Notes for Using "Imfit"

Peter Erwin
MPE and USM
erwin@sigmaxi.net

June 15, 2011

Contents

1 What Is It?

`Imfit` is a program for fitting astronomical images — more specifically, for fitting images of galaxies, though it can in principle be used for fitting other sources. The user specifies a set of one or more 2D 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.

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 inividual 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 Krist [1995][1], or an actual stellar image).

A key characteristic of `imfit` is a modular, object-oriented design that allows relatively 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), modifying one additional file to include references to the new function, and re-compiling the program. Notes are [WILL BE] provided to guide the interested user in adding new functions; in most cases, a basic working knowledge of C should suffice.

[1] `http://www.stsci.edu/hst/observatory/focus/TinyTim`

Additional auxiliary programs built from the same codebase exist for generating artificial galaxy images (using the same input/output parameter-file format as `imfit`).

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

System Requirements: `Imfit` has been built and tested on Intel-based MacOS X and Linux (Ubuntu) systems. It uses standard C++ and should work on any Unix-style system with a modern C++ compiler and the Standard Template Library (e.g., GCC v4 or higher). It relies on two external, open-source libraries: version 3 of the CFITSIO library[2] for FITS image I/O and version 3 of the FFTW (Fastest Fourier Transform in the West) library[3] for PSF convolution.

MORE CREDITS – e.g., Craig Markwardt's lmfit code; Differential Evolution code; C++ interface to DE.

2 Getting/Installing `Imfit`

2.1 Pre-Compiled Binaries

Pre-built binaries for Intel-based MacOS X and Linux systems are available XXX.

2.2 Compiling from Source: Outline

1. Install CFITSIO
2. Install FFTW
3. (Optional) Install GNU Scientific Library (GSL) — this is only necessary if you wish to use 2D image functions that rely on GSL; currently, the only such function is the EdgeOnDisk (`func_edge-on-disk.cpp`) component.
4. Install SCons
5. Build `imfit`

2.3 Building `Imfit` from Source

Assuming that CFITSIO and FFTW (and optionally GSL) are already installed on your system, XXX

`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. SCons can be downloaded from `http://www.scons.org/`.

Assuming things are simple, you should build `imfit` and the companion program `makeimage` with the following commands:

[2] `http://heasarc.nasa.gov/fitsio/`
[3] `http://www.fftw.org/`

```
$ scons imfit
$ scons makeimage
```

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

If you do not have GSL installed, you will get compilation errors; use the following commands instead:

```
$ scons define=NO_GSL imfit
$ scons define=NO_GSL makeimage
```

2.3.1 Options: Multithreaded Programs

Mainly useful for making PSF convolution go faster ... Requires that FFTW have been compiled with threading support turned on.

```
$ scons define=FFTW_THREADING imfit
$ scons define=FFTW_THREADING makeimage
```

3 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.

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 ("`-csomefile.dat`" is *not* a valid substitute for "`-c somefile.dat`").

Some notable/useful comand-line flags and options include:

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

- `--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 fitting process (by default, pixel values in the noise map are assumed to be sigma values).
- `--errors-are-variances` — indicates that pixel values in the noise map are variances (sigma^2) instead of sigmas.
- `--errors-are-weights` — indicates that pixel values in the noise map should be interpreted as weights, not as sigmas or variances.

- `--sky` *sky-level* — specifies an original sky background level (in counts/pixel) that was subtracted from the image; used for internal computation of the noise map.
- `--gain` *sky-level* — specifies the A/D gain (electrons/pixel) of the input image; used for internal computation of the noise map.
- `--readnoise` *value* — specifies the read noise (electrons) of the input image; used for internal computation of the noise map.
- `--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 should be used to specify the number of original images; used for internal computation of the noise map. 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 (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.

- `--de` — use Differential Evolution instead of Levenberg-Marquardt as the minimization technique (WARNING: much slower!)

- `--chisquare-only` — Evalute 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.

- `--list-functions` — list all the functions `imfit` can use.
- `--list-parameters` — list all the individual parameters (in correct order) for each functions that `imfit` can use.

4 Trying It Out

In the `examples/` directory are XXX.

The simplest way to run `imfit` is:

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

This converges to a fit in about 2.5 seconds on a 2009 MacBook Pro (2.8 GHz Core 2 Duo processor).

Note that you can specify a subset of an image, thus:

> imfit ic3478rss_256.fits[45:150,200:310] [rest of options]

This will fit columns 45–150 and rows 200–310 of the image (column and row numbering starts at 1); pixel coordinates in the configuration file should refer to locations within the full image.

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 the companion program `makeimage` and the configuration file `makeimage_config_moffat_psf_51_for_ic3478rss.dat`:

> makeimage --ncols=51 --nrows=51 -o psf_moffat_51.fits makeimage_config_moffat_psf_51_for_ic3478rss.dat

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 L-M, required for DE fitting), 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 after a "#" is ignored, which allows for comments at the end of lines.

A model for an image is specified by one or more "function blocks", 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 blocks, 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
      ii. one or more parameter-specification lines

This probably sounds more complicated than it is in practice. Here is a very basic, 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, 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 values.

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

```
# 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 be ≥ 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 blocks, with the first block having two individual functions (so this could be a model for, e.g., simultaneously fitting two galaxies, one as Sérsic + exponential, the other with just an exponential):

```
# 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 block:  Sersic + exponential
X0    150.1    148,152
Y0    149.5    148,152
FUNCTION Sersic # A Sersic function
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
PA    95.0     0,180
ell    0.45    0,1
I_0    90.0    fixed
h     15.0

# This is the second function block:  just a single exponential
X0    225.0    224,226
Y0    181.7    180,183
FUNCTION Exponential # a different exponential!
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 block 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 block 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 block 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 $(0.5, 0.5)$.

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 $+y$ 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. Restricted to lie 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
```

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.) 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 fuller discussions of each function, including their parameters.

- FlatSky — a uniform sky background.
- Gaussian — an elliptical 2D Gaussian function.
- Moffat — an elliptical 2D Moffat function.
- Exponential — an elliptical 2D exponential function.
- Exponential_GenEllipse — an elliptical 2D exponential function using generalized ellipses ("boxy" to "disky" shapes) for the isophote shapes.
- Sérsic — an elliptical 2D Sérsic function.
- Sersic_GenEllipse — an elliptical 2D Sérsic function using generalized ellipses ("boxy" to "disky" shapes) for the isophote shapes.
- FlatExponential — similar to Exponential, but with an inner radial zone of constant surface brightness for $r < R_{\rm break}$.
- 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 $r > R_{\rm ring}$).
- 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. Note that this function requires that the GNU Scientific Library (GSL) be installed; if the GSL is not installed, `imfit` will be compiled without the this function.

- 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.
- EdgeOnRing2Sdie — like EdgeOnRing, but using an asymmetric Gaussian for the radial profile (see description of GaussianRing2Side).

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

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

and the complete list of function parameters for each function can always be obtained by running `imfit` with the "`--list-parameters`" option:

```
$ imfit --list-paremeters
```

7 Extras for Fitting Images

7.1 Masks

7.2 Noise, Variance, or Weight Maps

Note that `imfit` does *not* [, xxx by default,] try to obtain information from the FITS header of an image, for two reasons. First, 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. Second, XXX

7.3 PSF Convolution

8 Rolling Your Own Functions

8.1 A Simple Example

To illustrate how one might make a new function, 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 accomodate the new algorithm.

8.1.1 Create and Edit the Header File

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
```

8.1.2 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 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:

This is the key place where your new function's algorithm is implemented: the computation of the intensity as a function of (scaled) radius. Replace the original version of this method with the following:

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

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

In this simple example, we aren't changing the isophote geometry (i.e., we're still assuming a perfectly elliptical shape), so we don't need to change the GetValue method, which converts pixel position to a scaled radius value. It probably doesn't make sense to change the CalculateSubsamples method, either, so we can leave that alone.

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.

8.1.3 Edit add_functions.cpp

We need to do three 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. Modify the list of function names: XXX
3. 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>();
```

8.1.4 Edit the SConstruct File

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

9 Makeimage

`Imfit` has a companion program called `makeimage`, which will generate model images using the same functions (and parameter files) as `imfit`. In fact, 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 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.

9.1 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/useful comand-line flags and options include:

- `--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.
- `--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

9.2 Configuration Files for `Makeimage`

The configuration file for `makeimage` has essentially the same format as that for `imfit`; parameter limits 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)

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 are in pixels.

A sample function specifiction (giving the parameters in their proper order) is listed for each function description.

Common parameters:

- `PA` = position angle, measured in degrees CCW from the image +y axis.
- `ell` = ellipticity ($1 - b/a$, where a and b are semi-major and semi-minor axes of the ellipse, respectively).

A.1 FlatSky

A constant background.

```
FUNCTION FlatSky
I_sky
```

A.2 Gaussian

An elliptical 2D Gaussian function.

```
FUNCTION Gaussian
PA
ell
I_0
sigma
```

A.3 Moffat

an elliptical 2D Moffat function.

```
FUNCTION Moffat
PA
ell
I_0
fwhm
beta
```

A.4 Exponential

an elliptical 2D exponential function.

```
FUNCTION Exponential
PA
ell
I_0
h
```

A.5 Exponential_GenEllipse

an elliptical 2D exponential function using generalized ellipses ("boxy" to "disky" shapes).

```
FUNCTION Exponential_GenEllipse
PA
ell
c0
I_0
h
```

A.6 Sérsic

an elliptical 2D Sérsic function.

```
FUNCTION Sersic
PA
ell
n
I_e
r_e
```

A.7 Sersic_GenEllipse

an elliptical 2D Sérsic function using generalized ellipses ("boxy" to "disky" shapes).

```
FUNCTION Sersic_GenEllipse
PA
ell
c0
n
I_e
r_e
```

A.8 FlatExponential

similar to Exponential, but with an inner radial zone of constant surface brightness.

```
FUNCTION FlatExponential
PA
ell
I_0
h
r_break
alpha
```

A.9 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.

```
FUNCTION BrokenExponential
PA
ell
I_0
h1
h2
r_break
alpha
```

A.10 GaussianRing

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

```
FUNCTION GaussianRing
PA
ell
A
R_ring
sigma_r
```

A.11 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
ell
A
R_ring
sigma_r_in
sigma_r_out
```

A.12 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. Note that this function requires that the GNU Scientific Library (GSL) be installed; if the GSL is not installed, `imfit` will be compiled without the this function.

```
FUNCTION EdgeOnDisk
PA
I_0
h
alpha
z_0
```

A.13 EdgeOnRing

A simplistic model for an edge-on ring, using two offset components located at distance $\pm$`r` from the center of the function block. Gaussian with size `sigma_r` for the radial profile and a Gaussian with size `sigma_z` for the vertical profile.

```
FUNCTION EdgeOnRing
PA
```

```
I_0
r
sigma_r
sigma_z
```

A.14 EdgeOnRing2Side

Similar to EdgeOnRing, but now the radial profile is asymmetric, with size `sigma_r_in` for the inner side of the ring and `sigma_r_out` for the outer side.

```
FUNCTION EdgeOnRing2Side
PA
I_0
r
sigma_r_in
sigma_r_out
sigma_z
```

References

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.

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

van der Kruit, P. C., & 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