SourceXtractor++ Documentation

Release 1.0.0

SourceXtractor++ team

September 22, 2021

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Introduction

SourceXtractor++ (Source-Extractor ++) is a program that extracts a catalog of sources from astronomical images. It is the successor to the original SExtractor package [1]. **SourceXtractor++** has been completely rewritten in C++ and improves over its predecessor in many ways:

- Support for multiple "measurement" images
- Optimized multi-object, multi-frame model-fitting engine
- Possibility to define complex priors and dependencies for model parameters
- Flexible, Python-based configuration file
- Efficient image data caching
- Multithreaded processing
- Modular code design with support for third-party plug-ins

SourceXtractor++ is a collaborative effort between the Astronomy Department, Université de Genève, the Faculty of Physics, LMU Munich, and the IAP (CNRS/Sorbonne Université).

1.2 License

1.2.1 Code

The SourceXtractor++ code is licensed under a LGPL v3 license:

SourceXtractor++ *is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3.0 of the License, or (at your option) any later version.*

SourceXtractor++ *is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.*

1.2.2 Documentation

1.3 Installing the software

1.3.1 Hardware requirements

Terminal

SourceXtractor++ runs in (ANSI) text-mode from a shell. A graphical environment is not necessary to operate the software.

Memory

Memory requirements depend mostly on the number of images to be analyzed. A rule of thumb is that **SourceXtractor++** requires about 100MB of resident memory per input frame. Extra-memory is taken advantage of through caching of pixel data.

CPUs

SourceXtractor++'s measurement pipeline is multithreaded and can take advantage of multiple CPU cores. As of version 0.13, **SourceXtractor++** scales reasonably well up to 8 cores.

1.3.2 Obtaining SourceXtractor++

From Fedora or EPEL repositories fedora v0.15

SourceXtractor++ is available in Fedora 30 and greater, and in EPEL 7 as well. On those platforms, you can use your package manager to install in the usual way:

```
# dnf install sourcextractor++
```

From Anaconda Cloud astrorama v0.15

SourceXtractor++ is also available for Linux and MacOS via Anaconda cloud. You can install it as follows:

```
$ conda install -c conda-forge -c astrorama sourcextractor
```

Or you can, of course, add permanently the `astrorama` channel to your conda configuration:

```
$ conda config --add channels conda-forge
$ conda config --add channels astrorama
$ conda install sourcextractor
```

It is recommended to install **SourceXtractor++** into its own environment to avoid dependency conflicts.

From Sources tag v0.15

The source package may be downloaded from the official GitHub repository. Links to binary packages for a selection of operating systems are also available at the URL above.

SourceXtractor++ relies on the **Elements** and **Alexandria** packages, which may be downloaded from Astrorama repositories. They are also available in Fedora and EPEL as `elements` and `elements-alexandria`.

It also requires the following packages for installation:

- Boost
- CCFITS
- CFITSIO
- FFTW3
- Log4cpp
- OpenCV
- Python (Python3.x recommended)
- Python-AstroPy
- WCSlib

The following packages are optional:

- Readline
- Ncurses (progress bar display)
- **LevMar** (model fitting engine)
- **GSL** (model fitting engine)

All or most of these packages are available in the main Linux distributions. You will also need the `-devel` version of the packages if you compile **SourceXtractor++** from the source.

If you have installed Elements and Alexandria from your distribution repository (right now, only in Fedora), you can skip the following section.

```
# dnf install elements-devel elements-alexandria-devel
```

Elements is a C++ and CMake framework. It is capable of managing the dependencies of the projects that are based on it, as long as it knows where to find them. By convention, this location usually is `~/Work/Projects`, but you can choose any other.

```
$ export CMAKE_PROJECT_PATH=~/Work/Projects
$ export CMAKE_PREFIX_PATH=$CMAKE_PROJECT_PATH/Elements/cmake
$ mkdir -p $CMAKE_PROJECT_PATH
```

Since Python 2 has been deprecated since the 1st of January 2020, it is recommended to tell Elements and other projects to use Python 3 instead. You can also disable the build of documentation with `-DUSE_SPHINX=OFF`.

```
$ export CMAKEFLAGS="-DPYTHON_EXPLICIT_VERSION=3 -DUSE_SPHINX=OFF"
```

You can always add for convenience these environment variables to your `.bashrc`, or the one corresponding to your shell.

Elements can handle multiple versions of a given project, or with just one. For the later, it suffices to clone the project into `$CMAKE_PROJECT_PATH`.

```
$ cd $CMAKE_PROJECT_PATH
$ git clone https://github.com/astrorama/Elements.git --branch 5.10
$ cd Elements
$ make install -j # -j uses multiple cores
```

Similarly for Alexandria:

```
$ cd $CMAKE_PROJECT_PATH
$ git clone https://github.com/astrorama/Alexandria.git --branch 2.15
$ cd Alexandria
$ make install -j
```

If you have compiled and installed Elements and Alexandria as described on the above section, make sure you are using the proper environment.

```
$ export CMAKE_PROJECT_PATH=~/Work/Projects
$ export CMAKE_PREFIX_PATH=$CMAKE_PROJECT_PATH/Elements/cmake
```

For system-wide installs, this step is not necessary.

Configure the project flags if you have not yet done it:

```
$ export CMAKEFLAGS="-DPYTHON_EXPLICIT_VERSION=3 -DUSE_SPHINX=OFF"
```

As a reminder, that tells **SourceXtractor++** to compile with Python 3, and not to build the documentation. **SourceXtractor++** can be built with Python 2, but this is strongly discouraged.

The sources can be obtained either from the releases as a compressed archive, or cloned with `git` the usual way.

If you download an archived version, you need to uncompress it:

```
$ unzip SourceXtractorPlusPlus-<version>.zip
```

A new directory called `SourceXtractorPlusPlus-<version>` should now appear at the current location on your disk.

If you opt for cloning the repository, make sure you are pointing to the latest commit in the `master` branch.

```
$ git pull
```

Either way, enter the directory you have just created, and build the software.

```
$ cd SourceXtractorPlusPlus
$ make -j4
```

After the binaries are now compiled and available, **SourceXtractor++** can be run with:

```
$ ~/Work/Projects/SourceXtractorPlusPlus/build/run sextractor++ --help
```

1.4 Using SourceXtractor++

SourceXtractor++ is run from the shell with the following syntax:

```
$ sourcextractor++  [--<option> [<arg>]]
```

The parts enclosed within brackets are optional. Any `--<option> <arg>` or `--<option>=<arg>` statement in the command-line overrides the corresponding definition in the *configuration file* or any default value (see *configuration section*).

1.4.1 Input files

SourceXtractor++ accepts images stored in FITS (Flexible Image Transport System) [2]. Contrary to the original **SExtractor**, **SourceXtractor++** relies on the FITSIO library and is therefore compatible with all standard variants of FITS, including compressed images. Both "Basic FITS" (one single header and one single body) and MEF (Multi-Extension FITS) files are recognized. Binary **SourceXtractor++** catalogs produced from MEF images are MEF files themselves. If the catalog output format is set to ASCII, all catalogs from the individual extensions are concatenated in one big file.

Currently, only the first data-plane of multichannel images with NAXIS > 2, is loaded.

In **SourceXtractor++**, as in all similar programs, FITS axis #1 is traditionally referred to as the x axis, and FITS axis #2 as the y axis.

Detection and measurement images

SourceXtractor++ distinguishes between two kinds of science images, detection and measurement images:

- **The detection image** is where the sources are extracted. Rough estimates of position and shapes are obtained from the detection image; they define the photometric apertures and initial guesses for model-fitting parameters that will be applied to the measurement images. Currently there can be only one detection image per run, and sources are detected from a single channel.
- **Measurement images are where the sources are measured**. There can be hundreds (or even more) of them. They need not share the same pixel grid or have the same size as the detection image, which they may only partially overlap. However if the pixel grid is different, both detection and measurement image headers must contain valid WCS information [3, 4]. **SourceXtractor++** uses that information to precisely match celestial positions and areas on all images. A scaling parameter for pixel values can be applied to any measurement image independently, provided either as a `flux_scale` optional argument to *`load_fits_image()`* in the Python measurement configuration file, or as the value of a FITS header keyword (FLXSCALE by default). Note that sources need not be detectable at all on measurement images for the software to work.

Weight-maps

Both detection and measurement images can have their own weight-maps. Weight-maps are companion images that indicate how "noisy" every science pixel is, in terms of variance. The **SourceXtractor++** `--weight-image` command line option and the *`load_fits_image()`* measurement configuration script command may be used to specify the names of the input detection and measurement weight-maps, respectively. See the *Weighting section* for details.

Flag-maps

Flag maps are images in integer format having the same dimensions as the science images, with pixel values that can be used to flag some pixels (for instance, "bad" or noisy pixels). Flag map usage is described in the *flagging section*.

PSF models

PSF models are required by the *model-fitting module*. PSF (Point Spread Function) model files must be provided as FITS binary tables in the PSFEx format. There must be one PSF model per measurement image.

1.4.2 Output

Catalog

SourceXtractor++ writes out a catalog file with a filename set by the `--output-catalog-filename` option. Note that if no `output-catalog-filename` is set in the command line nor in the configuration file, then the catalog is printed in ASCII to the standard output.

The `--output-catalog-format` option sets the format of the output catalog. Currently available options are *FITS* for a FITS binary table, and *ASCII* for an ASCII table.

Output properties

SourceXtractor++ computes only what is required to produce the output catalog and diagnostics. While the original **SExtractor** would automatically trigger the measurement processes depending on the columns the user would like to be included in the catalog, **SourceXtractor++** adopts a more coarse-grained and modular approach through the concept of *properties*.

Properties regroup several scalar or vector values obtained in the context of a specific image measurement or processing. Properties are requested by giving as argument to the `--output-properties` a coma-separated list of keywords. For example, requesting the PixelCentroid property (the default) adds both the `pixel_centroid_x` and `pixel_centroid_y` columns to the output catalog. Note that additional, custom columns may be inserted in the catalog through calls to *`add_output_column`* in the *measurement script*.

The `--list-output-properties` option prints out the complete list of available properties known to **SourceXtractor++**. The lists of catalog columns for the requested properties and for all properties are displayed using `--property-column-mapping` and `--property-column-mapping--all` options, respectively.

The complete list of columns and properties currently known to **SourceXtractor++** is given below.

Diagnostic files

Additional files can be generated by **SourceXtractor++**, providing diagnostics about the source extraction and measurement processes. Currently only *check-images*, which are FITS images containing maps of various types, are produced. Each of the following options, followed by a filename, may be set to trigger the production of a check-image:

- `check-image-model-fitting`: noiseless image of the best-fitting source models
- `check-image-residual`: model-fitting residuals
- `check-image-background`: bicubic-spline-interpolated background model
- `check-image-variance`: variance map
- `check-image-segmentation`: map of detected objects after the segmentation stage
- `check-image-partition`: map of detected objects after the partition stage

- `check-image-grouping`: map of detected objects after the grouping stage
- `check-image-filtered`: filtered copy of the detection image
- `check-image-thresholded`: filtered, thresholded copy of the detection image
- `check-image-auto-aperture`: map of automatic photometric apertures
- `check-image aperture`: map of disk aperturess
- `check-image-moffat`: noiseless image of the best-fitting extended Moffat models
- `check-image-psf`: PSF map

1.4.3 Configuration

There are two types of configuration settings: those dedicated purely to measurements, and those related to the source extraction process and to global operations of **SourceXtractor++**. Measurement settings can be rather complex and require a *Python configuration script* (see the *Measurement section*). Global configuration settings may be changed using command-line options (prefixed with a `--`), however a configuration file is often more convenient for storing settings that do not change from run to run.

Note: Command-line options can be abbreviated, provided that there is no more than one full matching keyword. Example: `--config-file=foo.conf` may be abbreviated as `--conf=foo.conf`.

The configuration file

SourceXtractor++ searches for the configuration file at the beginning of a run, starting from system repositories (which makes it possible to apply specific, system-wide configuration settings), to the current repository. Each time it is run, SourceXtractor++ looks for a configuration file. The name of a dedicated configuration files is given with:

```
$sourcextractor++ --config-file sepp.config
```

Creating a configuration file

SourceXtractor++ dumps all parameters with their default values into a configuration files with:

```
$sourcextractor++ --dump-default-config > my_sepp.config
```

Format of the configuration file

Configuration instructions follow the `<option> = <value>` format. There must be no more than one `<option> = <value>` instance per line. Comments must be preceded with a `#`. Boolean parameters are set/unset with 1/0 such as: `output-flush-unsorted=1`, and the individual parameters of a parameter list are separted with a “,” such as `output-properties=SourceIDs,PixelCentroid,WorldCentroid`

Configuration parameter list

Here is a complete list of all the configuration parameters known to **SourceXtractor++**. Please refer to the next sections for a detailed description of their meaning.

Option	Default	Use
Generic options		
`version`		Print version string
`help`		Produce help message
`config-file`	—	Name of a configuration file
`log-level`	`INFO`	Log level: FATAL, ERROR, WARN, INFO (default), DEBUG
`log-file`	—	Name of a log file
`list-output-properties`		List the possible output properties for the given input parameters and exit
`property-column-mapping-all`		Show the columns created for each property
`property-column-mapping`		Show the columns created for each property, for the given configuration
`progress-min-interval`	5	Minimal interval to wait before printing a new log entry with the progress report
`progress-bar-disable`		Disable progress bar display
Auto (Kron) photometry options		
`auto-kron-factor`	*2.5*	Scale factor for AUTO (Kron) photometry
`auto-kron-min-radius`	*3.5*	Minimum radius for AUTO (Kron) photometry
Background modelling		
`background-cell-size`	*64*	Background mesh cell size to determine a value
`smoothing-box-size`	*3*	Background median filter size
Check images		
`check-image-model-fitting`	—	Path to save the model fitting check image
`check-image-residual`	—	Path to save the model fitting residual check image
`check-image-background`	—	Path to save the background check image
`check-image-variance`	—	Path to save the variance check image
`check-image-segmentation`	—	Path to save the segmentation check image
`check-image-partition`	—	Path to save the partition check image
`check-image-grouping`	—	Path to save the grouping check image
`check-image-filtered`	—	Path to save the filtered check image
`check-image-thresholded`	—	Path to save the thresholded check image
`check-image-auto-aperture`	—	Path to save the auto aperture check image
`check-image-aperture`	—	Path to save the aperture check image
`check-image-moffat`	—	Path to save the moffat check image
`check-image-psf`	—	Path to save the PSF check image
Cleaning		
`use-cleaning`		Enable the cleaning of sources (remove false detections near bright objects)
`cleaning-minarea`	*3*	Minimum # of pixels above threshold
Detection image		
`background-value`	—	Background value to be subtracted from the detection image
`detection-threshold`	*1.5*	Detection threshold above the background
`segmentation-algorithm`	`LUTZ`	Segmentation algorithm to be used. Currently LUTZ is the only choice
`segmentation-disable-filtering`		Disables filtering
`segmentation-filter`	—	Loads a filter
`detection-image`	—	Path to a fits format image to be used as detection image.
`detection-image-gain`	*0*	Detection image gain in e-/ADU (0 = infinite gain)
`detection-image-flux-scale`		Detection image flux scale
`detection-image-saturation`		Detection image saturation level (0 = no saturation)
`detection-image-interpolation`	*1*	Interpolate bad pixels in detection image
`detection-image-interpolation-gap`	*5*	Maximum number if pixels to interpolate over

continues on next page

Table 1.1 – continued from previous page

Option	Default	Use
External flag options		
`flag-image-*`	—	FITS file containing the external flag
`flag-type-*`	—	The combination type of the external flag (OR, AND, MIN, MAX, MOST)
Extraction		
`detect-minarea`	*3*	Minimum # of pixels above threshold
`use-attractors-partition`		Enables the use of attractors for partitioning
Grouping		
`grouping-algorithm`	`NONE`	Grouping algorithm to be used (NONE, MOFFAT)
`grouping-moffat-threshold`	*0.02*	Threshold used for Moffat grouping.
Magnitude		
`magnitude-zeropoint`	*0*	Magnitude zero point calibration
Measurement config		
`python-config-file`	—	Measurements Python configuration file
`python-arg`	—	Parameters to pass to Python via *sys.argv*
Memory usage		
`tile-memory-limit`	*512*	Maximum memory used for image tiles cache in megabytes
`tile-size`	*256*	Image tiles size in pixels
Model Fitting		
`model-fitting-iterations`	*1000*	Maximum number of iterations allowed for model fitting
Multi-threading		
`thread-count`	*4*	Number of worker threads (0=disable all multithreading)
Multi-thresholding		
`partition-multithreshold`		Activate multithreshold partitioning
`partition-threshold-count`	*32*	of thresholds
`partition-min-area`	*3*	Minimum area in pixels to consider partitioning
`partition-min-contrast`	*0.005*	Minimum contrast for partitioning
Output configuration		
`output-catalog-filename`	—	The file to store the output catalog
`output-catalog-format`	`FITS`	The format of the output catalog, one of ASCII or FITS
`output-properties`	`PixelCentroid`	The output properties to add in the output catalog
Plugin configuration		
`plugin-directory`	—	Path to a directory that contains the plugins
`plugin`	—	Defines a plugin to load (without file extension). Can be used multiple times
Variable PSF		
`psf-filename`	—	PSF image file (FITS format)
`psf-fwhm`	—	Generate a Gaussian PSF with the given full-width half-maximum (in pixels)
`psf-pixel-sampling`	—	Generate a Gaussian PSF with the given pixel sampling step size
Weight map		
`weight-image`	—	Path to a FITS format image to be used as weight image

continues on next page

Table 1.1 – continued from previous page

Option	Default	Use
`weight-absolute`	*0*	Is the weight map provided as absolute values or relative to background
`weight-type`	`BACKGROUND`	Weight image type (BACKGROUND, RMS, VARIANCE, WEIGHT)
`weight-scaling`	*1*	Weight map scaling factor
`weight-threshold`	—	Threshold for pixels to be considered bad pixels. In same units as weight map.
`weight-usesymmetry`	*1*	Use object symmetry to replace pixels above the weight threshold for photometry

1.5 Processing

The complete analysis of an image is fully automated (Fig. 1.1). There are three main steps in the processing: detection, collection and measurement:

- During the detection step, image pixels from the detection image are background-subtracted, filtered and segmented on-the-fly. The extracted source candidates are then deblended.
- During the collection step, a source model is fit to the detection image. Overlapping sources are identified, grouped and pruned ("cleaned").
- During the measurement step, sources are analysed individually or as part of their group. Measurements are written to the output catalog.

The following sections describe each of these operations in more detail.

1.5.1 Modeling the background

On linear detectors, the value measured at each pixel is the sum of a "background" signal and light coming from the sources of interest. To be able to detect the faintest objects and make accurate measurements, **SourceXtractor++** needs first computing a precise estimate of the background level at any position of the image: a *background map*. Strictly speaking, there should be one background map per source, that is, what would the image look like if that very source was missing. However, one can start by assuming that most discrete sources do not overlap too severely — which is generally the case for high galactic latitude fields —, and that the background varies smoothly across the field. **SourceXtractor++**'s current background model is essentially the same as that of **SExtractor** [1].

Background estimation

To compute the background map, **SourceXtractor++** makes a first pass through the pixel data, estimating the local background in each cell of a square grid that covers the whole frame. The background estimator is a combination of $\kappa\,\sigma$ clipping and mode estimation, similar to Stetson's **DAOPHOT** program [5, 6].

Briefly, the local background histogram is clipped iteratively until convergence at $\pm 3\sigma$ around its median. The mode of the histogram is estimated using:

$$\mathrm{Mode} = 2.5 \times \mathrm{Median} - 1.5 \times \mathrm{Mean}. \tag{1.1}$$

Using simulated images, the expression above was found more accurate with clipped distributions :cite:1996AAS_117_393B than the usual approximation (e.g., [7]):

$$\mathrm{Mode} = 3 \times \mathrm{Median} - 2 \times \mathrm{Mean}. \tag{1.2}$$

Fig. 1.2 shows that the mode estimation in (1.1) is considerably less affected by source crowding than a simple clipped mean [8, 9] but it is $\approx 30\%$ noisier. Obviously (1.1) is not valid for any distribution; **SourceXtractor++** falls back to a simple median for estimating the local background value if the mode and the median disagree by more than 30%.

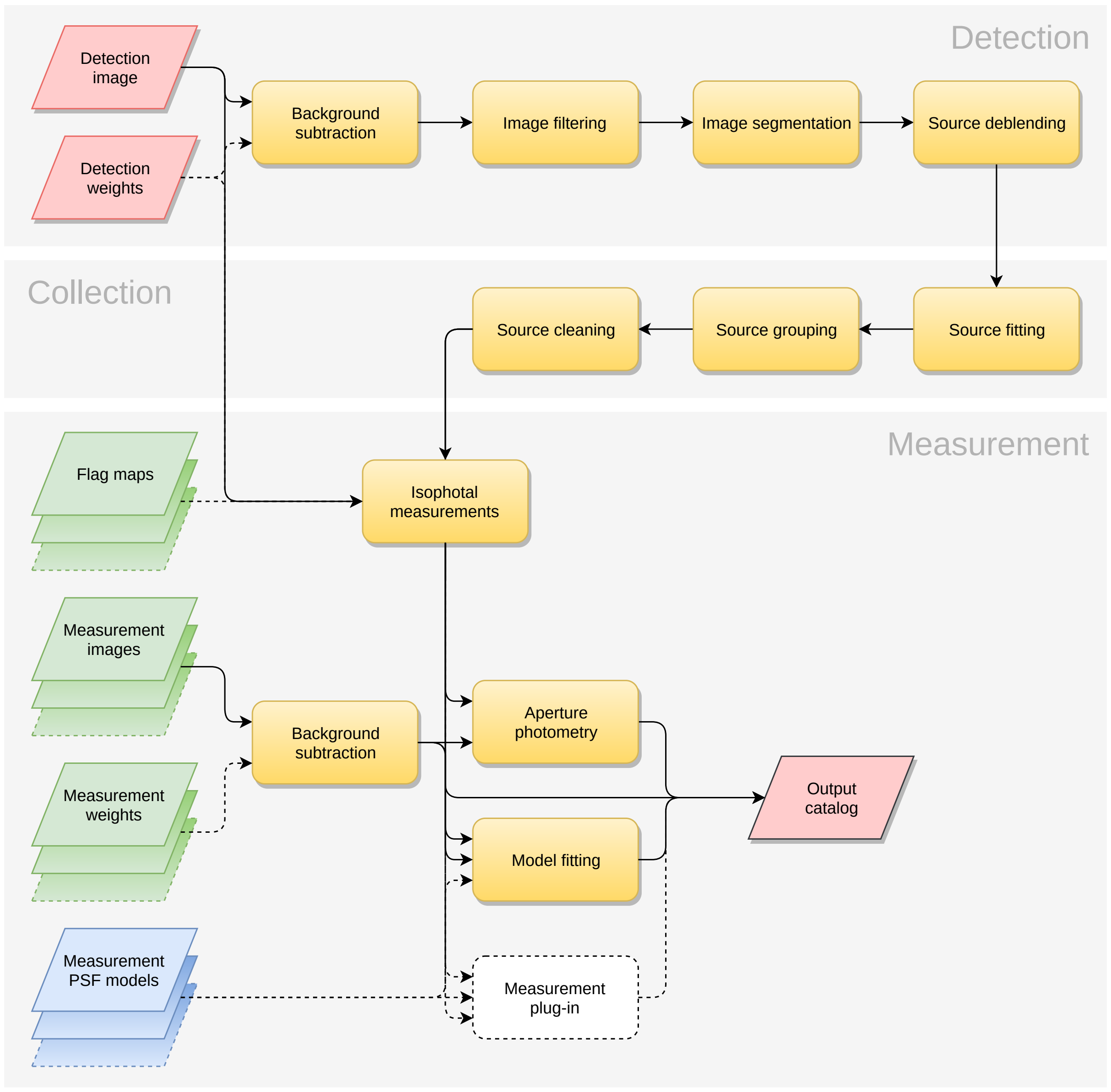

Fig. 1.1: Layout of the main **SourceXtractor++** procedures. *Dashed arrows* represent optional inputs.

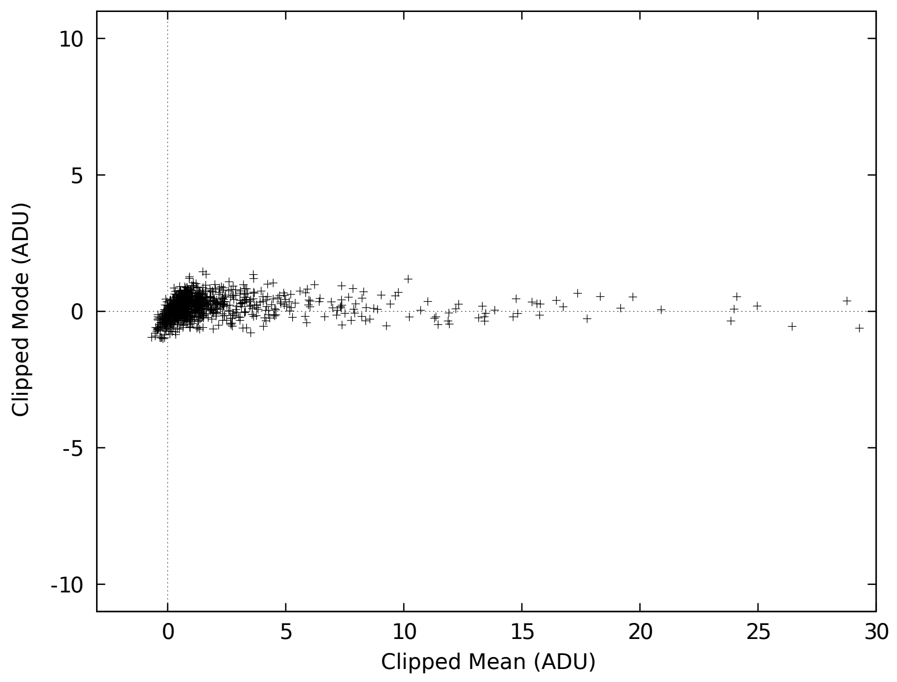

Fig. 1.2: Simulations of 32×32 pixels background cells contamined by random Gaussian profiles. The true background lies at 0 ADUs. While being a bit noisier, the clipped "mode" gives a more robust estimate than the clipped mean in crowded regions.

Background map

Once the values of all background cells have been estimated, a median box filter is applied to suppress possible local overestimations due to bright stars. Median filtering helps reducing possible ringing effects of the bicubic-spline around bright features. The final background map is a (natural) bicubic-spline interpolation between grid cells.

In parallel with the making of the background map, a *background noise map*, that is, a map of the background noise standard deviation is produced. It is used as an internal weight map when the `weight-type` configuration parameter is set to `weight-type=background`, which is the default.

Configuration and tuning

Note: All background configuration parameters also affect background noise maps.

The choice of the `background-cell-size` is very important. If it is too small, the background estimation is affected by the presence of objects and random noise. Most importantly, part of the flux of the most extended objects can be absorbed into the background map. If the cell size is too large, it cannot reproduce the small scale variations of the background. Therefore a good compromise must be found by the user. For reasonably sampled images, cell sizes between 32 to 512 pixels are generally appropriate.

The user has some control over background map filtering by specifying the size of the median filter box . The `smoothing-box-size` configuration parameter sets the box size in pixels. `smoothing-box-size=3` is the default and is sufficient in most cases. `smoothing-box-size=1` deactivates filtering. Larger dimensions may occasionally be used to compensate for small background cell sizes, or in the presence of large image artifacts.

By default, the computed background maps are automatically subtracted from input images. However there are situations where subtracting a *constant* from the detection image may be more relevant, e.g., for images with strongly non-Gaussian background noise PDF (Probability Density Function)s). The `background-value` configuration parameter may be set to subtract a specific value from the input detection image, instead of the background map. It is unset by default.

Computing cost

The background estimation operation is generally I/O (Input/Output)-bound, unless the image file already resides in the disk cache.

1.5.2 Weighting

The noise level in astronomical images is often fairly constant, that is, constant values for the gain, the background noise and the detection thresholds can be used over the whole frame. Unfortunately in some cases, like strongly vignetted or composited images, this approximation is no longer good enough. This leads to detecting clusters of detected noise peaks in the noisiest parts of the image, or missing obvious objects in the most sensitive ones. **SourceXtractor++** is able to handle images with variable noise. It does it through *weight maps*, which are frames having the same size as the images where objects are detected or measured, and which describe the noise intensity at each pixel. These maps are internally stored in units of *absolute variance* (in ADU^2). We employ the generic term *weight map* because these maps can also be interpreted as quality index maps: infinite variance ($\geq 10^{30}$ by definition in **SourceXtractor++**) means that the related pixel in the science frame is totally unreliable and should be ignored.

The variance format was adopted as it linearizes most of the operations done over weight maps (see below). However noise covariances between pixels are ignored.

Effect of weighting

Weight-maps modify the working of **SourceXtractor++** in the several ways:

- The detection threshold t above the local sky background in the detection image is adjusted for each pixel i with variance σ_i^2: t_i =`detection-threshold`$\times\sqrt{\sigma_i^2}$, where `detection-threshold` is expressed in units of standard deviations of the background noise. Pixels having a variance *above* the threshold set with the `--weight-threshold` command line option are never detected. This may result in splitting objects crossed by a group of bad pixels. Interpolation should be used to avoid this problem. If convolution filtering is applied for detection then the variance map is also convolved.
- The cleaning process takes into account the exact individual thresholds assigned to each pixel before deciding whether a faint detections must be merged with a larger one.
- Bad pixels are discarded from the background statistics. If more than 50% of the pixels in a background cell are bad, then the background and the standard deviation values are discarded and interpolated from the nearest valid cells.
- Measurement uncertainties assume that the standard deviation of the local background noise is simply $\sqrt{\sigma_i^2}$.
- Detection image pixels with weights beyond `weight-threshold` (above or below depending on the `--weight-type` option argument, see below) are discarded.

Caution: Although raw Charge-Coupled Device exposures have essentially white noise, this is not the case for the resampled images that enter image stacks; resampling induces a significant amount of noise correlation between neighboring pixels. Despite the fact that the correlation length is often smaller than the patterns to be detected or measured, and constant over the image, noise correlations bias the weight rescaling process. It is therefore recommended to deactivate the weight-map rescaling when working with resampled image and use weight-maps with absolute values whenever possible.

Weight-map formats

Detection weight-map

SourceXtractor++ reads and converts to its internal variance format several types of weight-maps. The detection weight-map can either be read from a FITS file specified by the `--weight-image` command line option, or computed internally. The type of detection weight-map must be selected using the `--weight-type` command-line option. Valid `weight_type` values are:

- **`background`** : The science image itself is used to compute internally a variance map (the `--weight-type` command-line option is ignored). Robust (3σ-clipped) variance estimates are first computed within the same background cells as the *background model*[1]. The resulting low-resolution variance map is then bicubic-spline-interpolated on the fly to produce the actual full-size variance map.
- **`rms`** : The FITS image specified using the `--weight-image` option is a weight-map in units of standard deviations (in ADUs per pixel).
- **`variance`** : The FITS image specified using the `--weight-image` option is a weight-map in units of relative variance.
- **`weight`** : The FITS image specified using the `--weight-image` option is a weight-map in units of inverse variance.

Measurement weight-map

Measurement weight-maps are managed identically to detection weight-maps, except that configuration must be done through the *`load_fits_image()`* command options in the *measurement configuration script*, instead of command line options.

1.5.3 Flagging

Flags are binary attributes of the detected sources. They are set to indicate, e.g., that a source is saturated or that it has been deblended. Flags are grouped in columns in the **SourceXtractor++** output catalog. Each column element is an integer, comprising several flag bits as a sum of powers of 2. Many properties produce flag columns: AperturePhotometry (aperture_flags), AutoPhotometry (auto_flags), FlexibleModelFitting (fmf_flags), … For example, the source_flags column is produced by the SourceFlags property; a saturated detection close to an image boundary will have source_flags = 4+8 = 12 in decimal.

External flags

External flags are derived from *Flag-maps*. Flag maps are images in (unsigned) integer format having the same size as the detection image, where each integer represents a set of flag bits as a sum of powers of 2.

Flag images are specified as arguments to the `--flag-image-*` option(s), where the `*` represents an arbitrary character or string; for instance: `--flag-image-cosmic cosmic.flag.fits`.

Different combinations of flags can be applied within the isophotal footprint that defines objects, to produce a unique integer that will be written to the catalog. How the flags are combined within the isophotal footprint can bet set with the `--flag-type-*` option(s). Valid `flag-type` values are:

- `or` : An arithmetic (bitwise) OR between pixels is applied to all flags independently. For instance if a pixel is set to $9 = 1001_2$ and another to $5 = 0101_2$, then the result will be $1001_2 \vee 0101_2 = 1101_2 = 13$
- `and` : An arithmetic AND between pixels is applied to all flags independently. For instance with inputs from the previous example the result will be $1001_2 \wedge 0101_2 = 0001_2 = 1$
- `min` : The result is the minimum value of the flag combination within the isophotal footprint. For instance with inputs from the previous example the result will be $\min(9, 5) = 5$

[1] The *background map filtering procedure* is also applied to the variance map.

- `max` : The result is the maximum value of the flag combination within the isophotal footprint. For instance with inputs from the previous example the result will be $\max(9, 5) = 9$
- `most`: The result is the (non-zero) flag combination that is most represented within the isophotal footprint. For instance if a pixel is set to 9 and two other pixels are set to 5, then the result will be 5.

1.6 Measuring

Once sources have been detected and deblended, they enter the measurement phase.

SourceXtractor++ performs three categories of measurements: isophotal, aperture, and model-fitting.

Isophotal measurements are done *on the detection image* only. They exclusively take into account object pixels with values exceeding the detection threshold. They run quick and are reasonably immune to light contamination by neighbors. However as such they generally only provide heavily biased estimates of physical quantities, especially for fluxes. Therefore their usage should be restricted to rough estimation and to make first guesses. Isophotal measurements do not require any specific configuration. They depend only on detection, deblending and background-subtraction settings.

Aperture measurements are done on the measurement images. They involve pixels within geometric apertures, such as disks and ellipses. The size and shape of the apertures may be fixed, or adaptive to every object. Aperture measurements are largely sub-optimal from the point-of-view of SNR (Signal-to-Noise Ratio). However they are generally tolerant to object shape variations, and offer a good control of measurement biases. They can be very sensitive to contamination by the light of neighbors, although this can be mitigated (see ??)

Model-fitting measurements are performed on the measurement images. They are close to optimum from the point-of-view of SNR, but they require accurate PSF models (one per image). PSF models can be computed using the **PSFEx** package. **SourceXtractor++** can fit mixtures of models to clumps of overlapping objects, which is generally effective at taking care of the contamination by the light of neighbors. The main inconvenient of model-fitting is that it is much slower than aperture photometry.

1.6.1 The measurement configuration script

Measurement settings, as well as *grouping procedures* and catalog outputs must be defined in a configuration script that uses the Python language: the measurement configuration script. The script filename is set with the `python-config-file` configuration option. Thanks to its flexibility, the Python language makes it possible to set up arbitrarily complex rules to finely control the measurement process. **SourceXtractor++**'s configuration library classes and functions must be imported at the beginning of the script:

```python
from sourcextractor.config import *
```

The same goes for other Python libraries that might be needed for the current configuration, such as glob for filename expansion, math or NumPy for numerical computations, …

Caution: The measurement configuration script is excusively meant to be executed by the **SourceXtractor++** built-in Python interpreter, and cannot be run on its own.

Measurement images

Measurement images are scientific images stored as FITS files and used only for measurements. Every measurement image can be associated a PSF model (**PSFEx**'s `.psf` file), and a weight map. The *load_fits_image()* function creates a measurement image from a FITS image filename; for example:

```
mesimage = load_fits_image("image.fits")
```

The list of currently loaded measurement images can be displayed using the *print_measurement_images()* function. The following will print the list of measurement images on screen (more precisely, on the stderr channel):

```
print_measurement_images()
```

In practice, one will generally want to load several images with a single instruction, not one by one. However before addressing this point we need to introduce the concept of *grouping*.

Grouping and splitting

SourceXtractor++ measures the properties of sources from instances spread across multiple exposures. Each exposure has its own context defined by a specific combination of filter, instrument, epoch, etc. Depending on science goals, one may choose to group images by time of observation (e.g., to build light curves), filter (for regular photometry), . . . This is the purpose of **SourceXtractor++**'s grouping (and splitting) mechanisms.

There are two types of groups: image groups and measurement groups.

Image groups

Image groups offer a convenient way to load many images, and group them according to specific criteria. The most efficient way to instantiate an image group is through the *load_fits_images()* function:

```
imagegroup = load_fits_images(["image_01.fits", "image_02.fits"])
```

In practice, the easier way to load large series of images, PSF models and weight maps is through filename expansion and sorting, for instance:

```
imagegroup = load_fits_images(
    sorted(glob("image_??.fits")),
    psfs=sorted(glob("image_??.psf")),
    weights=sorted(glob("image_??.weight.fits")),
    weight_type='weight'
)
```

All non-empty lists must contain the same number of items. It is possible to add images or another group to a pre-existing image group using the *add_images()* method, e.g.:

```
anotherimage = load_fits_image(
    "anotherimage.fits",
    psf="anotherimage.psf",
    weight="anotherimage.weight.fits",
    weight_type='weight'
)
imagegroup.add_images(anotherimage)
```

Splitting

Now, within an image group one may want to create subgroups based on, e.g., band pass filters or epochs. This is easily accomplished using the *split()* method. Splitting may be done according to a FITS header keyword with the *ByKeyword()* callable. For instance, to generate subgroups each with a different filter (assuming all image headers contain the FILTER keyword):

```
imagegroup.split(ByKeyword('FILTER'))
```

ByPattern() also checks for a FITS header keyword, with the difference that a regular expression provided by the user is applied to the keyword value. The first matching group acts as the grouping key, which means that a 'capturing group (within parentheses) <https://www.regular-expressions.info/brackets.html>'_ must be present in the regular expression. For instance, the following command groups images by year of observation, ignoring the rest of the date:

```
imagegroup.split(ByPattern('DATE-OBS', "^(\d{4})"))
```

Note: Subgroups can themselves be split into subgroups, at any level.

Accessing subgroups

Image subgroups are custom maps. One can access individual subgroups much like a Python dictionary. In the first *group splitting example above*, the subgroup of images from year 2007 (if they exists) is simply imagegroup['2007'], and looping through all subgroups is as simple as:

```
for year, subgroup in imagegroup:
  print(int(year), subgroup)
```

Measurement groups

Measurements are applied to measurement groups and subgroups. Measurement groups are similar to image groups, although they maintain additional information and their image and subgroup content cannot be updated. A measurement group (*MeasurementGroup*) is instanciated from a completed image groups, e.g.,

```
mesgroup = MeasurementGroup(imagegroup)
```

Like image subgroups, measurement subgroups are custom maps and can be managed in a similar way.

1.6.2 Measurements

Isophotal measurements

Position and shape

The following quantities are derived from the spatial distribution $\mathcal{S}$ of pixels detected above the detection threshold (see *description*).

Important: Unless otherwise noted, the pixel values used for computing "isophotal" positions and shapes are taken from the filtered, background-subtracted detection image.

Photometry

Aperture photometry

Besides *isophotal*, PSF and *model-fitting* flux estimates, **SourceXtractor++** can currently perform two types of aperture flux measurements: *fixed-aperture* and *adaptive-aperture*.

Both types are applied to the measurement image(s). In **SourceXtractor++** runs with only a detection image it is necessary to define the detection image as measurement in a python configuration files such as:

```
from glob import glob
import numpy as np
from sourcextractor.config import *

top = load_fits_image(
   '../EUC_MER_BGSUB-MOSAIC-VIS_TILE79171-SM.fits',
   weight='../EUC_MER_MOSAIC-VIS-RMS_TILE79171-SM.fits',
   weight_type='rms',
   constant_background = 0.0,
   weight_absolute=1
)

mesgroup = MeasurementGroup(top)

all_apertures = []
for img in mesgroup:
   all_apertures.extend(add_aperture_photometry(img, 10 ) )
add_output_column('aperture', all_apertures)
```

Fixed-aperture flux

The fixed aperture flux measurements are requested with `output-properties=...,AperturePhotometry,....` The diameter of the aperture is specified in the python configuration file (see above) with `all_apertures.extend(add_aperture_photometry(img,<diameter [pix]>))`. It is possible to give a vector with several diameters `[diam_1, diam_2, diam_3, ...]`.

It is also necessary to append the aperture measurements to the output with a given column name such as `add_output_column('aperture', all_apertures)`. The measurements have the dimension `n x m` for each object with `n` the number of measurement images and `m` the number of diameters.

The fixed aperture checkimage is specified with `check-image-aperture=<name.fits>` and provides a visual impression of the apertures for each measurement image.

Automatic aperture flux

SourceXtractor++'s automatic aperture photometry routine derives from Kron's "first moment" algorithm [10]:

[10] and [9] have shown that for stars and galaxy profiles convolved with Gaussian seeing, $\geq 90\%$ of the flux is expected to lie inside a circular aperture of radius $kr_{\rm Kron}$ with $k = 2$, almost independently of the magnitude. Experiments have shown [1] that this conclusion remains unchanged if one replaces the circular aperture with the "Kron elliptical aperture" $\mathcal{K}$ with reduced pseudo-radius $kr_{\rm Kron}$.

Automatic aperture flux measurements are requested with `output-properties=...,AutoPhotometry,....` The scale factor k for the Kron radius $r_{\rm Kron}$ and the minimal Kron radius $r_{\rm Kron,min}$ can be adjusted with the parameters `auto-kron-factor=` and `auto-kron-min-radius=`, respectively. The measurements have the dimension `n` for each object with `n` the number of measurement images.

The automatic aperture checkimage is specified with `check-image-auto-aperture=<name.fits>` and provides a visual impression of the automatic apertures for each measurement image.

Model fitting

Fitting procedure

SourceXtractor++ can fit models to the images of detected objects. The fit is performed by minimizing the loss function

$$\lambda(\boldsymbol{q}) = \sum_i \left(g\left(\frac{p_i - \tilde{m}_i(\boldsymbol{q})}{\sigma_i} \right) \right)^2 + \sum_j \left(\frac{f_j(\boldsymbol{q}) - \mu_j}{s_j} \right)^2 \tag{1.3}$$

with respect to components of the model parameter vector $\boldsymbol{q}$. $\boldsymbol{q}$ comprises parameters describing the shape of the model and the model central coordinates $\boldsymbol{x}$.

Modified least squares

The first term in (1.3) is a modified weighted sum of squares that aims at minimizing the residuals of the fit. p_i, $\tilde{m}_i(\boldsymbol{q})$ and σ_i are respectively the pixel value above the background, the value of the resampled model, and the pixel value uncertainty at image pixel i. $g(u)$ is a derivable monotonous function that reduces the influence of large deviations from the model, such as the contamination by neighbors (Fig. 1.3):

$$g(u) = u_0 \operatorname{arsinh} \frac{u}{u_0}. \tag{1.4}$$

u_0 sets the level below which $g(u) \approx u$. In practice, choosing $u_0 = \kappa\sigma_i$ with $\kappa = 10$ makes the first term in (1.3) behave like a traditional weighted sum of squares for residuals close to the noise level.

Caution: The cost function in (1.3) is optimized for noise distributions with a Gaussian core and makes model-fitting in **SourceXtractor++** appropriate only for image noise with a PDF symmetrical around the mean.

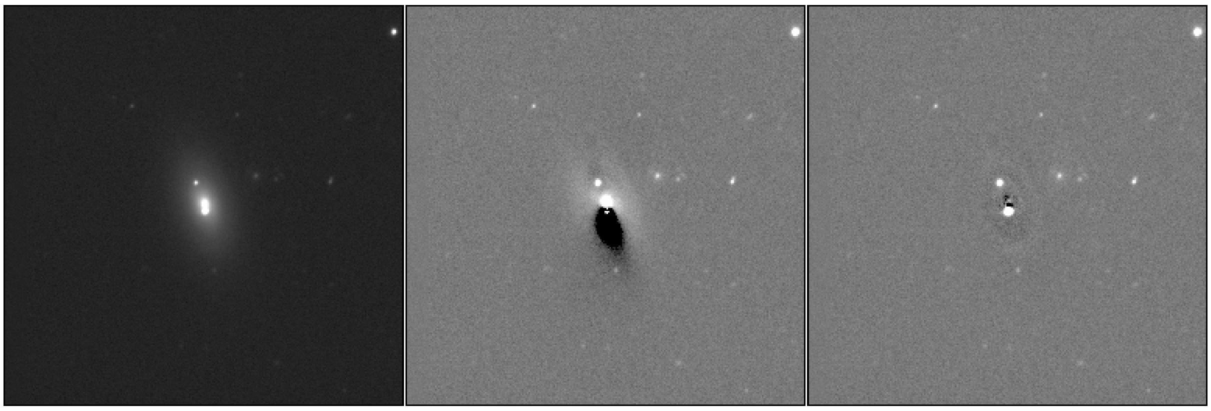

Fig. 1.3: Effect of the modified least squares loss function on fitting a model to a galaxy with a bright neighbor. *Left*: the original image; *Middle*: residuals of the model fitting with a regular least squares ($\kappa = +\infty$); *Right*: modified least squares with $\kappa = 10$.

The vector $\tilde{\boldsymbol{m}}(\boldsymbol{q})$ is obtained by convolving the high resolution model $\boldsymbol{m}(\boldsymbol{q})$ with the local PSF model $\boldsymbol{\phi}$ and applying a resampling operator $\mathbf{R}(\boldsymbol{x})$ to generate the final model raster at position $\boldsymbol{x}$ at the nominal image resolution:

$$\tilde{\boldsymbol{m}}(\boldsymbol{q}) = \mathbf{R}(\boldsymbol{x})(\boldsymbol{m}(\boldsymbol{q}) * \boldsymbol{\phi}). \tag{1.5}$$

$\mathbf{R}(\boldsymbol{x})$ depends on the pixel coordinates $\boldsymbol{x}$ of the model centroid:

$$\mathbf{R}_{ij}(\boldsymbol{x}) = h\left(\boldsymbol{x}_j - \eta.(\boldsymbol{x}_i - \boldsymbol{x})\right), \tag{1.6}$$

where h is a 2-dimensional interpolant (interpolating function), $\boldsymbol{x}_i$ is the coordinate vector of image pixel i, $\boldsymbol{x}_j$ the coordinate vector of model sample j, and η is the image-to-model sampling step ratio (sampling factor) which is by default defined by the PSF model sampling. We adopt a Lánczos-4 function [11] as interpolant.

Configuring model-fitting

The model-fitting process can be precisely defined and tuned in the *measurement configuration Python script*.

Model parameters

In **SourceXtractor++**, any of the model parameters q_j may be a constant parameter, a free parameter, or a dependent parameter.

Constant parameters

In the model fitting configuration, constant parameters are declared using the *ConstantParameter()* construct:

```
size = ConstantParameter(42)
```

One can also use a lambda expression based on actual measurements for the current object:

```
size = ConstantParameter(lambda o: 2 * o.get_radius())
```

Free parameters

Free parameters are adjusted by minimizing $\lambda(\boldsymbol{q})$ in (1.3). They are declared using the *FreeParameter()* construct. The free parameter initial value follows the same rules as the constant parameter value: it may be defined as a simple number supplied by the user, e.g.:

```
size = FreeParameter(42, range)
```

or as a lambda expression, such as:

```
size = FreeParameter(lambda o: 2.5 * o.get_radius(), range)
```

Many model parameters are valid only over a restricted domain. Fluxes, for instance, cannot be negative. The purpose of the range argument is to define the boundaries of the domain. In **SourceXtractor++**, this domain restriction is achieved through a change of variables, applied individually to every model parameter:

$$q_j = f_j(q_j^{(\mathrm{min})}, q_j^{(\mathrm{max})}, Q_j). \tag{1.7}$$

The "model" variable q_j is bounded by the lower limit $q_j^{(\mathrm{min})}$ and the upper limit $q_j^{(\mathrm{max})}$ by construction. The "engine" variable Q_j can take any value, and is actually the parameter that is being adjusted in the fit, although it does not have any physical meaning.

Parameter range

The *Range()* construct is used to set $q_j^{(\mathrm{min})}$, $q_j^{(\mathrm{max})}$, and $f_j()$:

```
range = Range((-1,1), RangeType.LINEAR)
```

The first argument is a tuple of 2 numbers specifying the lower and upper limits of the range. The range type defines $f_j()$. Currently supported range types are linear (RangeType.LINEAR) and exponential (RangeType.EXPONENTIAL). Linear ranges are appropriate for parameters such as positions or shape indices. Exponential ranges are better suited to strictly positive parameters with a large dynamic range, such as fluxes and aspect ratios:

```
range = Range((0.01,100), RangeType.EXPONENTIAL)
```

The relation between model and engine parameters is plotted Fig. 1.4 for both examples. Table 1.2 details the formula applied for currently supported range types.

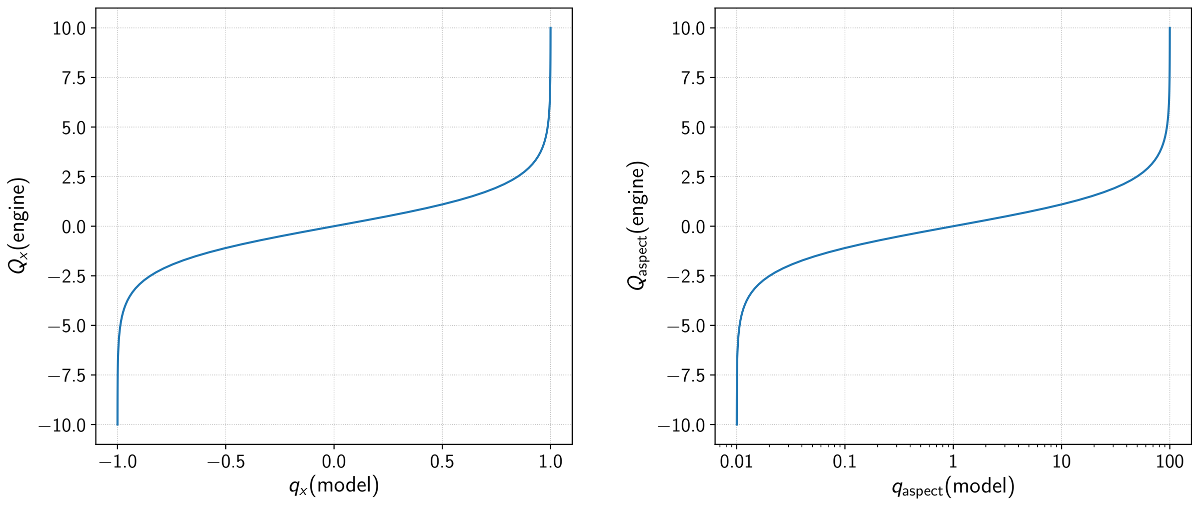

Fig. 1.4: Engine vs model parameters. *Left:* example of a linear range (x position parameter in the range $]-1,1[$). *Right*: example of an exponential range (aspect ratio parameter in the range $]0.01,100[$). Note the logarithmic scale of the abcissa in the second example.

Table 1.2: Types of changes of variables applied to model parameters

Type	Model $\xrightarrow{f^{-1}}$ Engine	Engine $\xrightarrow{f}$ Model	Examples
Unbounded (linear)	$Q_j = q_j$	$q_j = Q_j$	Position angles
RangeType.LINEAR	$Q_j = \ln \frac{q_j - q_j^{(\mathrm{min})}}{q_j^{(\mathrm{max})} - q_j}$	$q_j = \frac{q_j^{(\mathrm{max})} - q_j^{(\mathrm{min})}}{1 + \exp - Q_j} + q_j^{(\mathrm{min})}$	positions Sersic index
RangeType.EXPONENTIAL	$Q_j = \ln \frac{\ln q_j - \ln q_j^{(\mathrm{min})}}{\ln q_j^{(\mathrm{max})} - \ln q_j}$	$q_j = q_j^{(\mathrm{min})} \frac{\ln q_j^{(\mathrm{max})} - \ln q_j^{(\mathrm{min})}}{1 + \exp - Q_j}$	fluxes aspect ratios

In practice, we find this approach to ease convergence and to be much more reliable than a box constrained algorithm [12].

Predefined free parameters

SourceXtractor++ comes with two pre-defined free parameters for easily initializing positions (*get_pos_parameters()*) and fluxes (*get_flux_parameter()*):

```
def get_pos_parameters():
  return (
      FreeParameter(lambda o: o.get_centroid_x(), Range(lambda v,o: (v-o.get_radius(),
↪ v+o.get_radius()), RangeType.LINEAR)),
      FreeParameter(lambda o: o.get_centroid_y(), Range(lambda v,o: (v-o.get_radius(),
↪ v+o.get_radius()), RangeType.LINEAR))
  )
```

```
def get_flux_parameter(type=FluxParameterType.ISO, scale=1):
  func_map = {
      FluxParameterType.ISO : 'get_iso_flux'
  }
  return FreeParameter(lambda o: getattr(o, func_map[type])() * scale, Range(lambda v,
↪o: (v * 1E-3, v * 1E3), RangeType.EXPONENTIAL))
```

Note the use of lambda functions in the first argument to *Range()*.

Dependent parameters

It is often useful to define dependencies between parameters. Dependent parameters are declared using the *DependentParameter()* construct. For instance, one may wish to adjust the sizes of two components of a model in parallel:

```
size1 = FreeParameter(lambda o: 2 * o.get_radius(), Range((0.01, 100.0), RangeType.
↪EXPONENTIAL))
size2 = DependentParameter(lambda s: 1.2 * s, size1)
```

Parameter dependencies are useful for computing new output columns from fitted parameters:

```
MAG_ZEROPOINT = 33.568
mag = DependentParameter(lambda f: -2.5 * np.log10(f) + MAG_ZEROPOINT, flux)
```

Parameter dependencies may also be used to apply changes of variables for setting complex priors, as we shall see in the next section.

Models

The full models $\boldsymbol{m}(\boldsymbol{q})$ are defined as a sum of one or several 2D functions. These model components are added using the *add_model()* function. For instance, for adding a point source model component one may insert

```
add_model(group, PointSourceModel(x, y, flux))
```

Composite models (e.g., a galaxy bulge plus a disk component) may be generated by inserting multiple `add_model()` with the same *measurement group*. Model components need not be coaxial.

SourceXtractor++ currently supports the following models.

Constant: *ConstantModel*

This is the simplest model, which applies a constant offset to the image:

$$m_{\texttt{Constant}} = m_0 \tag{1.8}$$

Point source: *PointSourceModel*

The point source model is appropriate for representing unresolved sources such as stars. It is a scaled delta function, and as such depends only on a coordinate vector $\boldsymbol{r} = (x, y)$ and a flux m_0.

$$\boldsymbol{m}_{\texttt{POINTSOURCE}}(\boldsymbol{r}) = m_0 \delta(\boldsymbol{r}) \tag{1.9}$$

Sérsic: *SersicModel*

The Sérsic model has an elliptical shape with aspect ratio ρ and position angle θ, and a Sérsic profile [13] with effective radius R_e and Sérsic index n:

$$m_{\texttt{Sersic}}(r) = m_0 \exp\left(-b(n)\left(\frac{R}{R_e}\right)^{1/n}\right), \tag{1.10}$$

where $b(n)$ is the solution to

$$2\gamma[2\,n, b(n)] = \Gamma(2\,n) \tag{1.11}$$

An accurate approximation for the solution to $b(n)$ in (1.11) is [14]:

$$b(n) = 2\,n - \frac{1}{3} + \frac{4}{405\,n} + \frac{46}{25515\,n^2} + \frac{131}{1148175\,n^3}$$

Exponential: *ExponentialModel*

The exponential model is a Sérsic model with fixed $n = 1$. It is generally appropriate for describing galaxy disks.

de Vaucouleurs: *DeVaucouleursModel*

The de Vaucouleurs model is a Sérsic model with fixed $n = 4$. It can be a good fit to elliptical galaxies and galaxy bulges.

Regularization

Although minimizing the (modified) weighted sum of least squares gives a solution that fits best the data, it does not necessarily correspond to the most probable solution given what we know about celestial objects. The discrepancy is particularly significant in very faint (SNR ≤ 20) and barely resolved galaxies. For instance, there is a tendency to overestimate the elongation of such galaxies, known as the "noise bias" in the weak-lensing community [15, 16, 17, 18]. The second term in (1.3) implements a simple Tikhonov regularization scheme to mitigate this issue. This penalty term acts as a Gaussian prior on the selected (free or dependent) parameters.

Priors are inserted using the *add_prior()* function. For instance, one can put a Gaussian prior on the flux, centered on the initial isophotal guess from the detection image, with a 10% standard deviation:

```
flux = get_flux_parameter()
add_prior(flux, lambda o: o.get_iso_flux(), lambda o: 0.1 * o.get_iso_flux())
```

Non-Gaussian priors

Sometimes a Gaussian prior is unsuited to some parameters (e.g., to disfavor excessively low or high values, but not both). In that case a change of variable must be applied. This is easily accomplished using a *dependent parameter*. For example, to penalize Sérsic index values n above n_0, one can apply the change of variable:

$$N = e^{n-n_0}, \tag{1.12}$$

which can be implemented as (assuming $n_0 = 4$ and a Gaussian prior on N centered on 0 with unit standard deviation):

```
n = Freeparameter(2.0, Range((0.3, 8.0), RangeType.LINEAR))
n0 = 4
nc = DependentParameter(lambda nt: np.exp(nt - n0), n)
add_prior(nc, 0.0, 1.0)
```

Fig. 1.5 shows the resulting prior on n , and its regularizing effect on the solution.

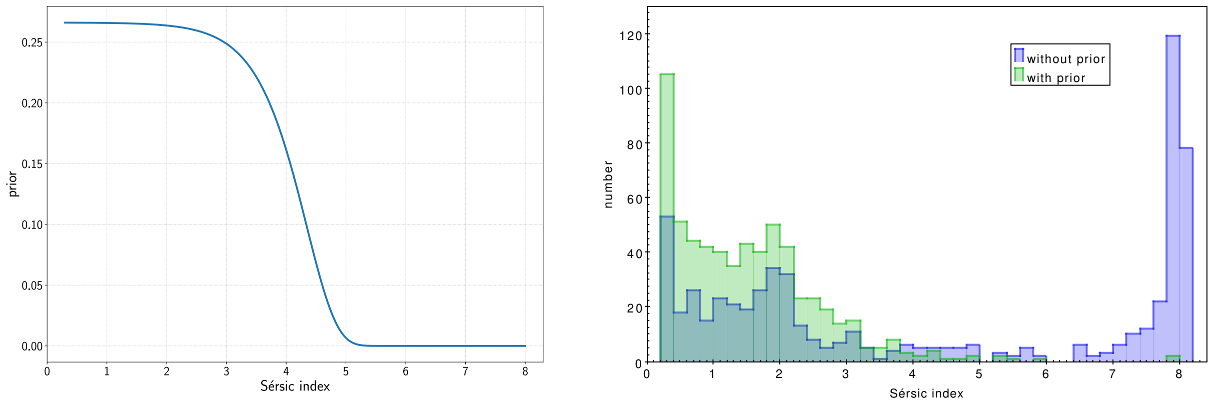

Fig. 1.5: Effect of a non-Gaussian prior generated through a change of variable (see text). *Left:* Effective prior on the Sérsic index. *Right*: Distribution of Sérsic indices obtained for Sérsic fits on noisy data before (blue) and after (green) applying the prior.

Caution: Although any function valid over the parameter range could in principle be used for the change of variable, only functions producing concave priors are likely to play nice with the current minimization algorithm and lead to consistent results.

Combinations of parameters

Priors are also useful for penalizing some values taken by specific combinations of parameters. For example, the following code applied to images taken in the g and r bands sets a Gaussian prior on the color index $g - r$ of the detected point sources (centered on $g - r = 0.5$, with standard deviation 0.3):

```
...
MAG_ZEROPOINT = 32.19
flux_g = get_flux_parameter()
flux_r = get_flux_parameter()
mag_g = DependentParameter(lambda f: -2.5 * np.log10(f) + MAG_ZEROPOINT, flux_g)
mag_r = DependentParameter(lambda f: -2.5 * np.log10(f) + MAG_ZEROPOINT, flux_r)
col = DependentParameter(lambda g,r: g - r, mag_g, mag_r)
add_prior(col, 0.5, 0.3)

group_g = mesgroup['g']
group_r = mesgroup['r']
add_model(group_g, PointSourceModel(x, y, flux_g))
add_model(group_r, PointSourceModel(x, y, flux_r))
add_output_column('mag_g', mag_g)
add_output_column('mag_r', mag_r)
...
```

Fig. 1.6 shows the impact of the prior on the color-magnitude diagram.

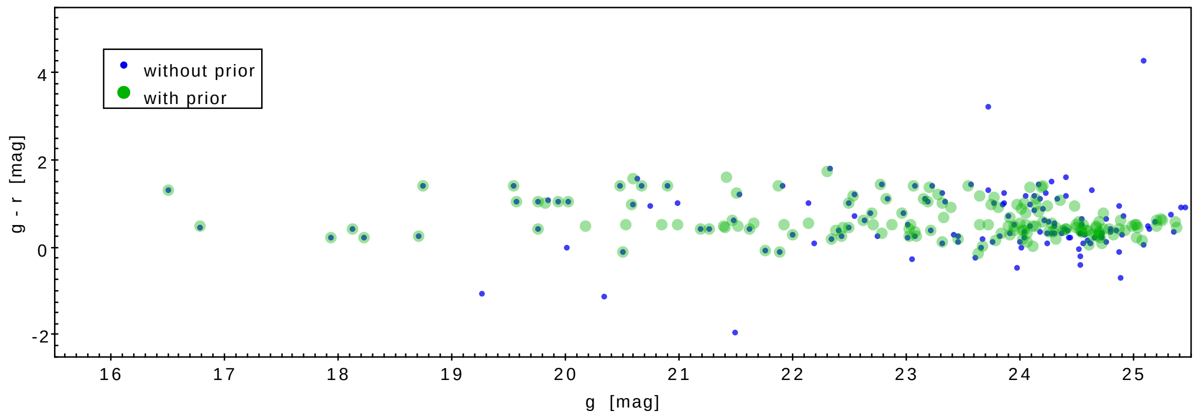

Fig. 1.6: Effect of a Gaussian prior (centered on 0.5 mag with standard deviation 0.3 mag) on $g - r$ source color measurements.

Minimization

Minimization of the loss function $\lambda(\boldsymbol{q})$ is carried out using the Levenberg-Marquardt algorithm. Two Levenberg-Marquardt engines are supported in the current version of **SourceXtractor++**: the **LevMar** library [19], and the **GSL** implementation. **LevMar** approximates the Jacobian matrix of the model from finite differences using Broyden's [20] rank one updates, which results in less evaluations of the model and ≈ 30% faster processing. The default engine is `levmar`, but can be switched to `gsl` if preferred using the *`set_engine()`* function, e.g.:

```
set_engine('gsl')
```

The fit is done inside a disk which diameter is scaled to include the isophotal footprint of the object, plus the FWHM of the PSF, plus a 20 % margin.

1 σ uncertainty estimates are provided for most measurement parameters; they are obtained from the full covariance matrix of the fit, which is itself computed by inverting the approximate Hessian matrix of $\lambda(\boldsymbol{q})$ at the solution.

1.7 Configuration API Reference

1.7.1 `argv`

class Arguments(***kwargs*)

Bases: `object`

This helper class automatically parses the arguments received by the Python script via sys.argv, using the set of key-value parameters received on its constructor to cast the types and define the defaults.

Parameters kwargs – A set of key-values with the defaults, or the types of the expected values. For instance, it is valid to do both:

key1 = 32. key2 = float

For the first case - a value -, if no value is passed via sys.argv, this will be the default. Otherwise, its type (float) will be used to cast the received value.

For the second case - a type -, if no value is passed via sys.argv, the parameter will default to None. Otherwise, the type will be used to cast the received value.

Callables are also accepted. They should accept to be called with no parameters, returning the default, or with a single string, returning the parsed value.

Values can be accessed later as attributes:

instance.key1 and instance.key2

Raises

- **`ValueError`** – If the cast of the parameter failed (i.e. an invalid integer or float format)
- **`KeyError`** – If additional unknown parameters are received

`class` **`FileList`**(*value*)

Bases: `object`

Helper class for receiving a globing pattern as a parameter, defining a list of files - i.e. measurement images or PSFs. It is an iterable, and can be passed directly to load_fits_images.

Parameters **`value`** (*`str`*) – A file globing expression. i.e "*.psf", "band_[r|i|g]_*.fits" or similar. The result is always stored and returned in alphabetical order, so the order between two file lists - i.e frame image and PSF - is consistent and a matching can be done easily between them.

See also:

`glob.glob` Return a list of paths matching a pathname pattern.

1.7.2 `output`

`add_output_column`(*name*, *params*)

Add a new set of columns to the output catalog.

Parameters

- **`name`** (*`str`*) – Name/prefix of the new set of columns
- **`params`** (*`list of columns`*) – List of properties to add to the output with the given name/prefix. They must be subtype of one of the known ones: ParameterBase for model fitting, or Aperture for aperture photometry.

Raises

- **`ValueError`** – If the name has already been used
- **`TypeError`** – If any of the parameters are not of a known type (see params)

See also:

`aperture.add_aperture_photometry`, `model_fitting.ParameterBase`

`print_output_columns`(*file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)

Print a human-readable representation of the configured output columns.

Parameters **`file`** (*`file object`*) – Where to print the representation. Defaults to sys.stderr

1.7.3 `measurement_images`

`class` **`ByKeyword`**(*key*)

Bases: `object`

Callable that can be used to split an ImageGroup by a keyword value (i.e. FILTER).

Parameters **`key`** (*`str`*) – FITS header keyword (i.e. FILTER)

See also:

`ImageGroup.split`

`class` **`ByPattern`**(*key*, *pattern*)

Bases: `object`

Callable that can be used to split an ImageGroup by a keyword value (i.e. FILTER), applying a regular expression and using the first matching group as key.

Parameters

- **`key`** (*`str`*) – FITS header keyword
- **`pattern`** (*`str`*) – Regular expression. The first matching group will be used as grouping key.

See also:

`ImageGroup.split`

class `FitsFile`(**args: Any*, ***kwargs: Any*)
Bases: `_SourceXtractorPy.`

class `ImageGroup`(***kwargs*)
Bases: `object`

Models the grouping of images. Measurement can *not* be made directly on instances of this type. The configuration must be "frozen" before creating a MeasurementGroup

See also:

`MeasurementGroup`

`add_images`(*images*)
Add new images to the group.

Parameters **`images`** (*`list of, or a single,`* `MeasurementImage`) –

Raises **`ValueError`** – If the group has been split, no new images can be added.

`add_subgroup`(*name*, *group*)
Add a subgroup to a group.

Parameters

- **`name`** (*`str`*) – The new of the new group
- **`group`** (`ImageGroup`) –

`is_leaf`()

Returns True if the group is a leaf group

Return type bool

`print`(*prefix=''*, *show_images=False*, *file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)
Print a human-readable representation of the group.

Parameters

- **`prefix`** (*`str`*) – Print each line with this prefix. Used internally for indentation.
- **`show_images`** (*`bool`*) – Show the images belonging to a leaf group.
- **`file`** (*`file object`*) – Where to print the representation. Defaults to sys.stderr

`split`(*grouping_method*)
Splits the group in various subgroups, applying a filter on the contained images. If the group has already been split, applies the split to each subgroup.

Parameters **`grouping_method`** (*`callable`*) – A callable that receives as a parameter the list of contained images, and returns a list of tuples, with the grouping key value, and the list of grouped images belonging to the given key.

See also:

`ByKeyword`, *`ByPattern`*

Raises **`ValueError`** – If some images have not been grouped by the callable.

class MeasurementGroup(*image_group*, *is_subgroup=False*)

Bases: `object`

Once an instance of this class is created from an ImageGroup, its configuration is "frozen". i.e. no new images can be added, or no new grouping applied.

Parameters **image_group** (ImageGroup) –

is_leaf()

Returns True if the group is a leaf group

Return type bool

print(*prefix=''*, *show_images=False*, *file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)

Print a human-readable representation of the group.

Parameters

- **prefix** (*str*) – Print each line with this prefix. Used internally for indentation.
- **show_images** (*bool*) – Show the images belonging to a leaf group.
- **file** (*file object*) – Where to print the representation. Defaults to sys.stderr

class MeasurementImage(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

A MeasurementImage is the processing unit for SourceXtractor++. Measurements and model fitting can be done over one, or many, of them. It models the image, plus its associated weight file, PSF, etc.

Parameters

- **fits_file** (*str or FitsFile object*) – The path to a FITS image, or an instance of FitsFile
- **psf_file** (*str*) – The path to a PSF. It can be either a FITS image, or a PSFEx model.
- **weight_file** (*str or* FitsFile) – The path to a FITS image with the pixel weights, or an instance of FitsFile
- **gain** (*float*) – Image gain. If None, *gain_keyword* will be used instead.
- **gain_keyword** (*str*) – Keyword for the header containing the gain.
- **saturation** (*float*) – Saturation value. If None, *saturation_keyword* will be used instead.
- **saturation_keyword** (*str*) – Keyword for the header containing the saturation value.
- **flux_scale** (*float*) – Flux scaling. Each pixel value will be multiplied by this. If None, *flux_scale_keyword* will be used instead.
- **flux_scale_keyword** (*str*) – Keyword for the header containing the flux scaling.
- **weight_type** (*str*) – The type of the weight image. It must be one of:
  - **none** The image itself is used to compute internally a constant variance (default)
  - **background** The image itself is used to compute internally a variance map
  - **rms** The weight image must contain a weight-map in units of absolute standard deviations (in ADUs per pixel).
  - **variance** The weight image must contain a weight-map in units of relative variance.
  - **weight** The weight image must contain a weight-map in units of relative weights. The data are converted to variance units.
- **weight_absolute** (*bool*) – If False, the weight map will be scaled according to an absolute variance map built from the image itself.

- `weight_scaling` (*float*) – Apply an scaling to the weight map.
- `weight_threshold` (*float*) – Pixels with weights beyond this value are treated just like pixels discarded by the masking process.
- `constant_background` (*float*) – If set a constant background of that value is assumed for the image instead of using automatic detection
- `image_hdu` (*int*) – For multi-extension FITS file specifies the HDU number for the image. Default 0 (primary HDU)
- `psf_hdu` (*int*) – For multi-extension FITS file specifies the HDU number for the psf. Defaults to the same value as image_hdu
- `weight_hdu` (*int*) – For multi-extension FITS file specifies the HDU number for the weight. Defaults to the same value as image_hdu

`load_fits_image`(*image*, *psf=None*, *weight=None*, ***kwargs*)

Creates an image group with the images of a (possibly multi-HDU) single FITS file.

If image is multi-hdu, psf and weight can either be multi hdu or lists of individual files.

In any case, they are matched in order and HDUs not containing images (two dimensional arrays) are ignored.

Parameters

- `image` – The filename of the FITS file containing the image(s)
- `psf` – psf file or list of psf files
- `weight` – FITS file for the weight image or a list of such files

Returns A ImageGroup representing the images

`load_fits_images`(*images*, *psfs=None*, *weights=None*, ***kwargs*)

Creates an image group for the given images.

Parameters

- `images` (*list of str*) –

  A list of relative paths to the images FITS files. Can also be single string in which case, this function acts like load_fits_image
- `psfs` (*list of str*) – A list of relative paths to the PSF FITS files (optional). It must match the length of image_list or be None.
- `weights` (*list of str*) – A list of relative paths to the weight files (optional). It must match the length of image_list or be None.

Returns A ImageGroup representing the images

Return type *ImageGroup*

Raises `ValueError` – In case of mismatched list of files

`print_measurement_images`(*file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)

Print a human-readable representation of the configured measurement images.

Parameters `file` (*file object*) – Where to print the representation. Defaults to sys.stderr

1.7.4 aperture

add_aperture_photometry(*target*, *apertures*)

Flux measurement from the image above the background inside a circular aperture.

Parameters

- **target** (*MeasurementImage object, or leaf MeasurementGroup object with a single image, or a list of either*) – Target images on which to measure the aperture photometry. Leaf MeasurementGroup with a single image are accepted as a convenience.
- **apertures** (*float, or list of float*) – Diameter of the aperture. As different MeasurementImage may not be aligned, nor have equivalent pixel size, the aperture is interpreted as diameter in pixels of a circle on the detection image. A transformation will be applied for each frame, so the covered area is equivalent.

Returns An Aperture object is an internal representation of a property on the measurement frame that contains the apertures. To actually get the measurements on the output catalog, you need to add explicitly them to the output.

Return type list of Aperture objects

See also:

`add_output_column`

Notes

This property will generate five columns with the prefix specified by *add_output_column*: - `_flux` and `_flux_err`, for the flux and its associated error - `_mag` and `_mag_err`, for the magnitude and its associated error - `_flags`, to mark, for instance, saturation, boundary conditions, etc.

For M apertures and N images, the cells on the output column will be an array of MxN fluxes.

Examples

```
>>> measurement_group = MeasurementGroup(load_fits_images(frames, psfs))
>>> all_apertures = []
>>> for img in measurement_group:
>>>     all_apertures.extend(add_aperture_photometry(img, [5, 10, 20]))
>>> add_output_column('aperture', all_apertures)
```

1.7.5 model_fitting

class ConstantModel(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

A model that is constant through all the image.

Parameters **value** (`ParameterBase` *or float*) – Value to add to the value of all pixels from the model.

to_string(*show_params=False*)

Return a human readable representation of the model.

Parameters **show_params** (*bool*) – If True, include information about the parameters.

Returns

Return type str

class ConstantParameter(**args: Any*, ***kwargs: Any*)
Bases: `_SourceXtractorPy.`

A parameter with a single value that remains constant. It will not be fitted.

Parameters **value** (*float, or callable that receives a source and returns a float*) – Value for this parameter

get_value()

Returns Receives a source and returns a value for the parameter

Return type callable

class CoordinateModelBase(**args: Any*, ***kwargs: Any*)
Bases: `_SourceXtractorPy.`

Base class for positioned models with a flux. It can not be used directly.

Parameters

- **x_coord** (*ParameterBase or float*) – X coordinate (in the detection image)
- **y_coord** (*ParameterBase or float*) – Y coordinate (in the detection image)
- **flux** (*ParameterBase or float*) – Total flux

class DeVaucouleursModel(**args: Any*, ***kwargs: Any*)
Bases: `_SourceXtractorPy.`

Model a source with a De Vaucouleurs profile (Sersic model with an index of 4)

Parameters

- **x_coord** (*ParameterBase or float*) – X coordinate (in the detection image)
- **y_coord** (*ParameterBase or float*) – Y coordinate (in the detection image)
- **flux** (*ParameterBase or float*) – Total flux
- **effective_radius** (*ParameterBase or float*) – Ellipse semi-major axis, in pixels on the detection image.
- **aspect_ratio** (*ParameterBase or float*) – Ellipse ratio.
- **angle** (*ParameterBase or float*) – Ellipse rotation, in radians

to_string(*show_params=False*)
Return a human readable representation of the model.

Parameters **show_params** (*bool*) – If True, include information about the parameters.

Returns

Return type str

class DependentParameter(**args: Any*, ***kwargs: Any*)
Bases: `_SourceXtractorPy.`

A DependentParameter is not fitted by itself, but its value is derived from another Parameters, whatever their type: FreeParameter, ConstantParameter, or other DependentParameter

Parameters

- **func** (*callable*) – A callable that will be called with all the parameters specified in this constructor each time a new evaluation is needed.
- **params** (*list of ParameterBase*) – List of parameters on which this DependentParameter depends.

Examples

```
>>> flux = get_flux_parameter()
>>> mag = DependentParameter(lambda f: -2.5 * np.log10(f) + args.mag_zeropoint,␣
↪flux)
>>> add_output_column('mf_mag_' + band, mag)
```

class ExponentialModel(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Model a source with an exponential profile (Sersic model with an index of 1)

Parameters

- **x_coord** (ParameterBase *or float*) – X coordinate (in the detection image)
- **y_coord** (ParameterBase *or float*) – Y coordinate (in the detection image)
- **flux** (ParameterBase *or float*) – Total flux
- **effective_radius** (ParameterBase *or float*) – Ellipse semi-major axis, in pixels on the detection image.
- **aspect_ratio** (ParameterBase *or float*) – Ellipse ratio.
- **angle** (ParameterBase *or float*) – Ellipse rotation, in radians

to_string(*show_params=False*)

Return a human readable representation of the model.

Parameters **show_params** (*bool*) – If True, include information about the parameters.

Returns

Return type str

class FluxParameterType

Bases: `object`

Possible flux types to use as initial value for the flux parameter. Right now, only isophotal is supported.

class FreeParameter(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

A parameter that will be fitted by the model fitting engine.

Parameters

- **init_value** (*float or callable that receives a source, and returns a float*) – Initial value for the parameter.
- **range** (*instance of Range or* Unbounded) – Defines if this parameter is unbounded or bounded, and how.

See also:

Unbounded, *Range*

Examples

```
>>> sersic = FreeParameter(2.0, Range((1.0, 7.0), RangeType.LINEAR))
```

`get_init_value()`

Returns Receives a source, and returns an initial value for the parameter.

Return type callable

`get_range()`

Returns

Return type *Unbounded* or *Range*

class ModelBase(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Base class for all models.

class ParameterBase(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Base class for all model fitting parameter types. Can not be used directly.

class PointSourceModel(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Models a source as a point, spread by the PSF.

Parameters

- **x_coord** (`ParameterBase` *or* `float`) – X coordinate (in the detection image)
- **y_coord** (`ParameterBase` *or* `float`) – Y coordinate (in the detection image)
- **flux** (`ParameterBase` *or* `float`) – Total flux

to_string(*show_params=False*)

Return a human readable representation of the model.

Parameters **show_params** (*bool*) – If True, include information about the parameters.

Returns

Return type str

class Prior(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Model a Gaussian prior on a given parameter.

Parameters

- **param** (`ParameterBase`) – Model fitting parameter
- **value** (*float or callable that receives a source and returns a float*) – Mean of the Gaussian
- **sigma** (*float or callable that receives a source and returns a float*) – Standard deviation of the Gaussian

class Range(*limits*, *type*)

Bases: `object`

Limit, and normalize, the range of values for a model fitting parameter.

Parameters

- **`limits`** (*a tuple (min, max), or a callable that receives a source, and returns a tuple (min, max)*) –
- **`type`** (`RangeType`) –

Notes

RangeType.LINEAR Normalized to engine space using a sigmoid function

$$engine = \ln \frac{world - min}{max - world}$$
$$world = min + \frac{max - min}{1 + e^{engine}}$$

RangeType.EXPONENTIAL Normalized to engine space using an exponential sigmoid function

$$engine = \ln \left(\frac{\ln(world/min)}{\ln(max/world)} \right)$$
$$world = min * e^{\frac{\ln(max/min)}{(1+e^{-engine})}}$$

`get_limits()`

Returns Receives a source, and returns a tuple (min, max)

Return type callable

`get_type()`

Returns

Return type *RangeType*

`class` **`RangeType`**(*value*)

Bases: `enum.Enum`

An enumeration.

`class` **`SersicModel`**(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Model a source with a Sersic profile.

Parameters

- **`x_coord`** (`ParameterBase or float`) – X coordinate (in the detection image)
- **`y_coord`** (`ParameterBase or float`) – Y coordinate (in the detection image)
- **`flux`** (`ParameterBase or float`) – Total flux
- **`effective_radius`** (`ParameterBase or float`) – Ellipse semi-major axis, in pixels on the detection image.
- **`aspect_ratio`** (`ParameterBase or float`) – Ellipse ratio.
- **`angle`** (`ParameterBase or float`) – Ellipse rotation, in radians
- **`n`** (`ParameterBase or float`) – Sersic index

`to_string`(*show_params=False*)

Return a human readable representation of the model.

Parameters **`show_params`** (`bool`) – If True, include information about the parameters.

Returns

Return type str

`class` **`SersicModelBase`**(**args: Any*, ***kwargs: Any*)

Bases: `_SourceXtractorPy.`

Base class for the Sersic, Exponential and de Vaucouleurs models. It can not be used directly.

Parameters

- **`x_coord`** (`ParameterBase` *or* `float`) – X coordinate (in the detection image)
- **`y_coord`** (`ParameterBase` *or* `float`) – Y coordinate (in the detection image)
- **`flux`** (`ParameterBase` *or* `float`) – Total flux
- **`effective_radius`** (`ParameterBase` *or* `float`) – Ellipse semi-major axis, in pixels on the detection image.
- **`aspect_ratio`** (`ParameterBase` *or* `float`) – Ellipse ratio.
- **`angle`** (`ParameterBase` *or* `float`) – Ellipse rotation, in radians

`class` **`Unbounded`**(*normalization_factor=1*)

Bases: `object`

Unbounded, but normalize, value of a model fitting parameter

Parameters **`normalization_factor`** (*`float, or a callable that receives the initial value parameter value and a source,`*) – and returns a float The world value which will be normalized to 1 in engine coordinates

`get_normalization_factor`()

Returns Receives the initial parameter value and a source, and returns the world value which will be normalized to 1 in engine coordinates

Return type callable

`class` **`WorldCoordinate`**(*ra*, *dec*)

Bases: `object`

Coordinates in right ascension and declination

Parameters

- **`ra`** (*`float`*) – Right ascension
- **`dec`** (*`float`*) – Declination

`add_model`(*group*, *model*)

Add a model to be fitted to the given group.

Parameters

- **`group`** (`MeasurementGroup`) –
- **`model`** (`ModelBase`) –

`add_prior`(*param*, *value*, *sigma*)

Add a prior to the given parameter.

Parameters

- **`param`** (`ParameterBase`) –
- **`value`** (*`float or callable that receives a source and returns a float`*) – Mean of the Gaussian
- **`sigma`** (*`float or callable that receives a source and returns a float`*) – Standard deviation of the Gaussian

`get_flux_parameter`(*type=1*, *scale=1*)
Convenience function for the flux parameter.

Parameters

- **type** (*int*) – One of the values defined in FluxParameterType
- **scale** (*float*) – Scaling of the initial flux. Defaults to 1.

Returns **flux** – Flux parameter, starting at the flux defined by *type*, and limited to +/- 1e3 times the initial value.

Return type *FreeParameter*

`get_pos_parameters`()
Convenience function for the position parameter X and Y.

Returns

- **x** (*FreeParameter*) – X coordinate, starting at the X coordinate of the centroid and linearly limited to X +/- the object radius.
- **y** (*FreeParameter*) – Y coordinate, starting at the Y coordinate of the centroid and linearly limited to Y +/- the object radius.

Notes

X and Y are fitted on the detection image X and Y coordinates. Internally, these are translated to measurement images using the WCS headers.

`get_position_angle`(*x1*, *y1*, *x2*, *y2*)
Get the position angle in sky coordinates for two points defined in pixels on the detection image.

Parameters

- **x1** –
- **y1** –
- **x2** –
- **y2** –

Returns

Return type Position angle in degrees, normalized to -/+ 90

`get_separation_angle`(*x1*, *y1*, *x2*, *y2*)
Get the separation angle in sky coordinates for two points defined in pixels on the detection image.

Parameters

- **x1** (*float*) –
- **y1** (*float*) –
- **x2** (*float*) –
- **y2** (*float*) –

Returns

Return type Separation in degrees

`get_sky_coord`(*x*, *y*)
Transform an (X, Y) in pixel coordinates on the detection image to astropy SkyCoord.

Parameters

- **x** (*float*) –
- **y** (*float*) –

Returns

Return type SkyCoord

`get_world_parameters`(*x*, *y*, *radius*, *angle*, *ratio*)

Convenience function for generating five dependent parameters, in world coordinates, for the position and shape of a model.

Parameters

- **x** (`ParameterBase`) –
- **y** (`ParameterBase`) –
- **radius** (`ParameterBase`) –
- **angle** (`ParameterBase`) –
- **ratio** (`ParameterBase`) –

Returns

- **ra** (*DependentParameter*) – Right ascension
- **dec** (*DependentParameter*) – Declination
- **rad** (*DependentParameter*) – Radius as degrees
- **angle** (*DependentParameter*) – Angle in degrees
- **ratio** (*DependentParameter*) – Aspect ratio. It has to be recomputed as the axis of the ellipse may have different ratios in image coordinates than in world coordinates

Examples

```
>>> flux = get_flux_parameter()
>>> x, y = get_pos_parameters()
>>> radius = FreeParameter(lambda o: o.get_radius(), Range(lambda v, o: (.01 * v,
↪ 100 * v), RangeType.EXPONENTIAL))
>>> angle = FreeParameter(lambda o: o.get_angle(), Range((-np.pi, np.pi),␣
↪RangeType.LINEAR))
>>> ratio = FreeParameter(1, Range((0, 10), RangeType.LINEAR))
>>> add_model(group, ExponentialModel(x, y, flux, radius, ratio, angle))
>>> ra, dec, wc_rad, wc_angle, wc_ratio = get_world_parameters(x, y, radius,␣
↪angle, ratio)
>>> add_output_column('mf_world_angle', wc_angle)
```

`get_world_position_parameters`(*x*, *y*)

Convenience function for generating two dependent parameter with world (alpha, delta) coordinates from image (X, Y) coordinates.

Parameters

- **x** (`ParameterBase`) –
- **y** (`ParameterBase`) –

Returns

- **ra** (*DependentParameter*)
- **dec** (*DependentParameter*)

See also:

`get_pos_parameters`

Examples

```
>>> x, y = get_pos_parameters()
>>> ra, dec = get_world_position_parameters(x, y)
>>> add_output_column('mf_ra', ra)
>>> add_output_column('mf_dec', dec)
```

`pixel_to_world_coordinate`(*x*, *y*)
Transform an (X, Y) in pixel coordinates on the detection image to (RA, DEC) in world coordinates. :param x: :type x: float :param y: :type y: float

Returns

Return type *WorldCoordinate*

`print_model_fitting_info`(*group*, *show_params=False*, *prefix=''*, *file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)
Print a human-readable representation of the configured models.

Parameters

- **group** (`MeasurementGroup`) – Print the models for this group.
- **show_params** (`bool`) – If True, print also the parameters that belong to the model
- **prefix** (`str`) – Prefix each line with this string. Used internally for indentation.
- **file** (`file object`) – Where to print the representation. Defaults to sys.stderr

`print_parameters`(*file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)
Print a human-readable representation of the configured model fitting parameters.

Parameters **file** (`file object`) – Where to print the representation. Defaults to sys.stderr

`radius_to_wc_angle`(*x*, *y*, *rad*)
Transform a radius in pixels on the detection image to a radius in sky coordinates.

Parameters

- **x** (`float`) –
- **y** (`float`) –
- **rad** (`float`) –

Returns

Return type Radius in degrees

`set_coordinate_system`(*cs*)
Set the global coordinate system. This function is used internally by SourceXtractor++.

`set_engine`(*engine*)

Parameters **engine** (`str`) – Minimization engine for the model fitting : levmar or gsl

`set_max_iterations`(*iterations*)

Parameters **iterations** (`int`) – Max number of iterations for the model fitting.

`set_modified_chi_squared_scale`(*scale*)

Parameters **scale** (`float`) – Sets u0, as used by the modified chi squared residual comparator, a function that reduces the effect of large deviations. Refer to the SourceXtractor++ documentation for a better explanation of how residuals are computed and how this value affects the model fitting.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[1] E. Bertin and S. Arnouts, 1996. SExtractor: Software for source extraction. *A&AS*, 117:393–404.

[2] D. C. Wells, E. W. Greisen, and R. H. Harten, 1981. FITS - a Flexible Image Transport System. *A&AS*, 44:363.

[3] E. W. Greisen and M. R. Calabretta, 2002. Representations of world coordinates in FITS. *A&A*, 395:1061–1075.

[4] M. R. Calabretta and E. W. Greisen, 2002. Representations of celestial coordinates in FITS. *A&A*, 395:1077–1122.

[5] P. B. Stetson, 1987. DAOPHOT - A computer program for crowded-field stellar photometry. *PASP*, 99:191–222.

[6] G. S. Da Costa, 1992. Basic Photometry Techniques. In S. B. Howell, editor, *Astronomical CCD Observing and Reduction Techniques*, volume 23 of Astronomical Society of the Pacific Conference Series, 90. 1992.

[7] A. Stuart and K. Ord. *Kendall's Advanced Theory of Statistics: Volume 1: Distribution Theory*. Number vol. 1 ;vol. 1994 in Kendall's Advanced Theory of Statistics. Wiley, 2009. ISBN 9780340614303. URL: https://books.google.fr/books?id=tW18thQWJQIC.

[8] J. F. Jarvis and J. A. Tyson, 1981. FOCAS - Faint Object Classification and Analysis System. *AJ*, 86:476–495.

[9] L. Infante, 1987. A faint object processing software - Description and testing. *A&A*, 183:177–184.

[10] R. G. Kron, 1980. Photometry of a complete sample of faint galaxies. *ApJS*, 43:305–325.

[11] C. E. Duchon, 1979. Lanczos Filtering in One and Two Dimensions. *Journal of Applied Meteorology*, 18:1016–1022.

[12] C. Kanzow, N. Yamashita, and M. Fukushima, 2004. Levenberg-Marquardt methods with strong local convergence properties for solving nonlinear equations with convex constraints. *Journal of Computational and Applied Mathematics*, 172(2):375–397.

[13] J. L. Sérsic. *Atlas de Galaxias Australes*. Cordoba, Argentina: Observatorio Astronomico, 1968, 1968.

[14] L. Ciotti and G. Bertin, 1999. Analytical properties of the R^(1/m) law. *A&A*, 352:447–451.

[15] C. M. Hirata, R. Mandelbaum, U. Seljak, J. Guzik, N. Padmanabhan, C. Blake, J. Brinkmann, T. Budávari, A. Connolly, I. Csabai, R. Scranton, and A. S. Szalay, 2004. Galaxy-galaxy weak lensing in the Sloan Digital Sky Survey: intrinsic alignments and shear calibration errors. *MNRAS*, 353:529–549.

[16] P. Melchior and M. Viola, 2012. Means of confusion: how pixel noise affects shear estimates for weak gravitational lensing. *MNRAS*, 424:2757–2769.

[17] A. Refregier, T. Kacprzak, A. Amara, S. Bridle, and B. Rowe, 2012. Noise bias in weak lensing shape measurements. *MNRAS*, 425:1951–1957.

[18] T. Kacprzak, J. Zuntz, B. Rowe, S. Bridle, A. Refregier, A. Amara, L. Voigt, and M. Hirsch, 2012. Measurement and calibration of noise bias in weak lensing galaxy shape estimation. *MNRAS*, 427:2711–2722.

[19] M.I.A. Lourakis, 2004. Levmar: levenberg-marquardt nonlinear least squares algorithms in C/C++.

[20] C. Broyden, 1965. *Mathematics of Computation*, 19:577–593.

PYTHON MODULE INDEX

INDEX