Notes for Using Imfit
(Version 1.9.0)

Peter Erwin
MPE and USM
erwin@sigmaxi.net

November 27, 2022

Contents

1 What Is It?

Imfit is a program for fitting astronomical images – specifically, for fitting images of galaxies, though it can certainly be used for fitting other sources. The user specifies a set of one or more 2D surface-brightness functions (e.g., elliptical exponential, elliptical Sérsic, circular Gaussian) which will be added together in order to generate a model image; this model image will then be matched to the input image by adjusting the 2D function parameters via nonlinear minimization of the total χ^2 (or of the total Cash or Poisson-MLR statistics in the alternate case of Poisson statistics).

The 2D functions can be grouped into arbitrary sets sharing a common (x, y) position on the image plane; this allows galaxies with off-center components or multiple galaxies to be fit simultaneously. Parameters for the individual functions can be held fixed or restricted to user-specified ranges. The model image can (optionally) be convolved with a point spread function (PSF) image to better match the input image; the PSF image can be any square, centered image the user supplies – e.g., an analytic 2D Gaussian or Moffat, a *Hubble Space Telescope* PSF generated by the TinyTim program[1] [Krist, 1995], or an actual stellar image. (A higher-resolution, "oversampled" PSF image can also be used for one or more subsections of the full image.)

A key part of **Imfit** is a modular, object-oriented design that allows easy addition of new, user-specified 2D image functions. This is accomplished by writing C++ code for a new image-function class (this can be done by copying and modifying an existing pair of `.h/.cpp` files for one of the pre-supplied image functions), making small modifications to two additional files to include references to the new function, and re-compiling the program.

Imfit is an open-source project; the source code is freely available under the GNU Public License (GPL).

A note on names: The **Imfit** package consists of three programs:

- `imfit` – the main image-fitting program;
- `imfit-mcmc` – a program to do Markov chain Monte Carlo (MCMC) analysis;
- `makeimage` – an auxiliary program which can be used to generate artificial galaxy images (using the same input/output parameter-file format that `imfit` uses).

In this document, I use **Imfit** (in boldface) to refer to the whole package and `imfit` to specific use of the actual fitting program, although I'm not entirely consistent about this; in most cases this distinction should be meaningless. (I.e., in most cases you're using the `imfit` binary to fit images.)

System Requirements: Imfit has been built and tested on Intel-based MacOS X (or macOS) 10.8–12 (Mountain Lion through Monterey)[2] and Linux (Ubuntu) systems. The Intel-based Mac binary will also run on Apple Silicon (aka arm64) Mac computers via the Rosetta 2 facility. It uses standard C++ or C++11 and should work on most Unix-style systems with a modern C++

[1] `http://www.stsci.edu/software/tinytim/`

[2] Versions 1.3 and earlier were also built for MacOS X 10.6 and 10.7; it's unclear whether the current version could still be compiled for those systems.

compiler and the Standard Template Library (e.g., GCC v4.8.1 or higher). It relies on three external, open-source libraries: version 3 of the CFITSIO library[3] for FITS image I/O, version 3 of the FFTW (Fastest Fourier Transform in the West) library[4] for PSF convolution, and version 2.0 or higher of the GNU Scientific Library (GSL),[5]. Some of the fitting algorithms require the NLopt library[6], but the program can also be built without it.

Imfit also includes modified versions of Craig Markwardt's `mpfit` code (an enhanced version of the MINPACK-1 Levenberg-Marquardt least-squares fitting code); the Differential Evolution fitting code of Rainer Storn and Kenneth Price (more specifically, a C++ wrapper written by Lester E. Godwin); and an implementation of the MCMC DREAM algorithm in C++ by Gabriel Leventhal.[7]

[3] `https://heasarc.nasa.gov/fitsio/`
[4] `http://www.fftw.org/`
[5] `https://www.gnu.org/software/gsl/`
[6] `https://nlopt.readthedocs.io/en/latest/`
[7] `https://github.com/gaberoo/cdream`

2 Getting and Installing Imfit

2.1 Pre-Compiled Binaries

Pre-built binaries for Intel-based MacOS X and Linux systems, along with the source code, are available at `https://www.mpe.mpg.de/~erwin/code/imfit/`. The pre-compiled binaries included statically linked versions of the CFITSIO, FFTW, GSL, and NLopt libraries, so you do not need to have those installed. (The pre-built Linux binaries *do* require that the system GNU C library be version 2.15 or more recent; try typing `ldd --version` at the command line to find out.)

Inside the binary-install tarball, there are three binary executable files: `imfit`, `imfit-mcmc`, and `makeimage`. Copy these to some convenient place on your path.

Useful Extras

The binary-install tarball has a `docs/` subdirectory with a copy of the **Imfit** manual in PDF and LaTeX form.

There is also an `examples/` subdirectory with sample files; see the `README_examples.txt` file there for simple examples of how to use **Imfit** with those files. (This includes the files used in the online tutorial.[8])

If you use the bash shell, there is a tab-completion file in the `extras/` subdirectory. Copy the file `imfit_completions.bash` to somewhere convenient and add the following to your `.profile`, `.bash_profile`, or `.bashrc` file:

```
source /path/to/imfit_completions.bash
```

You should then (assuming you restart your shell, open a new terminal window, or something similar) be able to type "`imfit --`", press the TAB key a couple of times, and get a listing of the command-line flags and options; typing part of a flag/option and then pressing TAB should complete the command. (And similarly for `imfit-mcmc` and `makeimage`.)

2.2 Building Imfit from Source: Outline

The standard source-code distribution for **Imfit** can be found at the main **Imfit** web site (see above). (The "useful extras" listed above are of course included in the source-code distribution.) An expanded version of the source code (including extra tests, notes, and auxiliary programs) can be found at **Imfit**'s GitHub site: `https://github.com/perwin/imfit`.

The basic outline for building **Imfit** from source is:

1. Install the CFITSIO library (version 3.0 or higher).

2. Install the GNU Scientific Library (GSL; version 2.0 or higher).

3. Install the FFTW library (version 3.0 or higher). Note that if you have a multi-core CPU (or multiple CPUs sharing main memory), you should build and install the threaded version of FFTW as well (this will probably involve running its "configure" script with the `--enable-threaded` option; see the FFTW manual for details), since this speeds up PSF convolution considerably. Building FFTW with SSE2 and AVX support – assuming you have

[8] `https://www.mpe.mpg.de/~erwin/code/imfit/markdown/index.html`

an Intel CPU – will provide a ~ 10–20% speedup for PSF convolutions, so that's also recommended, though not as strongly.

4. (Optional, but strongly recommended) Install the NLopt library – this is necessary if you wish to use the Nelder-Mead minimization algorithm. **Imfit** can be built without this, if for some reason you don't have access to the NLopt library.

5. Install SCons (if needed; see below).

6. Build `imfit`, `imfit-mcmc`, and `makeimage`.

7. (Optional) Run test scripts `do_imfit_tests`, `do_mcmc_tests`, and `do_makeimage_tests`.

2.3 Building Imfit from Source: Details

Assuming that CFITSIO, FFTW, NLopt, and GSL have already been installed on your system (steps 1–4 from the outline in the previous section), unpack the source-code tarball (imfit-x.x-source.tar.gz) in some convenient location. (Or, if you prefer, use 'git clone' to download the full repo from GitHub.)

Building with SCons

By default, **Imfit** uses SCons for the build process. SCons is a Python-based build system that is somewhat easier to use and more flexible than the traditional `make` system; it can be downloaded from `https://www.scons.org/` or installed via your favorite package manager, including Python's Pip, e.g.

```
$ pip install scons
```

If things are simple, you should be able to build **Imfit** and the companion program `makeimage` with the following commands:

```
$ scons imfit
$ scons imfit-mcmc
$ scons makeimage
```

This will produce three binary executable files: `imfit`, `imfit-mcmc`, and `makeimage`. Copy these to some convenient place on your path.

If you do not have the NLopt library installed, you will get compilation errors; this can be dealt with by using:

```
$ scons --no-nlopt imfit
$ scons --no-nlopt imfit-mcmc
$ scons --no-nlopt makeimage
```

Various other compilation options may be useful, particularly for telling SCons where to look for library files; these are explained in the next subsections (note that all the SCons options can be combined on the command line).

Tests

Finally, there are three shell scripts – `do_imfit_tests`, `do_mcmc_tests`, and `do_makeimage_tests` – which can be run to do some very simple sanity checks (e.g., do the programs fit some simple images correctly, are common config-file errors caught, etc.). They make use of files and data in the `tests/` subdirectory.

Differences in output at the level of the least significant digit compared to the reference files may occur when running the tests on a Linux system; these should not be considered problems. (This sort of thing appears to depend on subtle differences in how different compilers and libraries handle floating-point computations and rounding.)

For the full set of tests to run, you should have Python version 2.6, 2.7, or 3.5+ installed, along with the Python libraries numpy[9] and astropy.[10] If these are not available, then the parts of the tests which compare output images with reference versions will be skipped.

Telling the Compiler Where to Find Header Files and Libraries

By default, the SConstruct file (the equivalent of a Makefile for SCons) included in the source distribution for **Imfit** tells SCons to look for header files in `/usr/local/include` and library files in `/usr/local/lib`. If you have the FFTW, CFITSIO, GSL, and/or (optionally) NLopt and headers and libraries installed somewhere else, you can tell SCons about this by using the `--header-path` and `--lib-path` options:

```
$ scons --header-path=/some/path ...
$ scons --lib-path=/some/other/path ...
```

(note that "`...`" is meant to stand for the rest of the compilation command, whatever that may be). This will add the specified directories to the header and library search paths (`/usr/local/include` and `/usr/local/lib` will still be searched as well).

Multiple paths can be specified if they are separated by colons, e.g.

```
$ scons --lib-path=/some/path:/some/other/path ...
```

Options: Compiling Without OpenMP Support

By default, `imfit` and `makeimage` are compiled to take advantage of OpenMP compiler support, which speeds up image computation (a *lot*) by splitting it up across multiple CPUs (and multiple cores within multi-core CPUs). The code uses OpenMP 2.5 options, which are available with versions of GCC 4.2 or higher. (Though since the code also uses C++11 features, you really need GCC 4.8.1 or higher.)

If your compiler does not support OpenMP – or you want, for whatever reason, a version that does not include OpenMP support – you can disable it by compiling with the following commands:

```
$ scons --no-openmp ...
```

[9] `https://www.numpy.org`
[10] `https://www.astropy.org`

Compiling on macOS

The standard compiler tools for macOS (formerly Mac OS X) are those which come with Apple's Xcode, which for C and C++ are based on LLVM's Clang. (Confusingly, the Clang-based tools include "`gcc`" and "`g++`" aliases which are wrappers around `clang` and `clang++`. To be certain this is the case, try `gcc -v`; if you see something like "Apple LLVM version", then you know it's really the Clang compiler.)

Prior to version 9 of Xcode (introduced in late 2017), the Apple-supplied Clang compiler did *not* support OpenMP (even though the general non-Apple version did); this meant that compiling **Imfit** with OpenMP required a separate installation of GCC. Now, there *is* support for OpenMP, though you will still need to install an additional library (see below).

Currently, there are two approaches for compiling **Imfit** on macOS:

1. Use a Separate GCC (or LLVM) Installation For current versions of **Imfit**, you can use a separate installation of GCC (this should be version 4.8.1 or later), such as those available via package managers such as Homebrew or MacPorts. (Note that Homebrew does not *replace* the pseudo-"gcc" and "g++" aliases provided by Xcode, which will still point to Apple's Clang compilers; instead, it installs version-specific names such as "gcc-12" and "g++-12".) Another alternative is to install a version of LLVM/Clang that *does* support OpenMP, e.g., from one of the aforementioned package managers.

To tell SCons to use a specific compiler, use the `--cpp` option:

```
$ scons --cpp=<C++_COMPILER> ...
```

2. Use Apple's Clang As mentioned above, versions of Apple's Clang from Xcode 9 or later provide support for OpenMP. However, this is not (quite) automatic: you still need to install a copy of `libomp`, which is the LLVM version of the OpenMP library. This can be done via a package manager like Homebrew or MacPorts. For example, to install it using Homebrew, just do this:

```
$ brew install libomp
```

Since supporting this in **Imfit** is still a bit experimental, you need to tell SCons to explicitly use this approach via the `--clang-openmp` option:

```
$ scons --clang-openmp ...
```

Options: Compiling Without FFT Multithreading

By default, the **Imfit** binaries are compiled to take advantage of multi-core CPUs (and other shared-memory multiple-processor systems) when performing PSF convolutions by using the multithreaded version of the FFTW library. If you do not have (or cannot build) the multithreaded FFTW library, you can remove multithreaded FFT computation by compiling with the following commands:

```
$ scons --no-threading imfit
$ scons --no-threading imfit-mcmc
$ scons --no-threading makeimage
```

3 Trying It Out

This section has some basic notes on how to use **Imfit**; for a somewhat more comprehensive introduction, see the online tutorial at `https://www.mpe.mpg.de/~erwin/code/imfit/markdown/index.html`.

In the `examples/` directory are some sample galaxy images, masks, PSF images, and configuration files. To give **Imfit** a quick spin (and check that it's working on your system), change to the `examples/` directory and execute the following on the command line (assuming that the `imfit` binary is now in your path; if it isn't you can access it via `../imfit`):

```
$ imfit ic3478rss_256.fits -c config_sersic_ic3478_256.dat --sky=130.14
```

This converges to a fit in about a second or less (e.g., about 0.14 seconds on a 2019 MacBook Pro with a 2.3 GHz Core i9 processor). In addition to being printed to the screen, the final fit is saved in a file called `bestfit_parameters_imfit.dat`.

The preceding command told `imfit` to fit using every pixel in the image and to estimate the noise assuming an original (previously subtracted) sky level of 130.14, an A/D gain of 1.0, and zero read noise (the latter two are default values). A better approach would be to include a mask (telling `imfit` to ignore, e.g., pixels occupied by bright stars) and to specify more accurate values of the gain and read noise:

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

If you want to see what the best-fitting model looks like, you can use the companion program `makeimage` on the output file:

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

This will generate and save the model image in a file called `modelimage.fits`. (The `imfit` program can itself save the best-fitting model image at the end of the fitting process if the `--save-model` option is used.)

You can also fit the image using PSF convolution, by adding the "`--psf`" option and a valid FITS image for the PSF; the `examples/` directory contains a Moffat PSF image which matches stars in the original image fairly well:

```
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
```

The PSF image was generated using `makeimage` and the configuration file `config_makeimage_moffat_psf.dat`:

```
makeimage -o psf_moffat_51.fits config_makeimage_moffat_psf.dat
```

4 Using Imfit

Basic use of **Imfit** from the command line looks like this:

```
$ imfit -c config-file input-image [options]
```

where *config-file* is the name of the configuration file which describes the model (the combination of 2D functions, initial values for parameters, and possible limits on parameter values) and *input-image* is the FITS image we want to fit with the model.

Note that both *config-file* and *input-image* – as well as other file names specified by options discussed below – can be plain filenames in the current working directory, relative paths (e.g., `data_dir/image.fits`), or absolute paths.

The "options" are a set of command-line flags and options; use "`imfit -h`" or "`imfit --help`" to see the complete list. Options must be followed by an appropriate value (e.g., a filename, an integer, a floating-point number); this can be separated from the option by a space, or they can be connected with an equals sign. In other words, both of the following are valid:

```
imfit --gain 2.5
imfit --gain=2.5
```

Note that **Imfit** does not follow the full GNU standard for command-line options and flags (as implemented by, e.g., the GNU `getopt` library): you cannot merge multiple one-character flags into a single item (if "`-a`" and "`-b`" are flags, "`-a -b`" will work, but "`-ab`" will *not*), and you cannot merge a one-character option and its target ("`-cfoo.dat`" is *not* a valid substitute for "`-c=foo.dat`" or "`-c foo.dat`").

4.1 Command-line Flags and Options

Some notable and useful command-line flags and options include:

- `-c, --config` *config-file* – the only *required* command-line option, which tells `imfit` the name of the configuration file. (Actually, if you don't supply this option, `imfit` will look for a file called "imfit_config.dat", but it's best to explicitly specify your own configuration files.)

- `--psf` *psf-image* – specifies a FITS image to be convolved with the model image.

- `--overpsf` *psf-image* – specifies an oversampled FITS image to be convolved with (a subregion of) the model image.

- `--overpsf_scale` *scale* – specifies the oversampling factor of the oversampled PSF image (an integer > 1). E.g., for a 5 × 5 oversampled PSF image (25 PSF pixels for each data-image pixel), the scale is 5.

- `--overpsf_region` *x1:x2,y1:y2* – specifies the subsection of the image where oversampled PSF convolution will be done. Multiple oversampled regions (each convolved with the same oversampled PSF) can be specified by repeating this command.

- `--mask` *mask-image* – specifies a FITS image which marks bad pixels to be ignored in the fitting process. By default, zero values in the mask indicate *good* pixels, and positive values indicate bad pixels.

- `--mask-zero-is-bad` – indicates that zero values (actually, any value < 1.0) in the mask correspond to *bad* pixels, with values ≥ 1.0 being good pixels.

- `--noise` *noisemap-image* – specifies a pre-existing noise or error FITS image to use in the χ^2 fitting process (by default, pixel values in the noise map are assumed to be Gaussian σ values).

- `--errors-are-variances` – indicates that pixel values in the noise map are variances (σ^2) instead of sigmas.

- `--errors-are-weights` – indicates that pixel values in the noise map should be interpreted as weights (e.g., $1/\sigma^2$), not as sigmas or variances.

  Note that none of these three options is usable with Cash statistic or Poisson-MLR statistic minimization.

- `--sky` *sky-level* – specifies an original constant sky background level (in counts/pixel) *that was previously subtracted from the image*; this is used for internal computation of the noise map (for χ^2 minimization) or for correcting the Cash statistic or Poisson-MLR statistic computation. (If the units of the pixel values are counts/sec, then the sky level should also be in those units, and you should use the "`--exptime`" option to specify the original exposure time.)

- `--gain` *value* – specifies the A/D gain (in electrons/ADU) of the input image; used for internal computation of the noise map (for χ^2 minimization) or for correcting the Cash statistic and Poisson-MLR statistic computation.

- `--readnoise` *value* – specifies the read noise (in electrons) of the input image; used for internal computation of the noise map for χ^2 minimization (this is ignored in the case of Cash statistic or Poisson-MLR statistic minimization).

- `--exptime` *value* – specifies the exposure time of the image; this should **only** be used *if* the image has been divided by the exposure time (i.e., if the pixel units are counts/sec).

- `--ncombined` *value* – if values in the input image are the result of averaging (or computing the median of) two or more original images, then this option should be used to specify the number of original images; this is used for internal computation of the noise map (for χ^2 minimization) or for correcting the Cash statistic and Poisson-MLR statistic computation. If multiple images were *added* together with no rescaling, then do not use this option.

- `--save-params` *output-filename* – specifies that parameters for best-fitting model should be saved using the specified filename (the default is for these to be saved in a file named `bestfit_parameters_imfit.dat`).

- `--save-model` *output-filename* – the best-fitting model image will be saved using the specified filename.
- `--save-residual` *output-filename* – the residual image (input image – best-fitting model image) will be saved using the specified filename.

- `--nm` – use Nelder-Mead simplex instead of Levenberg-Marquardt as the minimization technique (WARNING: slower).
- `--de` – use Differential Evolution instead of Levenberg-Marquardt as the minimization technique (WARNING: much slower!).
- `--de-lhs` – use Differential Evolution with Latin hypecube sampling (as opposed to standard uniform sampling) for the initial guesses.
- `--nlopt` *algorithm-name* – use one of the "local derivative-free" minimization algorithms from the NLopt library;[11] recognized options are COBYLA, BOBYQA, NEWUOA, PRAXIS, and SBPLX. These are generally slower and/or less robust than the Nelder-Mead simplex algorithm (which is also part of the NLopt library, and can be specified with `--nlopt NM`, though it's simpler just to use `--nm`).

- `--model-errors` – use the model image pixel values (instead of the data values) to estimate the individual-pixel dispersions σ_i for χ^2 minimization.
- `--poisson-mlr` or `--mlr` – use the Poisson Maximum-Likelihood-Ratio (MLR) statistic for minimization (this is the same as the "CSTAT" statistic in the X-ray packages XSPEC and Sherpa) This is especially useful in the case of Poisson statistics and low or zero read noise, but it also provides less biased fits than the χ^2 approach even when count levels are high.
- `--cashstat` – use Cash statistic C instead of χ^2 as the fit statistic for minimization. This is essentially the same as the Poisson MLR statistic, and should give identical results. However, since values of C can be < 0, it cannot be used with the (default) Levenberg-Marquardt minimization technique.
- `--ftol` *FTOL-value* – specify tolerance for fractional improvements in the fit statistic value; if further iterations do not reduce the fit statistic by more than this, the minimization is considered a success and halted (default value = 10^{-8}).

- `--bootstrap` *n-iterations* – Do *n-iterations* rounds of bootstrap resampling after the fit, to estimate parameter errors.
- `--save-bootstrap` *filename* – file where individual best-fit parameter values from the bootstrap resampling (one line per iteration) will be saved.

- `--quiet` – Suppress printing of intermediate fit-statistic values during the fitting process.

[11] See `https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/`.

- `--loud` – Print intermediate parameter values during the fitting process. Currently applicable to L-M fitting (current best-fit parameter values are printed once per iteration) and N-M simplex fitting (best-fit values printed once per 100 interations).

- `--chisquare-only` or `--fitstat-only` (these are synonyms) – Evaluate the χ^2 value for the initial input model as a fit to the input image, *without* doing any minimization to find a better solution. (If `--poisson-mlr` or `--cashstat` is also specified, then the appropriate statistic is evaluated instead.)

- `--max-threads` *n-threads* – specifies the maximum number of CPU threads to use during computation (the default is to use *all* available CPU threads); has no effect if **Imfit** was compiled without OpenMP or FFTW multithreading support. For best performance with PSF convolution, this should be set = the number of physical CPU cores (*not* the number of logical cores). For best performance on Apple Silicon computers, this should be set = the number of "performance" cores. (As of version 1.9, **Imfit** will default to this mode.) You can use this to reduce the number of threads that **Imfit** will use if you want to reduce system load or power consumption, at the cost of slower fits.

- `--seed` *N* – specifies a specific integer seed to use with random number generation; applies to DE fits and also to bootstrap resampling. This is mainly for testing purposes, to ensure that the same sequence of pseudo-random numbers is generated. (By default, the current system time is used as a seed, ensuring that no two runs have the same random seed.)

- `--sample-config` – generates and saves a simple example of an **Imfit** configuration file (named `config_imfit_sample.dat`).

- `--list-functions` – list all the functions **Imfit** can use.

- `--list-parameters` – list all the individual parameters (in correct order) for each of the functions that **Imfit** can use. You can copy and paste pieces of the output of this command to help construct a configuration file.

5 The Configuration File

Imfit always requires a configuration file, which specifies the model which will be fit to the input image, initial values for model parameters, any limits on parameter values (optional for fitting with the Levenberg-Marquardt solver, but required for fitting with the Differential Evolution solver), and possibly additional information (e.g, gain and read noise for the input image).

The configuration file should be a plain text file. Blank lines and lines beginning with "#" are ignored; in fact, anything on the same line after a "#" is ignored, which allows for comments at the end of lines. (A minor exception: on lines defining a function name, an optional label can be specified as part of the comment.)

A model for an image is specified by one or more **function sets**,[12] each of which is a group of one or more 2D image functions sharing a common (x, y) spatial position. Each function-specification consists of a line beginning with "FUNCTION" and containing the function name, followed by one or more lines with specifications for that function's parameters.

More formally, the format for a configuration file is:

1. Optional specifications of general parameters and settings (e.g., the input image's A/D gain and read noise)
2. One or more function sets, each of which contains:
    a) X-position parameter-specification line
    b) Y-position parameter-specification line
    c) One or more function + parameters specifications, each of which contains:
        i. `FUNCTION` + function-name line [+ optional label]
        ii. one or more parameter-specification lines

This probably sounds more complicated than it is in practice. Here is a very bare-bones example of a configuration file:

```
X0   150.1
Y0   149.5
FUNCTION Exponential
PA   95.0
ell   0.45
I_0   90.0
h   15.0
```

This describes a model consisting of a single elliptical exponential function, with initial values for the x and y position on the image of the object's center, the position angle (PA), the ellipticity (ell), the central intensity (I_0) in counts/pixel, and the exponential scale length in pixels (h). None of the parameters have limits on their possible values.

Here is the same file, with some additional annotations and with limits on some of the parameters (comments are colored red for clarity):

[12] In versions of **Imfit** prior to 1.8, these were generally referred to as "function blocks".

```
# This line is a comment

X0    150.1   148,152
Y0    149.5   148,152 # a note
FUNCTION Exponential   # here is a comment
PA   95.0   0,180    # limits on the position angle
ell   0.45   0,1     # ellipticity should always be 0--1
I_0   90.0   fixed     # keep central intensity fixed
h   15.0
```

Here we can see the use of comments (lines or parts of lines beginning with "#") and the use of parameter limits in the form of "lower,upper": the X0 and Y0 parameters are required to remain ≥ 148 and ≤ 152, the position angle is limited to 0–180, the ellipticity must stay ≥ 0 and ≤ 1, and the central intensity I_0 is held fixed at its initial value.

Finally, here is a more elaborate example, specifying a model that has two function sets, with the first set having two individual functions (this could be a model for, e.g., simultaneously fitting two galaxies in the same image, one as Sérsic + exponential, the other with just an exponential). In addition, we specify labels for the functions. These labels are purely for the convenience of the user; they have no effect on the fitting, but are reproduced in output files.

```
# This line is a comment

GAIN 2.7 # A/D gain for image in e/ADU
READNOISE 4.5 # image read-noise in electrons

# This is the first function set:  Sersic + exponential
X0    150.1    148,152
Y0    149.5    148,152
FUNCTION Sersic # LABEL galaxy 1 bulge
PA   95.0    0,180
ell   0.05   0,1
n    2.5     0.5,4.0    # Sersic index
I_e   20.0    # intensity at the half-light radius
r_e   5.0    # half-light radius in pixels
FUNCTION Exponential # LABEL galaxy 1 disk
PA   95.0    0,180
ell   0.45   0,1
I_0   90.0   fixed
h    15.0

# This is the second function set:  just a single exponential
X0    225.0    224,226
Y0    181.7    180,183
FUNCTION Exponential # LABEL galaxy 2
PA   22.0    0,180
```

```
ell    0.25   0,1
I_0    10.0
h      20.0
```

5.1 Parameter Names, Specifications, and Values

The X0/Y0 position lines at the start of each function set and the individual parameter lines for each function all share a common format:

parameter-name initial-parameter-value optional-limits

The separation between the individual pieces must consist of one or more spaces and/or tabs. The final piece specifying the limits is optional (except that fitting in Differential Evolution mode *requires* that there be limits for each parameter).

Parameter Names: The X0/Y0 positional parameters for each function set must be labeled "X0" and "Y0". Names for the parameters of individual functions can be anything the user desires; only the order matters. Thus, the position-angle parameter could be labeled "PA", "PosAngle", "angle", or any non-space-containing string – though it's a good idea to have it be something relevant and understandable.

Important Note: *Do not change the order of the parameters for a particular function!* Because the strings giving the parameter names can be anything at all, `imfit` actually ignores them and simply assumes that all parameters are in the correct order for each function.

Note that any output which `imfit` generates will use the default parameter names defined in the individual function code (use "`--list-parameters`" to see what these are for each function).

Values for Positional Parameter (X0, Y0): The positional parameters for each function set are pixel values – X0 for the column number and Y0 for the row number. **Imfit** uses the IRAF pixel-numbering convention: the center of first pixel in the image (the lower left pixel in a standard display) is at $(1.0, 1.0)$, with the lower-left corner of that pixel having the coordinates $(0.5, 0.5)$.

Note that these positions are in the coordinate system of the entire image. That is, if you specify a subsection of the image to be fit (e.g., "image.fits[150:350,425:700]"), the X0 and Y0 values must still be absolute positions referring to the original image, *not* relative positions within the subsection.

General Parameter Values for Functions: The meaning of the individual parameter values for the various 2D image functions is set by the functions themselves, but in general:

- position angles are measured in degrees counter-clockwise from the image's vertical ($+y$) axis (i.e., degrees E of N if the image has standard astronomical orientation);
- ellipticity = $1 - b/a$, where a and b are the semi-major and semi-minor axes of an ellipse;
- intensities are in counts/pixel;
- lengths are in pixels.

If you write your own functions, you are encouraged to stick to these conventions.

5.2 Parameter Limits

Individual parameters can be limited in two ways:

1. Held fixed;
2. Bounded between lower and upper limits.

To hold a parameter fixed, use the string "fixed" after the initial-value specification. E.g.:

```
X0  442.85  fixed
```

To specify lower and upper limits for a parameter, include them as a comma-separated pair following the initial-value specification. E.g.:

```
X0  442.85  441.0,443.5
```

Note that specifying *equal* lower and upper limits for a parameter (or a lower limit which is higher than the upper limit) is not allowed; to specify that a parameter value should remain constant, used the "fixed" keyword as described above. **Warning:** there should be *no* spaces before or after the comma. E.g., use "441.0,443.5", not "441.0, 443.5" or "441.0 , 443.5".

5.3 Function Labels

Optional labels can be attached to each function. This has no effect on the fits, but since the labels are copied to the output printouts and files, they can be helpful in cases where models become complicated.

The basic idea is a comment is added after the function name, where the first non-whitespace element after the comment character is the word "LABEL" (must be in all-caps), followed by whitespace, and then the rest of the line is treated as the label.

```
FUNCTION function-name # LABEL label-text
```

For example,

```
FUNCTION Exponential # LABEL nuclear disk
```

5.4 Optional Image-Description Parameters

The configuration file can, optionally, contain one or more specifications of parameters describing the whole image, which take the place of certain command-line options for computing the internal noise map. The specifications should be placed *at the beginning* of the configuration file, *before* the first function set is described. The format is the same as for other parameters in the configuration file: the name of the parameter, followed by one or more spaces and/or tabs, followed by a numerical value. E.g.,

```
GAIN 2.7
READNOISE 4.5
```

The currently available image-description parameters are (see Section 4.1 for more details about the corresponding command-line options):

- GAIN – same as command-line option `--gain` (A/D gain in electrons/ADU)
- READNOISE – same as command-line option `--readnoise` (read noise in electrons)
- EXPTIME – same as command-line option `--exptime`
- NCOMBINED – same as command-line option `--ncombined`
- ORIGINAL_SKY – same as command-line option `--sky` (original background level that was subtracted from the image)

In situations where a configuration file contains one of these specifications and the corresponding command-line option is also used, *the command-line option always overrides whatever value is in the configuration file.*

6 Standard Image Functions

Imfit comes with the following 2D image functions, each of which can be used as many times as desired. (As mentioned above, **Imfit** is designed so that constructing and using new functions is a relatively simple process.) Most of these functions use a specified radial intensity profile (e.g., Gaussian, exponential, Sérsic) with elliptical isophote shapes. Note that elliptical functions can always be made circular by setting the "ellipticity" parameter to 0.0 and specifying that it be held fixed. See Appendix A for more complete discussions of all functions, including their parameters.

- FlatSky – a uniform sky background.
- TiltedSkyPlane – a sky background in the form of an inclined plane.
- Gaussian – an elliptical 2D Gaussian function.
- Moffat – an elliptical 2D Moffat function.
- PointSource – a scaled copy of the user-supplied PSF image.
- PointSourceRot – as for PointSource, but allowing arbitrary rotation
- Exponential – an elliptical 2D exponential function.
- Exponential_GenEllipse – an elliptical 2D exponential function using generalized ellipses ("boxy" to "disky" shapes) for the isophote shapes.
- Sersic – an elliptical 2D Sérsic function.
- Sersic_GenEllipse – an elliptical 2D Sérsic function using generalized ellipses ("boxy" to "disky" shapes) for the isophotes.
- Core-Sersic – an elliptical 2D Core-Sérsic function [Graham et al., 2003, Trujillo et al., 2004].
- BrokenExponential – similar to Exponential, but with *two* exponential radial zones (with different scalelengths) joined by a transition region at $R_{\rm break}$ of variable sharpness.
- GaussianRing – an elliptical ring with a radial profile consisting of a Gaussian centered at $r = R_{\rm ring}$.
- GaussianRing2Side – like GaussianRing, but with a radial profile consisting of an asymmetric Gaussian (different values of σ for $r < R_{\rm ring}$ and $r > R_{\rm ring}$).
- GaussianRingAz – like GaussianRing, but with the peak intensity of the radial Gaussian profile varying (sinusoidally) with azimuth.
- EdgeOnDisk – the analytical form for a perfectly edge-on exponential disk, using the Bessel-function solution of van der Kruit & Searle [1981] for the radial profile and the generalized sech function of van der Kruit [1988] for the vertical profile.
- EdgeOnRing – a simplistic model for an edge-on ring, using a Gaussian for the radial profile and another Gaussian (with different σ) for the vertical profile.

- EdgeOnRing2Side – like EdgeOnRing, but using an asymmetric Gaussian for the radial profile (see description of GaussianRing2Side).
- FerrersBar2D – a 2D version of the Ferrers ellipsoid.
- FlatBar – An ad-hoc description of the outer (vertically thin) part of a relatively face-on bar in a massive galaxy (with the "flat" radial profile first identified by Elmegreen & Elmegreen [1985]).

In addition, four "3D" functions are available. With these, the intensity value for each pixel comes from line-of-sight integration through a 3D luminosity-density model, generating a projected 2D model image given input specifications of the orientation and inclination to the line of sight.

- ExponentialDisk3D – uses a 3D luminosity-density model of an axisymmetric disk with an exponential radial profile and a generalized sech function for the vertical profile (as for the EdgeOnDisk function), observed at an arbitrary inclination, to generate a projected surface-brightness image.
- BrokenExponentialDisk3D – similar to ExponentialDisk3D, except that the radial profile of the luminosity density is a broken exponential.
- GaussianRing3D – uses a 3D luminosity-density model of an elliptical ring with Gaussian radial and exponential vertical profiles.
- FerrersBar3D – uses a 3D luminosity-density model of a triaxial Ferrers [1877] ellipsoid.

A list of the currently available functions can always be obtained by running `imfit` with the "`--list-functions`" option:

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

The complete list of function parameters for each function (suitable for copying and pasting into a configuration file) can always be obtained by running `imfit` with the "`--list-parameters`" option:

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

7 Images

Imfit is designed to fit 2D astronomical images in FITS format, where pixel values are some form of linear surface-brightness (or surface density) measurement. The default internal error calculations (see Section 8.2, below) assume that pixel values are integrated counts (e.g., ADUs), which can be converted to detected photons using the A/D gain (provided by the "`--gain`" option, or by the GAIN keyword in a configuration file). However, since **Imfit** can also accept a user-supplied noise/error image in FITS format, you can use any linear pixel values as long as the corresponding noise image is appropriately scaled to match (and you're using χ^2 statistics).

If your image is in counts/second, you can either multiply it by the exposure time to recover the integrated counts, or include the actual exposure time via the "`--exptime`" option (or the EXPTIME keyword in a configuration file).

If the image is an *average* of N input images of the same exposure time, you can either multiply the image by N *or* use the "`--ncombined`" option to tell **Imfit** how to adjust the error estimations. The latter option is slightly better, because **Imfit** will also scale the read noise accordingly (if χ^2 statistics are being used and the read noise is nonzero).

Imfit does *not* assume the presence of any particular header keywords in the FITS file.

7.1 Specifying Image Subsections, Compressed Images, etc.

Image Subsections

In many cases, you may want to fit an object which is much smaller than the whole image. You can always make a smaller cutout image and fit that, but it may be convenient to specify the image subsection directly. You can do this using a subset of the image-section syntax of CFITSIO (which will be familiar to you if you've ever worked with image sections in IRAF). An example:

```
ic3478rss_256.fits[45:150,200:310]
```

This will fit columns 45–150 and rows 200–310 of the image (column and row numbering starts at 1 and is inclusive, so "[1:10]" means column or row numbers 1 through 10). Pixel coordinates in the configuration (and output) files refer to locations within the *full* image, not relative positions within the subsection. Note that this works as-is when running **Imfit** from within the Bash shell; for the C and Z shells, you need to enclose the image name + specification in quotation marks, or (Z shell only, I believe) just precede the entire `imfit` command with "`noglob`".

The only kind of image section specification that's allowed is a simple [x1:x2,y1:y2] format, though you can specify all of a particular dimension using an asterisk (e.g., [*,y1:y2] to specify the full range of x values). Exception: you can also specify an image extension number for a multi-extension FITS file; see below.

Obviously, if you are also using a mask image (and/or a noise image), you should specify the same subsection in those images!

Image Extensions; Compressed Files

By default, **Imfit** will try to read the first ("primary") Header Data Unit (HDU) in a FITS file. If this is *not* a proper 2D image – e.g, if it is an empty, header-only HUD; if it is a 1D spectrum; if it is a table; etc. – then **Imfit** will report an error and quit.

You *can* specify a particular extension in a multi-extension FITS file using the CFITSIO format, e.g.:

ic3478rss.fits[2]

ic3478rss.fits[2][45:150,200:310]

This particular numbering scheme is zero-based: the primary HDU is 0, the first extension (which is the second HDU) is 1, etc.

You can also fit (or generate) images which have been compressed with gzip or Unix compress – e.g., `ic3478rss_256.fits.gz`. Images, masks, etc., can even be read via `http://` or `ftp://` URLs which point directly to accessible FITS files; you cannot *save* files to URLs, however.

8 Extras for Fitting Images

8.1 Masks (and automatic masking of certain pixels)

A mask image can be supplied to `imfit` by using the command-line option `--mask`. The mask image should be a FITS file with the same dimensions as the image being fitted (IRAF `.pl` mask files are not recognized, but these can be converted to FITS format within IRAF). The default is to treat zero-valued pixels in the mask image as *good* and pixels with values > 0 as *bad* (i.e., to be excluded from the fit); however, you can specify that zero-valued pixels are *bad* with the command-line flag `--mask-zero-is-bad`.

Any pixels in the data image, the noise/error/weight image (see below), *or* the mask image which have non-finite values (i.e., NaN, $\pm\infty$) will automatically be considered part of the mask (i.e., corresponding pixels in the data image will be excluded from the fit).

8.2 Noise, Variance, or Weight Maps

(See Section 9.1 for background on the different fit statistics.)

By default, `imfit` uses χ^2 as the statistic for minimization. As part of this process, `imfit` normally calculates an internal weight map, using the input pixel intensities, the A/D gain, any previously subtracted background level, and the read noise to estimate Gaussian errors σ_i for each pixel i. In formal terms, the error-based weight map is $w_i = 1/\sigma_i^2$, with the dispersion (in ADU) defined as

$$\sigma_i^2 \;=\; (I_{d,i} + I_{\rm sky})/g_{\rm eff} \;+\; N_{\rm c}\,\sigma_{\rm rdn}^2/g_{\rm eff}^2\,, \tag{1}$$

where $I_{d,i}$ is the data intensity in counts/pixel, $I_{\rm sky}$ is the original subtracted sky background (if any), $\sigma_{\rm rdn}$ is the read noise (in electrons), $N_{\rm c}$ is the number of separate images combined (averaged or median) to form the data image, and $g_{\rm eff}$ is the "effective gain" (the product of the A/D gain, $N_{\rm c}$, and optionally the exposure time, if the pixel units are counts/sec). As an alternative, the *model* intensity values $I_{m,i}$ can be used instead of $I_{d,i}$, by specifying the `--model-errors` command-line flag. This is marginally slower (since the weights must be recalculated every time the model image is updated), but can lead to smaller biases in fitted parameters [see Humphrey, Liu, & Buote, 2009, Erwin, 2015].

The weights are then used in the χ^2 calculation:

$$\chi^2 \;=\; \sum_{i=0}^{N} w_i\,(I_{m,i} - I_{d,i})^2, \tag{2}$$

where $I_{m,i}$ and $I_{d,i}$ are the model and data intensities in counts/pixel, respectively. (Masking is handled by setting $w_i = 0$ for masked pixels.)

If you have a pre-existing error map as a FITS image, you can tell `imfit` to use that instead, via the `--noise` command-line option. By default, the pixel values in this image are assumed to be errors σ_i in units of ADU/pixel. If the values are *variances* (σ_i^2), you can specify this with the `--errors-are-variances` flag. You can also tell `imfit` that the pixel values in the noise map are actual *weights* w_i (e.g., if the values are inverse variances) via the `--errors-are-weights` flag, if that happens to be the case. (If a mask image is supplied, the weights of masked pixels will still be set to 0, regardless of their individual values in the weight image.)

Internally, the weight map for χ^2 calculations is actually stored using the *square root* $\sqrt{w_i}$ of the weights as defined above. This is because the Levenberg-Marquardt solver relies on access to

a vector of "deviate" values, which are the square roots of individual terms in the formal χ^2 sum:

$$\sqrt{w_i}\,(I_{m,i} - I_{d,i}). \tag{3}$$

Important Note: Prior to version 1.3, `imfit` incorrectly interpreted the `--errors-are-weights` flag as indicating that the pixel values of the input "noise map" were $\sqrt{w_i}$, and the `--save-weights` option caused `imfit` to output a FITS file with $\sqrt{w_i}$ values. Starting with version 1.3, `imfit` treats those two commandline-flags/options as referring to w_i as defined above, and conversions to and from the internal $\sqrt{w_i}$ values are handled correctly.

In the case of minimizing the Poisson-MLR statistic or the related Cash statistic C, the only relevant external maps are masks. For the Cash statistic C, the actual minimized quantity is:

$$C \;=\; 2\sum_{i=0}^{N} w_i\,(I'_{m,i} \;-\; I'_{d,i} \ln I'_{m,i}), \tag{4}$$

where $I'_{m,i}$ and $I'_{d,i}$ are the model and data intensities in counts/pixel, multiplied by the effective gain $g_{\rm eff}$ as defined above. In this case, all weights are automatically = 1, except for masked pixels, which are still set = 0. (Note that any attempt to specify the read noise is ignored, since Gaussian noise terms cannot be accommodated in the Cash-statistic or Poisson-MLR statistic computation; see, e.g., Erwin 2015.)

The Poisson-MLR statistic includes extra terms based on the data values:

$$\mathrm{PMLR} \;=\; 2\sum_{i=1}^{N} w_i\,\left(I'_{m,i} - I'_{d,i} \ln I'_{m,i} + I'_{d,i} \ln I'_{d,i} - I'_{d,i}\right). \tag{5}$$

Note that `imfit` does *not* try to obtain information (such as the A/D gain or read noise) from the FITS header of an image. This is primarily because there is little consistency in header names across the wide range of astronomical images, so it is difficult pick one name, or even a small set, and assume that it will be present in a given image's header; this is even more true if an image is the result of a simulation.

8.3 PSF Convolution (Including Oversampled PSFs)

To simulate the effects of seeing and other telescope resolution effects, model images can be convolved with a PSF (point-spread function) image. This uses an input FITS file which contains the point spread function. The actual convolution uses Fast Fourier Transforms of the internally-generated model image and the PSF image to compute the output convolved model image.

PSF images should be square, ideally with width N = an odd number of pixels, and the PSF should be centered in the central pixel: $x = y = N$ **div** 2 + 1, where **div** is integer division (remember that **Imfit** uses the IRAF/DS9 convention for pixel numbering: the center of the first pixel in the image is at $(x, y) = (1.0, 1.0)$, etc.). E.g., for a 25 × 25-pixel PSF, the center should be at $(x, y) = (13.0, 13.0)$. An off-center PSF can certainly be used, but the resulting convolved model images will be shifted! The PSF does *not* need to be normalized, as `imfit` will automatically normalize the PSF image internally (you can use the "`--no-normalize`" flag to tell `imfit` to skip normalizing the PSF).

Although **Imfit** uses a multi-threaded version version of the FFTW library, which is itself quite fast, adding PSF convolution to the image-fitting process *does* slow things down considerably. **See Section 11.1 for notes on how to maximize multi-threaded performance with PSF convolution.**

Convolution with Oversampled PSFs

The default convolution performed by **Imfit** uses a PSF image and a model image which have the same spatial scaling as the data image. While this is adequate for most purposes, it is a somewhat crude approximation to the true situation, since in reality the PSF convolution process is effectively infinite-resolution and takes place prior to the sampling of the image in pixel space by the detector.

A better approach would be to construct a model image at higher resolution and convolve it with a higher-resolution PSF, then downsample the resulting image to the same pixel scale as the data image. Unfortunately, this can be quite time-consuming (and memory-consuming), due to the larger images required.

A compromise is to perform a standard convolution on the entire image, and then replace a small subset of the model image (e.g., around a bright point source where more accurate PSF convolution is desirable) with the result of a higher-resolution convolution. This can be done in **Imfit** with the "`overpsf`" command-line options.

To convolve an image region with an oversampled PSF, you need to use *three* command-line options:

- `--overpsf` to specify the FITS file which holds the oversampled PSF image;
- `--overpsf_scale` to specify the *amount* of oversampling corresponding to the PSF image (i.e., how many oversampled pixels fit into an original pixel along one axis; this must be an integer);
- `--overpsf_region` to specify what part of the image should have convolution with the oversampled PSF applied to the model.

For example,

```
$ imfit --overpsf psf5.fits --overpsf_scale 5 \
--overpsf_region 490:500,620:630
```

tells **Imfit** that you want the region [490:500,620:630] within the image to use oversampled PSF convolution, that the oversampled PSF is contained in the file `psf5.fits`, and that the oversampling scale is a factor of 5 (i.e., one normal pixel in the image corresponds to 5×5 pixels in the oversampled model and the PSF).

As is true for the X0 and Y0 parameter values, the oversampled-region specification refers to the coordinate system of the entire input image, *not* relative coordinates within a subsection.

More than one oversampled region can be specified by repeating the `--overpsf_region` command; this will use the same oversampled PSF and oversampling scale.

In some cases, one may wish to use *different* oversampled PSF images for different parts of the image, particularly if it is known that the PSF varies significantly across the image (and if one has representative oversampled PSF images). In this case, multiple oversampled PSF images (with, potentially, different oversampling scales) can be specified, each with its corresponding image region. To do this, extra invocations of the `--overpsf` and `--overpsf_scale` commands should be given, along with one `--overpsf_region` per oversampled PSF.

To summarize, one can do the following with `imfit-mcmc`, `imfit-mcmc`, or `makeimage`:

1. Convolve one subsection of the model image with an oversampled PSF image, e.g.

```
  --overpsf=oversampled_psf.fits --overpsf_scale=4 \
--overpsf_region=200:205,260:265
```

2. Convolve several subsections of the model image with the same oversampled PSF image, e.g.

```
  --overpsf=oversampled_psf.fits --overpsf_scale=4 \
--overpsf_region=200:205,260:265 --overpsf_region=310:315,581:590
```

3. Convolve several subsections of the model image, each with a different oversampled PSF image.

```
  --overpsf=oversampled_psf1.fits --overpsf_scale=4 \
--overpsf_region=200:205,260:265 \
--overpsf=oversampled_psf2.fits --overpsf_scale=4 \
--overpsf_region=830:835,1010:1015
```

9 What Gets Minimized During a Fit, and How to Minimize It

9.1 Fit Statistics: χ^2 or Poisson Statistics

(See Erwin [2015] for a more detailed discussion of the statistical background and the effects of minimizing different fit statistics.)

Imfit attempts to find the best-fitting model via a maximum-likelihood approach. In practice, this is done by *minimizing* a maximum-likelihood statistic defined as $-2\ln\mathcal{L}$, where $\mathcal{L}$ is the likelihood of a given model. The rest of this section describes the different versions of $-2\ln\mathcal{L}$ which can be minimized to achieve the best fit. Note that this is equivalent to a Bayesian approach which assumes constant priors on all model parameters.

By default, **Imfit** uses a Gaussian-based maximum likelihood statistic which is the total χ^2 for the model. This is defined by computing the sum over individual pixels i:

$$-2\ln\mathcal{L} \;=\; \chi^2 \;=\; \sum_{i=0}^{N} w_i\,(I_{m,i} - I_{d,i})^2, \tag{6}$$

where N is the total number of pixels, $I_{m,i}$ and $I_{d,i}$ are the model and data intensities for the individual pixels, and w_i are per-pixel weights. These weights can be supplied directly by the user in a weight map, but normally they are derived from the dispersions σ_i of the per-pixel Gaussian errors:

$$w_i \;=\; 1/\sigma_i^2. \tag{7}$$

There are three possible sources for the σ_i values. The default approach is to estimate them from the data values, assuming that the counts per pixel follow the Gaussian approximation of Poisson statistics, so that $\sigma_i = \sqrt{I_{d,i}}$. An alternate method is to use the *model* values (via the `--model-errors` flag) instead: $\sigma_i = \sqrt{I_{m,i}}$. Finally, the user can supply a "noise" image (via the `--noise` option) which specifies the individual σ_i (or σ_i^2) values directly.

An arguably more accurate approach – especially in the regime of low counts per pixel, as is often the case for, e.g., X-ray images – is to minimize a maximum likelihood statistic based directly on Poisson statistics (rather than Gaussian approximation thereof). One way of doing this involves computing the Cash statistic C:

$$-2\ln\mathcal{L} \;=\; C \;=\; 2\sum_{i=0}^{N} w_i\,(I_{m,i} \;-\; I_{d,i}\ln I_{m,i}). \tag{8}$$

In this case, all weights are automatically = 1, except for masked pixels, which are still set = 0. Specifying the Cash statistic is done with the `--cashstat` command-line flag.

A drawback of the Cash statistic is that it *cannot* be minimized with the Levenberg-Marquardt algorithm, since the latter assumes that all terms in the summation are ≥ 0, which is usually not true for the Cash statistic. Fortunately, there is also the Poisson-MLR (maximum likelihood ratio) statistic PMLR, which includes extra terms based on the data values:

$$-2\ln\mathcal{L} \;=\; \mathrm{PMLR} \;=\; 2\sum_{i=1}^{N} w_i\;\left(I_{m,i} - I_{d,i}\ln I_{m,i} + I_{d,i}\ln I_{d,i} - I_{d,i}\right). \tag{9}$$

Since the extra terms do not depend on the model values, they remain unchanged during the minimization and do not affect it. However, they have the key advantage of ensuring that all individual-pixel values are always ≥ 0, which makes it possible to use the Levenberg-Marquardt

algorithm to minimize PMLR. Use of PMLR is done via the `--poisson-mlr` flag (or its shortened version `--mlr`).[13]

(See Section 8.2 above for more details on how the data and model intensities, σ_i values, and weights are actually calculated, including contributions from previously-subtracted backgrounds, A/D gain, Gaussian read noise, and combination with masks.)

9.2 Minimization Options: Levenberg-Marquardt, Nelder-Mead Simplex, Differential Evolution

The default method for χ^2 (or PMLR) minimization used by **Imfit** is the Levenberg-Marquardt algorithm, based on the classic MINPACK-1 implementation [Moré, 1978] with enhancements by Craig Markwardt.[14] This is very fast and robust, and is the most extensively tested algorithm in **Imfit**, but requires an initial guess for the parameter values and can sometimes become trapped in local minima in the χ^2 landscape. (In addition, it is *not* appropriate if one is using the Cash statistic C, since the latter can have both per-pixel and total values < 0, which least-squares algorithms like Levenberg-Marquardt cannot handle.)

An alternative algorithm, available via the `--nm` flag, is a version of the Nelder-Mead simplex, as implemented by the NLopt library.[15] This is significantly slower than Levenberg-Marquardt minimization (~ 5–10 times slower for fits with one or two components), but significantly faster than Differential Evolution (below). Like the Levenberg-Marquardt method, it requires an initial guess for the parameter values, but is considered less likely to become trapped in local minima in the fit-statistic landscape. Unlike the L-M algorithm, it can be used for minimizing C (as well as χ^2 and PMLR). If you compile **Imfit** from source and want to use this algorithm, you need to have the NLopt library installed on your system.

The second alternate algorithm is available via the `--de` flag. This minimizes the fit-statistic using Differential Evolution (DE) [Storn & Price, 1997], a genetic-algorithms approach.[16] It has the drawback of being ~ an order of magnitude slower than the Nelder-Mead simplex method, and *much* slower (~ two orders of magnitude) than Levenberg-Marquardt minimization. For example, fitting a single Sérsic function to the 256 × 256 image in the `examples/` subdirectory takes ~ 60 times as long when using Differential Evolution as it does when using L-M minimization. It does has the advantage of being the least likely (at least in principle) of being trapped in local minima in the fit-statistic landscape; it also does *not* require or use initial guesses for the parameter values.

The Differential Evolution algorithm does, however, *require* lower and upper limits for *all* parameters in the configuration file (see Section 5.2); this is because DE generates initial parameter-value "genomes" by random sampling from within the ranges specified by the parameter limits. The format of the configuration file still requires that initial-guess values be present for all parameters as well, but these are actually ignored by the DE algorithm. (This is to ensure that the same configuration file can be used with all minimization routines.)

The standard implementation of Differential Evolution uses simple uniform sampling to generate the initial guesses: the initial value for a given parameter in a given "genome" is generated by uniformly sampling from within that parameter's limits. In some cases, this might leave the full parameter space unevenly sampled, which *might* make it harder to find the global fit-statistic

[13] PMLR is actually defined as $-2\ln(\mathcal{L}/\mathcal{L}_{\max})$; see Section 4.1.3 of Erwin [2015].

[14] Original C version available at `https://www.physics.wisc.edu/~craigm/idl/cmpfit.html`

[15] `https://nlopt.readthedocs.io/en/latest/`

[16] `http://www.icsi.berkeley.edu/~storn/code.html`

minimum if there are multiple local minima. As an alternative, you can use the `--de-lhs` option, which tells **Imfit** to use Latin hypercube sampling when generating the initial parameter guesses. In practice, it's not clear this improves the convergence speed of typical DE fits, but it might be useful in some cases (ideally, one should probably run experiments to see if it makes things better or worse).

(An additional set of minimization algorithms – all of the "local derivative-free" algorithms in the NLopt library[17] – can be invoked with the `--nlopt` option. Most if not all of these seem to be slower and/or less robust than the Nelder-Mead simplex algorithm (which is in fact one of these NLopt algorithms) for image-fitting purposes, so they are probably not worth using. Nonetheless, it's possible that some image-fitting scenarios might benefit from one of these algorithms. See Section 4.1 for details on the use of the `--nlopt` option.)

TBD. [more details of DE implementation]

Note that the N-M simplex and DE algorithms do *not* produce uncertainty estimates for the best-fitting parameter values, in contrast to the Levenberg-Marquardt approach. However, the L-M error estimates are themselves only reliable if the minimum in the χ^2 landscape is symmetric and parabolic, and if the errors for the input image are truly Gaussian and well-determined. See Sections 10.2 and 13 for an alternative way of estimating the parameter uncertainties.

The fact that the minimization algorithms are relatively decoupled from the rest of the code means that future versions of **Imfit** could include other minimization techniques, or that ambitious users can add such techniques themselves.

9.3 Controlling the Tolerance for Minimization

All three minimization algorithms have stop conditions based on fractional changes in the fit-statistic (χ^2, C, or PMLR) value: if further iterations do not improve the current value by more than FTOL × the fit statistic, then the algorithm declares success and terminates. (In the case of DE, the test condition is actually no further improvement after 30 generations.) The default value of FTOL is 10^{-8}, which seems to do a reasonable job for typical images (in fact, it's probably overkill). If you want to experiment with different values of FTOL, you can do this via the command-line option `--ftol`.

There are also built-in stopping conditions based on a maximum number of iterations for the L-M algorithm (1000), a maximum number of generations for DE (600), or a maximum number of function evaluations (i.e., computations of model images) for the Nelder-Mead simplex algorithm (10000 times the number of free parameters).

[17] `https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/`

10 Outputs

10.1 Main Outputs

Outputs Prior to the Fit

By default, **Imfit** prints a set of running comments to the command line as it processes the input parameters. This includes file names, command-line options, and general values from the config file (e.g, image gain). This is potentially useful for debugging failed fits: e.g., check the output to make sure the correct image file was specified, that any mask file was recognized (and its pixel values treated appropriately), etc.

As part of this setup phase, **Imfit** will make an estimate of the total memory needed for the fit (including the intermediate arrays used by the fast Fourier transforms if PSF convolution is being done, etc.). This should be treated as an order-of-magnitude estimate (and in fact the actually memory used will be slightly larger than the estimate), but is useful to catch cases where a proposed fit might run up against the memory limits of your computer. An extra warning message will be printed if the estimated memory use is > 1 GB, though **Imfit** will still go ahead and attempt the fit. (If the memory use really is too large for the computer, the program will probably be killed by the operating system, which may or may not print anything useful about *why* the program was killed.)

Outputs from the Fit

Assuming that the fitting process converges, **Imfit** will print a summary of the results, including the final, best-fitting parameter values. The output parameter list is in the same format as the configuration file, except that if the L-M algorithm is used, its error estimates are listed after each parameter value.[18] These error estimates are separated from the parameter values by "#"; this means that you can copy and paste the parameter list into a text file and use that file as an input configuration file for `imfit`, `imfit-mcmc`, or `makeimage`, since those programs will treat everything after "#" on a line in the configuration file as a comment.

The best-fitting parameters will also be written to an output text file (the default name for this is `bestfit_parameters_imfit.dat`; use `--save-params` to specify a different name). The output file will also include a copy of the original command used to start **Imfit** and the date and time it was generated; these are commented out so that the file can be subsequently used as an **Imfit** or `makeimage` configuration file without modification.

The final value of the fit statistic is also printed; if the fit statistic was χ^2 or PMLR, then the "reduced" χ^2 or PMLR (which accounts for the total number of unmasked pixels and non-fixed parameter values)[19] is also printed. Finally, two alternate measures of the fit are also printed: the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC). These two are, in principle, useful for comparing different models fit to the same data. They are defined in the usual sense (for AIC, **Imfit** uses the "corrected" version, which accounts for cases of few data

[18] No in-line error estimates are produced if the Nelder-Mead simplex or Differential Evolution algorithms were used for the minimization.

[19] The reduced values should be interpreted with caution; it is valid in absolute terms only if the noise has been correctly estimated *and* if all differences between the model and the data are solely due to noise, which is rarely true for galaxies and other astronomical objects.

points):

$$\text{AIC} = -2\ln\mathcal{L} + 2k + \frac{2k(k+1)}{n-k-1} \quad (10)$$

$$\text{BIC} = -2\ln\mathcal{L} + k\ln n, \quad (11)$$

where $\mathcal{L}$ is the likelihood of the best-fit model, k is the number of free parameters, and n is the number of data points (i.e., unmasked pixels). The value of $-2\ln\mathcal{L}$ is equal to the best-fit value of the chosen fit statistic: e.g., $-2\ln\mathcal{L} = \chi^2$ for fits that minimize χ^2, and = PMLR for fits using that statistic.

A short summary of the results of the fitting process – minimization algorithm and its final status, fit-statistic value, etc. – is also included in the output text file; these lines (like the ones listing the original command line and the timestamp) are commented out.

By default, no model or residual (image − model) images are saved, but the command-line options `--save-model` and `--save-residual` can be used to specify output names for those images, and corresponding FITS files will be saved to disk.

10.2 Uncertainties on Parameter Values: L-M Estimates vs. Bootstrap Resampling

The (default) Levenberg-Marquardt minimization algorithm used by **Imfit** automatically generates a set of (symmetric) uncertainty estimates for each free parameter at the conclusion of the fitting process; as noted above, these are printed to the terminal as part of the summary output.

These values come from the covariance matrix derived from the final Hessian matrix corresponding to the best-fit solution, and should be viewed with caution: for example, they assume that the χ^2 landscape in the vicinity of the best-fit solution is parabolic. In practice, they should probably be seen as *lower limits* on the uncertainty.

The other fitting algorithms used by **Imfit** do not compute gradients in the fit landscape, and so they do not produce automatic uncertainty estimates. Although one could certainly take the solution of a χ^2 or PMLR fit done with the N-M simplex or DE algorithms and use it as input to a L-M run of **Imfit**, thus generating L-M–based uncertainties, this is not possible when using the Cash statistic C.

As an alternative to the L-M uncertainty estimates, **Imfit** offers the option of bootstrap resampling. This is done with the "`--bootstrap`" command-line option, which takes the *number* of iterations as its corresponding value; e.g.

```
imfit someimage.fits -c config.dat [...] --bootstrap=200
```

Each iteration of bootstrap resampling generates a new data image by sampling pixel values (with replacement) from the original data image, and then re-runs the fit to generate a new set of parameter values.[20] After n iterations, the combined set of bootstrapped parameter values can be used as a distribution for estimating confidence intervals; **Imfit** finds and outputs the 68.3% confidence intervals and the standard deviation for each parameter. While this approach may produce more plausible uncertainties for the best-fit parameters – and can be used with both χ^2 and Cash-statistic or Poisson-MLR statistic minimization and as an adjunct to any of the three minimization algorithms – it *is* slow, as one is essentially repeating (a somewhat faster version

[20] To speed things up, the original best-fit parameters are used as starting values for the new fit.

of) the fitting process *n* times. Ideally, one should do at least 200 iterations – 1000 or more is preferable – to get reasonably consistent confidence intervals. Since this can take *much* longer than the original fitting process, it is probably not a good idea to use bootstrap resampling when one is engaged in exploratory fitting, but to instead postpone it until one is reasonably certain one has the final fit. To keep things as fast as possible, **Imfit** automatically chooses L-M minimization for the bootstrap process – unless the Cash statistic is being used, in which case the N-M simplex method is used.[21]

Using the `--save-bootstrap` command, one can provide a filename for saving the best-fitting parameters from all the individual resampled fits; these are written as one line per fit. This allows more detailed analysis of the parameter distributions, including potential correlations between parameters.

The python module `imfit.py` (in the `python` subdirectory) contains a function (GetBootstrapOutput) which reads in a bootstrap save file and returns a list of the parameter names and a Numpy array with the saved bootstrap fits.

See also Section 13 for how to do Markov Chain Monte Carlo analysis using **Imfit** models.

[21] If **Imfit** was compiled without the NLopt library, then the N-M simplex method is not available and **Imfit** uses DE instead – which will make the bootstrap estimation *very* slow.

11 Miscellaneous Notes

11.1 Faster PSF Convolution with Multithreading

If **Imfit** is compiled with multi-threading support (the default), there is a subtle "gotcha" when it comes to using PSF convolution. If multi-threading is enabled, then by default **Imfit** will choose the maximum possible number of threads to use, which is the maximum number of hardware threads that the operating system reports. In older multi-core CPUs, this is the number of actual (physical) cores on the CPU chip (or on all CPU chips if there are multiple CPUs). This is ideal for such systems.

However, many modern multi-core CPUs include "simultaneous multithreading" (called "hyperthreading" for Intel CPUs). This involves executing multiple (usually 2) threads *per core*, and the operating system will usually include this in its hardware thread (or "logical core") count. (E.g., an 8-core Intel i9 CPU will be reported as having 16 threads available, two for each core.)

For computations that *don't* involve PSF convolution, this is fine, and the results are (slightly) faster than if one used one thread per physical core. However, this does *not* work for PSF convolution: the computations can sometimes be *slower* if more threads than physical cores are used – sometimes a factor of several times slower!

A somewhat related problem seems to occur with the new Apple Silicon (aka M1, M2) CPUs in modern Mac computers. Although these do *not* have simultaneous multithreading, they do have different CPU core *types*: half or more of the cores are "high-performance" (fast), while the remainder are "high-efficiency" (useful for low power consumption, but slower).

Because using more threads than there are physical (or high-efficiency) cores seems to either be no faster than, or signficantly slower than, using the same number of threads as there are physical (or high-efficiency) cores, **Imfit** now (as of version 1.9) defaults to the latter case. (This requires some trickery in the case of Linux, as it is not easy for a program to learn how many physical cores there are.)

For versions of **Imfit** prior to 1.9, you should use the `--max-threads` option and set this equal to the number of *physical* cores (or high-performance cores on an Apple Silicon CPU) on your computer.

11.2 Memory Usage

As noted above, **Imfit** will print an estimate of the memory needed for a fit when it is starting up. In most cases, this will be of merely academic interest, unless you are planning to fit very large images and/or convolve with very large PSFs.

If you want to have an idea of what the memory usage might be *before* running **Imfit**, you can start by estimating the memory needed (in bytes) for the data image

$$N_{\mathrm{data}} = 8\, N_x N_y, \tag{12}$$

where N_x and N_y are the image dimensions in pixels.

In the case of no PSF convolution, the total memory (in bytes) will be a minimum of $\sim 4\, N_{\mathrm{data}}$. If Levenberg-Marquardt minimization (the default) is being used, then the total memory needed will actually be of order $(7 + n_{\mathrm{free}})\, N_{\mathrm{data}}$, where n_{free} is the number of free parameters in the model. (This is mostly due to the Jacobian matrix and associated arrays used internally by the L-M minimizer.) Thus, fitting a 1000×1000 pixel image using Levenberg-Marquardt minimization and a single elliptical Sérsic model with no fixed parameters ($n_{\mathrm{free}} = 7$) will require ~ 106 MB.

If PSF convolution (using a PSF image with dimensions $N_{x,\mathrm{PSF}} \times N_{y,\mathrm{PSF}}$) is being done, then more memory will be needed. In this case, you should also calculate the memory needed for the internal model image

$$N_{\mathrm{model}} = 8\, N_{x,m} N_{y,m}, \tag{13}$$

where $N_{x,m} = N_x + 2\, N_{x,\mathrm{PSF}}$ and $N_{y,m} = N_y + 2\, N_{y,\mathrm{PSF}}$, and the memory needed for the complex FFT-related arrays

$$N_{\mathrm{FFT}} = 16\, N_{x,\mathrm{FFT}} N_{y,\mathrm{FFT}}, \tag{14}$$

where $N_{x,\mathrm{FFT}} = N_x + 3\, N_{x,\mathrm{PSF}}$ and $N_{y,\mathrm{FFT}} = N_y + 3\, N_{y,\mathrm{PSF}}$. The total memory needed will then be (for non–L-M minimization) of order

$$N_{\mathrm{data}} + 3\, N_{\mathrm{model}} + 6\, N_{\mathrm{FFT}} \tag{15}$$

or

$$(4 + n_{\mathrm{free}}) N_{\mathrm{data}} + 3\, N_{\mathrm{model}} + 6\, N_{\mathrm{FFT}} \tag{16}$$

when doing L-M minimization. Using the same example as before (fitting a 1000×1000 pixel image with a single Sérsic model) but including convolution with a 100 × 100 pixel PSF image would thus require ~ 270 MB, or ~ 195 MB using a minimization algorithm other than L-M. Fitting, say, a 5000×5000 pixel image using a 1000×1000 pixel PSF image would require almost 9 GB of memory!

The estimation done by **Imfit** when starting up is slightly more accurate, but these notes should suffice to help you figure out whether you're in the general vicinity of running out of memory with a particular fit.

12 Makeimage

Imfit has a companion program called `makeimage`, which will generate model images using the same functions (and parameter files) as **Imfit**. In fact (as noted above), the output "best-fitting parameters" file generated by **Imfit** can be used as input to `makeimage`, as can an **Imfit** configuration file.

`Makeimage` *does* require an output image size. This can be specified via command-line flags ("`--ncols`" and "`--nrows`"), via specifications in the configuration file (see below), or by supplying a reference FITS image ("`--refimage` *image-filename*"); in the latter case, the output image will have the same dimensions as the reference image.

`Makeimage` can also be run in a special mode to estimate the magnitudes and fractional luminosities of different components in a model.

12.1 Using `Makeimage`

Basic use of `makeimage` from the command line looks like this:

`$ makeimage` [options] *config-file*

where *config-file* is the name of the **Imfit**-style configuration file which describes the model.

As for **Imfit**, the "options" are a set of command-line flags and options (use "`makeimage -h`" or "`makeimage --help`" to see the complete list). Options must be followed by an appropriate value (e.g., a filename, an integer, a floating-point number); this can be separated from the option by a space, or they can be connected with an equals sign.

Some notable and useful command-line flags and options include:

- `-o, --output` *filename* – filename for the output model image (default = "modelimage.fits").
- `--refimage` *filename* – existing reference image to use for determining output image dimensions.
- `--ncols` *N_columns* – number of columns in output image
- `--nrows` *N_rows* – number of rows in output image
- `--psf` *psf-image* – specifies a FITS image to be convolved with the model image. (The "oversampled PSF" options that `imfit` uses can also be used with `makeimage`.)
- `--nosave` – do *not* save the model image (useful for testing purposes, or when estimating fluxes with `makeimage`)
- `--timing` *N* – generates the specified model image *N* times and computes the average time taken by the computation (no output image will be saved)
- `--list-functions` – list all the functions `makeimage` can use
- `--list-parameters` – list all the individual parameters (in correct order) for each functions that `makeimage` can use

12.2 Configuration Files for `Makeimage`

The configuration file for `makeimage` has essentially the same format as that for **Imfit**; any parameter limits that might be present are ignored.

Optional general parameters like GAIN and READNOISE are ignored, but the following optional general parameters are available:

- NCOLS – number of columns for the output image (x-size)
- NROWS – number of rows for the output image (y-size)

12.3 Generating Single-Function Output Images

`Makeimage` can also output individual images for each function in the configuration file. For example, if the configuration file specifies a model with one Sérsic function and two exponential functions, `makeimage` can generate three separate FITS files, in addition to the (standard) sum of all three functions. This is done with the `--output-functions` option:

`--output-functions` *root-name*

where *root-name* is a string that all output single-function filenames will start with. The single-function filenames will be sequentially numbered (starting with 1) according to the order of functions in the configuration file, and the name of each function will added to the end; the resulting filenames will have this format:

*root-name*N_*function-name*`.fits`

Using the example specified above (a model with one Sérsic and two exponential functions), one could execute the following command

`$ makeimage` *config-file* `--output-functions mod`

and the result would be three FITS files, named `mod1_Sersic.fits`, `mod2_Exponential.fits`, and `mod3_Exponential.fits` (in addition to `model.fits`, which is the sum of all three functions).

12.4 Using `Makeimage` to Estimate Fluxes and Magnitudes, B/T Ratios, etc.

Given a configuration file, you can use `makeimage` to estimate the total fluxes and magnitudes of different model components. For some components – e.g., the purely elliptical versions of the Gaussian, Exponential, and Sérsic functions – there are analytical expressions which could be used. But since **Imfit** and `makeimage` are designed to use arbitrary functions, including ones which do not have analytical expressions for total flux, `makeimage` normally estimates the flux for each component by internally constructing a large model image for each component function in the configuration file, with the component centered within this image, and then summing the pixel values of that image. (For certain image functions – e.g., Gaussian, Exponential, Sersic – the analytic total flux value is calculated instead.) The output includes a list of total and relative fluxes (i.e., what fraction of the total flux each component makes up) for each component in the model image – and their magnitudes, if a zero point is supplied.

There are two command-line options to do this. The first prints the results to the console; the second saves the results to an output text file.

```
$ makeimage config-file --print-fluxes
```

```
$ makeimage config-file --save-fluxes filename
```

Useful command-line flags and options:

- `--estimation-size` *N_columns_and_rows* – size of the (square) image to construct (the default size is 5000 pixels on a side)
- `--zero-point` *value* – zero point Z for converting total counts to magnitudes:

$$m = Z - 2.5 \log_{10}(\text{counts}) \tag{17}$$

This enables you to compute things like bulge/total ratios – but it's up to you to determine which component(s) should be considered "bulge", "disk", etc.

When run in this mode, `makeimage` will *not* produce an output image file.

13 Markov Chain Monte Carlos Analysis with `imfit-mcmc`

Imfit includes a separate program for doing Markov chain Monte Carlo (MCMC) analysis of galaxy image models: `imfit-mcmc`. This uses the exact same surface-brightness models, statistical options, and input configuration files as `imfit`, but instead of solving for the "best-fitting" model, it produces an estimate of the posterior-probability (i.e., likelihood) distribution. Although it certainly *can* be used to "fit" models to data – if the user is willing to adopt some convention for identifying "best-fit" values from the posterior distribution[22] – it is probably more useful as a way of determining uncertainty ranges for, and correlations between, model parameters.

The inputs for `imfit-mcmc` are the same as for `imfit`: data image, model configuration file, optional PSF image(s), statistical model (χ^2 based on data, model, or input error map; Poisson MLR statistic), etc. The outputs are different: instead of a best-fitting parameter file and optional outputs such as best-fit model image, `imfit-mcmc` saves multiple output text files, each containing one of the Markov chains (there will be one chain, and thus one file, for each free parameter in the model). These are meant to be analyzed subsequently by the user; some simple Python code to help with this is provided.

In essence, you can use the same data and configuration file to do both a standard fit with `imfit` and MCMC analysis of the model; the latter can then be used to determine confidence intervals, look for multiple modes in the posterior distribution, identify correlations between parameters, etc.

It is important to bear in mind that MCMC analysis takes *much* longer than a simple fit, even a fit done using the Differential Evolution (DE) minimizer. For example, the `examples/` subdirectory that comes with **Imfit** contains a 256 × 256-pixel SDSS *r*-band image of the dE galaxy IC 3478 and a configuration file for a single-Sérsic fit to this image. The basic fit (without PSF convolution) takes approximately 115 model-image computations and about 0.3 seconds to finish[23] using the default Levenberg-Marquardt minimizer. Using DE, the fit takes ~11,000 model-image computations and about 20 seconds. An MCMC analysis of the same model requires *several hundred thousand* model-image computations and approximately ten minutes to reach convergence.

The MCMC approach that `imfit-mcmc` uses is the DREAM (DiffeRential Evolution Adaptive Metropolis) algorithm of Vrugt et al. [2009], which is based on the earlier adaptation of DE to MCMC by ter Braak [2006]; more details can be found in Vrugt [2016]. The specific implementation used by `imfit-mcmc` is based on C++ code published by Gabriel Rosenthal.[24] The algorithm uses multiple separate Markov chains (by default, a number equal to the number of free parameters in the model) and a DE-based scheme for generating new proposals for each chain using differences between the current states of other (randomly chosen) chains. A version of the Gelman-Rubin convergence diagnostic [Gelman & Rubin, 1992] is used to estimate when convergence is reached, by examining the most recent 50% of each chain; the algorithm terminates after a user-specified maximum number of generations if convergence is not achieved before then.

In simplified terms, creating a new proposal for a chain involves scaled offsets from the current state of the chain (i.e., the set of parameter values x_i, with $i = 1, .., N_{\mathrm{params}}$), with the base offsets Δ_i taken from the difference in parameter values between the current states of two (or more) *other* chains. A random subset of the parameter values in the current state are updated, with the

[22] E.g., using mean, median, or mode from the marginal distribution of each parameter.
[23] On a MacBook Pro 2012 with a 2.7 GHz Intel i7 quad-core CPU.
[24] `https://github.com/gaberoo/cdream`

remaining parameters left unchanged.[25] For each parameter that is updated, the new value z_i is

$$z_i = x_i + (1 + U)\,\gamma\,\Delta_i + N(0, s), \tag{18}$$

where U is a uniformly sampled random number between $\pm u$ and $N(0, s)$ is a random sample from a Gaussian with mean of 0 and dispersion s. Both u and s can be specified by the user, though their default values (0.01 and 10^{-6}, respectively) are probably good starting points. The scale parameter γ is

$$\gamma = 2.38/\sqrt{2\delta d'}, \tag{19}$$

where δ is the number of pairs used to determine the base offset (δ = 1, 2, or 3; the actual value used is randomly permuted by the algorithm) and d' is the number of parameters being updated for the current proposal. Every five generations γ is set = 1, to encourage occasional longer jumps outside the current location in parameter space.

The stationary posterior-probability distribution (after convergence) is proportional to the likelihood of the data given the model, multiplied by an assumed prior probability which is constant between the parameter limits specified in the configuration file and zero outside those limits.

The usual caveats about "convergence" of MCMC chains apply: the convergence test is not guaranteed to identify true convergence, and it is always possible that if run long enough, the composite chain will discover new modes in the posterior distribution. However, since most uses of MCMC for galaxy image fitting will probably be for purposes of identifying the distribution of parameter values in the vicinity of the overall "best-fit" state, it's probably not worth worrying too much about outlying low-probability alternate modes.

13.1 Using `imfit-mcmc`

Use of `imfit-mcmc` on the command line is almost identical to use of `imfit` (Chapter 4). The main difference is the optional use of `-o`/`--output` to specify the root name for the Markov-chain output files (the program defaults to using `mcmc_out` as the root name if `-o` is not used), and some extra options for tweaking the MCMC algorithm.

The following `imfit` options are *not* available with `imfit-mcmc`:

- Minimizer selection and control: `--nm`, `--nlopt`, `--de`, `--ftol`
- "Best-fit" output: `--save-params`, `--save-model`, `--save-residual`, `--save-weights`
- Bootstrap-related: `--boostrap`, `--save-bootstrap`
- `--chisquare-only`, `--fitstat-only`

Extra options for `imfit-mcmc`

- `-o, --output` *root-name* – the root name for output MCMC chain files. The actual output files will be named <*root-name*>`.1.txt`, <*root-name*>`.2.txt`, etc. The default is "mcmc_out".
- `--nchains` N_{chains} – specifies the number of MCMC chains to calculate (must be at least 3). This defaults to the number of free parameters in the model, which is the recommended value.

[25] The proposal can thus be seen as a "crossover" between a parameter vector with completely new values and the current parameter vector; this is part of the general DE algorithm.

- `--append` – specifies that pre-existing MCMC chain files should be read in and the MCMC process continued from their final states.
- `--max-chain-length` N – the maximum number of generations (with one likelihood evaluation per generation) per chain. The program will quit if if reaches this value. The default value is 100,000 (which means the total number of likelihood evaluations – i.e., model-image computations – will be 100,000 $\times N_{\rm chains}$).
- `--burnin-length` N – the number of generations of the initial "burn-in" phase, where larger jumps are taken to ensure adequate exploration of the posterior landscape. If outlier chains are subsequently discovered, burn-in will be re-entered for the same number of generations. Default value: 5,000.
- `--gelman-evals` N – Gelman-Rubin convergence tests will be done every N generations (after the burn-in phase has finished). Default value: 5,000.
- `--uniform-offset` u – the limit on variations in the multiplicative scaling applied to proposed parameter offsets, such that offsets are multiplied by uniformly sampled numbers in the interval $[1-u, 1+u]$. Default value: 0.01.
- `--gaussian-offset` s – Gaussian σ value for additive variations applied to proposed parameter offsets, such that $N(0, s)$ is added to each offset (i.e., a Gaussian with mean = 0 and dispersion = s). Default value: 10^{-6}.

13.2 Configuration Files for `imfit-mcmc`

Configuration files for `imfit-mcmc` are identical to those for **Imfit**, with the restriction that *parameter limits must be supplied for all non-fixed function parameters.* (This is similar to how **Imfit** works with the Differential Evolution miminizer.)

TBD.

13.3 Analysis of `imfit-mcmc` output

The result of running `imfit-mcmc` is a set of $N_{\rm chains}$ output text files, each of which contains one of the Markov chains. The files contain one column for each of the model parameters (plus several additional columns with diagnostics of the MCMC process: the likelihood value for a given set of parameters, whether the chain was in the burn-in phase, etc.), and one row for each generation in the chain.

Each individual file can be inspected to see how the MCMC process evolved, to check for lack of convergence, etc. To get a proper evaluation of the estimated posterior/likelihood landscape, all the chains should be combined, excluding the burn-in phase(s). Since the Gelman-Rubin convergence test checks the last half of each chain, you could plausibly use the last half of all the output chains, assuming that `imfit-mcmc` terminated because convergence was detected. For some analyses, this may be overkill, and using the last few thousand generations from each chain may suffice.

Some simple Python code for reading in the MCMC output is included in the `python/` subdirectory of the distribution, in the file `imfit.py`. (This file in turn depends on the file `imfit_funcs.py` and the Scipy package, but the MCMC-related functions themselves do not, and can be extracted into a separate module if that makes things easier.) There are two functions of interest:

- `GetSingleChain`, which reads in a single output file;
- `MergeChains`, which reads in *all* the output files and concatenates them into a single composite chain, with user-specified limits (e.g., merge only generations *n* or later, or merge only the last *n* generations).

Some basic examples follow.

Assuming that `imfit-mcmc` has run and produced converged output files named mcmc_out.1.txt, mcmc_out.2.txt, etc., the following Python code will read in the first of these files (note that in version 1.4, the output chains were numbered starting with 0 instead of with 1):

```
>>> import imfit
>>> columnNames, chain1 = imfit.GetSingleChain("mcmc_out.1.txt")
```

The result is a list of the parameter names (columnNames) and a Numpy array (chain1) with shape = $(n_{\rm rows}, n_{\rm cols})$, where each row is one generation from the chain. The parameter names map to the columns in the Numpy array.

The following bit of Python code reads in the last 5,000 generations from each chain and merges them together, returning a list and a Numpy array with the same form as for `GetSingleChain`. It then uses the `corner` package[26] [Foreman-Mackey, 2016] to make a scatterplot matrix (a.k.a. "corner plot") of the different parameter distributions:

```
>>> import imfit, corner
>>> columnNames, allchains = imfit.MergeChains("mcmc_out", last=5000)
>>> corner.corner(allchains, labels=columnNames)
```

[26] `https://corner.readthedocs.io/en/latest/`

14 Rolling Your Own Functions

14.1 Basic Requirements

A new image function should be implemented in C++ as a subclass of the FunctionObject base class (`function_object.h`, `function_object.cpp`). At a minimum, it should provide its own implementation of the following public 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 (x_0, y_0) pixel values for the center) prior to determining intensity values for individual pixels. This is a convenient place to do any general calculations which 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 classes, 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;
- `PARAM_UNITS` – **[optional]** a vector of string labels describing units (if any) 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, as it would be used in a configuration file.

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

1. including the header file for the new class;
2. adding 2 lines to the PopulateFactoryMap() function to add the ability to create an instance of the new class.

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 `functionobject_obj_string` multi-line string definition.

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.

14.2 A Simple Example

To illustrate, we'll make a new version of the Moffat function (which already exists, so this is purely for pedagogical purposes) by copying and modifying the code for the Gaussian function.

We need to make three sets of changes:

- Change the class name from "Gaussian" to our new name ("NewMoffat");
- Change the relevant code which computes the function;
- Rename, add, or delete variables to accommodate the new algorithm.

Create and Edit the Header File

Change directory to the directory with the **Imfit** source code, and then to the subdirectory named "`function_objects`". Copy the file `func_gaussian.h` and rename it to `func_new-moffat.h`. Edit this file and change the following lines:

```
#define CLASS_SHORT_NAME  "Gaussian"
```

(replace `"Gaussian"` with `"NewMoffat"`)

```
class Gaussian : public FunctionObject
```

(replace `Gaussian` with `NewMoffat`)

```
    Gaussian( );
```

(replace `Gaussian` with `NewMoffat`)

And, finally, edit the list of class data members, changing this:

```
  private:
    double  x0, y0, PA, ell, I_0, sigma;   // parameters
    double  q, PA_rad, cosPA, sinPA;   // other useful (shape-related) quantities
```

to this:

```
  private:
    double  x0, y0, PA, ell, I_0, fwhm, beta;   // parameters
    double  alpha;
    double  q, PA_rad, cosPA, sinPA;   // other useful (shape-related) quantities
```

Create and Edit the Class File

Copy the file `func_gaussian.cpp` and rename it to `func_new-moffat.cpp`.

Initial changes, including parameter number and names:

Edit this file and change the following lines (changed text indicated in red):

```
#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";
```

Change references to class name:

Change all class references from "Gaussian" to "NewMoffat" (e.g., `Gaussian::Setup` becomes `NewMoffat::Setup`).

Changes to Setup method:

In the Setup method, you need to change how the input is converted into parameters, and do any useful pre-computations. So the initial processing of the "params" input changes from this:

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

to this:

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

and at the end we replace this:

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

with this:

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

Changes to 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);
}
```

Changes to 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 `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.

Edit add_functions.cpp

We need to do two simple things here:

1. Include the header file for our new function. Add the following line near the top of the file, where the other header files are included:
   `#include "func_new-moffat.h"`

2. Add code to generate an instance of our new class as part of the function-factory map. Inside the function PopulateFactoryMap, add the following lines:

   ```
   NewMoffat::GetClassShortName(classFuncName);
   input_factory_map[classFuncName] = new funcobj_factory<NewMoffat>();
   ```

Edit the SConstruct File

In the SConstruct file, locate the place where the variable "functionobject_obj_string" is defined (e.g., search for the string "functionobject_obj_string ="). This variable is bound to a string containing a compact list of all the filenames containing function-object code. Insert our new function's name ("func_new-moffat") into the list.

That's it! You should now be able to recompile **Imfit** and `makeimage` (see Section 2.3) to use the new function. (Assuming there aren't any bugs in your new code....)

14.3 Further Notes on Writing Your Own Functions

TBD.

Some potentially useful auxiliary functions can be found in the file `helper_funcs.h` in the "`function_objects`" subdirectory, such as a function which calculates the b_n parameter for a Sérsic function, and a function which calculates the effective radius for a generalized ellipse.

New in Version 1.9: Parameter Units

Version 1.9 of **Imfit** added the (optional!) ability to print parameter units when outputting best-fit parameters. This is purely cosmetic (it has no effect on image computation or fitting) and is meant to be helpful to users who might be uncertain as to the meaning of certain parameters of an image function.

To add parameter units to your function all you really need to do is define the units in the constructor, and set the `parameterUnitsExist` data member to `true`. Note that all of the standard image functions included in **Imfit** all have (as of v1.9) parameter unit definitions, so you can look at the constructors for any of those classes for examples of how to do this.

Units are defined as short strings; if a parameter does *not* have a unit (e.g., ellipticities or axis ratios, indices, exponents, etc.), then use an empty string (`""`).

New in Version 1.9: Optional Meta-Parameters

Version 1.9 of **Imfit** introduced the optional ability to specify that an image function takes optional "meta-parameters" – parameters that define certain specializations of a function, but which do *not* change during a fit. An example is the `PointSource` function, which has an optional meta-parameter (`mode`) specifying the interpolation algorithm to use (bicubic spline (the default) versus Lanczos2).

TBD.

15 Acknowledging Use of Imfit

A paper describing **Imfit** [Erwin, 2015] has been published in *The Astrophysical Journal* (`https://ui.adsabs.harvard.edu/abs/2015ApJ...799..226E/abstract`) and can be found on the arXiv at `https://arxiv.org/abs/1408.1097`; you can also reference the current URL (`https://www.mpe.mpg.de/~erwin/code/imfit/`) and/or the Github URL (`https://github.com/perwin/imfit/`) if **Imfit** has been useful in your research.

A Standard Functions in Detail

Unless otherwise noted, all "intensity" parameters (`I_sky`, `I_0`, `I_e`, etc.) are in units of counts per pixel, and all lengths (radii, semi-major axes, etc.) are in pixels. In addition, unless otherwise noted, position angles ("PA") are in degrees, measured CCW from the +y axis of the image (negative values are valid).

A sample function specification (giving the parameters in their proper order), as you would use in a configuration file, is listed for each function description.

"Elliptical" functions are defined to have an intensity which is constant on concentric, similar ellipses (with specified ellipticity and major-axis position angle); the intensity profile is defined as a function of the semi-major axis a.

A.1 2D Functions

The main set of image functions provided create 2D intensity distributions directly. These include most of the usual suspects used in 2D image fitting: constant background, Gaussian, exponential, Sérsic, etc.

Common parameters:

- `PA` = position angle (e.g., of the major axis), measured in degrees CCW from the image +y axis. This is equivalent to standard astronomical position angles *if* your image has standard astronomical orientation (N up, E to the left).
- `ell` = ellipticity ($1 - b/a$, where a and b are semi-major and semi-minor axes of the ellipse, respectively).

FlatSky

A uniform background: $I(x, y) = I_{\rm sky}$ everywhere.

```
FUNCTION FlatSky
I_sky     # counts/pixel
```

Note that this is considered a "background" function, and so no "total flux" calculation is done when using `makeimage`'s `--print-fluxes` or `savefluxes` options.

TiltedSkyPlane

A background in the form of an inclined plane: $I(x, y) = I_0$ at the center (X0,Y0), with x and y slopes given by m_x and m_y.

$$I(x, y) = I_0 + m_x(x - X_0) + m_y(y - Y_0). \tag{20}$$

```
FUNCTION TiltedSkyPlane
I_0     # counts/pixel
m_x
m_y
```

Note that this is considered a "background" function, and so no "total flux" calculation is done when using `makeimage`'s `--print-fluxes` or `savefluxes` options.

Gaussian

This is an elliptical 2D Gaussian function, with the major-axis intensity profile given by

$$I(a) = I_0 \exp(-a^2/(2\sigma^2)). \tag{21}$$

```
FUNCTION Gaussian
PA
ell
I_0     # counts/pixel
sigma
```

Moffat

This is an elliptical 2D Moffat [1969] function, with the major-axis intensity profile given by

$$I(a) = \frac{I_0}{(1+(a/\alpha)^2)^\beta}, \tag{22}$$

where α is defined as

$$\alpha = \frac{\mathrm{FWHM}}{2\sqrt{2^{1/\beta}-1}}. \tag{23}$$

In practice, FWHM describes the overall width of the profile, while β describes that strength of the wings: lower values of β mean more intensity in the wings than is the case for a Gaussian (as $\beta \to \infty$, the Moffat profile approaches a Gaussian).

The Moffat function is often a good approximation to typical telescope PSFs (see, e.g., Trujillo et al. 2001), and `makeimage` can easily be used to generate Moffat PSF images.

```
FUNCTION Moffat
PA
ell
I_0     # counts/pixel
fwhm
beta
```

PointSource

This function approximates a point source (e.g., star, AGN, etc.) by taking the user-input PSF image and generating an interpolated, intensity-scaled copy of it at the X0,Y0 position of the parent function set. It has only one free parameter: I_{tot}, which is the factor that the PSF image is multiplied by, and which will usually correspond to the total flux of the point source. Of course, it also *requires* that a PSF image be supplied!

For standard model-image generation, the main user-supplied PSF image is used. If PSF oversampling is specified, then any PointSource objects which fall within an oversampling region (specified via `--overpsf_region`) will interpolate the corresponding *oversampled* PSF image (specified via `--overpsf`) instead.

The interpolation is by default done using the 2D bicubic function (`gsl_interp2d_bicubic`) from the GNU Scientific Library.

```
FUNCTION PointSource
I_tot     # counts
```

An alternate interpolation method is Lanczos2, which can be specified by using the *optional parameter* specification, like so:

```
FUNCTION PointSource
I_tot     # counts
OPTIONAL_PARAMS_START
method          lanczos2
OPTIONAL_PARAMS_END
```

If you want to explicitly specify the (default) bicubic interpolation, you can replace "lanczos2" with "bicubic".

PointSourceRot

This function is basically identical to PointSource, except that it allows for arbitrary rotation of the user-input PSF image about its center (the center being the central pixel of the image, so this is best used with PSF images that are square and an odd number of pixels in size).

```
FUNCTION PointSourceRot
I_tot     # counts
PA
```

As with PointSource, you can optionally specify the interpolation method (the default bicubic or the alternative Lanczos2); see the description of PointSource above for what this looks like.

ModifiedKing

This is an elliptical 2D function with the intensity profile given by a "modified King" function [Elson, 1999, Peng et al., 2010], which is a generalization of the original King profile [King, 1962]:

$$I(a) = I_0 \left[1 - \frac{1}{(1+(r_t/r_c)^2)^{1/\alpha}}\right]^{-\alpha} \times \left[\frac{1}{(1+(r/r_c)^2)^{1/\alpha}} - \frac{1}{(1+(r_t/r_c)^2)^{1/\alpha}}\right]^{\alpha}, \qquad (24)$$

where I_0 is the central intensiy, r_c is the "core" radius, and r_t is the "tidal" or "truncation" radius (outside of which the intensity is 0). This reduces to the original King profile when $\alpha = 2$.

```
FUNCTION ModifiedKing
PA
ell
I_0     # counts/pixel
r_c
r_t
alpha
```

ModifiedKing2

This is an alternate interface for the ModifiedKing profile. It produces exactly the same surface-brightness model, but uses the "concentration" $c = r_t/r_c$ as a free parameter (in place of the tidal/truncation radius r_t, which can be recovered as $r_t = cr_c$).

```
FUNCTION ModifiedKing2
PA
ell
I_0     # counts/pixel
r_c
c
alpha
```

Exponential

This is an elliptical 2D exponential function, with the major-axis intensity profile given by

$$I(a) = I_0 \exp(-a/h), \tag{25}$$

where I_0 is the central surface brightness and h is the scale length.

```
FUNCTION Exponential
PA
ell
I_0     # counts/pixel
h
```

Exponential_GenEllipse

Similar to the Exponential function, but using generalized ellipses ("boxy" to "disky" shapes) instead of pure ellipses for the isophote shapes. Following Athanassoula et al. [1990], the shape of the elliptical isophotes is controlled by the `c0` parameter, such that a generalized ellipse with ellipticity $= 1 - b/a$ is described by

$$\left(\frac{|x|}{a}\right)^{c_0+2} + \left(\frac{|y|}{b}\right)^{c_0+2} = 1, \tag{26}$$

where $|x|$ and $|y|$ are distances from the ellipse center in the coordinate system aligned with the ellipse major axis (c_0 corresponds to $c - 2$ in the original formulation of Athanassoula et al). Thus, values of $c_0 < 0$ correspond to disky isophotes, while values > 0 describe boxy isophotes; $c_0 = 0$ corresponds to a perfect ellipse.

```
FUNCTION Exponential_GenEllipse
PA
ell
c0
I_0     # counts/pixel
h
```

Sersic

This is an elliptical 2D Sérsic function with the major-axis intensity profile given by

$$I(a) = I_e \exp\left\{-b_n\left[\left(\frac{a}{r_e}\right)^{1/n} - 1\right]\right\}, \tag{27}$$

where I_e is the surface brightness at the effective (half-light) radius r_e and n is the Sérsic index controlling the shape of the intensity profile. The value of b_n is formally given by the solution to the transcendental equation

$$\Gamma(2n) = 2\gamma(2n, b_n), \tag{28}$$

where $\Gamma(a)$ is the gamma function and $\gamma(a, x)$ is the incomplete gamma function. However, in the current implementation b_n is calculated via the polynomial approximation of Ciotti & Bertin [1999] when $n > 0.36$ and the approximation of MacArthur, Courteau, & Holtzman [2003] when $n \leq 0.36$.

Note that the Sérsic function is equivalent to the de Vaucouleurs "$r^{1/4}$" profile when $n = 4$, to an exponential when $n = 1$, and to a Gaussian when $n = 0.5$.

```
FUNCTION Sersic
PA
ell
n
I_e     # counts/pixel
r_e
```

Sersic_GenEllipse

Similar to the Sersic function, but using generalized ellipses ("boxy" to "disky" shapes) instead of pure ellipses for the isophote shapes. See the discussion of the `Exponential_GenEllipse` function above for details of the isophote shapes.

```
FUNCTION Sersic_GenEllipse
PA
ell
c0
n
I_e     # counts/pixel
r_e
```

Core-Sersic

This generates an elliptical 2D function with the major-axis intensity profile given by the Core-Sérsic model [Graham et al., 2003, Trujillo et al., 2004]. This has a Sérsic profile (parameterized by n and r_e) for radii > the break radius r_b and a single power law with index $-\gamma$ for radii < r_b. The transition between the two regimes is mediated by the parameter α: for low values of α, the transition is very gradual and smooth, while for high values of α the transition becomes very abrupt (a perfectly sharp transition can be approximated by setting α = some large number, such as 100). The overall intensity scaling is set by I_b, the intensity at the break radius r_b.

```
FUNCTION Core-Sersic
PA
ell
n
I_b     # counts/pixel
r_e
r_b
alpha
gamma
```

BrokenExponential

Similar to Exponential, but with *two* exponential radial zones (with different scalelengths) joined by a transition region at R_b of variable sharpness:

$$I(a) \;=\; S\, I_0\, e^{-\frac{a}{h_1}} \left[1 + e^{\alpha(a - R_b)}\right]^{\frac{1}{\alpha}\left(\frac{1}{h_1} - \frac{1}{h_2}\right)}, \tag{29}$$

where I_0 is the central intensity of the inner exponential, h_1 and h_2 are the inner and outer exponential scale lengths, R_b is the break radius, and α parameterizes the sharpness of the break. (See Erwin, Pohlen, & Beckman [2008].) Low values of α mean very smooth, gradual breaks, while high values correspond to abrupt transitions. S is a scaling factor, given by

$$S \;=\; \left(1 + e^{-\alpha R_b}\right)^{-\frac{1}{\alpha}\left(\frac{1}{h_1} - \frac{1}{h_2}\right)}. \tag{30}$$

Note that the parameter α has units of length^{-1} (i.e., pixels^{-1}).

```
FUNCTION BrokenExponential
PA
ell
I_0     # counts/pixel
h1
h2
r_break
alpha
```

GaussianRing

An elliptical ring with a radial profile consisting of a Gaussian centered at $r = R_{\rm ring}$.

```
FUNCTION GaussianRing
PA
ell
A     # counts/pixel
R_ring
sigma_r
```

GaussianRing2Side

Similar to GaussianRing, but now using an asymmetric Gaussian (different values of σ for $r < R_{\rm ring}$ and $r > R_{\rm ring}$).

```
FUNCTION GaussianRing2Side
PA     # deg (CCW from +y axis)
ell
A     # counts/pixel
R_ring     # pixels
sigma_r_in     # pixels
sigma_r_out     # pixels
```

GaussianRingAz

Similar to GaussianRing, except that the peak radial intensity is modulated by a cosine function, varying from $A_{\rm maj}$ along the ellipse major axis to $A_{\rm min} = A_{\rm min\_rel} A_{\rm maj}$. $A_{\rm maj}$ is a standard intensity variable (counts/pixel); $A_{\rm min\_rel}$ defines the minor-axis intensity relative to $A_{\rm maj}$; $A_{\rm min\_rel} = 1$ will produce the same result as a standard GaussianRing component.

```
FUNCTION GaussianRingAz
PA
ell
A_maj     # counts/pixel
A_min_rel
R_ring
sigma_r
```

EdgeOnDisk

This function provides the analytical form for a perfectly edge-on disk with a radial exponential profile, using the Bessel-function solution of van der Kruit & Searle [1981] for the radial profile and the generalized sech function of van der Kruit [1988] for the vertical profile.[27] The position angle parameter (PA) describes the angle of the disk major axis; there is no ellipticity parameter.

In a coordinate system aligned with the edge-on disk, r is the distance from the minor axis (parallel to the major axis) and z is the perpendicular direction, with $z = 0$ on the major axis. (The latter corresponds to height z from the galaxy midplane.) The intensity at (r, z) is given by

$$I(r,z) \;=\; \mu(0,0)\;(r/h)\;K_1(r/h)\;\mathrm{sech}^{2/n}(n\,z/(2\,z_0)) \tag{31}$$

where h is the exponential scale length in the disk plane, z_0 is the vertical scale height, and K_1 is the modified Bessel function. The central surface brightness $\mu(0,0)$ is given by

$$\mu(0,0) \;=\; 2\,h\,L_0, \tag{32}$$

where L_0 is the central luminosity *density*, referred to as "counts/voxel" in the units string (see van der Kruit & Searle 1981). Note that L_0 is the actual parameter required by the function; $\mu(0,0)$ (which has units of counts/pixel) is calculated internally.

[27] This model was used by Yoachim & Dalcanton [2006] for 2D modeling of thin and thick disks in edge-on galaxies, though typically with n fixed to values of 1 or 2.

When $n = 1$, this becomes the familiar sech^2 model for the vertical distribution of a disk (with z_0 corresponding to 1/2 of the z_0 in the original definition of van der Kruit & Searle [1981]). As $n \to \infty$, the vertical distribution approaches an exponential with $\exp(-z/z_0)$. In practice, the code substitutes a pure exponential function for the $\mathrm{sech}^{2/n}$ term whenever

$$\frac{n}{2}\frac{z}{z_0} > 100. \tag{33}$$

```
FUNCTION EdgeOnDisk
PA
L_0     # counts/voxel
h
n
z_0
```

EdgeOnRing

A simplistic model for an edge-on ring, using two offset components located at distance ±`r` from the center of the function set. Each component (i.e., each side of the ring) is a symmetric Gaussian with size `sigma_r` for the radial profile and a symmetric Gaussian with size `sigma_z` for the vertical profile. (See GaussianRing3D for a similar component which does line-of-sight integration through a 3D luminosity-density model of a ring.)

```
FUNCTION EdgeOnRing
PA
I_0     # counts/pixel
r
sigma_r
sigma_z
```

EdgeOnRing2Side

Similar to EdgeOnRing, but now the radial profile for the two components is asymmetric: the inner ($|R| < R_{\mathrm{ring}}$) side of each component is a Gaussian with radial size `sigma_r_in`, while the outer side has radial size `sigma_r_out`.

```
FUNCTION EdgeOnRing2Side
PA
I_0     # counts/pixel
r
sigma_r_in
sigma_r_out
sigma_z
```

FerrersBar2D

This produces a 2D analog of the classic Ferrers [1877] ellipsoid (see FerrersBar3D in Section A.2 for the original 3D version), where the intensity is constant on (generalized) ellipses, with the intensity going to zero outside a specified semi-major axis (= a_{bar} along the major axis).

The surface-brightness function is

$$I(m) = \begin{cases} I_0\,(1-m^2)^n & \text{if } m < 1 \\ 0 & \text{otherwise} \end{cases} \tag{34}$$

where m is the elliptical radius, defined (assuming ellipticity = $1 - b/a$) by:

$$m^2 = \left(\frac{|x|}{a}\right)^{c_0+2} + \left(\frac{|y|}{b}\right)^{c_0+2}. \tag{35}$$

This is inspired by functions used in 2D decompositions by Laurikainen, Salo, & Buta [2005] and Aguerri, Méndez-Abreu, & Corsini [2009]. It is similar to the "Ferrer" component in GALFIT, except that it lacks the latter's "central slope" parameter β.

```
FUNCTION FerrersBar2D
PA
ell
c0
I_0     # counts/pixel
n
a_bar
```

FlatBar

TBD.

see Erwin et al. [2021], especially Appendix A.

```
FUNCTION FlatBar
PA
ell
deltaPA_max
I_0     # counts/pixel
h1
h2
r_break
alpha
```

A.2 3D Functions

The following are image functions which use line-of-sight integration through a 3D luminosity-density model to create a projected 2D image.

The functions are defined so as to have a primary plane (e.g., the equatorial plane for a disk galaxy); the orientation of this plane is defined by the `PA` and `inc` parameters, which specify the angle of the line of nodes (in degrees CCW with respect to the image +y axis) and the inclination to the line of sight (also in degrees), respectively. Thus, `PA` = 0 will align the line of nodes vertically, while `PA` = 90 will make it horizontal (parallel to the image x-axis).[28] The inclination is defined

[28] The goal is to ensure that the orientation of the component's line of nodes follows the same conventions as for the 2D functions, so that an inclined ExponentialDisk3D function with PA = 30 will have the same orientation as an elliptical 2D Exponential function with PA = 30.

in the usual astronomical sense: $i = 0$ for a face-on system and $i = 90$ for an edge-on system. See Section 6.2 of Erwin [2015] for details of how the line-of-sight integration is handled.

For the GaussianRing3D and FerrersBar3D functions, which are not axisymmetric, there is an additional "position angle" parameter (`PA_ring` or `PA_bar`), which defines the position of the ring's or bar's major axis *in the primary plane* (i.e., prior to any projection) with respect to the primary plane's +x axis. (The logic behind this is that when the primary plane's line of nodes is *horizontal* – i.e., PA = 90 – the orientation of the ring's major axis follows the usual orientation conventions with respect to the image +y axis. You are, of course, free to change this if you write 3D components of your own, though I will probably continue to follow it in the future.)

A note on units: since these are luminosity-density functions, which yield standard surface-brightness or intensity units (usually counts/pixel) by integrating along the line of sight, the brightness is controlled by a luminosity-density parameter (J_0) which has units of "counts/voxel". This slightly odd name indicates that the line-of-sight integration (integrating using units of pixels) will produce counts/pixel in the final image. (Technically, we could speak of a linear 1D unit of "pixel-size", with the 2D image pixels having area units of pixel-size^2 – but that's being way too pedantic!)

ExponentialDisk3D

This function implements a 3D luminosity density model for an axisymmetric disk with an exponential radial profile and a sech$^{2/n}$ vertical profile (as for the EdgeOnDisk function), using line-of-sight integration to create the projected surface-brightness profile for arbitrary inclinations.

In a cylindrical coordinate system (r, z) aligned with the disk (where the disk midplane has $z = 0$), the luminosity density at radius r from the central axis and at height z from the midplane is given by

$$j(r, z) \;=\; J_0 \;\exp(-r/h)\;\mathrm{sech}^{2/n}(n\,z/(2\,z_0)) \tag{36}$$

where h is the exponential scale length in the disk plane, z_0 is the vertical scale height, and J_0 is the central luminosity density.

When $n = 1$, the vertical distribution is the familiar sech^2 model (with z_0 corresponding to 1/2 of the z_0 in the original definition of van der Kruit & Searle [1981]). As $n \to \infty$, the vertical distribution approaches an exponential with $\exp(-z/z_0)$; in practice, the can be approximated by setting n equal to some fixed, large number.

```
FUNCTION ExponentialDisk3D
PA
inc
J_0     # counts/voxel
h
n
z_0
```

Because this function performs numerical integration for each pixel value, it will be slower than the analytic EdgeOnDisk function (though the latter is correct only in the $i = 90°$ case), and even slower than the standard Exponential function.

BrokenExponentialDisk3D

This function is identical to the ExponentialDisk3D function, except that the radial profile of the luminosity density follows a broken-exponential function (e.g., Section A.1) instead of a simple exponential. Consequently, it has the following parameters:

```
FUNCTION BrokenExponentialDisk3D
PA
inc
J_0     # counts/voxel
h1
h2
r_break
alpha
n
z_0
```

GaussianRing3D

This function does line-of-sight integration through an elliptical ring. The ring is defined as having luminosity density with a radial Gaussian profile (centered at `a_ring` along ring's major axis, with in-plane width σ) and a vertical exponential profile (with scale height `h_z`). The ring can be envisioned as residing in an (invisible) plane which has a line of nodes at angle `PA` and inclination `inc` (as for the ExponentialDisk3D function, above); within this plane, the ring's major axis is at position angle `PA_ring` *relative to the perpendicular to the line of nodes*, and the ring has an ellipticity given by `ell`.

```
FUNCTION GaussianRing3D
PA
inc
PA_ring
ell
J_0     # counts/voxel
a_ring
sigma
h_z
```

FerrersBar3D

This function does line-of-sight integration through a Ferrers [1877] ellipsoid, where the luminosity density is constant on concentric, stratified ellipsoidal shells, with the density going to zero outside a specified ellipsoidal radius (= $R_{\rm bar}$ along the major axis). The potential corresponding to the mass-density version of this function has commonly been used to represent galactic bars in theoretical studies of orbits in barred galaxies.

As with the GaussianRing3D component, the ellipsoid can be imagined as lying in an (invisible) "equatorial" plane – defined by the first two axes of the ellipsoid – with a line of nodes at angle `PA` and inclination `inc` (as for the ExponentialDisk3D function, above); within this plane, the ellipsoid's major axis is at position angle `barPA` *relative to the perpendicular to the line of nodes*

(i.e., at 90+barPA degrees relative to the line of nodes). The shape of the ellipsoid is defined by the axis ratios $q = b/a$ and $q_z = c/a$, where a (= $R_{\rm bar}$) and b are the semi-major and semi-minor axes in the equatorial plane and c is in the direction perpendicular to the equatorial plane. For simplicity, it is assumed that $q \leq 1$ and $q_z \leq 1$; however, it is not required that q_z must be $\leq q$.

The luminosity density function is

$$j(m) = \begin{cases} J_0 \, (1 - m^2)^n & \text{if } m < 1 \\ 0 & \text{otherwise} \end{cases} \tag{37}$$

where m is the ellipsoidal radius:

$$m^2 = \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2}, \tag{38}$$

with $a = R_{\rm bar}$, $b = qR_{\rm bar}$, and $c = q_z R_{\rm bar}$. Typical values of n used in theoretical studies are 1 and 2.

Note that this is related to but different from the "Ferrer" component in GALFIT, which defines a 2D surface-brightness function using Eqn. 37.

```
FUNCTION FerrersBar3D
PA
inc
barPA
J_0     # counts/voxel
R_bar
q
q_z
n
```

B Acknowledgments

Major inspirations for **Imfit** include both GALFIT [Peng et al., 2002, 2010] and BUDDA [de Souza, Gadotti, & dos Anjos, 2004, Gadotti, 2008].

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

B.1 Data Sources

Sample FITS images for demonstration and testing use are taken from Data Release 7 [Abazajian et al., 2009] of the Sloan Digital Sky Survey [York et al., 2000]. Funding for the creation and distribution of the Sloan Digital Sky Survey Archive has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Aeronautics and Space Administration,

the National Science Foundation, the U.S. Department of Energy, the Japanese Monbukagakusho, and the Max Planck Society. The SDSS Web site is `http://www.sdss.org/`.

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

B.2 Specific Software Acknowledgments

Minpack

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

B.3 Miscellaneous Useful Software

The development of **Imfit** has benefitted from the C++ unit-testing framework CxxTest,[29] and also from the static analyzers Cppcheck[30] and the Clang Static Analyzer.[31]

Bibliography

Abazajian, K. N.., et al. 2009, "The Seventh Data Release of the Sloan Digital Sky Survey", *Astrophys.J. Supplement* **182**: 182.

Aguerri, J. A. L., Méndez-Abreu, J., and Corsini, E. M. 2009, "The population of barred galaxies in the local universe. I. Detection and characterisation of bars", *Astron. Astrophys.* **495**: 491.

Athanassoula, E., Morin, S., Wozniak, H., Puy, D., Pierce, M. J., Lombard, J., and Bosma, A. 1990, *Monthly Notices of the Royal Astronomical Society* **245**: 130.

Ciotti, L., and Bertin, G. 1999, "Analytical properties of the $R^{1/m}$ law", *Astron. Astrophys.* **352**: 447.

de Souza, R. E., Gadotti, D. A., and dos Anjos, S. 2004, "BUDDA: A New Two-dimensional Bulge/Disk Decomposition Code for Detailed Structural Analysis of Galaxies", *Astrophys.J. Supplement* **153**: 411.

Elmegreen, D. M., and Elmegreen, B. G. 1985, "Properties of barred spiral galaxies", *Astrophys.J.* **288**: 438.

Elson, R. A. W. 1999, "Stellar dynamics in globular clusters", in *Globular clusters: X Canary Islands Winter School of Astrophysics*, C. Martínez Roger, I. Perez Fournón, and F. Sánchez, eds., (Cambridge: Cambridge U. Press), 209.

[29] `https://cxxtest.com`
[30] `http://cppcheck.sourceforge.net`
[31] `https://clang-analyzer.llvm.org`

Erwin, P., Pohlen, B., and Beckman, J. E. 2008, "The Outer Disks of Early-Type Galaxies. I. Surface-Brightness Profiles of Barred Galaxies", *Astron.J.* **135**: 20.

Erwin, P. 2015, "Imfit: A Fast, Flexible New Program for Astronomical Image Fitting", *Astrophys.J.* **799:** 226.

Erwin, P., et al. 2021, "Composite Bulges – II. Classical Bulges and Nuclear Disks in Barred Galaxies: The Contrasting Cases of NGC 4608 and NGC 4643", *Monthly Notices of the Royal Astronomical Society*, **502:** 2446.

Ferrers, N. M. 1877, "On the Potentials of Ellipsoids, Ellipsoidal Shells, Elliptic Laminae, and Elliptic Rings, of Variable Densities", *Quarterly Journal of Pure and Applied Mathematics* **14**: 1.

Foreman-Mackey, D. 2016, "corner.py: Scatterplot matrices in Python", *The Journal of Open Source Software* **1:** 24.

Gadotti, D. A. 2008, "Image decomposition of barred galaxies and AGN hosts", *Monthly Notices of the Royal Astronomical Society* **384**: 420.

Gelman, A., and Rubin, D. B. 1992, "Inference from iterative simulation using multiple sequences", *Statistical Science* **7**: 457.

Graham, A., Erwin, P., Trujillo, I., and Asensio Ramos, A. 2003, "A New Empirical Model for the Structural Analysis of Early-Type Galaxies, and A Critical Review of the Nuker Model", *Astron.J.* **125**: 2951.

Humphrey, P. J., Liu, W., and Buote, D. A. 2009, "χ^2 and Poissonian Data: Biases Even in the High-Count Regime and How to Avoid Them", *Astrophys.J.* **693**: 822.

King, I. 1962, "The structure of star clusters. I. an empirical density law", *Astron.J.* **67**: 471.

Krist, J. 1995, "Simulation of HST PSFs using Tiny Tim", in *Astronomical Data Analysis Software and Systems IV*, R.A. Shaw, H.E. Payne, and J.J.E. Hayes, eds., *ASP Conference Series* **77**: 349.

Laurikainen, E., Salo, H., and Buta, R. 2005, "Multicomponent decompositions for a sample of S0 galaxies", *Monthly Notices of the Royal Astronomical Society* **362**: 1319.

MacArthur, L. A., Courteau, S., and Holtzman, J. A. 2003, "Structure of Disk-dominated Galaxies. I. Bulge/Disk Parameters, Simulations, and Secular Evolution", *Astrophys.J.* **582**: 689.

Moffat, A. F. J. 1969, "A Theoretical Investigation of Focal Stellar Images in the Photographic Emulsion and Application to Photographic Photometry", *Astron. Astrophys.* **3**: 455.

Moré, J. J. 1978, "The Levenberg-Marquardt algorithm: Implementation and theory", in *Numerical Analysis*, G.A. Watson, ed., *Lecture Notes in Mathematics* **630**: 105.

Peng, C. Y., Ho, L. C., Impey, C. D., and Rix, H. W. 2002, "Detailed Structural Decomposition of Galaxy Images", *Astron.J.* **124**: 266.

Peng, C. Y., Ho, L. C., Impey, C. D., and Rix, H. W. 2010, "Detailed Decomposition of Galaxy Images. II. Beyond Axisymmetric Models", *Astron.J.* **139**: 2097.

Sérsic, J.-L. 1968, *Atlas de Galaxias Australes* (Cordoba: Obs. Astron.).

Storn, R., and Price, K. 1997, "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization Over Continuous Spaces", *Journal of Global Optimization* **11**: 314.

ter Braak, C.J.F. 2006. "A Markov chain Monte Carlo version of the genetic algorithm differential evolution: easy Bayesian computing for real parameter spaces", *Statistics and Computing* **16** (4): 239–249.

ter Braak, C., and J. Vrugt 2008. "Differential Evolution Markov Chain with snooker updater and fewer chains", *Statistics and Computing* **18** (4): 435–446.

Trujillo, I., Aguerri, J. A. L., Cepa, J., and Gutiérrez, C. M. 2001, "The effects of seeing on Sérsic profiles – II. The Moffat PSF", *Monthly Notices of the Royal Astronomical Society* **328**: 977.

Trujillo, I., Erwin, P., Asensio Ramos, A., and Graham, A. 2004, "Evidence for a New Elliptical-Galaxy Paradigm: Sérsic and Core Galaxies", *Astron.J.* **127**: 1917.

van der Kruit, P. C., and Searle, L. 1981, "Surface Photometry of Edge-on Spiral Galaxies: I. A Model for the Three-dimensional Distribution of Light in Galactic Disks", *Astron. Astrophys.* **95**: 105.

van der Kruit, P. 1988, "The Three-dimensional Distribution of Light and Mass in Disks of Spiral Galaxies", *Astron. Astrophys.* **192**: 117.

Vrugt, J. A., ter Braak, C. J. F., Diks, C. G. H., Robinson, B. A., Hyman, J. M., and Higdon, D. 2009. "Accelerating Markov chain Monte Carlo simulation by differential evolution with self-adaptive randomized subspace sampling", *International Journal of Nonlinear Sciences and Numerical Simulation* **10** (3): 273–290.

Vrugt, J. A. 2016. "Markov chain Monte Carlo simulation using the DREAM software package: Theory, concepts, and MATLAB implementation", *Environmental Modelling and Software* **75**: 273–316.

Yoachim, P., and Dalconton, J. J. 2006, "Structural Parameters of Thin and Thick Disks in Edge-On Disk Galaxies", *Astron.J.* **131**: 226.

York, D. G., et al. 2000, "The Sloan Digital Sky Survey: Technical Summary", *Astron.J.* **120**: 1579.