lmfit

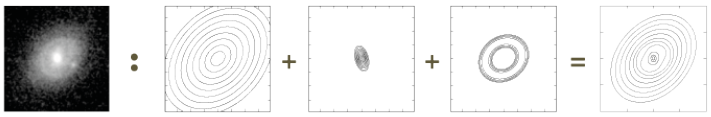
Release 1.9

Peter Erwin

Nov 27, 2022

USER DOCUMENTATION:

IMFIT



Imfit is an open-source C++ program for fitting models to astronomical images (primarily of galaxies). It is fast, flexible, and designed to be easily extended with new functions for components of the model image.

Examples of Use:

Fitting an image (with a model specified in the configuration file `model_description.txt`):

```
$ imfit someimage.fits --config model_description.txt
```

Fitting a subsection of the image and convolving the model with a Point-Spread-Function image, using Differential Evolution as the solver:

```
$ imfit someimage.fits[500:750,600:800] --config model_description.txt --psf
psf.fits --de
```

Markov Chain Monte Carlo (MCMC) analysis of the same image and model:

```
$ imfit-mcmc someimage.fits[500:750,600:800] --config model_description.txt
--psf psf.fits
```

Useful things:

- `imfit -h` – lists command-line flags and options
- `imfit --list-functions` – lists all available image functions for models
- `imfit --list-parameters` – lists the individual parameters for all image functions
- `imfit --sample-config` – writes a sample configuration file to the current directory

The main documentation for Imfit is in the PDF file imfit_howto.pdf.

Where to Get It:

Pre-compiled binaries for Linux and macOS, along with the source code for compilation, are available here.

The full source-code distribution is available at Imfit's Github page.

CHAPTER

ONE

CONTENTS:

1.1 Preliminaries

To get started with Imfit, you need to download the pre-compiled binary distribution for your platform (Mac or Linux), or else download and compile the source code; links to both can be found on the main imfit page. Notes on how to compile the source code can be found in the Imfit documentation.

In both the binary-only and source-code distributions, there is a subdirectory called "examples", which has some images we'll be using in this tutorial – ic3478rss_256.fits, ic3478rss_256_mask.fits, and psf_moffat_51.fits – as well as some configuration files (names starting with "config_"). You can go ahead and work in the examples subdirectory, or copy the files there to another directory and work there.

If you want to download just the examples directory and its files, you can find it here.

1.2 Fitting Your First Image

Imfit requires, as a minimum, two things:

1. An image in FITS format containing the data to be fit;
2. A configuration file describing the model you want to fit to the data.

So to start off, we'll try fitting the image file ic3478rss_256.fits (a 256 x 256-pixel cutout from a DR7 SDSS *r*-band image of the dwarf elliptical galaxy IC 3478) with a simple exponential model, which is described in the configuration file config_exponential_ic3478_256.dat. To do the fit, just type (all on one line):

```
imfit ic3478rss_256.fits -c config_exponential_ic3478_256.dat
  --sky=130.14
```

The `--sky=130.14` is a note to Imfit that the image had a background sky level of 130.14 counts/pixel which was previously subtracted; if we don't include this, Imfit will get confused by the fact that some of the pixels in the image have slightly negative values. (Note that you can use = or a space to connect an option with its argument on the command line.)

(Note that, as is normal for Unix-style commands, all invocations of `imfit` should be entered as a single line, even though in this and some of the other examples it's displayed on two or more lines to fit the web page better.)

Imfit will print some preliminary information, confirming which files are being used, the size of the image being fit, the image functions used in the model, and so forth. It will then call the minimization routine, which prints a minimal set of updates for each iteration. At the end, a summary of the fit is printed (final 2, etc.), along with the best-fitting parameters of the model and some crude estimates of the uncertainties for each parameter. These parameters are also saved in a text file: "bestfit_parameters_imfit.dat", which includes a record of how imfit was called and a short summary of the fit. (You can specify a different name for this output file via the `--save-params` option.)

```
Reduced Chi^2 = 0.450366
AIC = 29524.514967, BIC = 29579.055814

X0      128.8530 # +/- 0.0517
Y0      129.1035 # +/- 0.0633
FUNCTION Exponential
PA      19.7364 # +/- 0.467417
ell     0.231428 # +/- 0.00333939
I_0     315.173 # +/- 1.33252
h       20.5726 # +/- 0.0748152
```

Congratulations; you've fit your first image!

1.3 Inspecting the Fit: Model Images and Residuals

So what kind of fit did we get, and how good was it? When you run imfit, you can tell it to save the best-fitting model image, and the residual image (data - model) as well, using the `--save-model` and `--save-residual` commandline options:

```
imfit ic3478rss_256.fits -c config_exponential_ic3478_256.dat
  --sky=130.14 --save-model=model.fits --save-residual=resid.fits
```

These are FITS files with the same dimensions as the data image.

If you look at the residual image (below, right), you can see it's systematically bright in the center, with an oval region of negative pixels outside. This is a pretty good indication that the exponential model isn't actually a good match to the data, something we'll try to address in a bit.

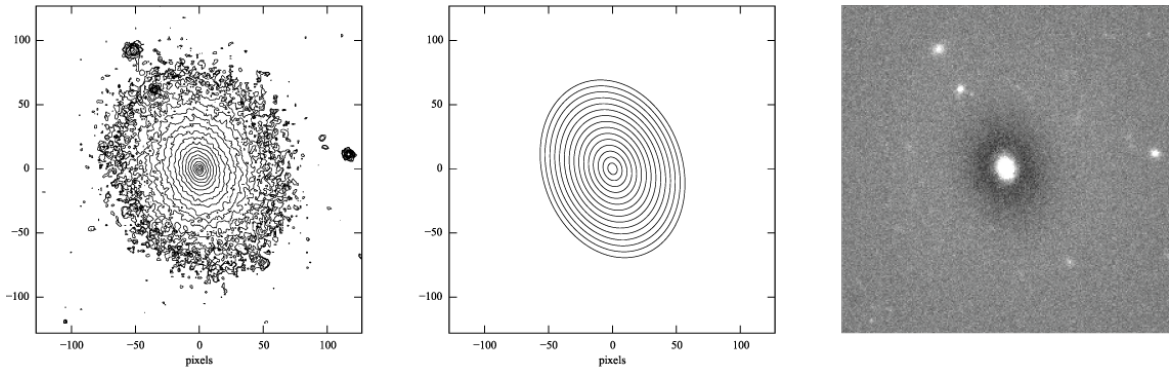


Figure 1: log-scaled isophotes for original SDSS image (left) and best-fitting exponential model (middle), along with linear-scaled residual image (data - model, right).

CHAPTER

TWO

GENERATING MODEL IMAGES WITH MAKEIMAGE

You can also generate a copy of the model image using the "makeimage" program which comes with imfit; it can take a best-fit parameter file produced by imfit as its own input. To run makeimage, you need:

1. An input configuration file;
2. Some specification for the size of the output image (this can be included in the configuration file, if you wish).

To run makeimage, you can type

```
makeimage bestfit_parameters_imfit.dat --refimage=ic3478rss_256.fits
```

This tells makeimage to make an image with the same dimensions as the "reference image" (ic3478rss_256.fits, in this case). You can also use the commandline parameters `--ncols` and `--nrows` to directly specify the output image size, or you can edit the input configuration file so it specifies the image size there (see the main documentation). By default, this saves the model image using the filename "model.fits"; you can use the `-o` commandline parameter to specify your own name for the outer file.

2.1 Better Fits: Telling Imfit the Truth About the Image

Leaving aside the question of mismatches between an exponential model and the actual galaxy, this isn't the best possible fit yet for our model. (You may have noticed that imfit reported a reduced 2 value of ~ 0.45, which is a sign something odd is going on.) For one thing, we've deceived imfit about the nature of the data. The default 2 minimization process that imfit uses is based on the Gaussian approximation to Poisson statistics, and assumes that the pixel values in the image are detected photoelectrons (or N-body particles, or something else that obeys Poisson statistics). In reality, our image deviates from this ideal in three ways:

1. There was a sky background that was previously subtracted from the image;
2. The pixel values are counts (ADUs), not detected photoelectrons;
3. The image has some Gaussian read noise.

To fix this, we can tell imfit three things:

1. The original background level (which we're already doing, via the `--sky` option);
2. The A/D gain in electrons/count, via the `--gain` option;
3. The read noise value (in electrons), via the `--readnoise` option

In the case of this SDSS image, the corresponding tsField FITS table (from the SDSS DR7 archive) has information about the A/D gain and the read noise (or "dark variance") and tells us that the gain and read noise are 4.725 and 4.3 electrons, respectively, for the *r*-band image.

So we can re-run the fit with the following command:

```
imfit ic3478rss_256.fits -c config_exponential_ic3478_256.dat
  --sky=130.14 --gain=4.725 --readnoise=4.3
```

Now the reduced 2 is about 2.1, which isn't necessarily that good, but is at least statistically plausible!

```
Reduced Chi^2 = 2.082564
AIC = 136482.400611, BIC = 136536.941458

X0      128.8540 # +/- 0.0239
Y0      129.1028 # +/- 0.0293
FUNCTION Exponential
PA      19.7266 # +/- 0.217212
ell     0.23152 # +/- 0.00155236
I_0     316.313 # +/- 0.619616
h        20.522 # +/- 0.0346742
```

2.2 Better Fits: Masking

If you look at the image (e.g., with SAOimage DS9 or another FITS-displaying program), you can see features that most likely aren't part of the galaxy – for example, there are certainly three (and possibly five) distinct, small objects near the galaxy which are probably foreground stars or background galaxies. Since they're relatively bright compared to the outer parts of the galaxy, they will bias the fit.

To prevent this from happening, you can mask out parts of an image. This is done with a separate mask image: an image of the same size as the data, but with pixel values = 0 for all the "good" pixels and >= 1 for all the "bad" pixels (i.e., those pixels you want Imfit to ignore).

The file ic3478rss_256_mask.fits in the examples directory is a mask image. You can use it in the fit with the "`--mask`" option:

```
imfit ic3478rss_256.fits -c config_exponential_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --sky=130.14 --gain=4.725
  --readnoise=4.3
```

(Again, note that options can be linked to their targets with "=" or with just a space, whichever make more sense to you.)

The reduced 2 is slightly smaller; in addition, the position angle, ellipticity, and scale length of the best-fitting model have changed slightly (the smaller scale length is because imfit is no longer trying to account for the excess light from the other sources by radially stretching the exponential).

```
Reduced Chi^2 = 1.964467
AIC = 124602.443320, BIC = 124656.787960

X0      128.8793 # +/- 0.0237
Y0      129.0589 # +/- 0.0289
FUNCTION Exponential
PA      18.7492 # +/- 0.23086
ell     0.220646 # +/- 0.00159077
I_0     321.631 # +/- 0.634224
h       20.0684 # +/- 0.034584
```

2.3 Better Fits: Trying Different Models

As noted above, it looks like the exponential model is not a good match to the galaxy. You can see the available model components ("image functions") by calling imfit with the `--list-functions` option:

```
imfit --list-functions
```

You can also see the full set of parameters for each image function using the `--list-parameters` option:

```
imfit --list-parameters
```

A model fit to an image can consist of multiple image functions (and multiple instances of each image function), but for now let's just try a Sérsic function with elliptical isophotes. This is encoded in the "config_sersic_ic3478_256.dat" file.

```
imfit ic3478rss_256.fits -c config_sersic_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --gain=4.725 --readnoise=4.3
  --sky=130.14
```

The result is a significantly better fit:

```
Reduced Chi^2 = 1.055366
AIC = 66946.393806, BIC = 67009.795665

X0      128.9321 # +/- 0.0130
Y0      129.0983 # +/- 0.0155
FUNCTION Sersic
PA      19.0449 # +/- 0.247618
ell     0.221656 # +/- 0.00171861
n        2.3108 # +/- 0.00818546
I_e     22.1351 # +/- 0.163568
r_e     56.2217 # +/- 0.256568
```

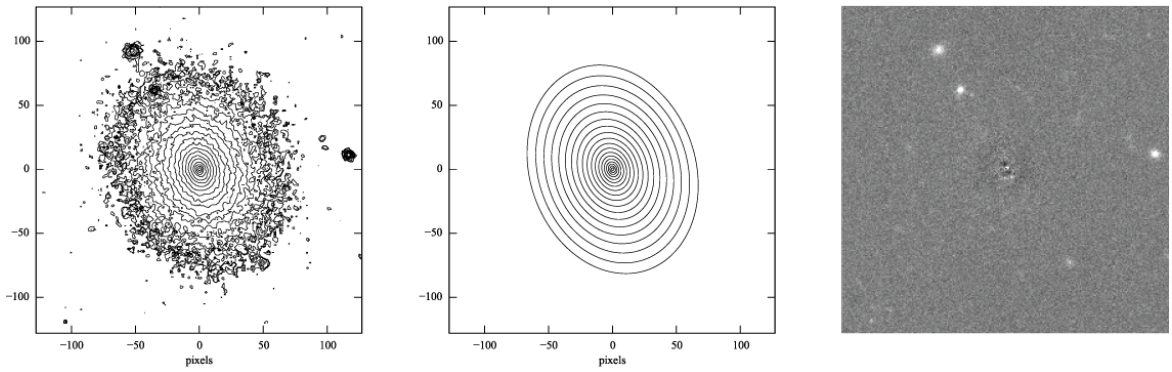


Figure 2: log-scaled isophotes for original SDSS image (left) and best-fitting Sérsic model (middle), along with linear-scaled residual image (data - model, right). Note that the residuals are much improved over the residuals for the exponential model (*Figure 1*).

This is clearly a *much* better fit!

2.4 More Correct Fits: PSF Convolution

Astronomical images obtained with telescopes are almost always affected by telescope optics, atmospheric seeing, and so forth, so that the actual recorded image – what we're trying to model – is really the convolution of an idealized "true" image with a point-spread function (PSF).

You can simulate this process in Imfit by providing a PSF image in FITS format, using the `--psf` option. This can be any square, centered image, based on observed stellar PSFs, produced by telescope modeling software, etc. Imfit will then convolve the internally generated model image with the PSF image before comparing the model with the data.

Here, we use a pre-generated 51 x 51-pixel PSF image which approximates the seeing in the SDSS image using a circular Moffat function:

```
imfit ic3478rss_256.fits -c config_sersic_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --gain=4.725 --readnoise=4.3
  --sky=130.14 --psf psf_moffat_51.fits

Reduced Chi^2 = 1.074154
AIC = 68137.906037, BIC = 68201.307896

X0      128.9174 # +/- 0.0147
Y0      129.0800 # +/- 0.0176
FUNCTION Sersic
PA      19.0576 # +/- 0.247209
ell     0.227617 # +/- 0.00175711
n       2.48051 # +/- 0.00983808
I_e     19.9097 # +/- 0.169477
r_e     59.5241 # +/- 0.309487
```

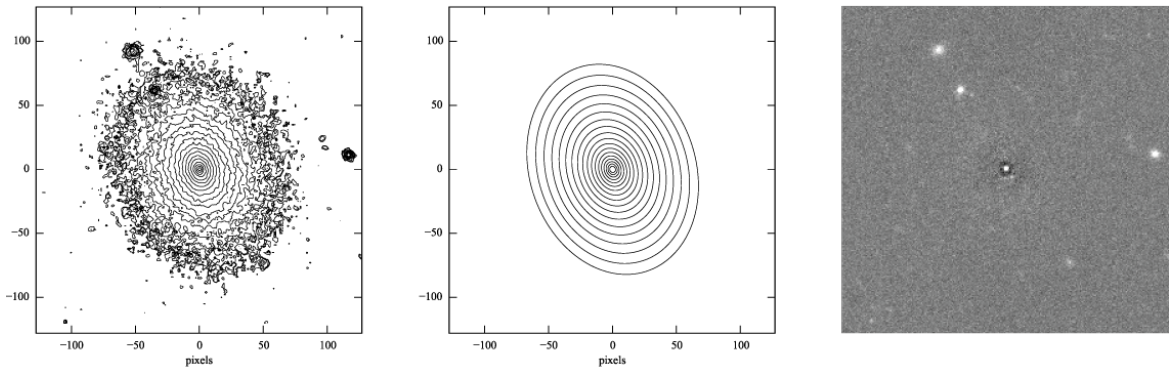


Figure 3: log-scaled isophotes for original SDSS image (left) and best-fitting, PSF-convolved Sérsic model (middle), along with linear-scaled residual image (data - model, right).

The residuals for the PSF-convolved fit (above right) are systematically somewhat *worse* than without the PSF (compare with *Figure 2*): there is a small central excess and a surrounding negative-pixel "moat". So the galaxy is probably a bit more complicated than just a single Sérsic function can accomodate. (In fact, Janz et al. 2014, working with a higher-resolution and higher-S/N *H*-band image, found that a Sérsic + exponential model is a better fit for this galaxy than just a Sérsic function by itself.)

CHAPTER

THREE

MAKEIMAGE AND PSF IMAGES

Makeimage can be used with PSF images to generate properly convolved model images, using the same `--psf` option that imfit uses. E.g.

```
makeimage bestfit_parameters_imfit.dat --refimage=ic3478rss_256.fits
  --psf=psf_moffat_51.fits
```

Makeimage can also be used to *generate* PSF images; in fact, the PSF image we used above was generated using the "config_makeimage_moffat_psf.dat" configuration file, which is included in the examples subdirectory (note that this file includes directives specifying the size of the output image, so the `--refimage` option isn't necessary in this case). A model PSF image can be constructed using any combination of the image functions that imfit and makeimage know about – Gaussian, Moffat, the *sum* of Gaussians and Moffats, etc.

3.1 Chi-Squared and All That: Using Different Fit Statistics

Fitting a model to an image involves some assumptions about the underlying *statistical* model that generated your data – i.e., what kind of statistical distributions the individual pixel values are drawn from. This in turn affects how the "fit statistic" – the quantity you are trying to minimize in order to get the best fit – is calculated.

By default, imfit uses a "data-based" 2 approach, which assumes that individual pixel values are drawn from the Gaussian approximation of a Poisson distribution. To compare a model pixel value to the data value, we assume that the Gaussian distribution has a mean equal to the model value, with the dispersion equal the square root of the *data* value. (If you provide a read-noise value, this is added in quadrature to the data-based dispersion.)

One alternative is to take the dispersion from the square root of the (current) *model* value, which you can do with the `--model-errors` flag:

```
imfit ic3478rss_256.fits -c config_sersic_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --gain=4.725 --readnoise=4.3
  --sky=130.14 --psf psf_moffat_51.fits --model-errors

Reduced Chi^2 = 1.075389
AIC = 68216.271136, BIC = 68279.672995

X0      128.9250 # +/- 0.0127
Y0      129.0750 # +/- 0.0171
FUNCTION Sersic
PA      19.0862 # +/- 0.247458
ell     0.227161 # +/- 0.00175713
n       2.59104 # +/- 0.0111591
```

(continues on next page)

(continued from previous page)

```
I_e      17.9857 # +/- 0.167361
r_e      63.6443 # +/- 0.360108
```

The result is not dramatically different, though both *n* and *r_e* are slightly larger and *I_e* is slightly smaller; this is expected due to the differing biases which apply to the data-based and model-based approaches (see Erwin 2015 and references therein).

You can *also* tell imfit to use an external "noise" or "error" map – an image whose pixel value are standard deviations, perhaps produced by a data pipeline. In this case, you use the `--noise` option to specify the corresponding FITS file. (If your noise/error map has units of *variance*, you can add the `--errors-are-variances` flag to tell imfit this.)

Finally, you can abandon the 2 Gaussian statistical model entirely and assume that your data comes from a pure Poisson process (rather than the Gaussian approximation of one). This involves a "Poisson maximum-likelihood ratio" (Poisson MLR) approach, and is especially appropriate for data with very low counts per pixel, where the Gaussian approximation really breaks down. Imfit allows you to do with the `--poisson-mlr` flag (or just `--mlr` for short):

```
imfit ic3478rss_256.fits -c config_sersic_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --gain=4.725 --sky=130.14
  --psf psf_moffat_51.fits --mlr

Reduced Chi^2 equivalent = 1.104470
AIC = 70060.584150, BIC = 70123.986009

X0       128.9218 # +/- 0.0146
Y0       129.0796 # +/- 0.0173
FUNCTION Sersic
PA       19.0826 # +/- 0.244875
ell      0.227176 # +/- 0.00173874
n        2.55157 # +/- 0.00999606
I_e      18.6469 # +/- 0.162048
r_e      62.1518 # +/- 0.331032
```

(Note that we leave off the `--readnoise` option, because the pure-Poisson approach cannot handle separate read-noise components. In most cases, this be done without affecting the fit in any significant way.)

The result is a fit which is in between the two 2 alternatives, though closer to the model-based approach. (Again, this is consistent with what we would expect from the different statistical models being used, with the pure-Poisson approach being the most unbiased.)

(See Erwin 2015 for more on the statistical background and the corresponding biases.)

3.2 Parameter Uncertainties and Correlations: Bootstrap and MCMC

As you probably noticed, part of the output of imfit is a set of 1-sigma parameter uncertainties for each fitted parameter in the model. These are automatically generated when using the default (Levenberg-Marquardt) minimizer. They're not usually all that accurate, they assume the uncertainties are all symmetric, and they don't provide any information about possible correlations or anti-correlations between different parameter values.

If you would like a better picture of what the parameter uncertainties and possible correlations are like, there are two options: one fast but noisy, the other slow but detailed:

1. **Boootstrap resampling**: This involves generating a new version of the data image by sampling from the original image with replacement (ignoring masked pixels) and re-running the fit. Do this several hundred (or ideally

several thousand) times, and you get a distribution of parameter values that can approximate the likelihood (e.g., the 2).

2. **Markov chain Monte Carlo (MCMC) analysis**: This involves computing Markov chains consisting of sequences of sets of parameter values. After an initial "burn-in" period, the distribution of points in parameter space represented by a chain should converge to something proportional to the likelihood. (The particular algorithm used by Imfit actually runs multiple chains in parallel.)

CHAPTER

FOUR

BOOTSTRAP RESAMPLING EXAMPLE

To save time, we'll use the model *without* PSF convolution (you can of course use PSF convolution with bootstrap resampling; it will just take longer):

```
imfit ic3478rss_256.fits -c config_sersic_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --gain=4.725 --readnoise=4.3
  --sky=130.14 --bootstrap 500 --save-bootstrap=bootstrap_output.dat
```

This will do the fit as before, print the result, and then start doing 500 rounds of bootstrap resampling and fits to the resampled data. When it's done (this takes about 30 seconds on a 2012 MacBook Pro with a quad-core CPU) it will print out a summary of the best-fit parameter values and their uncertainties; it will also save all 500 sets of parameter values in the file bootstrap_output.dat.

This file has one column per parameter; the column names are the parameters with numbers appended (e.g., `X0_1`, `n_1`) to 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. (I.e., all parameters for the first function will have `_1` appended, all parameters from the second will have `_2` appended, etc.)

In the `python/` subdirectory of the main Imfit package there are a couple of Python modules: imfit_funcs.py and imfit.py. The latter has a simple function to read in the bootstrap-resampling output file (`imfit.GetBootstrapOutput`), which will return a list of parameter names and a 2D Numpy array with the full set of parameter values.

There are many possible ways of analyzing the bootstrap-resampling output. 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 that also uses the `imfit.GetBootstrapOutput` function:

```
>>> import imfit, corner

>>> columnNames, bootstrapResults =
    imfit.GetBootstrapOutput("bootstrap_output.dat")
>>> corner.corner(bootstrapResults, labels=columnNames)
```

The result is shown below.

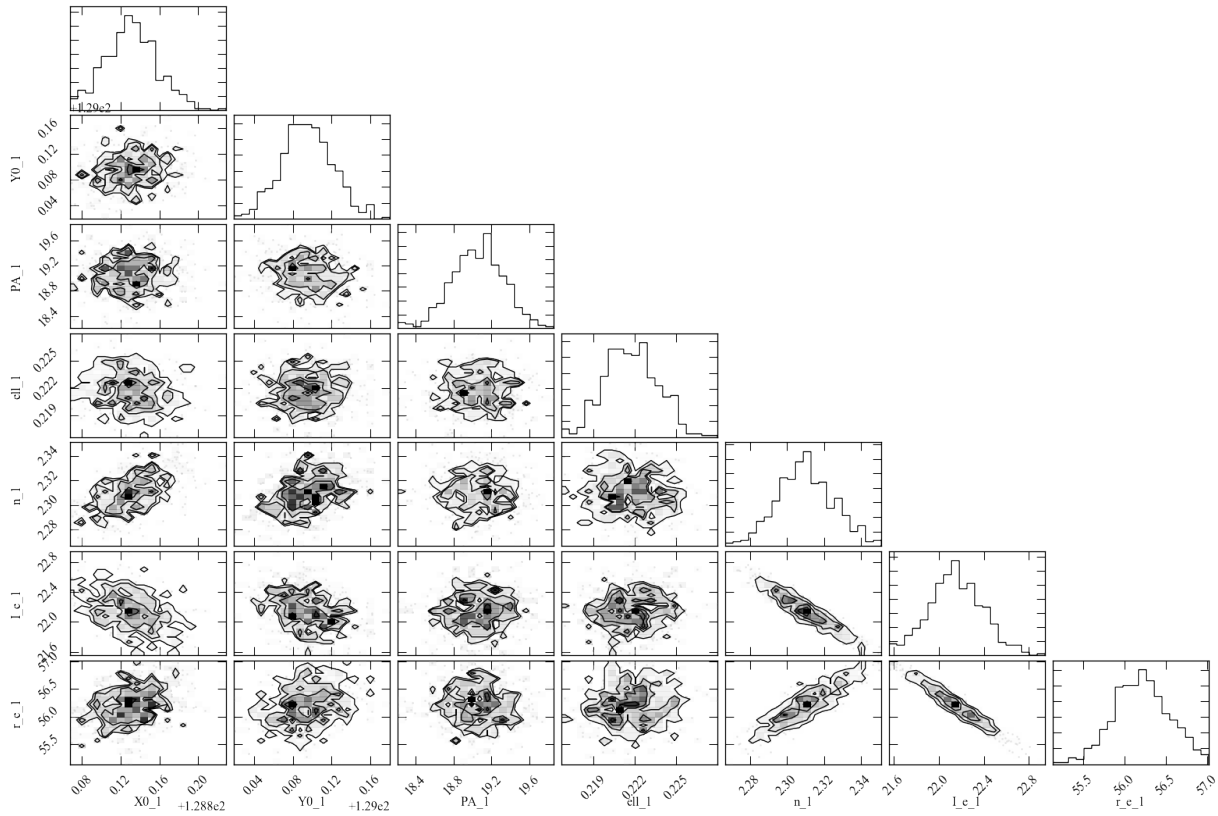


Figure 4: Scatterplot matrix of parameter values from 500 rounds of bootstrap resampling fits to the IC 3478 *r*-band image (Sérsic model, no PSF convolution). Note the clear correlations between the Sérsic model parameters (n, r_e, I_e).

CHAPTER

FIVE

MCMC EXAMPLE

MCMC analysis uses a separate program called `imfit-mcmc`. You can run it with the following command (note that it's identical to the regular `imfit` command, except for the option that specifies the root name for output files):

```
imfit-mcmc ic3478rss_256.fits -c config_sersic_ic3478_256.dat
  --mask ic3478rss_256_mask.fits --gain=4.725 --readnoise=4.3
  --sky=130.14 --output=mcmc_ic3478r
```

Warning: this will take several minutes! (On my 2012 MacBook Pro with a quad-core Intel i7 CPU, it takes about eight or ten minutes.)

Various updates will be printed as the program runs. Once a trial "burn-in" phase is over, `imfit-mcmc` will test for possible convergence of the chains every 5,000 generations by looking at the last half of each chain. If convergence is detected, the program will quit; otherwise, it will quit when it reaches 100,000 generations. (These values can be changed with command-line options.)

When it's done, you will have *seven* output text files, named mcmc_ic3478r.1.txt, mcmc_ic3478r.2.txt, etc., one for each of the individual chains. (By default, the total number of chains is equal to the number of free parameters in the model.) Each is similar to the bootstrap-resampling output file in format, with one column for each parameter in the model (plus some extra bookkeeping columns that you can ignore unless you're interested in details of the MCMC process), and one row for each generation in the chain; each chain will have several tens of thousands of generations.

The ideal thing to do is probably to take the last half of each chain and combine them all into one gigantic set of parameter values. There's a Python function for that in python/imfit.py, which returns the same kinds of output as imfit.GetBootstrap (i.e., a list of parameter names and a 2D Numpy array). Here's an example of using that, and then making a scatterplot matrix with the corner.py module, just as we did for the bootstrap output:

```
>>> import imfit, corner

>>> columnNames, allchains = imfit.MergeChains("mcmc_ic3478r",
    secondHalf=True)
>>> corner.corner(allchains, labels=columnNames)
```

The result is shown below.

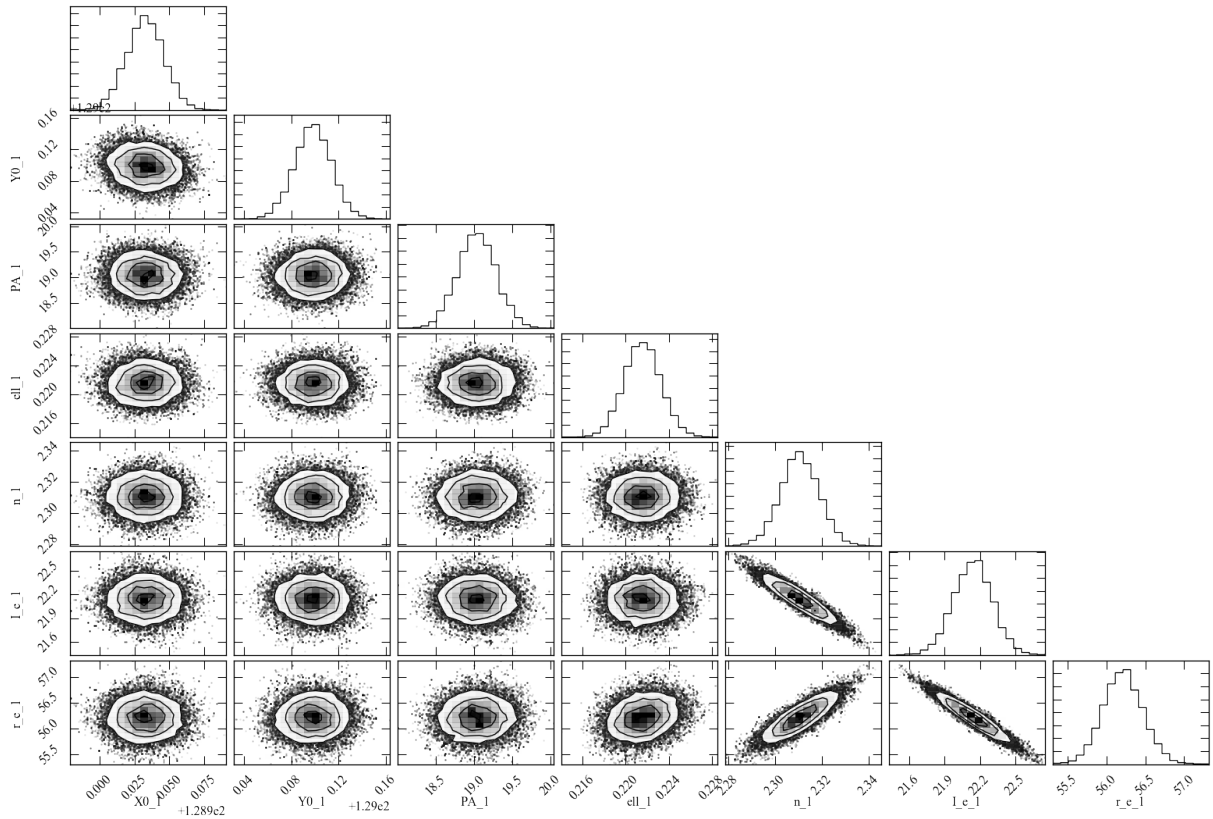


Figure 5: Scatterplot matrix of parameter values from Markov chain Monte Carlo analysis of the IC 3478 *r*-band image (Sérsic model, no PSF convolution). Note the strong correlations between the Sérsic model parameters (n, r_e, I_e), and the weaker correlation between r_e and ellipticity and between X0 and Y0. Since this plot is based on about 300,000 samples, it is considerably less noisy than the version based on 500 rounds of bootstrap resampling in *Figure 4*.

CHAPTER

SIX

CONFIGURATION FILE FORMAT

The `imfit`, `imfit-mcmc`, and `makeimage` programs *always* require a configuration file ("config file" for short): a text file which describes the model to be fit to or compared with the data (in the case of imfit or imfit-mcmc) or to be generated (in the case of makeimage), plus some optional information about the image itself.

Note that you can use the `--sample-config` command-line option with `imfit` or `makeimage` to have an example of a config file saved to the current directory; this can then be edited to taste and used with the appropriate program.

This page provides a basic description of the configuration file format.

(In what follows, I generally use "imfit" to refer to both `imfit` *and* `imfit-mcmc`, unless otherwise noted.)

6.1 Blank lines and Comments

Any blank lines in the file are ignored. Any lines beginning with the hash/pound symbol "#" are also ignored.

Finally, if a hash symbol is found in the middle of a line, then it and the rest of the line are ignored. This allows comments to be added on the same line as a meaningful entry (function declaration, parameter name and value, etc.).

(There is one minor exception to that last point, having to do with optional labels for funtions; see below.)

6.1.1 (Optional) Prelude: Describing the Image

The configuration file can start with a prelude which provides information about the data image (for imfit) or the output model image (in the case of makeimage). This is in the form of lines containing single "NAME value" pairs, e.g.

```
GAIN 4.5
```

These do not *have* to be provided, and you can provide as many or as few of these as are needed (the exception being the NCOLS and NROWS entries for a makeimage configuration file – both of those *should* be provided unless you know you will be supplying makimage with relevant info on the command line, via the `--nrows` and `--ncols` or `--refimage` options).

The allowed entries for an imfit configuration file are:

- GAIN – the A/D gain value (electrons/count) for the image
- READNOISE – any Gaussian read noise (electrons)
- EXPTIME – total integration time *if* pixel values are counts/sec
- NCOMBINED – number of images combined *if* pixel values are mean or median
- ORIGINAL_SKY – any original constant value (e.g., sky background) which has already been subtracted from the image

The allowed entries for a makeimage configuration file are:

- NCOLS – integer defining the width (number of columns) of the output image
- NROWS – integer defining the height (number of rows) of the output image

All of this information can *also* be provided via command-line options, which if present will override any corresponding values in the configuration file.

6.1.2 Main Section: Defining the Model

The main (and required) section of the configuration file is the part which describes the actual model to be fit (or generated and saved in the case of makeimage). This consists of one or more **function sets**.

6.2 Function Sets

A **function set** is a central-coordinate (x0,y0) specification followed by one or more **image-function declarations**. This specifies a set of image functions which share the same center.

```
X0  <initial_value>   [<limits>]
Y0  <initial_value>   [<limits>]
FUNCTION <function_name>
<parameter specifications ...>
```

The optional limit specifications for X0 and Y0, which are only used by imfit, can have one of two forms:

1. The word "fixed", which indicates that the parameter should be held fixed. E.g.

   ```
   X0  101.5   fixed
   ```

2. A comma-separated pair of numbers specifying lower and upper limits, e.g.

   ```
   X0  101.5    98.0,103.5
   ```

6.3 Image-Function Declarations

An **image-function declaration** consists of the word "FUNCTION" followed by the name of one of Imfit's image functions. (Recall that you can use `--list-functions` to get a list of the available image functions and `--list-parameters` to get that list along with the parameter list for each function.) This is followed by the list of parameter values; these are the values for generating a model image with makeimage, or the *initial* values for the fitting process in imfit.

```
FUNCTION <function_name>
<parameter_name1>    <initial_value>   [<limits>]
<parameter_name2>    <initial_value>   [<limits>]
[etc.]
```

Optionally, you can include an arbitrary text label for a given function on the same line as the world "FUNCTION" by adding `# LABEL <some text>` after the function name, e.g.:

```
FUNCTION <function_name>  # LABEL <some label text>
```

(These labels do not affect the fitting, but are copied to the output; they are for allowing the user to more easily keep track of intended meanings of different components in complicated models.)

A simple example:

```
FUNCTION Exponential  # LABEL main disk
PA    90
ell   0.5
I_0   100
h     15
```

The optional limit specifications for the parameters are exactly like those for the X0 and Y0 values (see above):

1. The word "fixed", which indicates that the parameter should be held fixed; or
2. A comma-separated pair of numbers specifying lower and upper limits.

Note that parameter-limit specifications are actually *required* by imfit-mcmc. They are also required when using imfit's Differential Evolution solver; they are optional for the other solvers (including the default Levenberg-Marquardt solver). Limit specifications are *ignored* by makeimage.

6.3.1 Examples

6.4 Single-function image

This is based on configuration file in the `examples/` subdirectory, where it is meant to be used to fit a single Sersic function to a 256 by 256-pixel cutout of an SDSS image. If used with makeimage, it generates a 2D Sersic-function image using the specified parameters. This example does *not* contain gain or read-noise specifications. Lower and upper limits are listed for all parameters for use with imfit.

```
X0   129.0    125,135
Y0   129.0    125,135
FUNCTION Sersic
PA    18.0    0,90
ell    0.2    0,1
n      1.5    0,5
I_e    15     0,500
r_e    25     0,100
```

6.5 Single function set with two functions

This is a modification of the previous configuration file, using an Exponential function along with the Sersic function. In addition, the Sersic index n is held fixed with a value of 4 (making the Sersic profile a de Vaucouleurs profile). Both functions share the same center, and are thus part of a single function set. This version also includes an image-description prelude.

```
GAIN          4.725
READNOISE     4.3
ORIGINAL_SKY  130.14

X0   129.0    125,135
Y0   129.0    125,135
```

(continues on next page)

(continued from previous page)

```
FUNCTION Sersic
PA     18.0     0,90
ell     0.2     0,1
n       4       fixed
I_e     15      0,500
r_e     25      0,100
FUNCTION Exponential
PA     18.0     0,90
ell    0.5      0,0.8
I_0    100      1,500
h      50       5,500
```

6.6 Multiple function sets

Multiple function sets can be included in a configuration file; these indicate different sets of image functions which share common centers (i.e, x0,y0 locations on the image).

A simple example, modifying the previous example by including a Sersic function representing a neighboring galaxy located approximately 110 pixels away in the X direction and 45 pixels away in Y:

```
GAIN           4.725
READNOISE      4.3
ORIGINAL_SKY   130.14

X0    129.0    125,135
Y0    129.0    125,135
FUNCTION Sersic
PA     18.0     0,90
ell     0.2     0,1
n       4       fixed
I_e     15      0,500
r_e     25      0,100
FUNCTION Exponential
PA     18.0     0,90
ell    0.5      0,0.8
I_0    100      1,500
h      50       5,500

X0    240.0    235,245
Y0    183.0    180,186
FUNCTION Sersic
PA     -40.0    -10,-60
ell     0.5     0,1
n       1       0.5,2.0
I_e     5      0,520
r_e     10      0,20
```

6.6.1 Using Imfit Output Files with Makeimage

When `imfit` successfully fits a model to an image, it saves the best-fitting parameters to an output file (by default this file is called `bestfit_parameters_imfit.dat`). This file has the same basic format as a config file, and can in fact be used as a config file by `makeimage` (though it will be missing the `NCOLS` and `NROWS` parameters, so you will have to add those to the file or else specify them with command-line options).

An `imfit` best-fit output file can even be used as input to another invocation of `imfit` itself, though it will lack any prelude parameters describing the data image (`GAIN`, etc.) and any parameter limits or "fixed" specifications.

6.6.2 Quick and Dirty Generation of Config Files

As noted above, you can always generate a bare-bones sample config file using the `--sample-config` command-line option.

Calling `imfit` or `makeimage` with the `--list-parameters` option will print a list of all the functions and their parameters. You can copy and paste the relevant parts of this output into a config file to make function entries (aside from needing to fill in the initial values and possible limits, of course!).

CHAPTER

SEVEN

GENERAL NOTES AND ADVICE FOR FITTING IMAGES

WARNING: this page is currently an incomplete work in progress!

7.1 Pixel values and uncertainties

It is important to have some idea of the underlying statistical model for your data.

The basic assumption is that the pixel values in a data image are the outcome of some statistical sampling process (usually Poisson or Gaussian) operating on an underlying model of the intensity distribution.

A very simple example of an underlying model might be a constant sky background plus an elliptical 2D Sersic function representing a galaxy. Sampling of this by a 2D detector results in pixel values which are a combination of the image model and a sampling model for the distribution of individual intensity value recorded for each pixel.

If your data processing pipeline has produced some kind of error image (sigma or variance values) which you trust, then you can go ahead and tell imfit about the error image (via the `--noise` command-line option, plus the `--errors-are-variances` flag if the pixel values in the error image are variances instead of sigmas).

For the common case where you do *not* have an error image, you can have imfit estimate the per-pixel uncertainties for you, either from the data image or from the model. The question is then whether to use proper Poisson statistics or the usual Gaussian approximation of Poisson statistics (i.e., sigma ~ sqrt(intensity).

If the original count levels (including background) are high (say, greater than 100 photo-electrons per pixel), then you can probably assume the Gaussian approximation of Poisson statistics is OK, and use some variant of chi^2 as the fit statistic.

7.1.1 IMPORTANT NOTE: Converting your pixel values to photo-electrons (or particles, or. . .)

In order for imfit to correctly estimate the per-pixel uncertainties, it is *very important* that the pixel values be convertible to units of the original detected quantities – i.e., integrated photo-electrons for a typical detector, or particles/pixel for an N-body simulation.

Since the actual recording of an astronomical image almost always involves conversion of detected photo-electrons to counts (also called "data numbers" or ADUs) via an A/D converter, the output image is already no longer in the proper units.

So you must tell imfit how to convert the pixel values back to the original values, at least in a general, image-wide sense. (It's probably overkill to worry about the effects of flat-fielding.)

The simple way to do this is to give imfit an effective gain factor, via the GAIN parameter in the config file, or the `--gain` command-line option. The effective gain is whatever number will, when multiplied by the image's pixel values, convert them back to, e.g., photo-electrons. For a processed (but not photometrically calibrated) image, where the pixel values

are in ADUs, this is simply the gain of the original A/D conversion, in units of electrons/ADU. Imfit will then multiply the pixel values by the GAIN parameter to turn them back into photo-electrons. (Note that some data-taking setups will record an inverse "gain" in units of ADU/electrons; you will need to invert this before passing it to imfit!)

If your image has photometrically calibrated flux units (Jy, nanomaggies, whatever), then the GAIN parameter must be a number that will convert these back to photo-electrons. If an image happens to be in units of magnitudes per square arc second, then it must be converted to *linear* intensity values first.

7.2 Special Notes for SDSS Images

7.2.1 DR7 (and earlier) Images

DR7 images are fairly straightforward; the only real potential for confusion is with the background level.

DR7 images do *not* have background subtraction applied (a crude estimate of the background level is included in the header of each image). However, each image has an artifical "soft bias" level of 1000 added to each pixel. Thus, if an image has a mean background level of 1210 counts/pixel, the *actual* observed sky background is only 210 counts/pixel. The additional constant value of 1000 should be removed *and not included in any sky-background levels input to imfit*.

Thus, if you determine that a particular image has a mean background level of, say, 1210.7 counts/pixel, you can either:

- Subtract 1210.7 from the image, and use 201.7 (*not* 1210.7) as the `ORIGINAL_SKY` (or `--sky` commandline option) parameter for imfit;
- OR: Subtract 1000 from the image, then include a background component (e.g., FlatSky) in the model with initial value = 210.7 (possibly fixed to that value, if you don't want imfit to vary the background).

The A/D gain values are included in the tsField FITS tables that go with each field; typical values can also be found in the table at the bottom of this page

7.2.2 DR 8+ Images

Images that are part of DR8 and later releases are more problematic, because they have been preprocessed in slightly complicated and potentially confusing ways.

First, the pixel values have been photometrically calibrated and transformed into flux values – specifically, "nanomaggies" (units of 3.631e-6 Jy). To fit the images properly, you will need to convert these back to counts, or else combine the flux conversion with the A/D gain to make an "effective gain" parameter as input to imfit.

The photometric calibration is recorded in the image header using the NMGY header keyword. You can convert the image back to ADUs via

```
image_ss_counts = image_ss_nmgy / NMGY
```

Second, a 2D model of the background is computed and subtracted from the image as part of the pipeline, so you will definitely need to include some approximation of the background value as an `ORIGINAL_SKY` parameter in the configuration file (or via the `--sky` command-line option). The 2D background model is included in the image files in a somewhat obscure fashion, as a series of floating-point values for interpolation, stored as a table in extension 2 of the An additional problem is that these sky-model values are in counts, *not* in the nanomaggy units of the processed data image.

An estimate of the original sky background can be obtained by averaging the interpolation values in the second extension. For example, using `numpy` and the `astropy.io.fits` module in Python:

```
>>> import numpy as np
>>> from astropy.io import fits
>>> hdu_list = fits.open('<path-to-SDSS-image-file>')
>>> sky_bintable = hdu_list[2]
>>> np.mean(sky_bintable.data['ALLSKY'])
```

The simplest approach is probably to convert the image from nanomaggies to counts, then use the standard A/D gain values and the mean sky value from the second FITS extension.

Alternately, you could combine the NMGY value with the gain to get an effective-gain value that converts the nanomaggie values directly to photo-electrons ("gain" = A/D gain / NMGY) – but then you will have to convert the sky value from counts to nanomaggies.

CHAPTER

EIGHT

HOW TO WRITE NEW IMAGE FUNCTIONS

8.1 Overview

Imfit comes with a variety of 2D surface-brightness functions ("image functions"). But it is designed to make it relatively simple to add *new* image functions, to describe objects or substructures which are not well modeled by the existing functions. This is done by writing additional C++ code and re-compiling the various Imfit programs (`imfit`, `imfit-mcmc`, `makeimage`) to include the new function or functions.

This page provides some guidelines on how to write new image functions (see also the "Rolling Your Own Image Functions" section of the Imfit manual, from which this page is excerpted). Even if you're not very familiar with C++, it should hopefully not be too difficult to write a new image function, since it can be done by copying and modifying one of the existing image functions.

A new image function is written in C++ as a subclass of the FunctionObject base class, which is declared and defined in the `function_object.h` and `function_object.cpp` source-code files (in the `function_objects/` subdirectory of the source-code distribution).

A new image-function class should provide its own implementation of (i.e., "override") the following public FunctionObject methods, which are defined as virtual methods in the base class:

- The class constructor – in most cases, the code for this can be copied from any of the existing FunctionObject subclasses, unless some special extra initialization is needed.
- `Setup()` – this is used by the calling program to supply the current set of function parameters (including the ($x0$,$y0$) pixel values for the center of the function set) prior to determining intensity values for individual pixels. The parameter values in the input array should be assigned to the appropriate data members of the class. This is also a convenient place to do any general calculations which depend on the parameter values but don't depend on the exact pixel (x,y) values.
- `GetValue()` – this is used by the calling program to obtain the surface brightness for a given pixel location (x,y). In existing FunctionObject subclasses, this method often calls other (private) methods to handle details of the calculation.
- `GetClassShortName()` – this is a class function which returns the short version of the class name as a string.

The new class should also redefine the following internal class constants:

- `N_PARAMS` — the number of input parameters (*excluding* the central pixel coordinates);
- `PARAM_LABELS` — a vector of string labels for the input parameters;
- `FUNCTION_NAME` — a short string describing the function;
- `className` — a string (no spaces allowed) giving the official name of the function.

The `add_functions.cpp` file should then be updated by:

- including the header file for the new class;

- adding 2 lines to the `PopulateFactoryMap()` function to add the ability to create an instance of the new class. (Look for the comment line "// ADD CODE FOR NEW FUNCTIONS HERE" in the file.)

Finally, the name of the C++ implementation file for the new class should be added to the `SConstruct` file to ensure it gets included in the compilation; the easiest thing is to add the file's name (without the `.cpp` suffix) to the multi-line `functionobject_obj_string` string definition. (Or look for the comment line "# ADD CODE FOR NEW FUNCTIONS HERE" in the `SConstruct` file.)

Existing examples of FunctionObject subclasses can be found in the `function_objects/` subdirectory of the source-code distribution, and are the best place to look in order to get a better sense of how to implement new FunctionObject subclasses.

8.2 A Simple Example

To demonstrate the basics of writing a new image function, we'll modify the existing Gaussian class to make a class called NewMoffat, which produces an elliptical structure with a Moffat radial profile. (Such a function already exists in Imfit under the name Moffat, so this is really a redundant exercise.)

Three basic changes to the existing `func_gauss.h/cpp` files are needed:

1. Change the class name (in this case, from "Gaussian" to "NewMoffat");
2. Change the code which actually computes the function;
3. Add, rename, and delete class data members to accommodate the new algorithm.

8.2.1 1. Create and Edit the Header File

Assuming we're in the Imfit source code directory, `cd` to the `function_objects/` subdirectory, copy the header file `func_gaussian.h` to `func_new-moffat.h`, and edit the file to change the following lines:

```
#define CLASS_SHORT_NAME "Gaussian"

class Gaussian : public FunctionObject

Gaussian( );
```

to

```
#define CLASS_SHORT_NAME "NewMoffat"

class NewMoffat : public FunctionObject

NewMoffat( );
```

8.2.2 2. Create and Edit the Class File

Copy the file `func_gaussian.cpp` to `func_new-moffat.cpp`.

A. Change the following lines in the beginning of the file:

```
#include "func_gaussian.h"

const int N_PARAMS = 4;
const char PARAM_LABELS[][20] = {"PA", "ell", "I_0", "sigma"};
const char  PARAM_UNITS[][30] = {"deg (CCW from +y axis)", "", "counts/pixel", "pixels"};
const char FUNCTION_NAME[] = "Gaussian function";
```

to

```
#include "func_new-moffat.h"

const int N_PARAMS = 5;
const char PARAM_LABELS[][20] = {"PA", "ell", "I_0", "fwhm", "beta"};
const char  PARAM_UNITS[][30] = {"deg (CCW from +y axis)", "", "counts/pixel",
                                 "pixels", ""};
const char FUNCTION_NAME[] = "Moffat function";
```

B. In the remainder of the file, change all references to the class name from `Gaussian` to `NewMoffat` (e.g., `Gaussian::Setup` becomes `NewMoffat::Setup`).

C. Change the `Setup` method. Here, you'll need to change how the input parameter array is converted into individual parameters, and do any useful pre-computations (i.e., computations that depend on the parameter values, but not on individual pixel values or values derived from the latter, like radius).

Change

```
PA = params[0 + offsetIndex];
ell = params[1 + offsetIndex];
I_0 = params[2 + offsetIndex];
sigma = params[3 + offsetIndex];
```

to

```
PA = params[0 + offsetIndex];
ell = params[1 + offsetIndex];
I_0 = params[2 + offsetIndex];
fwhm = params[3 + offsetIndex];
beta = params[4 + offsetIndex];
```

Then, at the end of the method, replaced this line

```
twosigma_squared = 2.0 * sigma*sigma;
```

with this (which computes the "alpha" parameter of the Moffat function)

```
double exponent = pow(2.0, 1.0/beta);
alpha = 0.5*fwhm/sqrt(exponent = 1.0);
```

D. Changes to the `CalculateIntensity` method:

Although it is the public method GetValue which is called by other parts of the program, we don't actually need to change the current version of that method in this example. The code in the original Gaussian version of GetValue converts pixel positions to a scaled radius value, given input values for the center, ellipticity, and position angle, and then calls the private method CalculateIntensity to determine the intensity as a function of the radius. Since we're still assuming a perfectly elliptical shape, we can keep the existing code. (GetValue also includes possible pixel subsampling, which is useful for cases where intensity changes rapidly one scales of a single pixel; we'll apply a simple modification for the Moffat function later on.)

So in this case we actually implement the details of the new function's algorithm in CalculateIntensity. Replace the original version of that method with the following:

```
double NewMoffat::CalculateIntensity( double r )
{
  double  scaledR, denominator;

  scaledR = r / alpha;
  denominator = pow((1.0 + scaledR*scaledR), beta);
  return (I_0 / denominator);
}
```

E. Changes to the `CalculateSubsamples` method:

Although pixel subsampling is performed in the GetValues method, the determination of whether or not to actually *do** the subsampling – and how much of it to do – is determined in CalcualteSubsamples.

For the Gaussian function, subsampling can be useful happen when $r < 1$ *and* sigma < 1. The equivalent for the Moffat function would be $r < 1$ and alpha < 1, so change the line in CalculateSubsamples that says

```
if ((sigma <= 1.0) && (r <= 1.0))
```

to say

```
if ((alpha <= 1.0) && (r <= 1.0))
```

At this point, most of the work is done. We only need to update the code in `add_functions.cpp` so it knows about the new function and update the `SConstruct` file so that the new function is included in the compilation.

8.3 Adding Parameter Unit Definitions

Imfit version 1.9 added the possibility of including strings describing the intended units for certain parameters of the image functions. This is purely cosmetic – the image functions ignore them – but it can be useful for the user as a reminder of what individual parameters mean. (Example: the Exponential image function has the units label "deg (CCW from +y axis)" for the PA (position angle) parameter, "counts/pixel" for I_0, and "pixels" for the scale length parameter h.)

You do not *have* to provide this information in any image functions you write. If you wish to, you need to properly define the `PARAM_UNITS` constant at the top of the source file (see 2.A above for an example); assign the unit-description strings (the empty string – "" – is used for an parameters without units) to the `parameterUnits` vector in the constructor (see any of the standard image functions for an example of how to do this), and set the `parameterUnitsExist` data member to `true` (again, see the constructor of any of the standard image functions for examples).

8.4 Other Potential Issues

If your new image function has an analytic expression for the total flux, then you might consider overriding the `CanCalculateTotalFlux` method to return `true` and then overriding the `TotalFlux` method so that it calculates and returns the total flux. (The default is to let `makeimage` estimate the total flux numerically, by generating a large image using the image function and summing all the pixel values.)

If your new image function is meant to represent the image *background* (as in the case of the built-in functions FlatSky and TiltedSkyPlane), then you may not want `makeimage` trying to calculate the "total flux" for the component. In this case, you should add a line to your class's constructor which sets the data member `isBackground` to `true` (as is done in, e.g., `func_flatsky.cpp`).

CHAPTER

NINE

IMFIT FREQUENTLY ASKED QUESTIONS

This is preliminary version of a page for assorted (potentially) Frequently Asked Questions about using Imfit.

9.1 How do I include a true point source (star, AGN, etc.) in a model?

Although point sources can be approximated by something like a Gaussian function with a very small size (e.g, 0.1 pixels), the best way to include point sources is with the PointSource function:

```
FUNCTION PointSource
I_0     <initial value  [limits]>
```

This uses the user-supplied PSF image, shifting it (via bicubic interpolation) and scaling its brightness using the `I_0` parameter (which corresponds to the total flux of the source).

If you are worried about possible rotation of the PSF for different positions within the image plane, you can use the PointSourceRot function, which adds a position-angle parameter.

9.2 How can I determine B/T (bulge/total) and other flux ratios?

Assuming you have *either* an `imfit`/`makeimage` config file *or* the best-fit-parameters output file produced by running `imfit` (which has the same general format), you can run `makeimage` in "–print-fluxes" mode, which will compute the fluxes of the individual components and their fractions of the total flux of the model:

```
$ makeimage config_or_bestfit_file.dat --print-fluxes
```

(In versions earlier than 1.9, you should also supply the "–nosave" option, which tells `makeimage` to skip saving the model image; this is the default in version 1.9 and later.)

The output will look something like this:

```
Component                Flux        Magnitude  Fraction    Label
Sersic                   1.0800e+06      ---     0.09038
Exponential              4.8448e+06      ---     0.40545    nuclear disk
Sersic_GenEllipse        5.6323e+05      ---     0.04713    bar?
Exponential              5.4612e+06      ---     0.45703    disk

Total                    1.1949e+07
```

Note that the flux units are in counts (i.e., the units of individual pixel values, summed over all pixels). To get outputs in magnitudes as well, see the next question. (The "Label" entries take their names from the optional "# LABEL " comments in the config/best-fit-parameters file.)

By default, `makeimage` computes component fluxes by generating a 5000 x 5000 pixel internal image and summing up the pixel values (a few functions – PointSource, Gaussian, Exponential, and Sersic – are able to compute their total fluxes analytically, which is faster). Should you be working with images larger than this size – or with models whose flux will extend significantly outside the default size – you can specify a larger (or smaller) summation image size with the "–estimation-size=", where the specified value is the size of the (square) image.

Finally, fluxes for "background" image functions (e.g., FlatSky), which are unbounded and would just scale with the summation image size, are treated as zero.

9.3 How can I determine magnitudes of model components?

Imfit works in units of counts/pixel, so it does not by default worry about magnitude systems, zero points, etc. However, you can ask the `makeimage` program, as part of the "–print-fluxes" mode, to compute magnitudes for individual components (and the whole model):

```
$ makeimage config_file.dat --print-fluxes --zero-point=<ZP>
```

The "ZP" value will be used to compute the magnitude of each component, as

```
m = ZP - 2.5 log10(flux)
```

The output will look something like this (using "–zero-point=25"):

```
Component                Flux         Magnitude  Fraction    Label
Sersic                   1.0800e+06    9.9164     0.09038
Exponential              4.8448e+06    8.2868     0.40545    nuclear disk
Sersic_GenEllipse        5.6323e+05   10.6233      0.04713    bar?
Exponential              5.4612e+06    8.1568     0.45703    disk

Total                    1.1949e+07    7.3066
```

CHAPTER

TEN

IMFIT DESIGN AND API

10.1 Design and Architecture of Imfit

10.1.1 General design and operation of imfit

The basic operation of imfit, as implemented in imfit_main.cpp, is:

1. Process command-line options
2. Read and process configuration file
    1. Optional data-image characteristics (A/D gain, read noise, original-sky background)
    2. Specifications of the model: functions, initial parameter values and limits
3. Read in user-supplied data from images
    1. Image to be fit
    2. Mask image, if any
    3. Noise/error/weight image, if any
    4. PSF image(s), if any
4. Create ModelObject instance and supply it with data
    1. List of image functions to use
        1. The ModelObject instance will then instantiate corresponding FunctionObject instances
    2. PSF image data (if supplied by user)
    3. Image to be fit
    4. Data image characteristics
    5. Oversampled PSF image (if supplied by user)
    6. Mask image data (if supplied by user)
    7. Type of fit statistic to be calculated
    8. Noise/error/weight image data (if supplied by user)
    9. Call FinalSetup() method on the ModelObject instance
5. Do the fit
    1. Set up initial parameter vector and parameter-limits structure (mp_par structure)
    2. Call the user-specified solver (Levenberg-Marquardt is default)

6. Print summary of fit
7. Optionally, do bootstrap resampling to get parameter uncertainty estimates
8. Save results
    1. Save best-fitting parameter values
    2. Optionally, save best-fitting model image and/or residual image

10.1.2 General design and operation of makeimage

The operation of makeimage is similar to (but simpler than) imfit.

The key difference in terms of generating the model image is that the user must specify the *size* of the output image, since there is no data image to use as a reference. There are three ways to do this:

1. Command-line options (`--ncols`, `--nrows`)
2. Specification within the configuration file (lines beginning NROWS and NCOLS)
3. Use a reference image (`--refimage` command-line option)

The basic operation of makeimage, as implemented in makeimage_main.cpp, is:

1. Process command-line options
2. Read and process configuration file
    1. Specifications of the model: functions and parameter values
    2. Optional image-size specifications
3. Read in PSF image(s), if supplied
4. Create ModelObject instance and supply it with data
    1. Give it list of image functions to use
    2. PSF image data (if supplied by user)
    3. Dimensions of model image
    4. Oversampled PSF data (if supplied by user)
5. Create model image:
    1. Set up parameter vector
    2. Call CreateModelImage() method of ModelObject instance, passing in the parameter vector
6. Save the image
    1. Generate string vector of comments for image header
    2. Call SaveVectorAsImage() function (image_io.h).

10.1.3 The ModelObject Class

The heart of Imfit is the ModelObject class. This is instantiated once in each program, and holds (among other things) the following objects and data ("**imfit only**" denotes data used in imfit or imfit-mcmc, but not needed when used in makeimage)

- Model image pixel values
- Vector of FunctionObject instances which define the model
- Array of parameter values for the model
- Image pixel data for PSFs, if any
- Convolver objects(s) for PSF convolution
- **imfit only**: Data image pixel values
- **imfit only**: other characteristics of data image (size, gain, read noise, etc.)
- **imfit only**: Error image (either supplied by user or internally generated)
- **imfit only**: Mask image, if supplied by user
- **imfit only**: Internal weight image (from combination of error image and weight image).

Not all of these data members are initialized or used – for example, none of the data-related ("**imfit only**") members are used by makeimage, and the error image is not generated or used if a fit uses Poisson-based statistics instead of chi^2.

The main functionality of the class includes:

- Generating a model image using the vector of FunctionObjects and the current array of parameter values
- Generating individual-function model images (i.e., the `--output-functions` option of makeimage)
- Computing individual-function and total fluxes for current model
- Computing deviances vector between current model image and data image (for use by Levenberg-Marquardt solver)
- Computing the fit statistic (chi^2, etc.) from comparison of the current model image and the data image
- Printing current parameter values
- Generating a bootstrap resampling pixel-index vector when bootstrap resampling is being done

10.1.4 FunctionObject Classes

A model image is the sum of individual model images computed using single image functions, each of which is an instance of a FunctionObject subclass. For example, a model which is the sum of a central point source, a Sersic bulge, and an exponential disk would be the sum of individual images created with the PointSource, Sersic, and Exponential classes. Multiple instances of each class can be used.

Each image function is a subclass of the abstract base class FunctionObject. The main methods of this class are:

- Setup() – Called by ModelObject to pass in the current parameter vector at the beginning of the computation of a model image; this allows the image function to store the relevant parameter values and do any useful computations that don't depend on pixel position.
- GetValue() – Called by ModelObject once for each pixel in the model image, to pass in the current pixel values (x,y); the image function uses these to compute and return the appropriate intensity value for that pixel.

10.1.5 Constructing a model image

The actual generation of pixel values in the model image depends on the vector of FunctionObjects and the current array of corresponding parameter values. (And, of course, convolution with a PSF if that is requested.) The FunctionObjects vector contains instantiations of one or more classes (e.g., Gaussian, Exponential, Sersic, ExponentialDisk3D) which are subclasses of the abstract base class FunctionObject (defined in function_object.h).

When a model image is constructed, the first step is to call the Setup() method on each FunctionObject and pass in the corresponding parameter values. This enables the FunctionObject instances to do any initial computations which don't depend on actual location within the image.

The value of an individual pixel in the model image is obtained by iterating over the individual FunctionObjects, calling its GetValue() method with the current pixel coordinates (x,y), and adding up all the return values. This all takes place within a loop which iterates over all the pixels in the model image; this loop is wrapped in OpenMP directives to allow parallelization across multiple CPU cores.

10.2 ModelObject

10.2.1 Overview

The heart of Imfit is the ModelObject class. This holds information about an image model and can compute images using the model and the current set of model parameter values; it can also hold a data image and its associated information, and can compute chi^2 and and other statistics by comparing the model and data images.

This class is instantiated once in each program, and holds (among other things) the following objects and data ("**imfit only**" denotes data used in imfit or imfit-mcmc, but not needed when used in makeimage)

- Model image pixel values
- Vector of FunctionObject instances which define the model
- Array of current parameter values for the model
- Image pixel data for PSF(s), if any
- Convolver objects(s) for PSF convolution
- **imfit only**: Data-image pixel values
- **imfit only**: other characteristics of the data image (size, gain, read noise, etc.)
- **imfit only**: Error image (either supplied by user or internally generated)
- **imfit only**: Mask image, if supplied by user
- **imfit only**: Internal weight image (from combination of error image and mask image).

Not all of these data members are initialized or used – for example, none of the data-related ("**imfit only**") members are used by makeimage, and the error image is not generated or used if a fit uses Poisson-based statistics instead of chi^2.

The main functionality of the class includes:

- Generating a model image using the vector of FunctionObjects and the current array of parameter values
- Generating individual-function model images (i.e., the `--output-functions` option of makeimage)
- Computing individual-function and total fluxes for current model
- Computing deviances vector between current model image and data image (for use by Levenberg-Marquardt solver)
- Computing the fit statistic (chi^2, etc.) from comparison of the current model image and the data image

- Printing current parameter values in different formats
- Generating a bootstrap resampling pixel-index vector when bootstrap resampling is being done

10.2.2 API

[**Warning**: This is currently somewhat incomplete!]

Files: `core/model_object.h`, `core/model_object.cpp`

class **ModelObject**

Main class holding data, model information, and code for generating model images, computing chi^2, etc.

Public Functions

ModelObject()

Constructor.

virtual **~ModelObject**()

Destructor.

void **SetVerboseLevel**(int verbosity)

Set the verbosity level.

void **SetDebugLevel**(int debuggingLevel)

Set the debugging level (must be 0 [default] or larger).

void **SetMaxThreads**(int maxThreadNumber)

Specify the maximum number of OpenMP threads to use in computations; also sets maximum number FFTW threads for convolutions.

void **SetOMPChunkSize**(int chunkSize)

Sets the chunk size for OpenMP.

virtual int **AddFunction**(*FunctionObject* *newFunctionObj_ptr)

Adds a *FunctionObject* subclass to the model.

int **SetupPsfInterpolation**(int interpolationType = kInterpolator_bicubic)

Specify that PSF interpolation (by PointSource functions) will be used; causes an internal PsfInterpolator object of the appropriate subclass to be allocated and set up with previously supplied PSF data. This should only be called *after* AddPSFVector has been called to supply the PSF data.

virtual void **DefineFunctionSets**(vector<int> &functionStartIndices)

Given a vector of indices specifying the start of individual function sets, set up the internal fsetStartFlags array; also compute nParamsTot

virtual void **SetZeroPoint**(double zeroPointValue)

Set the internal zero-point (for printing flux values)

virtual double **CashStatistic**(double params[])

Computes and returns the Cash statistic for the current set of model parameters (computes model image and compares it with the data image)

virtual void **PrintDescription**()

Prints the number of data values (pixels) in the data image.

void **GetFunctionNames**(vector<string> &functionNames)

Adds the names of image functions in the model (calling each function's GetShortName method) to the input vector of strings

void **GetFunctionLabels**(vector<string> &functionLabels)

Adds the names of image functions in the model (calling each function's GetShortName method) to the input vector of strings

void **GetImageOffsets**(double params[])

Given an input array (all zeros), this method returns the array with locations corresponding to each function set's imageOffset_X0 and imageOffset_Y0 values properly set (if there are no image offsets, then the corresponding values will remain zero). Intended for use in bootstrap_errors.cpp.

virtual string **GetParamHeader**()

Prints all function and parameter names in order all on one line; e.g., for use as header in bootstrap-parameters output file.

virtual int **PrintModelParamsToStrings**(vector<string> &stringVector, double params[], double errs[], const char *prefix = "", bool printLimits = false)

Prints description of model (e.g., best-fit result) as strings to the input vector of string. Optionally, the lower and upper limits defined in parameterInfo are also printed, OR associated lower and upper error bounds in errs can be printed.

If errs != NULL, then +/- errors are printed as well (only if printLimits is false)

If prefix != NULL, then the specified character (e.g., '#') is prepended to each output line.

If printLimits == true, then lower and upper parameter limits will be printed for each parameter (or else "fixed" for fixed parameters)

virtual string **PrintModelParamsHorizontalString**(const double params[], const string &separator = "\)

Like PrintModelParamsToString, but prints parameter values all in one line to a string (*without* parameter names or limits or errors), which is returned. Meant to be used in printing results of bootstrap resampling (imfit) or MCMC chains (imfit-mcmc)

void **PrintImage**(double *pixelVector, int nColumns, int nRows)

Basic function which prints an image to stdout. Mainly meant to be called by PrintInputImage, PrintModelImage, and PrintWeights.

virtual void **PrintInputImage**()

Prints the input data image to stdout (for debugging purposes).

virtual void **PrintModelImage**()

Prints the current computed model image to stdout (for debugging purposes).

virtual void **PrintWeights**()

Prints the current weight image to stdout (for debugging purposes).

virtual void **PrintMask**()

Prints the input mask image to stdout (for debugging purposes).

virtual void **PopulateParameterNames**()

Note that this is usually called by AddFunctions() in add_functions.cpp.

int **GetNFunctions**()

Prints the total number of image functions (instances of *FunctionObject* subclasses) making up the model.

int **GetNParams**()

Prints the total number of parameters making up the model.

long **GetNDataValues**()

Prints the number of data values (pixels, masked or unmasked) in the data image.

virtual long **GetNValidPixels**()

Prints the number of *valid* (i.e., unmasked) data values (pixels) in the data image.

bool **HasPSF**()

Returns true if the model has a PSF image.

bool **HasOversampledPSF**()

Returns true if the model has one or more oversampled regions (with oversampled PSFs).

bool **HasMask**()

Returns true if a mask image exists.

virtual double ***GetModelImageVector**()

Returns a pointer to the model image (matching the data image in size if convolution is being done). If the model image has not yet been computed, returns NULL.

double ***GetExpandedModelImageVector**()

This differs from *GetModelImageVector()* in that it always returns the full model image, even in the case of PSF convolution (where the full model image will be larger than the data image!)

double ***GetResidualImageVector**()

Computes residual image (data - model) using current model image (usually best-fit image), and returns pointer to residual-image vector. Returns NULL if memory allocation failed.

double ***GetWeightImageVector**()

Returns the weightVector converted to 1/sigma^2 (i.e., 1/variance) form. Returns NULL if memory allocation failed.

double ***GetDataVector**()

Returns a pointer to the data image.

double **FindTotalFluxes**(double params[], int xSize, int ySize, double individualFluxes[])

Estimate total fluxes for individual components (and entire model) by integrating over a very large image, with each component/function centered in the image. Total flux is returned by the function; fluxes for individual components are returned in individualFluxes.

virtual int **UseBootstrap**()

Tells ModelObject1d object that from now on we'll operate in bootstrap resampling mode, so that bootstrapIndices vector is used to access the data and model values (and weight values, if any). Returns the status from *MakeBootstrapSample()*, which will be -1 if memory allocation for the bootstrap-indices vector failed.

virtual int **MakeBootstrapSample**()

Generate a new bootstrap resampling of the data (more precisely, this generate a bootstrap resampling of the data *indices*) Returns -1 if memory allocation for the bootstrap indices vector failed, otherwise returns 0.

Protected Functions

bool **CheckParamVector**(int nParams, double paramVector[])

Returns true if all values in the parameter vector are finite.

bool **CheckWeightVector**()

Returns true if all pixels in the weight vector are finite *and* nonnegative.

bool **VetDataVector**()

Returns true if all non-masked pixels in the image data vector are finite; returns false if one or more are not, and prints an error message to stderr. ALSO sets any masked pixels which are non-finite to 0.

More simply: the purpose of this method is to check the data vector (profile or image) to ensure that all non-masked pixels are finite. Any non-finite pixels which *are* masked will be set = 0.

10.3 FunctionObject (and Subclasses)

10.3.1 Overview

Each image function used in the various Imfit programs is a subclass of an abstract base class called FunctionObject. Subclasses are defined for specific 2D image functions, each of which computes pixel intensity values based on input parameters and pixel coordinates. An instance of the ModelObject class holds a std::vector of pointers to one or more FunctionObjects, and constructs a model image by iterating over this vector.

When a model image is being computed, the first step is to call each FunctionObject instance's Setup() method and pass in the current array of parameter values, as well as the central position ($x0$,$y0$) for the current function block. In practice, the entire parameter vector for the model is passed in, along with an integer offset telling the FunctionObject instance where to look for *its* particular parameter values.

Then the caller requests intensity values for individual pixels by repeatedly calling the GetValue() method and passing in the current pixel coordinates.

The main methods of this class are:

- Setup() – Called by ModelObject to pass in the current parameter vector at the beginning of the computation of a model image; this allows the image function to store the relevant parameter values and do any useful computations that don't depend on pixel position.
- GetValue() – Called by ModelObject once for each pixel in the model image, to pass in the current pixel values (x,y); the image function uses these to compute and return the appropriate intensity value for that pixel.

  Many FunctionObject subclasses have additional private methods that do most of the calculations for each GetValue() call.

Note that the base class includes variant virtual methods for possible *1D* subclasses, though Imfit does not use any of these.

10.3.2 API

Files: function_objects/function_object.h, function_objects/function_object.cpp

class **FunctionObject**

Virtual base class for function objects (i.e., 2D image functions)

Subclassed by BrokenExponential, BrokenExponential2D, BrokenExponentialBar, BrokenExponentialDisk3D, CoreSersic, DoubleBrokenExponential, EdgeOnDisk, EdgeOnDiskN4762, EdgeOnDiskN4762v2, EdgeOnRing, EdgeOnRing2Side, Exponential, ExponentialDisk3D, FerrersBar2D, FerrersBar3D, FlatBar, FlatExponential, FlatSky, Gaussian, GaussianExtraParams, GaussianRing, GaussianRing2Side, GaussianRing3D, GaussianRingAz, GenExponential, GenExponential2, GenSersic, LogSpiral, LogSpiral2, LogSpiral3, LogSpiralArc, LogSpiralExp, LogSpiralGauss, ModifiedKing, ModifiedKing2, Moffat, N4608Disk, NaNFunc, PointSource, PointSourceRot, Sersic, SimpleCheckerboard, TiltedSkyPlane, TriaxBar3D

Public Functions

FunctionObject()

Basic constructor: sets functionName and shortFunctionName.

inline virtual bool **HasExtraParams**()

Boolean function: returns true if function can accept optional extra parameters (default = false)

inline virtual int **SetExtraParams**(map<string, string>&)

Set optional extra parameters.

inline virtual bool **ExtraParamsSet**()

Boolean function: returns true if optional extra parameters have been set.

inline virtual bool **IsPointSource**()

Returns true if function models point sources (e.g., PointSource class)

inline virtual void **AddPsfData**(double *psfPixels, int nColumns_psf, int nRows_psf)

Tell point-source function about PSF image data.

inline virtual void **AddPsfInterpolator**(PsfInterpolator *theInterpolator)

Pass in pointer to PsfInterpolator object (for point-source classes only)

inline virtual string **GetInterpolationType**()

Returns string with name of interpolation type (point-source classes only)

inline virtual void **SetOversamplingScale**(int oversampleScale)

Sets internal oversamplingScale to specified value (for use in oversampled regions)

virtual void **SetSubsampling**(bool subsampleFlag)

Turn pixel subsampling on or off (true = on, false = off).

virtual void **SetZeroPoint**(double zeroPoint)

Used to specify a magnitude zero point (for *1D* functions).

virtual void **SetLabel**(string &userLabel)

Used to specify a string label for a particular function instance.

virtual void **Setup**(double params[], int offsetIndex, double xc, double yc)

Base method for 2D functions: pass current parameters into the function object, storing them for when *GetValue()* is called, and pre-compute useful quantities. The parameter array params contains *all* parameters for *all* components in the overall model; offsetIndex is used to select the correct starting point for *this* component's parameters.

virtual void **Setup**(double params[], int offsetIndex, double xc)

Base method for 1D functions: pass current parameters into the function object, storing them for when *GetValue()* is called, and pre-compute useful quantities. The parameter array params contains *all* parameters for *all* components in the overall model; offsetIndex is used to select the correct starting point for *this* component's parameters.

virtual double **GetValue**(double x, double y)

Base method for 2D functions: Compute and return actual function value at pixel coordinates (x,y). This will be called once per pixel.

virtual double **GetValue**(double x)

Base method for 1D functions: Compute and return actual function value at specified value of independent variable x.

inline virtual bool **IsBackground**()

Returns true if class can calculate total flux internally.

inline virtual bool **CanCalculateTotalFlux**()

Returns true if class can calculate total flux internally.

inline virtual double **TotalFlux**()

Returns total flux of image function, given most recent parameter values.

virtual string **GetDescription**()

Return a string containing function name + short description.

virtual string &**GetShortName**()

Return a string containing just the function name.

virtual string &**GetLabel**()

Return a string containing just the function label (will be "" if not set).

virtual void **GetParameterNames**(vector<string> &paramNameList)

Add this function's parameter names to a vector of strings.

virtual void **GetParameterUnits**(vector<string> &paramUnitList)

Add this function's parameter unit names (if they exist) to a vector of strings.

virtual void **GetExtraParamsDescription**(vector<string> &outputLines)

Return lines describing the user-set extra/optional parameters, if they exist.

virtual int **GetNParams**()

Get number of parameters used by this function.

Protected Attributes

int **nParams**

number of input parameters that image-function uses

Protected Static Attributes

static const char **shortFuncName**[]

Class data member holding name of individual class.

10.4 Convolver

10.4.1 Overview

This class holds data and functions relating to FFT-based convolution of a point-spread-function (PSF) image with an input image (e.g., the model image computed by ModelObject).

An instance of this class is set up and used by main's ModelObject instance to handle standard PSF convolution of the whole model image. In addition, if there are any oversampled regions, then each OversampledRegion instance will have its own Convolver instance to handle convolution with the appropriate (oversampled) PSF image.

10.4.2 Setup and Use

To set up a Convolver object, you first call its SetupPSF method and pass in the PSF image data, size, and whether the PSF should be normalized. Then you call the SetupImage method to tell the Convolver object about the size of the image that will be convolved with the PSF, and finally you call the DoFullSetup to tell the Convolver object to do the necessary allocations and FFT setup.

To use the Convolver object, you simply call its ConvolveImage method with the image array as input; the input image will be updated in place.

(In Imfit, this is all done within the ModelObject class.)

10.4.3 API

Files: `core/convolver.h`, `core/convolver.cpp`

class **Convolver**

Class for handling PSF convolution (stores PSF, performs convolutions with input model images)

Public Functions

Convolver()

Constructor for *Convolver* class.

~Convolver()

Destructor for *Convolver* class.

void **SetMaxThreads**(int maximumThreadNumber)

Set maximum number of FFTW threads.

User specifies maximum number of FFTW threads to use (ignored if not compiled with multithreaded FFTW library)

void **SetupPSF**(double *psfPixels_input, int nColumns, int nRows, bool normalize = true)

Supply PSF image to *Convolver* object.

Pass in a pointer to the pixel vector for the input PSF image, as well as the image dimensions and whether PSF needs to be normalized.

void **SetupImage**(int nColumns, int nRows)

Pass in the dimensions of the image we'll be convolving with the PSF.

int **DoFullSetup**(int debugLevel = 0, bool doFFTWMeasure = false)

Do final setup work (allocate things, generate FT of PSF image, etc.)

General setup prior to actually supplying the image data and doing the convolution: determine padding dimensions; allocate FFTW arrays and plans; normalize, shift, and Fourier transform the PSF image.

void **ConvolveImage**(double *pixelVector)

Replace input model image (pixelVector) with convolution using stored PSF.

Given an input image (pointer to its pixel vector), convolve it with the PSF by: 1) Copying image to image_in_padded array (with zero-padding); 2) Taking FFT of image; 3) Multiplying transform of image by transform of PSF; 4) Taking inverse FFT of product; 5) Copying (and rescaling) result back into input image.

Private Functions

void **ShiftAndWrapPSF**()

Takes the input PSF (assumed to be centered in the central pixel of the image) and copy it into the (padded) image, with the PSF wrapped into the corners, suitable for convolutions.

10.5 OversampledRegion

10.5.1 Overview

This class handles computation of oversampled regions in the model image; there will be one instance for each oversampled region requested by the user (these are stored in ModelObject as a vector of pointers to individual OversampledRegion objects).

It holds information about the location, size, and pixel oversampling factor of the oversampled region; it also holds a Convolver object which performs PSF convolution using the appropriate oversampled PSF image.

Code contained in this class can generate an oversampled model-image region, using the FunctionObjects vector passed in by the caller (i.e., main's ModelObject instance); this subimage is then convolved with the oversampled PSF and

finally block-downsampled back to the main image's pixel scale and copied into the corresponding pixels of the main model image. OpenMP compiler directives are used to speed up computation of the oversampled model subimage.

10.5.2 Setup and Use

To set up an OversampledRegion object, you first call its AddPSFVector method to pass in the PSF image data, size, and whether the PSF should be normalized. Then you tell the OversampledRegion object about the size of the main image, the location of the oversampled region, and the oversampling scale by calling the SetupModelImage method.

(See ModelObject::AddOversampledPSFVector for an example of how this is done in practice.)

To *use* an OversampledRegion object in the model-image-generation process, you call its ComputeRegionAndDownsample method, passing in a pointer to the current model image and the vector of FunctionObjects which describe the model. You should be sure to update the FunctionObjects with the current set of parameter values before doing this. (See ModelObject::CreateModelImage for an example of use.)

(In Imfit, this is all done within the ModelObject class.)

10.5.3 API

Files: `core/oversampled_region.h`, `core/oversampled_region.cpp`

class **OversampledRegion**

Class for computing oversampled model image region and downsampling to match main image, with optional PSF convolution using oversampled PSF.

Public Functions

OversampledRegion()

Constructor for *OversampledRegion* class.

~OversampledRegion()

Destructor for *OversampledRegion* class.

void **AddPSFVector**(double *psfPixels_input, int nColumns, int nRows, bool normalizePSF = true)

Pass in a pointer to the pixel vector for the input PSF image, as well as the image dimensions. We assume this is an oversampled PSF, with the same oversampling scale as specified in the input to *SetupModelImage()*, below

void **SetMaxThreads**(int maximumThreadNumber)

User specifies maximum number of FFTW threads to use (ignored if not compiled with multithreaded FFTW library)

int **SetupModelImage**(int x1, int y1, int nBaseColumns, int nBaseRows, int nColumnsMain, int nRowsMain, int nColumnsPSF_main, int nRowsPSF_main, int oversampScale)

Pass in the dimensions of the image region, oversample scale, etc. x1,y1 = x,y location of lower-left corner of image region w/in main image (IRAF-numbering) nBaseColumns,nBaseRows = x,y size of region in main ("base") image nColumnsMain, nRowsMain = x,y size of full main model ("base") image

void **ComputeRegionAndDownsample**(double *mainImageVector, vector<*FunctionObject**> functionObjectVect, int nFunctionObjects)

This is the main method, which computes the oversampled (sub-region) model image, then downsamples it to the main image pixel scale and copies it into the main image (mainImageVector). We assume that

the FunctionObjects pointed to by functionObjectVect have already been set up with the current parameter values by calling their individual Setup() methods — e.g., by the method or function that is calling *this* method.

10.6 PsfOversamplingInfo

10.6.1 Overview

This is a data-container class meant to assist in the setup phase of imfit, makeimage, etc. It holds information about a single user-specified oversampling region: the location and of the region within the image, its size, the desired pixel oversampling scale, and the oversampled PSF image to be used with that region. In effect, it packages up the information provided by the following command-line options:

```
--overpsf, --overpsf_region, --overpsf_scale
```

In practice, a vector of one or more PsfOversamplingInfo objects is set up in main() based on the user inputs and is then passed to the SetupModelObject function, where the individual PsfOversamplingInfo objects are passed to the ModelObject instance via its AddOversampledPsfInfo method.

10.6.2 API

Files: `core/psf_oversampling_info.h`, `core/psf_oversampling_info.cpp`

class **PsfOversamplingInfo**

Public Functions

PsfOversamplingInfo()

Default constructor for *PsfOversamplingInfo* class.

PsfOversamplingInfo(double *inputPixels, int nCols, int nRows, int scale, string inputRegionString, int xOffset = 0, int yOffset = 0, bool isUnique = true, bool normalize = true)

Constructor (with inputs) for *PsfOversamplingInfo* class.

~PsfOversamplingInfo()

Destructor for *PsfOversamplingInfo* class.

void **AddPsfPixels**(double *inputPixels, int nCols, int nRows, bool isUnique)

Add data describing a PSF image.

void **AddRegionString**(string inputRegionString)

Add a region string (e.g., "[500:600,1080:1422]") defining a subsection of an image that will be oversampled

void **AddOversamplingScale**(int scale)

Specify the pixel oversampling scale for an oversampling region (must be an integer >= 1)

void **AddImageOffset**(int X0, int Y0)

Specify the offset of an overampling region (the location of its lower-left corner within the full image)

void **SetNormalizationFlag**(bool normalize)

Whether PSF image should be normalized or not.

int **GetNColumns**()

Returns x-size of PSF image.

int **GetNRows**()

Returns y-size of PSF image.

bool **PixelsArrayIsUnique**()

Returns true if PSF image is unique to this oversampling region; returns false if PSF image is shared with other oversampling regions

double ***GetPsfPixels**()

Returns a pointer to the PSF image pixel array.

bool **GetNormalizationFlag**()

Returns true if PSF is meant to be normalized.

10.7 getimages.h

10.7.1 Overview

These are higher-level functions, with specializations for reading in mask, noise, and PSF images; they make use of the functions in image_io.h.

10.7.2 API

Files: `core/getimages.h`, `core/getimages.cpp`

Functions for reading in (and checking) various images (other than main data image): mask, error, PSF, and oversampled PSF images. For use with makeimage, imfit, and imfit-mcmc.

Functions

std::tuple<double*, int> **GetAndCheckImage**(const string imageName, const string imageType, int nColumns_ref, int nRows_ref)

Main utility function: reads in image from FITS file imageName and checks dimensions against reference values (if latter are nonzero)

Main utility function: reads in image from FITS file imageName and checks dimensions against reference values (if latter are nonzero). Returns (nullptr, -1) if unable to read image-data from file; returns -2 (and frees allocated memory) if image dimensions do not match references dimensions (unless latter are both 0).

std::tuple<double*, double*, int> **GetMaskAndErrorImages**(int nColumns, int nRows, string &maskFileName, string &errorFileName, bool &maskPixelsAllocated, bool &errorPixelsAllocated)

Reads in mask and/or noise/error images.

Function which retrieves and checks dimensions for mask and/or noise/error images. Returns tuple of (maskPixels, errorPixels, status), where maxPixels and errorPixels are double * (and are = nullptr when image in question was not requested). In case of errors in retrieving image data — or if image dimensions do not match reference dimensions nColumns, nRows — then (nullptr, nullptr, -1) is returned. Return values: status = 1: mask image loaded, but no error image specified status = 2: error image loaded, but no mask image was specified status = 3: both images specified & loaded

std::tuple<double*, int, int, int> **GetPsfImage**(const string &psfFileName)

Reads in PSF image, returning dimensions as well.

Function which reads and returns data corresponding to requested PSF image, along with PSF image dimensions: tuple of (psfPixels, nColumns_psf, nRows_psf, status). In case of errors in retrieving image data, return value is (nullptr, 0, 0, -1)

int **GetOversampledPsfInfo**(const std::shared_ptr<OptionsBase> options, int xOffset, int yOffset, vector<*PsfOversamplingInfo**> &psfOversamplingInfoVect)

Reads in multiple oversampled PSF images, storing them (along with user-specified info like oversampling scale and image regions for oversampling) in a vector of (pointers to) *PsfOversamplingInfo* objects.

10.8 image_io.h

10.8.1 Overview

These are utility functions for inspecting and reading in FITS files; they are wrappers around calls to CFITSIO functions.

Note that ReadImageAsVector allocates the memory for the image it reads (via the FFTW3 utility routine fftw_alloc_real); the caller is responsible for freeing the memory (via fftw_free) when it is no longer needed.

10.8.2 API

Files: `core/image_io.h`, `core/image_io.cpp`

Public interfaces for the FITS image I/O routines.

Functions

bool **CheckHDUForImage**(fitsfile *imageFile_ptr, int hduNum, int *status_ptr)

Checks to see if current HDU in FITS file has proper 2D image.

int **CheckForImage**(const std::string filename, const bool verbose = false)

Checks to see if FITS file has proper 2D image in first HDU (or second, if first is empty)

Given the filename of a FITS image, this function opens the file and checks the first header-data unit to see if it is a 2D image. If it is *not* and a second HDU exists, then that is also checked.

Returns -1 if something goes wrong opening the file or if primary HDU is not a proper 2D image; otherwise, returns 1 if the specified FITS file (including implicitly specified extension number, if that was part of filename) is a proper2D image.

Currently uses int as return value for possible case of finding and returning a different HDU. FIXME: probably good idea to make this bool return value.

std::tuple<int, int, int> **GetImageSize**(const std::string filename, const bool verbose = false)

Gets dimensions (nColumns, nRows) of specified FITS image.

Given the filename of a FITS image, this function opens the file, reads the size of the image and returns the dimensions in nRows and nColumns.

Returns 0 for successful operation, -1 if a CFITSIO-related error occurred.

double *ReadImageAsVector(const std::string filename, int *nColumns, int *nRows, const bool verbose = false)

Reads image data from specified FITS image, returning it as 1D array (with image dimensions stored in nColumns, nRows)

Given a filename, it opens the file, reads the size of the image and stores that size in nRows and nColumns, then allocates memory for a 1-D array to hold the image and reads the image from the file into the array. Finally, it returns the image array — or, more precisely, it returns a pointer to the array; it also stores the image dimensions in the pointer-parameters nRows and nColumns.

Returns NULL (and prints error message) if a CFITSIO-related error occurred.

int SaveVectorAsImage(double *pixelVector, const std::string filename, int nColumns, int nRows, std::vector<std::string> comments)

Saves image data (1D array, logical dimensions nColumns x nRows) as FITS file, with comments added to header.

Saves specified 1D array (representing an image with size nColumns x nRows) in specified filename as a FITS image, with strings stored in comments vector written as comments to the FITS header.

Returns 0 for successful operation, -1 if a CFITSIO-related error occurred.

int CountHeaderDataUnits(fitsfile *imfile_ptr)

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

O

P

R

S