.chain, 3, 'ellipticity')
```

```
PlotAllWalkers(sampler.chain, 4, 'I_0')
```

Define the "converged" subset of the chains as step numbers ≥ 200, and merge all the individual walkers:

```
converged_samples = sampler.chain[:, 200:, :].reshape((-1, ndims))
print("Number of samples in \"converged\" chain = {0}".format(len(converged_samples)))
```

```
Number of samples in "converged" chain = 30000
```

8.4 Corner plot of converged MCMC samples

Define some nice labels and parameter ranges for the corner plot:

```
cornerLabels = [r"$X_{0}$", r"$Y_{0}$", "PA", "ell", r"$I_{0}$", r"$\sigma$"]

```

(continues on next page)

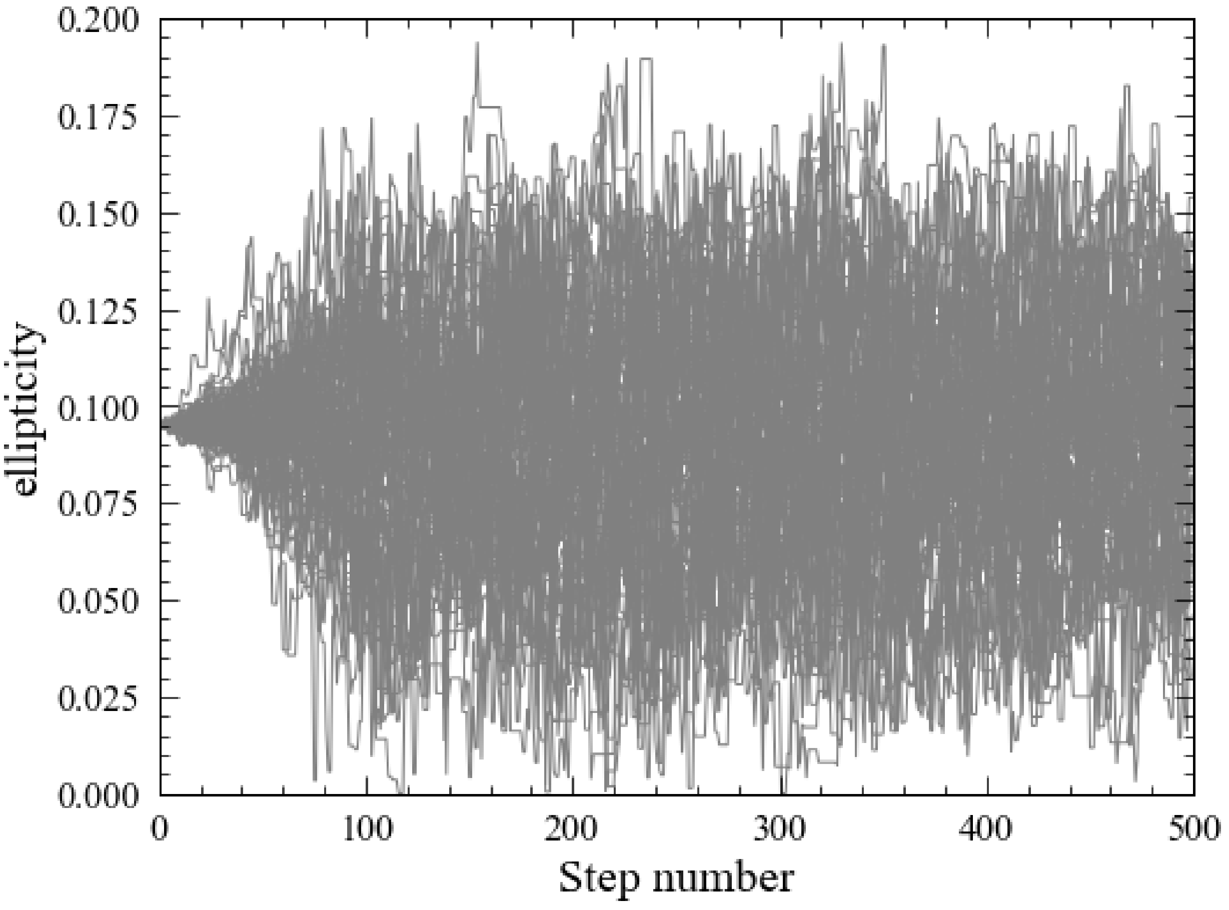
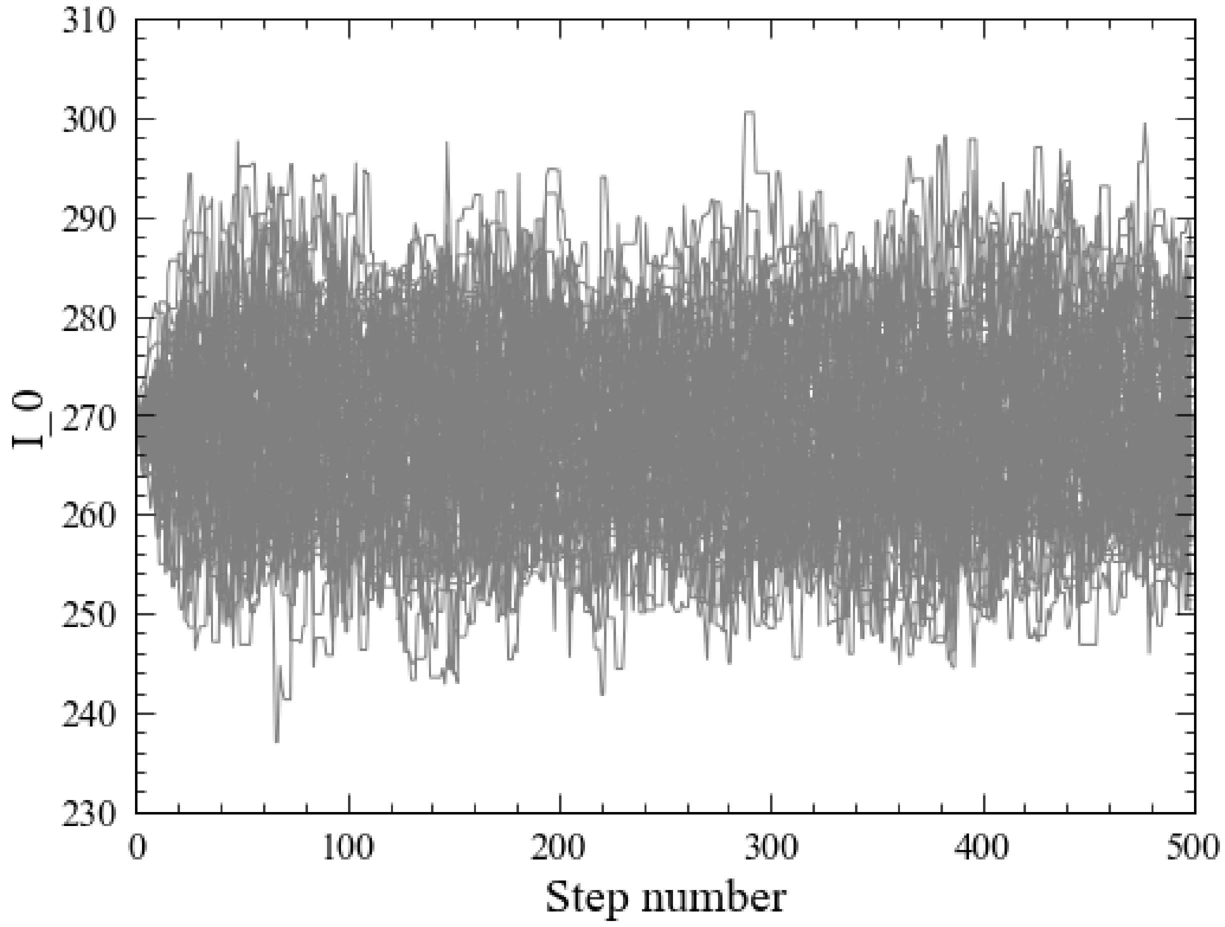

0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
ellipticity
0
100
200
300
400
500
Step number

Fig. 1: png



310
300
290
280
270
260
250
240
230
I_0
0
100
200
300
400
500
Step number

Fig. 2: png

(continued from previous page)

```
x0_range = (5.55, 5.73)
y0_range = (6.09, 6.29)
pa_range = (138,173)
ell_range = (0, 0.2)
i0_range = (240,300)
sigma_range = (0.92, 1.1)
ranges = [x0_range, y0_range, pa_range, ell_range, i0_range, sigma_range]
```

Make a corner plot; the thin blue lines/points indicate best-fit values from above. [Note that we have to explicitly capture the Figure instance returned by corner.corner, otherwise we'll get a duplicate display of the plot]:

```
fig = corner.corner(converged_samples, labels=cornerLabels, range=ranges, truths=p_
↪bestfit)
```

One thing to notice is that the PA values are running up against our (rather narrow) limits for that parameter, so a next step might be to re-run this with larger PA limits.

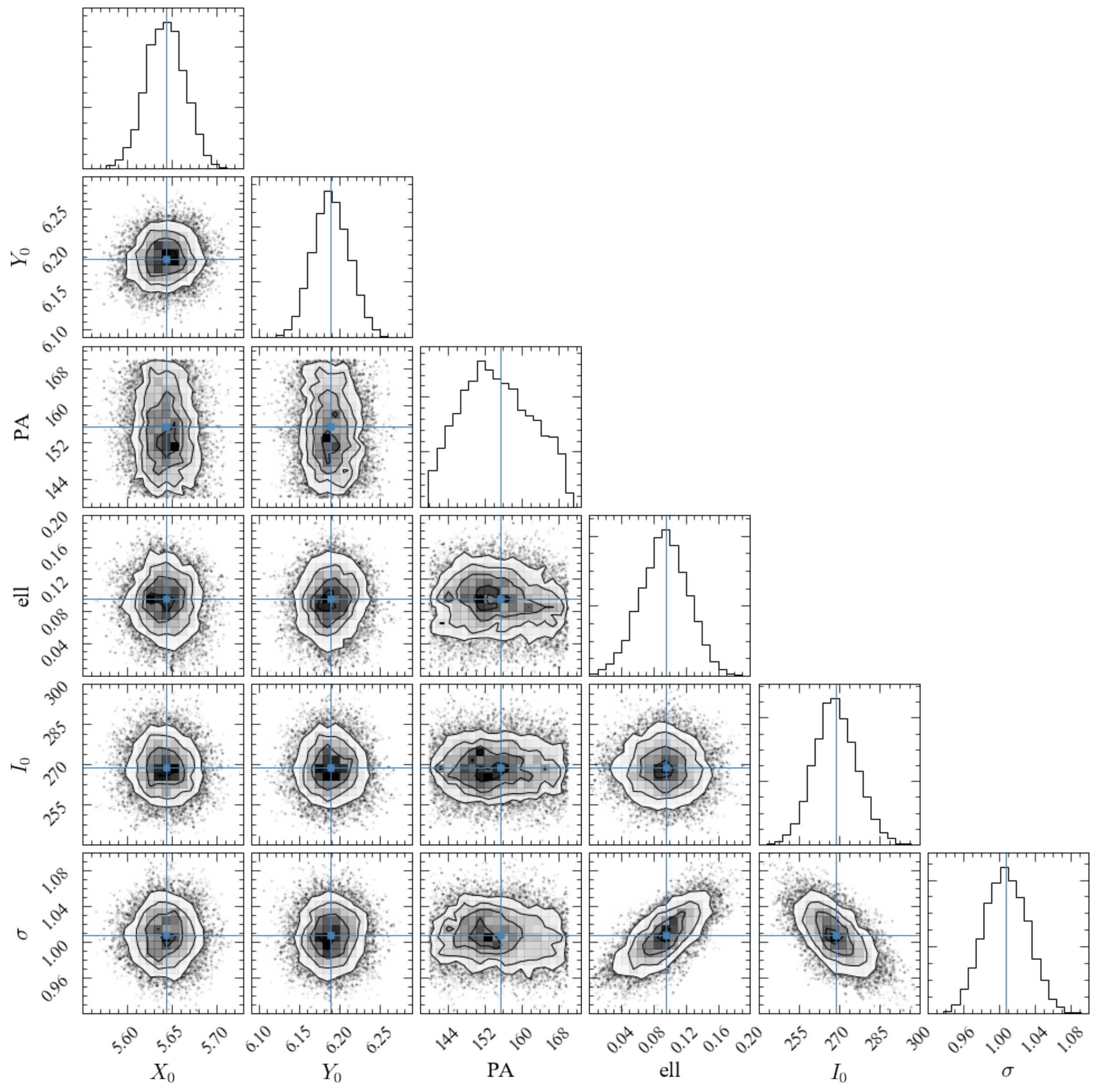

Y_0
PA
ell
I_0
σ
X_0
Y_0
PA
ell
I_0
σ
6.10
6.15
6.20
6.25
144
152
160
168
0.04
0.08
0.12
0.16
0.20
255
270
285
300
0.96
1.00
1.04
1.08
5.60
5.65
5.70

Fig. 3: png

CHAPTER 9

Example of using PyImfit to Estimate B/T Uncertainties

This is a Jupyter notebook demonstrating how to use PyImfit and bootstrap resampling to estimate uncertainties for derived quantities of fits, such as B/T values.

If you are seeing this as part of the readthedocs.org HTML documentation, you can retrieve the original .ipynb file here.

9.1 Introduction

PyImfit will estimate uncertainties for individual model parameters from a fit (if you use the default Levenberg-Marquardt minimizer) – e.g., position $X0, Y0$, position-angles, ellipticities, scale lengths, etc.. You can also estimate parameter uncertainties via bootstrap resampling, or by using an external Markov-Chain Monte Carlo algorithm (see here for an example of the latter).

Sometimes, you might also want to have some estimate of derived values based on a model, such as the total luminosity or the bulge/total (B/T) value (assuming you have some idea of which component in your model is the "bulge"). How do you determine the uncertainties for such quantities? This notebook shows a simple example of how one might do that, using PyImfit's bootstrap-resampling option.

The basic idea is to generate a set of model-parameter vectors via, e.g., bootstrap resampling (or from an MCMC chain). You then compute the resulting derived quantity from those parameter values. In this particular case, we use the `pyimfit.Imfit` object containing the model to compute "bulge" and total flux values for each parameter vector, and then take the ratio to get B/T values. By doing this for all the parameter vectors, you end up with a distribution for the derived quantity.

Preliminaries

Some initial setup for nice-looking plots:

```
%pylab inline

matplotlib.rcParams['figure.figsize'] = (8,6)
matplotlib.rcParams['xtick.labelsize'] = 16
```

(continues on next page)

(continued from previous page)

```
matplotlib.rcParams['ytick.labelsize'] = 16
matplotlib.rcParams['axes.labelsize'] = 20
```

```
Populating the interactive namespace from numpy and matplotlib
```

9.1.1 Create an image-fitting model using PyImfit

Load the pymfit package; also load numpy and astropy.io.fits (so we can read FITS files):

```
import numpy as np
import pyimfit
from astropy.io import fits
```

Load the data image (here, an SDSS r-band image cutout of VCC 1512) and corresponding mask:

```
imageFile = "./pyimfit_bootstrap_BtoT_files/vcc1512rss_cutout.fits"
image_vcc1512 = fits.getdata(imageFile)
maskFile = "./pyimfit_bootstrap_BtoT_files/vcc1512rss_mask_cutout.fits"
mask_vcc1512 = fits.getdata(maskFile)
```

Create a ModelDescription instance based on an imfit configuration file (which specifies a Sersic + Exponential model):

```
configFile = "./pyimfit_bootstrap_BtoT_files/config_imfit_vcc1512.dat"
model_desc = pyimfit.ModelDescription.load(configFile)
```

```
print(model_desc)
```

```
ORIGINAL_SKY    120.020408
GAIN    4.725000
READNOISE   4.300000
X0      60.0
Y0      73.0
FUNCTION Sersic
PA      155.0       90.0,180.0
ell     0.2     0.0,0.5
n       2.05        0.0,4.0
I_e     120.0       0.0,10000.0
r_e     4.5     0.0,20.0

FUNCTION Exponential
PA      140.0       90.0,180.0
ell     0.28        0.0,0.8
I_0     70.0        0.0,10000.0
h       20.0        0.0,200.0
```

Create an Imfit instance containing the model, and add the image and mask data. Note that we are *not* doing PSF convolution, in order to save time (this is not meant to be a particular accurate model).

```
imfit_fitter = pyimfit.Imfit(model_desc)
imfit_fitter.loadData(image_vcc1512, mask=mask_vcc1512)
```

Fit the model to the data (using the default Levenberg-Marquardt solver) and extract the best-fitting parameter values:

```
results = imfit_fitter.doFit(getSummary=True)
```

```
print(results)
```

```
           aic: 21156.824446201397
           bic: 21242.642276390998
  fitConverged: True
       fitStat: 21134.809840392267
fitStatReduced: 1.169219398118625
         nIter: 10
     paramErrs: array([0.01518161, 0.0167467 , 1.88166351, 0.00733777, 0.01613089,
      1.9553319 , 0.05896027, 0.65080573, 0.00529781, 1.11196358,
      0.18740197])
        params: array([6.04336387e+01, 7.32059007e+01, 1.61799952e+02, 1.18947666e-01,
      9.56352657e-01, 1.21814611e+02, 4.86558532e+00, 1.38986928e+02,
      2.73912311e-01, 8.13853830e+01, 2.08521933e+01])
    solverName: 'LM'
```

```
p_bestfit = results.params

print("Best-fitting parameter values:")
for i in range(len(p_bestfit) - 1):
    print("{0:g}".format(p_bestfit[i]), end=", ")
print("{0:g}\n".format(p_bestfit[-1]))
```

```
Best-fitting parameter values:
60.4336, 73.2059, 161.801, 0.118946, 0.956308, 121.821, 4.86538, 138.987, 0.273911,␣
↪81.389, 20.8517
```

9.1.2 Run bootstrap-resampling to generate a set of parameter values (array of best-fit parameter vectors)

OK, now we're going to do some bootstrap resampling to build up a set of several hundred alternate "best-fit" parameter values.

Note that you coul also generate a set of parameter vectors using MCMC; we're doing bootstrap resampling mainly because it's faster.

Run 500 iterations of bootstrap resamplng. More would be better; this is just to save time (takes about 1 minute on a 2017 MacBook Pro).

```
bootstrap_params_array = imfit_fitter.runBootstrap(500)
```

```
bootstrap_params_array.shape
```

```
(500, 11)
```

9.1.3 Use these parameter vectors to calculate range of B/T values

We define a function to calculate the B/T value, given a parameter vector (for this model it's simple, but you might have a more complicated model where the first component isn't necessarily the "bulge").

```
def GetBtoT( fitter, params ):
    """
    Get the B/T value for a model parameter vector (where "bulge" is the first␣
↪component
    in the model)

    Parameters
    ----------
    fitter : instance of PyImfit's Imfit class
        The Imfit instance containing the model and data to be fit
    params : 1D sequence of float
        The parameter vector corresponding to the model

    Returns
    -------
    B/T : float
    """
    total_flux, component_fluxes = fitter.getModelFluxes(params)
    # here, we assume the first component in the model is the "bulge"
    return component_fluxes[0] / total_flux
```

The $B/T$ value for the best-fit model:

```
GetBtoT(imfit_fitter, p_bestfit)
```

```
0.1557485598370547
```

Now calculate the $B/T$ values for the bootstrap-generated set of parameter vectors:

```
n_param_vectors = params_array.shape[0]
b2t_values = [GetBtoT(imfit_fitter, bootstrap_params_array[i]) for i in range(n_param_
↪vectors)]
b2t_values = np.array(b2t_values)
```

And now we can analyze the vector of B/T values . . .

For example:

```
np.mean(b2t_values)
```

```
0.15852423639167879
```

A histogram of the $B/T$ values (vertical line = best-fit value):

```
hist(b2t_values, bins=np.arange(0.14,0.2,0.0025));xlabel(r"$B/T$");ylabel(r"$N$")
axvline(GetBtoT(imfit_fitter, p_bestfit), color='k')
```

```
<matplotlib.lines.Line2D at 0x12c671a10>
```

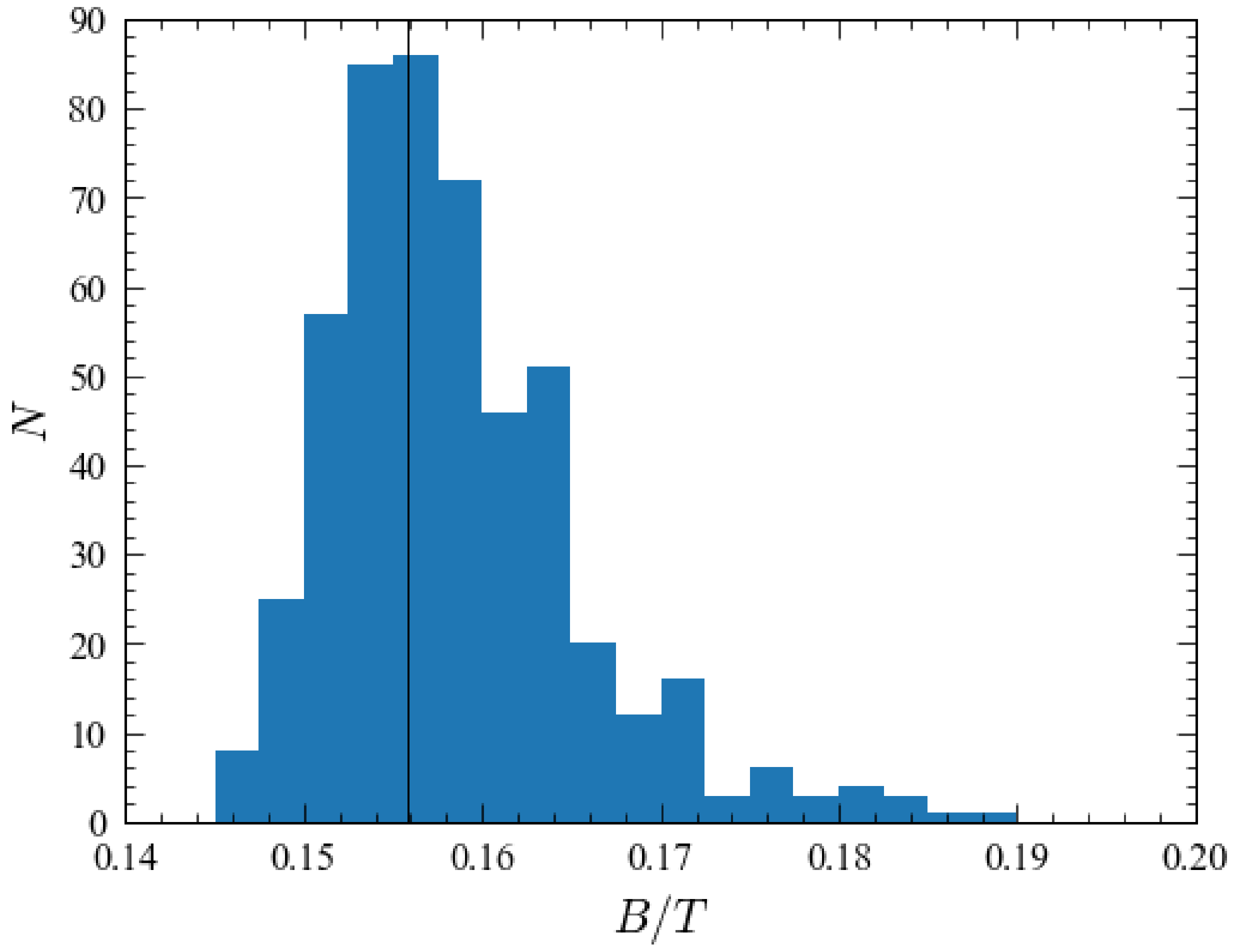


Fig. 1: png

CHAPTER 10

Acknowledgments

PyImfit is based on André Luiz de Amorim's python-imfit wrapper about an earlier version of **Imfit**.

10.1 Imfit-related Acknowledgements

Major inspirations for **Imfit** include both GALFIT and BUDDA.

Thanks also to Michael Opitsch and Michael Williams for being (partly unwitting) beta testers and for their feedback, to Martin Kuemmel for suggesting an early improvement (and finding a related bug), to Roberto Saglia for urging me to implement the Core-Sérsic function, and to Maximilian Fabricius for suggesting improvements to the documentation. Additional bug reports and suggestions from André Luiz de Amorim, Giulia Savorgnan, David Streich, Guillermo Barro, Sergio Pascual, Lee Kelvin, Colleen Gilhuly, Semyeong Oh, Benne Holwerde, David Wilman, Iskren Georgiev, Corentin Schreiber, Dan Prole, Alex Borlaff, and Daria Kozlova are gratefully appreciated.

10.1.1 Data Sources

Sample FITS images for demonstration and testing use are taken from Data Release 7 of the Sloan Digital Sky Survey. Funding for the creation and distribution of the Sloan Digital Sky Survey Archive has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Aeronautics and Space Administration, the National Science Foundation, the U.S. Department of Energy, the Japanese Monbukagakusho, and the Max Planck Society. The SDSS Web site is http://www.sdss.org/.

The SDSS is managed by the Astrophysical Research Consortium (ARC) for the Participating Institutions. The Participating Institutions are The University of Chicago, Fermilab, the Institute for Advanced Study, the Japan Participation Group, The Johns Hopkins University, the Korean Scientist Group, Los Alamos National Laboratory, the Max-Planck-Institute for Astronomy (MPIA), the Max-Planck-Institute for Astrophysics (MPA), New Mexico State University, University of Pittsburgh, University of Portsmouth, Princeton University, the United States Naval Observatory, and the University of Washington.

10.1.2 Specific Software Acknowledgments

PyImfit makes use of the following external libraries:

- FFTW3
- GNU Scientific Library (GSL)
- NLopt

Minpack

This product includes software developed by the University of Chicago, as Operator of the Argonne National Laboratory.

CHAPTER 11

PyImfit API

11.1 Description Classes

11.2 The Imfit class

11.3 Useful Functions

Imfit manual (PDF)