Balrog

An astronomical imaging simulation d(a)emon[1]

for those who dig too deeply

and too greedily into

their data...

Documentation written by Eric Suchyta

[1]Technically, the package is not a daemon, but I will invoke poetic license.

Contents

1. Introduction

Balrog is a package of Python simulation code for use with real astronomical imaging data. Its scheme is almost embarrassingly simple: objects are simulated into a survey's images, and measurement software is run over the simulated objects' images. We know what we put in, and Balrog allows us to derive the mapping between what we actually measure and the input truth. Conceptually, the story ends here. The rest is technical development to implement the scheme and statistical inference to perform science analyses.

Balrog's implementation wraps thoroughly tested codes, widely used throughout the Astronomy community. All object simulations are performed by GalSim (Rowe et al., 2014). Source extraction and measurement is executed by SExtractor (Bertin & Arnouts, 1996). Balrog facilitates the ease of running these codes en masse over many images, automating useful GalSim and SExtractor functionality, as well as filling in many bookkeeping steps along the way.

Since different users will have different needs, Balrog strives to be as flexible as possible. It includes a well defined framework capable of implementing an arbitrarily wide variety of parameter possibilities. The framework allows users to define their own arguments and functions to plug into Balrog when generating simulated object properties.

Balrog has been written making our best attempts at user-friendliness. To preserve an intuitive feel, its configuration framework mimics ordinary Python syntax. Furthermore, example files have been packaged with the code so that following installation, the pipeline is able to run out of the box without specifying any arguments. Users are encouraged to use and inspect the default example runs to become more familiar with using Balrog, then modify the examples to start creating their own runs. Balrog attempts to make debugging as painless as possible when users are developing their setup. Numerous errors and warnings are handled, printing informative messages when exception are raised. Several log files are automatically written.

In this brief introduction, we have merely scratched the surface explaining Balrog's capabilities. The remainder of the documentation elaborates further. Section 2 lays out the pipeline's workflow in full. Beyond Section 2, we begin to treat usage. Section 3 discusses what is necessary for installation. Section 4 presents an approach to hit the ground running and quickly get started with some key features of Balrog. The following sections build upon Section 4, offering more comprehensive Balrog usage details. Section 5 covers Balrog's built-in command line arguments. Section 6 treats the file we refer to as Balrog's `pyconfig` file, which is responsible for a number of customizations, including user-defined command line arguments and how the simulated objects' properties are generated. The format of Balrog's outputs are explained in Section 7. In general, the separate sections have been written to read fairly independently, so the entire document does not necessarily need to be read sequentially for the sections to make sense. Our approach is to build up the documentation conceptually and break it into manageably sized divisions. One thought to keep in mind is that depending on the user's intended application for Balrog, configuring Balrog runs can range being from very simple to being fairly involved. The documentation does its best to adequately explain the intricacies when necessary, but not at the cost of clarity for the simpler concepts.

2. Balrog's Workflow

Section 1 briefly introduced a high-level overview of Balrog. The purpose of this section is to characterize the workflow in full. The focus here is methodology, *not* usage instructions. The text is organized as follows. To begin, the workflow of the pipeline as a whole is described in Section 2.1. This is then subdivided into three topics to be discussed further. Section 2.2 explains Balrog's required input data. Section 2.3 describes simulating objects, offering a brief conceptual overview of how simulated properties can be generated, as well as comments about Balrog's GalSim usage. The functionality of Balrog's SExtractor implementation is considered in Section 2.4.

2.1 Overview

Figure 2.1 is a high-level flowchart of the Balrog pipeline, which offers a compact summary of text throughout this document. Green parallelograms are inputs and teal parallelograms are outputs. Rectangles are processes, with the orange rectangle calling GalSim and the two pink rectangles calling SExtractor. The single gray rectangle is pure Balrog code, without any calls to GalSim or SExtractor. Within Figure 2.1, we have included hyperlinks pointing to sections describing the selected flowchart item.

The Balrog pipeline begins by opening a log file and parsing the command line parameters. Next, Balrog parses how the user wants to generate simulated object properties. Section 2.3 fully prescribes the attributes of these objects. To summarize, each is composed of one or more Sérsic profiles. The user's directives are executed, and a truth catalog of the object parameters is written.

Balrog enforces a number of rules on the user's simulation configurations. If any errors or warnings occur they are directed to the log file. Errors are raised when users make a syntax error and Balrog cannot continue. Generally, warnings occur when something is missing, but the code is able to continue using an internal default. Warnings likely, but not always indicate something is not quite right. The log file remains open, recording messages through the full pipeline.

Balrog continues by reading in the imaging data into which the simulated objects will be drawn. Refer to Section 2.2 to define what is required of the imaging data in this context. By default, SExtractor is run over the input image. Please note, at this point no simulated objects have been inserted. However, this SExtractor call is potentially useful for reasons discussed further in Section 2.4. An image of each object, commonly known as a postage stamp, is generated and then added atop the input image. Postage stamp drawing is treated in Section 2.3. Once objects have been added to the image, SExtractor is called again. Details of SExtractor's implementation are covered in Section 2.4. SExtractor outputs a catalog of the simulated objects' measurements.

Technically, what has been described in this section is the default behavior for Balrog. However, the default configures a *full* Balrog *run*, meaning all the components of the pipeline execute. Balrog can be configured to turn off certain functionality with various command line arguments, described in Section 5. For example, Balrog can be told to only insert the simulated galaxies without actually running SExtractor or not to draw anything at all. One final convenience we have not mentioned is that Balrog can optionally operate over only a subsample of an image rather than the full area. The only benefit of this is that file sizes are smaller and SExtractor runs faster.

2.2 Input Data

Balrog reads in an astronomical image of pixelated flux values. Associated with the flux map is a required weight map, typically measuring inverse variance. The weight map is used by SExtractor when making its measurements. Both the flux and weight images must be FITS files; Balrog does not recognize any other

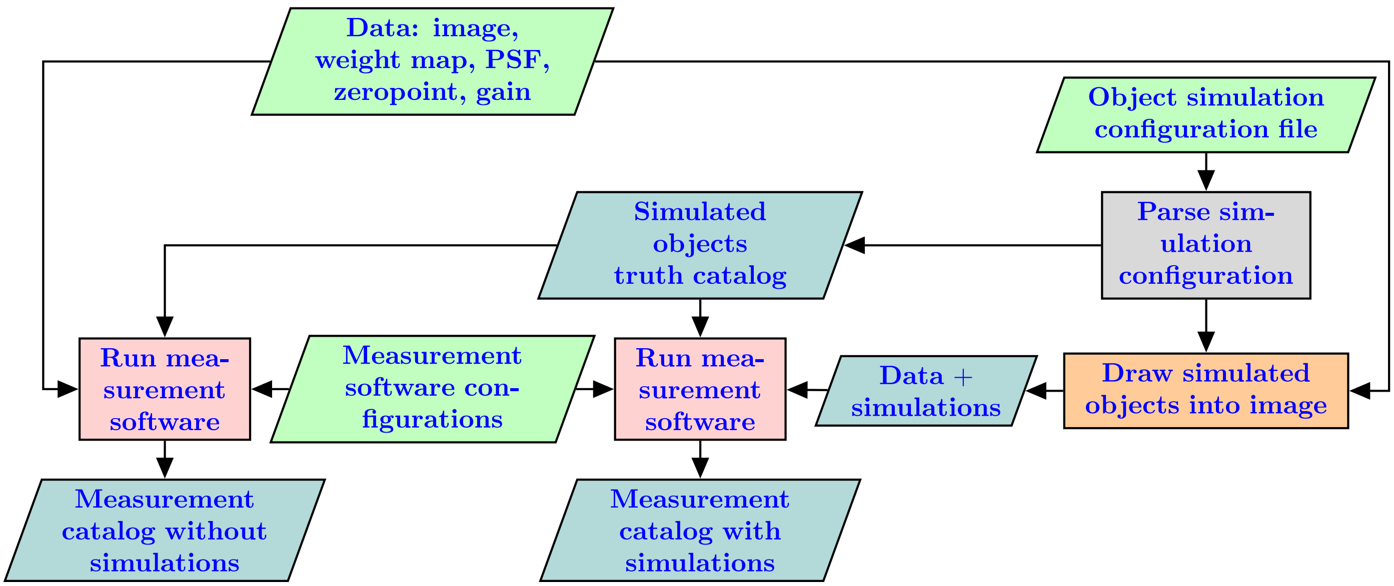

Figure 2.1: High-level overview of Balrog. Green parallelograms are inputs to Balrog and teal parallelograms are outputs. Rectangles are processes Balrog runs: orange by GalSim, pink by SExtractor, and gray by Balrog code itself. Blue text is hyperlinked to further details throughout the documentation.

Table 2.1: Brief summary of imaging inputs to Balrog.

Input	**Usage**
Image	Draw simulated objects into this image
Weight map	Used by SExtractor in measurements; Balrog does not change the weight map
PSF	Convolved with object to transform intrinsic brightness profile to observed profile
Zeropoint	Converts simulated magnitude values to image flux levels
Gain	Set the noise level in the simulated object's postage stamp

formats.

Also required with each image is a model of the image's point spread function (PSF). At a minimum, the PSF model is necessary for convolution with the simulated objects prior to embedding the objects in the given image. It is also required if attempting to fit deconvolved models of object profiles during the SExtractor measurement process. The only file type Balrog supports for input PSF models is that generated by PSFEx (Bertin, 2011). Readers should be aware that Balrog itself does not run PSFEx. Thus, generating PSFEx models constitutes a prerequisite users must complete prior to running Balrog.

The data should include a photometric calibration defining what one ADU flux count means physically. Object simulations are done in apparent magnitude space, and Balrog needs a zeropoint (m_z) to convert this to an image ADU level. The conversion is defined in the usual way, where the object's flux (F) and apparent magnitude (m) are related by: $m - m_z = -2.5 \log_{10} F$. A default zeropoint of 30 is assumed by Balrog if one is not otherwise set by the user.

Balrog uses GalSim functionality to add Poisson noise to the flux values in the simulated objects' postage stamps. The level of this noise is set by the effective electron/ADU gain level, with zero read noise. Higher gain decreases the noise level, with gain scaling like the square root of time. If the user does not set a gain, Balrog defaults it to unity.

Balrog simulations incorporate GalSim's WCS features, and hence the software requires each image contain a WCS solution. Balrog reads this solution from the image's header. If no WCS exists in the header, the pipeline enforces a fiducial WCS with a constant pixel scale of 0.263 arcsec/pixel.[1]

2.3 Simulating Objects

To simulate objects, Balrog makes use of GalSim. Throughout this section, we will make reference to various GalSim functionality. Because the GalSim team has written thorough documentation into their code repository, as well as a paper (Rowe et al., 2014), we will not go to great length to describe GalSim's underlying implementations. Please refer to their documentation or paper where necessary.

Balrog calls GalSim's class for implementing Sérsic objects to model surface brightness distributions as Sérsic profiles. By effectively adding together these Sérsic objects, Balrog allows objects to be composed of as many superimposed Sérsic components as desired. Each of these components has its own Sérsic index (n), half light radius (r_e), flux (F), minor to major axis ratio (b/a), and orientation angle (β). The half light radius is measured along the major axis, and the orientation angle is measured as the major axis' counter-clockwise rotation away from the $\hat{x}$-direction. Flux values are generated as magnitudes, then converted to ADU levels using the image's zeropoint: $m - m_z = -2.5 \log_{10} F$. In addition to its Sérsic components, each object shares five parameters which are identical among each Sérsic component: two centroid coordinate positions (x, y), two components of reduced shear (g_1, g_2), and magnification. The reduced shear follows the usual lensing notation convention, with positive g_1 along the $\hat{x}$-axis, negative g_1 along the $\hat{y}$-axis, and positive and negative g_2 rotated 45° from the respective g_1 counterparts. Magnification is the usual $\mu = 1 + \kappa$. To be explicitly clear, the shear and magnification are lensing effects applied to an object, while the others are intrinsic quantities, as they would be in the absence of lensing, prior to convolution with the PSF. When $\mu > 1$ is applied, the simulated objects' images appear bigger and brighter. However, the flux and half light radius reported in the Balrog truth catalog do not increase as μ increases. They are pre-lensing quantities. This is merely a choice on our part. At any rate, transformation of the truth quantities through magnification is simple: $\log F \to \log F + \kappa$, $\log r_e \to \log r_e + 2\kappa$.

Balrog presents users with a number of different options for how to generate the truth properties of their objects. Section 6.2 documents the full range of possibilities, including examples of how to implement them. Here, we briefly overview the functionality. Simple assignment types include a constant to be applied commonly to every object or an array containing an element for each object. Alternatively, values can be sampled from the columns in a catalog file. Multiple columns from the same table are automatically jointly sampled. Balrog ships with a version of the COSMOS mock catalog (CMC) (Jouvel et al., 2009), in which we have added a column for Sérsic index based on its early/late type. The CMC reaches faint depths, and can be used to build realistic input populations

[1] 0.263 arcsec/pixel is the fiducial pixel scale for DECam.

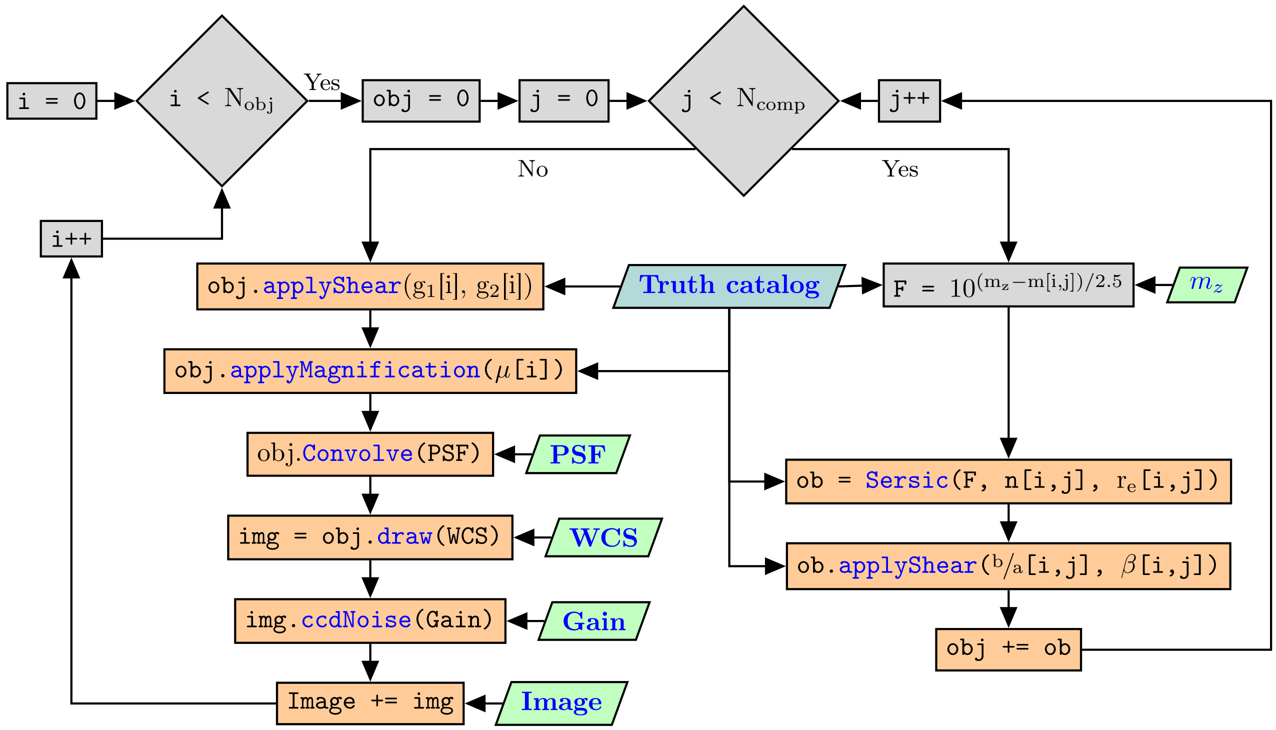

Figure 2.2: Flowchart of Balrog's use of GalSim to simulate objects. The colors and shapes are consistent with Figure 2.1: green parallelograms are Balrog inputs, the teal parallelogram is a Balrog output, orange rectangles call GalSim commands, and gray nodes are Python code in Balrog. Diamonds are decision points. Blue text hyperlinks to further documentation. This figure is effectively a "zoom in" of the "Draw simulated objects into image" node of Figure 2.1.

for shallower surveys. Last but certainly not least, users can define their own functions which determine the truth parameters of the simulated objects. This allows for arbitrary complexity to the simulations, giving users the flexibility to implement whatever they can code themselves in Python. Conveniently, the functions may operate over the objects' truth parameters themselves, allowing one parameter to be defined in terms of another.

Suchyta: Maybe we should include the version of the catalog with Claire's fits instead. I just haven't done this, but we could. Otherwise, Huff needs to add what he did to get Sérsic index.

Figure 2.2 is a flowchart of how Balrog uses GalSim to build simulated objects. The color coding of the nodes is the same as Figure 2.1: green parallelograms are Balrog inputs and the teal parallelogram is a Balrog output. Commands are inside rectangles, where the orange ones call GalSim and the gray ones do not. Diamonds are decision points. Blue text is hyperlinked to GalSim documentation or other sections of this document. The text lines in the orange nodes are not necessary valid GalSim commands as written, but meant to be indicative of the actual commands, presented here in a compact form.

The simulation process runs in a loop with a number of iterations equal to the number of objects to be simulated. A nested loop then iterates over the number of Sérsic profiles the objects are composed of. For each profile, a GalSim `Sersic` object is initialized with the Sérsic index and flux which were generated in the truth catalog. Initially these objects are circularly symmetric, then GalSim's `applyShear` method is called to stretch it to the given axis ratio and orient it in the given direction. The GalSim `Sersic` objects were originally initialized with half light radii which were multiplied by a factor of $\sqrt{a/b}$ compared to the truth catalog's half light radii, meaning the `applyShear` command has now stretched the originally circular objects into ellipses with half light radii along their major axes equal to truth catalog values.

Next, each object's stack of Sérsic components is summed to create the object's composite profile. Lensing is applied to this combined object, calling `applyShear` and `applyMagnfication` with the truth catalog's shear and magnification. The PSF model is opened as a `DES_PSFEx` object, with an implementation that operates in World Coordinates. The PSF is sampled at the object's coordinates, and GalSim's `Convolve` method applies the convolution. PSFEx's models include the contribution from the pixel response function, so no `Pixel` call is made in GalSim. The PSF models PSFEx generates are not guaranteed to be centered within a postage stamp, so Balrog applies a shift to center the model at the calculated centroid. After convolution, the `draw` method draws each object into a postage stamp image. Poisson noise is added to the object's postage stamp with the `CCDNoise` function, setting the gain equal to the input value given and the read noise to zero. The final step in the simulation process is to add the postage stamp to input image, centering the postage stamp onto the object's centroid coordinates by making uses of GalSim's `bounds` attributes for both images.

Within GalSim, the target accuracies for simulated Sérsic objects and convolutions is controlled with the `GSParams` argument. This specifies quantities like the maximum permissible FFT size and the fraction of flux allowed to be lost outside the postage stamp. Balrog allows users to adjust the `GSParams`, as described in Section 6.3. If an object fails to draw (e.g. too large of an FFT would be needed), Balrog will not exit, but will note in the truth catalog that it was not drawn.

2.4 SExtractor Implementation

Balrog uses SExtractor for object detection and measurement. Throughout the Balrog documentation, we will reference numerous SExtractor features, some of which are better documented than others. We will not attempt to redocument SExtractor, but will make efforts to fully clarify features relevant to Balrog. Please refer to the SExtractor user manual or the so-called Source Extractor for Dummies for SExtractor help. We have also written a brief SExtractor appendix.

Balrog's default behavior is to run SExtractor in *association mode*. Association mode means SExtractor will only make possible measurements at a predefined list of coordinates. Here, the list is given as the simulated objects' locations, meaning SExtractor is effectively aware of where in the image the simulated objects live. If SExtractor finds an object at one of the these coordinates, it extracts it; but it does not extract objects anywhere else. The attractiveness of running in association mode is the reduction in execution time it offers. It selects against the objects unrelated to the simulated objects that existed in the image prior to any simulation. By default, Balrog configures SExtractor to make a Sérsic fit to each detected object. The model fitting is a time expensive step, so association mode offers a significant decrease to SExtractor's run time when using this configuration. Balrog allows users to turn off association mode or model fitting, if desired. For a large dataset, running numerous Balrog realizations over the images with association mode disabled and model fitting enabled is likely time unfeasible.

In addition to running SExtractor over the image with the simulated objects, by default Balrog is configured to run SExtractor over the image prior to inserting the simulated objects. This is to confirm if the association mode functionality of SExtractor is genuinely measuring a simulated object. For example, Balrog may happen to place an object into the image at a location where a big, bright object already lives. If the simulated object is rather faint, SExtractor may just interpret its flux as part of the original object, having no idea the simulated object is even there. The SExtractor run over the pre-simulation image allows users to check for such occurrences if they would like. By default, this run does not include model fitting. For the case described here, there is no benefit to dedicate the additional time needed for model fitting.

Though it is not the default behavior, Balrog can tell SExtractor to run in *dual image mode*, meaning detection occurs in one image, and measurement in a different image. This is common in surveys; building a multi-band detection image increases the depth of detections. Dual image mode Balrog slightly differs from single image mode. Simulated objects are only drawn into the measurement image, not the detection image. Likely, wrappers will be necessary for dual image Balrog to be useful. For example, in the case of a multi-band coadd, one could run single image mode over each band contributing to the detection image, then coadd the set of resultant output Balrog images to make a "Balrog detection image". This could then be given as the detection image to a dual image Balrog run over any of the single band images, where the run has been configured to generate simulated objects in the same positions as before.

The SExtractor configuration settings are an important component of a SExtractor run. Users are free to give Balrog their own configuration files, and Balrog will configure SExtractor to use them. Two files which require further explanation are the `param` file and the `config` file. The `param` file sets which measurements will be in the output catalog and the `config` file contains settings like what thresholds to use, how to subtract the background, and what files to use. Anything in `config` can be overwritten from SExtractor's command line arguments. Not every keyword-value pair in `config` is relevant during Balrog runs, because Balrog sets some automatically, such as the image and weight map names. The top of Table 5.2 shows what is ignored in the `config` file. Turning association mode on or off requires modifications to both `config` and `assoc`, and Balrog automatically account for this. Balrog ignores any association lines that are in the user's `param` and `config` files, and generates the appropriate ones based on the user's Balrog command line arguments. We emphasize, there has been no loss generality for running SExtractor. We have just automated the process of creating an on/off switch for association mode, and moved some configurations to Balrog's command line rather than the SExtractor command line, because other Balrog functionality besides SExtractor depends on them as well.

3. Installation

Balrog consists of two dependencies: GalSim and SExtractor. Balrog itself is written entirely in Python and will run out of the box if *sufficiently recent* versions of GalSim and SExtractor have already been installed. Sufficiently recent GalSim means version `1.1` or newer. We have not thoroughly tested what constitutes a *recent enough* version of SExtractor. What we are able to say is that the oldest version we have successfully tested against is `2.17.0`. Balrog and GalSim both exist as repositories on GitHub and can be downloaded from the command line.

```
% git clone https://github.com/emhuff/Balrog.git
% git clone https://github.com/GalSim-developers/GalSim.git
```

This should automatically check out the master branch of each. Master is the only supported branch for Balrog. The master branch of GalSim will always be a version $\geq$ `1.1`. SExtractor is packaged as a tarball, which can be downloaded from its official website. We recommend choosing the most recent version.

The GalSim documentation includes an extensive installation guide, with many helpful hints and links to FAQ pages. We refer Balrog users to this guide for how to install GalSim. The GalSim team deserves a big thanks for the utility of their installation notes. Those who make up the the Balrog team are far from accomplished system administrators, but we were all independently successful installing GalSim on our systems by following these notes.

Section 3 of the SExtractor user manual contains a brief section explaining how to install the software. The code itself also includes a file called `INSTALL`, with slightly more detailed installation instructions. One item not explicitly stated in the installation notes is that SExtractor requires the ATLAS linear algebra package as a dependency. The ATLAS website includes the requisite source code and an extensive installation guide. Once ATLAS builds and installs, the SExtractor installation steps should be straightforward to follow and complete.

To test the Balrog installation, change directories to the directory `git` cloned for Balrog. The directory contains a file called `./balrog.py`. Run the file from the command line:

```
% ./balrog.py --fulltraceback
```

If installation was successful, this will run without printing any error messages.

4. Quick Start

Balrog has been designed with flexibility of use in mind. As a necessary consequence, a number of different configuration possibilities exist, each of which must be adequately explained, which quickly expands the length of the documentation. We realize parsing the entirety of this manual requires some time. Hence, the intent of the this section is to offer a short primer for a few of the most import features Balrog users will want to become familiar with. The comprehensive usage instructions are saved for later sections. We will refer readers to the relevant comprehensive sections for more details. Here we focus on running Balrog itself. Users less familiar with SExtractor may find it helpful to refer to our SExtractor appendix.

Throughout our documentation, we will supplement the discourse with concrete example calls users can run. In order to enable copying and pasting of the commands into the user's terminal and running them verbatim, users should create an alias to the Balrog executable Python file, `balrog.py`. This file is found in the directory which was pulled from GitHub during installation, which we will refer to as `$INSTALLDIR` for convenience. Adopting this convention, the alias statement looks as follows:

```
% alias runbalrog='$INSTALLDIR/balrog.py'
```

For convenient reference in later sections, we have effectively abbreviated what we have just defined in Table 4.1.

The fastest way to get started understanding how to configure and use Balrog is to run it using the example files which come packaged with the software, and then examine the input and output of the run. Balrog has been set up such that when the executable Python file is called without any command line arguments, it will run over the example files, filling in defaults as necessary. Thus, this initial call is as simple as:

```
% runbalrog
```

The rest of the examples here build off this call by adding additional configurations via command line arguments. To explore the command line arguments, one can run the following command to print a useful summary:

```
% runbalrog --help
```

For more extended help, the command line arguments built-in to Balrog are detailed further in Section 5. In brief, they are used to specify input images and their properties, as well as configuration files to use with SExtractor, and a few other variables every Balrog run will need to define. Each comes with a default Balrog will assume if the user does not supply one. The default example's image data (`--image`), weight map (`--weight`), and PSF (`--psf`) live in `$INSTALLDIR/default_example/`. The SExtractor configuration files live in `$INSTALLDIR/astro_config/`. File names are intended to be reasonably transparent. Balrog can be told to print messages more verbosely, and it is also instructive to repeat the example Balrog call in verbose mode.

```
% runbalrog --stdverbosity v
```

While the original call ran without printing anything, this one will write out some messages relevant to Balrog filling in defaults for parameters. Such messages can be helpful for users to recognize if they are not in fact using the configurations they think they are using. Closely related to `--stdverbosity` is `--logverbosity`. While `--stdverbosity` is for `stdout`/`stderr`, `--logverbosity` dictates the level of messaging logged to file. Run the the previous command, but now flag the logging to verbose:

Table 4.1: Definitions used throughout this manual to facilitate ease of command line calls.

Name	Meaning
`$INSTALLDIR`	Directory `git` clones for BALROG, i.e. where BALROG is installed
`runbalrog`	`$INSTALLDIR/balrog.py`

```
% runbalrog --stdverbosity v --logverbosity v
```

Opening the file found in `$INSTALLDIR/default_example/output/balrog_log/run.log.txt` should contain exactly the same statements printed to the command line. Saving output to file is useful when attempting to debug runs after-the-fact. Had any BALROG runtime errors occurred during the run, `run.log.txt` is the file they would be logged to. Additionally, BALROG logs the exact value of each command line argument in the file `$INSTALLDIR/default_example/output/balrog_log/args.log.txt` and redirects all SEXTRACTOR's `stdout` and `stderr` to `$INSTALLDIR/default_example/output/balrog_log/sex.log.txt`.

The command line arguments attempt to be rather intuitive as to what they mean, but one which may not be so obvious is `--pyconfig`. In order to flexibly handle how simulated object paratmeters will be generated, BALROG accepts defined blocks of Python code within the file specified by `--pyconfig`. Owing to its command line name, we will refer to this file simply as the `pyconfig` file throughout the documentation. Included within the `pyconfig` file is support for implementing custom functions and command line arguments to be called upon during the object parameter generation process. The core functions defined in `pyconfig` files have a strictly structured syntax which will be fully described in Section 6. The syntax is designed to be as natively Pythonesque as possible, so many users will likely be able to extrapolate directly from examples without reading lengthy documentation. In the current `runbalrog` examples, `--pyconfig` defaults to `$INSTALLDIR/config.py`. This file builds simulated galaxies as single component Sérsic objects, jointly sampling magnitudes, half light radii, and Sérsic index parameters from the included catalog. As an instructive guide, the file includes descriptive comment lines explaining what the different functions are used to do. Rather than copying the comments into this document, we encourage users to open the `pyconfig` file themselves. Reading through the file itself is a more efficient method to quickly get started than reading through anything else we could write here. A slightly more sophisticated `pyconfig` file can be found in `$INSTALLDIR/config2.py`. This generates galaxies with both a bulge and a disk component. Together, these two examples are designed to demonstrate the range of statements available to BALROG's `pyconfig` file.

All BALROG output is saved in subdirectories under the directory chosen by command line argument `--outdir`. This directory has defaulted to `$INSTALLDIR/default_example/output` for the example run. The complete set of output generated by BALROG is detailed in Section 7. In brief there are four subdirectories: `balrog_log`, `balrog_image`, `balrog_cat`, and `balrog_sexconfig`. We have already noted three log files written to `balrog_log`. In addition, an explicit copy of the user's `pyconfig` file is also written there. `balrog_image` includes copies of images BALROG either used or generated. Ones with `.sim` in their file names include the simulated ojects, and ones with `.nosim` do not. `balrog_cat` contains `example.truthcat.sim.fits`, the truth parameters assigned to simulated objects, as well as the SEXTRACTOR catalogs BALROG generates. Again, files including the simulated objects contain a `.sim` in their names. and the ones containing a `.nosim` do not. `balrog_sexconfig` copies the files BALROG used for configuring SEXTRACTOR. When running BALROG over files other than the defaults, the output file names are automatically generated based on the input file names. It may seem that a single run produces a substantial amount of output, but the idea is that any BALROG call be reproducible and debuggable from its output in case anything does go wrong.

5. Native Command Line Arguments

Balrog runs are configured via command line arguments. Two categories of arguments exist. First are the built-in ones, native to Balrog. These constitute the focus of this section. In addition, Balrog also supports a mechanism for users to define their own command line arguments. However, because the functionality is drawn from what we have deemed the `pyconfig` file, its treatment is saved for Section 6.1.

To generate a summary of Balrog's command line environment, run the following command, where `runbalrog` is defined in Table 4.1.

```
% runbalrog --help
```

This prints a listing of all the command line arguments (including those defined by the user), along with their available help strings. The built-in arguments' help strings are intended to be a useful quick reference tool, but in order to preserve suitable readability from the command line, they do not necessarily spell out every possible detail users may wish to know. Accordingly, we include a more extensive supplement within this section.

We direct the reader's attention to Table 5.1. It is a comprehensive listing, containing an entry for each of Balrog's built-in command line arguments. Table 5.1 is the core substance of this section, and is intended to be more or less readable on it own, apart from the text of the document. Ergo, we refrain from copying every detail from Table 5.1 into the text, preferring the orderly organization and conceptual clarity of the table over something which would require many more words to describe in paragraph form. Only a few general comments are necessary to put Table 5.1 into better context.

Each command line argument can be specified either with its full name or an abbreviated form, e.g. `--image` vs. `--i`. The full names attempt to be as intuitive as possible while maintaining a reasonably small number of characters. The abbreviations trade clarity for brevity for those who prefer compactness. Any parameter that is left out of a user's Balrog call assumes a default value in the code. Each of these defaults is listed in Table 5.1. In a few cases, the default values are conditionally defined, meaning Balrog will select one out of two possible defaults depending on how the run has been configured. Table 5.1 utilizes pseudocode to explain the behavior of how these defaults are ultimately chosen. When Balrog parses the user's command line, it performs a number of checks, thereby enforcing proper data types, file existence, etc. Warnings or errors are raised as needed. Table 5.1 has been grouped into subcategories of similar parameters. The divisions do not matter per se; they are merely an organizational guide. However, because Balrog contains quite a few possible arguments to adjust, we find them to be a useful aid in adding to tractability.

Some of the Balrog command line arguments are also SExtractor arguments. Section 2.4 noted that Balrog automatically sets some of the SExtractor command line argument based on the Balrog arguments. These have been summarized in Table 5.2.

Table 5.1: Command line arguments natively built-in to BALROG. Section heading lines are linked to text throughout the document with furter details about BALROG's behavior regarding those arguments.

Argument	Default

Output File Organization

`--outdir` / `-o` `$INSTALLDIR/default_example/output/`

Toplevel directory for BALROG output files. Files will be organized into intuitively named directories under `--outdir`.

`--clean` / `-c` Unflagged $\implies$ False

Delete intermediate image files (those in `--outdir/balrog_image`) when BALROG exits.

Logging

`--stdverbosity`/`--logverbosity` options:

`q`	Quiet	Errors only. (`--logverbosity q` does not exist.)
`n`	Normal	Errors and warnings
`v`	Verbose	Errors, warnings, and info statements
`vv`	Very Verbose	Errors, warnings, info, and debug statments. `vv` is not much different from `v`.

`--stdverbosity` / `-sv` n

Verbosity level for printing to `stdout`/`stderr`.

`--logverbosity` / `-lv` n

Verbosity level for logging to `--outdir/balrog_log/run.log.txt`.

`--fulltraceback` / `-ft` Unflagged $\implies$ False

Print the entire traceback when an exception forces BALROGto exit. By default, BALROG only prints the portion pointing to errors in `--pyconfig`. Problems in `pyconfig` might cause issues downstream, and seeing the full traceback could lead to confusion, thinking there is a coding error in other BALROG files.

Input Images

(All images must be given in FITS format. No other file types are supported.)

`--image` / `-i` `$INSTALLDIR/default_example/input/example.fits`

Image into which simulated galaxies are drawn.

`--weight` / `-w` `$INSTALLDIR/default_example/input/example.fits`

File containing the weight map associated with `--image`. This can be a separate file from `--image` or the same file, where the flux image and weight map live in different extensions.

`--psf` / `-p` `$INSTALLDIR/default_example/input/example.psf`

File containing the PSFEx PSF model for `--image`. This is a FITS file, but the convention uses `.psf` as the extension.

- Input Detection Images -

(All detection images must also be FITS files. Leave these unspecified to effectively run SExtractor in single image mode.)

`--detimage` / `-di` `--image`

File containing the input detection image. SExtractor uses this as the detection image in dual image mode. Nothing is drawn into this image.

`--detweight` / `-dw` `--weight`

File containing the weight map associated with `--detimage`. This can be a separate file from `--detimage` or the same file, where the flux image and weight map live in different extensions.

`--detpsf` / `-dp` `--psf`

File containing the PSFEx PSF model for `--detimage`. This is a FITS file, but the convention uses `.psf` as the extension.

- Input Images FITS Extensions -

(This is an integer position, not to be confused with a file name extension. Indexing begins at 0.)

`--imageext` / `-ie` 0

Extension of `--image` where the image flux data lives.

`--weightext` / `-we`
$$\begin{cases} \texttt{if --image != --weight: 0} \\ \texttt{else: --imageext + 1} \end{cases}$$

Extension of `--weight` where the weight map data lives.

- Input Detection Images FITS Extensions -

(All dectection extensions are also integers beginning at 0. Leave these blank if not trying to use SExtractor dual image mode.)

`--detimageext` / `-die` $\begin{cases} \texttt{if --detimage != --image: 0} \\ \texttt{else: --imageext} \end{cases}$

Extension of `--detimage` where the detection image flux data lives.

`--detweightext` / `-dwe` $\begin{cases} \texttt{if --detweight != --detimage: 0} \\ \texttt{else: --detimageext + 1} \end{cases}$

Extension of `--detimage` where the detection image flux data lives.

- Subsampling -

(Maintaining all defaults uses the entire image: $\{x \in [1, N_{\text{cols}}]\,,\ y \in [1, N_{\text{rows}}]\}$. If subsampling, Balrog will only operate over the reduced area, and the output image will only include the subsampled area. When subsampling, coordinates for simulated galaxies should be given in the original, unsubsampled coordinates.)

`--xmin` / `-x1` 1

Pixel x-coordinate for the lower bound of where to place simulated galaxies into `--image`.

`--xmax` / `-x2` N_{cols}

Pixel x-coordinate for the upper bound of where to place simulated galaxies into `--image`.

`--ymin` / `-y1` 1

Pixel y-coordinate for the lower bound of where to place simulated galaxies into `--image`.

`--ymax` / `-y2` N_{rows}

Pixel y-coordinate for the upper bound of where to place simulated galaxies into `--image`.

- Simulated Galaxy Generation -

`--pyconfig` / `-pc` `$INSTALLDIR/config.py`

Balrog's Python configuration file. It defines the user's custom command line arguments, determines how to generate simulated object properties, and allows for overriding specifications in `--sexconfig`.

`--ngal` / `-n` 50

Number of objects to simulate. `gal` is a bit of a misnomer; the objects need not be galaxies.

`--zeropoint` / `-z`

```
try: --zeropoint = --image[--imageext].header['SEXMGZPT']
except: --zeropoint = 30.0
```

Zeropoint for converting sampled simulation magnitudes to simulated fluxes. SExtractor will also use this zeropoint for magnitude measurements. `--zeropoint` can take two types values: a float explicitly defining the zeropoint, or a string referring to a keword written in the header of `--image[--imageext]`. If neither of these is successfully found, Balrog uses the default.

`--gain` / `-g`

```
try: --gain = --image[--imageext].header['GAIN']
except: --gain = 1.0
```

Gain [electron/ADU] for adding CCD noise to the simulated galaxies. `--gain` can take two types of values: a float explicitly defining the gain, or a string referring to a keword written in the header of `--image[--imageext]`. If neither of these is successfully found, Balrog uses the default.

`--seed` / `-s` Current time

Seed to give random number generator for any sampling which requires it, except noise realizations which are always different.

`--nodraw` / `-nd` Unflagged $\implies$ False

Do not actually draw simulated objects into the image. This will not write the truth catalog, but does run SExtractor.

- - - - - - - - - - - - - - - - SExtractor - - - - - - - - - - - - - - - -

(Refer to the SExtractor user manual or Source Extractor for Dummies for more help. We have also added a brief SExtractor appendix.)

`--sexpath` / `-sex` `sex`

Path to SExtractor executable.

`--sexconfig` / `-sc` `$INSTALLDIR/astro_config/sex.config`

Config file for running SExtractor.

`--sexparam` / `-sp` `$INSTALLDIR/astro_config/bulge.param`

Param file specifying which measurements SExtractor outputs.

`--sexnnw` / `-sn`   `$INSTALLDIR/astro_config/sex.nnw`

SExtractor neural network file for star-galaxy separation

`--sexconv` / `-sf`   `$INSTALLDIR/astro_config/sex.conv`

SExtractor filter convolution file when making detections.

`--noassoc` / `-na`   Unflagged $\implies$ use association mode.

Do not run SExtractor in association mode.

`--nonosim` / `-nn`   Unflagged $\implies$ perform the SExtractor run

Skip SExtractor run over the original image, prior to any simulation.

`--nosimsexparam` / `-nsp`   `$INSTALLDIR/astro_config/sex.param`.

Param file specifying which measurements SExtractor outputs during run over the original image, prior to any simulation.

`--catfitstype` / `-ct`   `ldac`

Type of FITS catalog file SExtractor writes out. `--catfitstype` $\in$ {`ldac`, `1.0`}.

`--indexstart` / `-is`   0

Identifying index for first Balrog simulated object, i.e. where to start `balrog_index`.

`--imageonly` / `-io`   Unflagged $\implies$ False

Do not run SExtractor in Balrog. No measurement catalogs will be written.

Table 5.2: Translation between Balrog command line argument names and how Balrog automatically configures SExtractor with these arguments.

Balrog		SExtractor
	Balrog arguments used to set SExtractor arguments	
--image	→	IMAGE
--weight	→	WEIGHT_IMAGE
--sexnnw	→	STARNNW_NAME
--sexconv	→	FILTER_NAME
--zeropoint	→	MAG_ZEROPOINT
--psf	→	PSF_NAME
--catfitstype	→	CATALOG_TYPE
--outdir	→	CATALOG_NAME
--sexnnw	→	STARNNW_NAME
--sexconv	→	FILTER_NAME
	Balrog's association mode configuration	
if !--noassoc:		
truth	→	ASSOC_DATA
truth['x'],truth['y']	→	ASSOC_PARAMS
		ASSOC_RADIUS = 2.0
		ASSOC_TYPE = NEAREST
		ASSOCSELEC_TYPE = MATCHED
Files slightly modified by Balrog to turn association mode on/off.		
--sexconfig	→	c
--sexparam	→	PARAMETERS_NAME
--nosimsexparam	→	PARAMETERS_NAME

6. Balrog's pyconfig File

As introduced in previous sections, Balrog's `pyconfig` file is a configuration file for Balrog made up of Python statements. Relying on the ease and simplicity of writing Python code, this file is what contributes a large portion of Balrog's flexibility. The authors cannot natively build-in to Balrog all the features every user would like to see. Thus, we attempt to create an environment which makes it easy for users to fold in additional content themselves. The `pyconfig` file plays a significant role in Balrog, and hence we will take care to provide adequate documentation.

As far as general comments go, we remark that the `pyconfig` file makes use of five core functions called: `CustomArgs`, `CustomParseArgs`, `SimulationRules`, `GalsimParams`, and `SextractorConfigs`, each of which will be described further in what follows. Balrog checks for functions of exactly these names. If one of them does not exist, Balrog continues by skipping the portion of the code where the function would have been called. An omission should not cause the Balrog run to fail; however, some level of flexibility will have been lost. We issue warning messages to alert the user when one or more of the functions are missing.

The procedual work flow of `pyconfig` is as follows. Users create command line arguments then parse them appropriately. These arguments, as well as the ones Balrog relies on natively, are then made available to functions which define how Balrog's object properties should be generated. We have included some convenience features to coordinate the coding of these functions. Within the `pyconfig` environment, it is also possible to override SExtractor command line arguments, and the GalSim `GSParams` if desired.

In the remainder of Section 6 we concentrate on `pyconfig`'s usage, with a particular focus on elucidating its syntax. Section 6.1 treats user-defined command line arguments within `pyconfig` and Section 6.2 illustrates how `pyconfig` specifies simulated object property generation. Section 6.3 describes the way in which `pyconfig` can be used to adjust GalSim's `GSParams`, and Section 6.4 addressing overriding SExtractor settings in `--sexconfig`, As an aid while reading this section, we strongly encourage users to open one of the example `pyconfig` files shipped with Balrog, `$INSTALLDIR/config.py` or `$INSTALLDIR/config2.py`. As mentioned in Table 4.1, `$INSTALLDIR` is the directory where Balrog is installed.

6.1 User-Defined Command Line Environment

Within the `pyconfig` file, the definitions for custom command line arguments occur inside the `CustomArgs` function. Passed to `CustomArgs` as a function argument is `parser`, an object made by Python's `argparse.ArgumentParser()`. Arguments can be added to parser according to the usual `argparse` syntax. For those unfamiliar with `argparse`, this tutorial contains many useful examples. A simple example of `CustomArgs` is copied below.

```
def CustomArgs(parser):
   parser.add_argument( "-cs", "--catalogsample", help="Catalog used to sample simulated
      galaxy parameter distributions from", type=str, default=None )
```

Users are allowed to add arguments by any names except those already defined as native Balrog command line arguments. Adding two arguments of the same name in the `ArgumentParser` class raises an exception. The utility of adding custom command line arguments is that the values given to the arguments are made available to the other core functions in `pyconfig`.

The user-defined arguments are parsed within `CustomParseArgs`. When this function is called, Balrog has already grabbed any of the user's custom arguments from the command line, and saved them into an object called `args`. `args` is passed as the single function argument to `CustomParseArgs`. It is equivalent to an object returned by `parser.parse_args()`; each one of the user's command line arguments becomes an attribute of `args`. `CustomParseArgs` allows users to apply any Python operations they so desire to their arguments. A simple version of `CustomParseArgs` has been included below:

```
def CustomParseArgs(args):
    thisdir = os.path.dirname( os.path.realpath(__file__) )
    if args.catalogsample==None:
        args.catalogsample = os.path.join(thisdir, 'cosmos.fits')
```

As the example demonstrates, one feature of the `CustomParseArgs` is a convenient way for users to effectively define new defaults for BALROG using commonly-known ordinary Python syntax. More generally, different BALROG runs are now capable of recognizing and appropriately parsing completely different sets of input parameters.

Returning to `args`, the object not only contains an attribute for each of the user's custom defined arguments; its attributes also include each of the native BALROG arguments. When `CustomParseArgs` is called, the built-in arguments have already been parsed, so arguments which were left out of the command line call have assumed their defaults. Inside `pyconfig`, users are allowed to modify the attributes of `args` derived from built-in BALROG arguments, if they so desire. However, in order to prevent changes from potentially causing unhandled exceptions later in BALROG, any modifications will not propagate outside `CustomParseArgs`. Downstream, BALROG's native arguments will behave according to how they were specified at the command line. Thus, allowing changes to the native attributes is more of a local convenience than anything else.

`args` will be given as a function argument to the `SimulationRules`, `GalsimParams`, and `SextractorConfigs` functions. Changes within these functions to `args`'s attributes, both those derived from native or custom arguments, are also effectively *local only*. Modifications made in one of the functions do not propagate outside that function. The code has been written this way so users do not need to consider the order in which the three functions execute in order to know exactly what `args` they are using in each function. The behavior is consistent with including the `CustomParseArgs` in the first place. The only place to make persisting changes to `args` is `CustomParseArgs`.

Please note, no parsing of any kind can be done inside `CustomArgs`. `CustomArgs` is for definitions only. It informs BALROG what it should be looking for when it receives a command line. No values have yet been assigned to any of the defined parameters when `CustomArgs` exits. `CustomParseArgs` on the other hand tells BALROG what do once it has actually received the command line and has saved each of the parameters into an attribute of `args`.

6.2 Defining How to Generate Simulated Object Properties

Deciding how the simulated object properties should be sampled occurs inside the function `SimulationRules`. Passed into `SimulationRules` are four arguments: `args`, `rules`, `sampled`, and `TruthCat`. As discussed in Section 6.1, `args` refers to the parsed command line arguments, both native BALROG and user-defined. `rules` is an object whose attributes are overwritten in order to specify how simulated object properties are sampled. `sampled` gives access to simulated object parameters after they have been sampled. `TruthCat` allows users to add additional columns of output to the BALROG truth catalog. `rules`, `sampled`, and `TruthCat`'s usage will become clearer to follow.

Section 2.3 laid out the parameters which characterize each simulated object. Briefly recapitulating, each object is made up of one or Sérsic profiles. Each Sérsic profile is composed of five components: Sérsic index (n), half light radius along the major axis(r_e), brightness (m), minor to major axis ratio (b/a), and orientation angle counter-clockwise from the $\hat{x}$-direction (β). Furthermore, a simulated object also shares fives additional parameters among each Sérsic profile: two centroid position coordinates, (x, y), two reduced shear coordinates (g_1, g_2), and magnification (μ). This totals ten "*variables*" which fully specify the object. `rules` and `sampled` are each composed of ten attributes, matching these same ten variables. The attribute names for `rules` and `sampled` have been transparently chosen. Nevertheless, for completeness the names and meanings are fully specified in Table 6.1. Table 6.1 uses `rules` as the concrete example. The attribute names of `sampled` are identical. For the time being, we will continue assuming each object has been simulated as a single Sérsic profile. We will return to handling two or more in a few paragraphs.

Users overwrite each of the `rules` object's attributes to define their sampling of simulation parameters. As-

Table 6.1: Attributes of the `rules` object in the `pyfconfig` file, defining the simulated object properties.

Attribute	Meaning
`rules.x`	Object centroid x-coordinate [pixels]
`rules.y`	Object centroid y-coordinate [pixels]
`rules.g1`	Reduced shear, g_1 component
`rules.g2`	Reduced shear, g_2 component
`rules.magnification`	Magnification, $1+\kappa$
`rules.sersicindex`	Sérsic index
`rules.halflightradus`	Sérsic half light radius (along major axis) [arcsec]
`rules.magnitude`	Brightness [mag]
`rules.axisratio`	Minor to major axis ratio
`rules.beta`	Orientation angle of major axis (measured from $\hat{x}$-axis) [degrees]

Table 6.2: Syntax examples for each of the simulation parameter sampling types Balrog understands.

Type	Example
Constant	`rules.g1 = 0`
Array	`rules.x = np.linspace(args.xmin, args.xmax, args.ngal)`
(Sampled	`rules.y = sampled.x)`
Catalog	`rules.magnitude = Catalog(file='cosmos.fits', ext=1, col='IMAG')`
Function	`rules.y = Function(function=rand, args=[args.xmin,args.xmax,args.ngal])`

Table 6.3: Defaults for the `rules` object, i.e. defaults for how to generate simulated object properties in the `pyconfig` file.

Attribute	Default
`rules.x`	`np.random.uniform(args.xmin, args.xmax, args.ngal)`
`rules.y`	`np.random.uniform(args.ymin, args.ymax, args.ngal)`
`rules.g1`	0
`rules.g2`	0
`rules.magnification`	1
- Sérsic -	
(If Sérsic attributes are multi-component, each list element uses this default)	
`rules.sersicindex`	`Catalog('cosmos.fits',1,'SERSIC_INDEX')`
`rules.halflightradius`	`Catalog('cosmos.fits',1,'HALF_LIGHT_RADIUS')`
`rules.magnitude`	`Catalog('cosmos.fits',1,'IMAG')`
`rules.axisratio`	1
`rules.beta`	0

signments can assume four (or possibly five depending how categorized) types of statements. The syntax for each is designed to be straightforward, intuitive, and Pythonesque. For quick reference, an example of each is included in Table 6.2. The first type of rule assignment is a constant, meaning each of the objects in the simulated sample will have the same value for the selected parameter, e.g. `rules.g1 = 0`. Second, rule types can also be given as an array, equal in length to the number of simulated objects. Simulated object i for the chosen parameter then assumes the value in element i of the array. The example in Table 6.2 is given as `rules.x = np.linspace(args.xmin, args.xmax, args.ngal)`, meaning objects will be evenly spaced along the $\hat{x}$-axis across the image.

The third available type for `rules` is sampling from a catalog. This makes use of a function Balrog has defined called `Catalog`. Calls to `Catalog` require three arguments: `file`, `ext`, and `col`. `file` is the file path to a FITS file; `ext` identifies which extension contains the data table to use; and `col` is the name of the column to draw from. If `Catalog` is used multiple times, multiple columns selected from the same data table are automatically jointly sampled. As a convenience for sampling multiple columns from the same catalog, we have created a shortcut function called `Table`. `Table` takes two arguments: `file` and `ext`, and returns an object that behaves like `Catalog`, which can be given any of the column names as a single argument. We have summarized the `Catalog`/`Table` functionality below.

```
#These two blocks of code are equivalent
rules.magnitude = Catalog(file='cosmos.fits', ext=1, col='IMAG')

tab = Table(file=args.catalog, ext=args.ext)
rules.magnitude = tab.Column('IMAG')
```

Last but not least, the fourth assignable sampling type to `rules` is a function. Users write their own function according to usual Python syntax, then feed this function and its necessary arguments as the arguments to Balrog's `Function` command. The one requirement of sampling functions is that they must return an array equal in length to the number of simulated objects. Like the array sampling type, object i will use element i of the returned array as its truth parameter. Here, we provide an example code snippet, illustrative of how one could place the simulated objects at random image positions.

```
def rand(minimum, maximum, ngal):
    return np.random.uniform( minimum, maximum, ngal )

def SimulationRules(args, rules, sampled, TruthCat):
    rules.x = Function(function=rand, args=[args.xmin, args.xmax, args.ngal])
    rules.y = Function(function=rand, args=[args.ymin, args.ymax, args.ngal])
```

`Function` affords a great deal of flexibility toward accomidating a wide variety of different simulation scenarios. We will make further statements regarding slightly more sophisticated usage of `Function`, but it is useful to consider a few other points first.

Let us now return to the `sampled` object which is passed to `SimulationRules` as an argument. `sampled` is used in conjunction with `rules` to define one parameter's rule in terms of another's. Conceptually speaking, `sampled` can be thought of an object whose attributes are arrays that have been set equal to the simulated object parameters post sampling. One might consider `sampled` its own sampling type, but at the same time, it is also functionally equivalent to an array. This is why a Sampled type has been added as a fifth entry in Table 6.2, in parentheses directly under the Array line. To better understand `sampled`, consider the following code sample:

```
def SimulationRules(args, rules, sampled, TruthCat):
    rules.x = Function(function=rand, args=[args.xmin, args.xmax, args.ngal])
    rules.y = sampled.x
```

The x-coordinates randomly sample, and then each objects's y-coordinate is set equal to exactly the same value as its x-coordinate. Please note, this is expressly different from the original example where both x and y sampled randomly. `sampled` can also be used as an argument within the `args` passed to a sampling function, as is the case in the proceeding example.

```
def SimulationRules(args, rules, sampled, TruthCat):
    rules.x = Function(function=rand, args=[args.xmin, args.xmax, args.ngal])
    rules.y = Function(function=rand, args=[args.ymin, args.ymax, args.ngal])
    rules.axisratio = Function(function=SampleFunction, args=[sampled.x, sampled.y,args.xmax,
        args.ymax])

def SampleFunction(x, y, xmax, ymax):
    dist = np.sqrt(x*x + y*y)
    max = np.sqrt(xmax*xmax + ymax*ymax)
    return dist/max
```

This functionality allows users to have access to the sampled object truth information in their functions. This particular example gave each object a position dependent axis ratio. As a further example, one could apply position dependent lensing effects in a similar fashion.

Thus far we have been operating with single component Sérsic models. However, two or more components can be implemented. Doing so requires a call to a Balrog function called `InitializeSersic` before any of `rules`'s five Sérsic attributes (`sersicindex`, `halflightradius`, `magnitude`, `axisratio`, `beta`) are reassigned. `InitializeSersic` takes `rules` and `sampled` as arguments as well as `nProfiles`, the number of superimposed Sérsic models, e.g. `InitializeSersic(rules, sampled, nProfiles=2)`. This has recast the Sérsic components of `rules` and `sampled` as length two lists. Now the elements of these attributes of `rules` must be updated accordingly, using syntax akin to that of any other Python list. For example, the following builds galaxies as bulge + disk models, using an exponential disk ($n = 1$) and a de Vaucouleurs bulge ($n = 4$).

```
def SimulationRules(args, rules, sampled, TruthCat):
    rules.x = Function(function=rand, args=[args.xmin, args.xmax, args.ngal])
    rules.y = Function(function=rand, args=[args.ymin, args.ymax, args.ngal])
    rules.g1 = 0
    rules.g2 = 0
    rules.magnfication = 1

    InitializeSersic(rules, sampled, nProfiles=2)
    rules.sersicindex = [1, 4]
    rules.halflightradius[0] = Catalog('cosmos.fits',1,'HALF_LIGHT_RADIUS')
    rules.halflightradius[1] = sampled.halflightradius[0]
    rules.magnitude[0] = Catalog('cosmos.fits',1,'IMAG')
    rules.magnitude[1] = sampled.magnitude[0]
    rules.beta[0] = Function(function=rand, args=[-90, 90, args.ngal])
    rules.beta[1] = sampled.beta[0]
    rules.axisratio[1] = sampled.axisratio[0]
    rules.axisratio[0] = Function(function=rand, args=[0.05, 1, args.ngal])
```

The example uses bulges and disks with identical half light radii, magnitudes, axis ratios, and orientation angles. The values for the half light radius and magnitude are sampled from the catalog `cosmos.fits`. The axis ratios and orientations angles are random, as are the centroid positions. No lensing is applied. We direct the reader's attention to the final two lines. One might ordinarily think of these as needing to be in the opposite order, but either is permissible and equivalent in BALROG. Also notice that the `x`, `y`, `g1`, `g2`, and `magnification` attributes are unaffected by the `InitializeSersic` statement. These attributes are never lists.

All attributes of `rules` have a default, which will be used if the attribute is not overwritten. Any time a default must be used, a warning message is issued. The defaults themselves are listed in Table 6.3. Notice that in some cases, the default is to sample from the `cosmos.fits` catalog which ships with BALROG. These should create realistic populations. By default a single component Sérsic model is adopted by BALROG. If the users sets the Sérsic mode to multi-components, each element of an attribute's list assumes the default in Table 6.3. Whenever one uses `InitializeSersic`, the Sérsic attributes are reinitialized to their defaults, so be careful if `InitializeSersic` is called more than once in `SimulationRules`.

Previously we mentioned we would elaborate further on some finer details of using `Function` to sample. We now do so. Inside `Function` statements, it is possible to make an element of `args` a `Catalog` statement. Here is a concrete example.

```
def SimulationRules(args, rules, sampled, TruthCat):
    nc = Catalog(file='cosmos.fits',ext=1,col='SERSIC_INDEX')
    n = Function(function=g, args=[1, 0.05, args.ngal, nc])
    rules.sersicindex = n

def g(avg, std, ngal, add):
    gaus = gaussian(avg, std, ngal)
    return gaus+add

def gaussian(avg, std, ngal):
    return np.random.normal( avg, std, ngal )
```

An amount which is sampled from the catalog is added to the Gaussian. This particular implementation is merely illustrative, not necessarily something one would realistically do scientifically. If the table given in `Catalog` was also used to sample any other parameters, everything is still automatically jointly sampled. `Function` can also take other `Function` statements as elements of `args`. To demonstrate, we form another example very similar to the previous one.

```
def SimulationRules(args, rules, sampled, TruthCat):
    ns = Function(function=exact, args=[np.ones(args.ngal)])
    n = Function(function=g, args=[4, 0.05, args.ngal, ns])
    rules.sersicindex = n

def exact(item):
    return item

def g(avg, std, ngal, add):
    gaus = gaussian(avg, std, ngal)
    return gaus+add

def gaussian(avg, std, ngal):
    return np.random.normal( avg, std, ngal )
```

Here, the usage of the `exact` function is trivial, but nevertheless representative of the proper usage. `Function` can take a third argument which we have not mentioned yet: `kwargs`. `kwargs` accomodates for any keyword arguments of the sampling function. Along the lines of the usual Python syntax, `kwargs` must be a dictionary. For example:

```
def SimulationRules(args, rules, sampled, TruthCat):
    ns = Function(function=exact, args=[np.ones(args.ngal)], kwargs={'i2':np.ones(args.ngal)})
    n = Function(function=g, args=[4, 0.05, args.ngal, ns])
    rules.sersicindex = n

def exact(item, i2=None):
    if i2==None:
        return item
    else:
        return item + i2

def g(avg, std, ngal, add):
    gaus = gaussian(avg, std, ngal)
    return gaus+add

def gaussian(avg, std, ngal):
    return np.random.normal( avg, std, ngal )
```

We now return to `SimulationRules`'s `TruthCat` argument. `TrutcCat`'s `AddColumn` method allows user's to append additional columns of output to Balrog's truth catalog, which will be described in Section 7.2. `AddColumn` takes up to three arguments. The first is the information to be appended. This can be of any of the four (five) sampling types `rules` understands. The other two arguments are `name` and `fmt`. `name` sets the column name in the output catalog, and `fmt` sets its FITS datatype (cf. Section 4.4 of the PyFITS manual). With `Catalog`/`Table`, `name` and `fmt` are optional, and read from the FITS header if not given. Here are a couple examples:

```
def SimulationRules(args, rules, sampled, TruthCat):
    tab = Table(file=args.catalog, ext=args.ext)
    TruthCat.AddColumn(tab.Column('Z'), name='redshift')
    TruthCat.AddColumn(Catalog('cosmos.fit',1,'IMAG'))
    TruthCat.AddColumn(args.seed, name='SEED', fmt='J')
```

`SimulationRules` is written to be handle errors. An exception is raised When users try to enter a sampling specification that Balrog does not understand. Exceptions are also raised if users attempt to reassign attributes of `sampled` or access an attribute of `rules` or `sampled` which does not actually exist. Such exceptions cause Balrog to terminate. The logic is that any Balrog `pyconfig` syntax errors are treated like ordinary Python syntax errors and thus force the execution thread to exit. The traceback will include line numbers for where the process was terminated.

6.3 GalSim Configuration Overrides

The final consideration to make about generating simulated objects is the function `GalsimParams`. This function is passed three arguments: `args`, `gsparams`, and `galaxies`. `args` is the same parsed command line argument object as passed to `SimulationRules`. Likewise, `galaxies` is the same object as `sampled`. It has just been renamed. `galaxies` is a slight misnomer, the simulated objects can be anything, not just galaxies. `gsparams` is an object which populates each objects's GalSim `GSParams`, having an attribute for each `GSParams` keyword. These attributes behave essentially the same as the non-Sérsic components of `rules`. Each attribute of `gsparams` can take on the same four (five) sampling types as `rules`: a constant, array, (a `galaxies` attribute, which is not particularly useful here on its own but could be inside `Function`), `Catalog`, or `Function`. All the same functionality applies to

`Function` as above.

Understanding all the `gsparams` attributes is a fairly technical subject, so we refer users who need to adjust them to GalSim's reference page for the complete details. In the context of Balrog these parameters are relevant for drawing the objects into postage stamp images. `alias_threshold`, for example, determines what maximum fraction of the object's flux may be lost outside the boundaries of the postage stamp when determining how large of a postage stamp is needed for drawing the object. Convolution with the PSF is done in Fourier Space. `maxk_theshold` determines the maximum k used in FFTs such that k-values must fall below `maxk_threshold` before cutting off the FFT. We will mention, there are also parameters for setting target accuracies.

If users do not overwrite a `gsparams` attribute, Balrog uses GalSim's default. In many cases, the defaults work well enough. The exception is when very bright objects ($\sim$ 15 mag, depending on the calibration) are being drawn, and even a small percentage of the flux totals to a significant number of ADU. When one thinks there might be an issue, it is useful to visually inspect the image into which the simulated objects have been drawn (c.f. Section 7) for signs of aliasing.

6.4 SExtractor Configuration Overrides

The final of the core functions in `pyconfig` which we are yet to discuss in detail is `SextractorConfigs`. This function is very simple. It is passed two arguments: `args` and `config`. As was the case in `SimulationRules` and `GalsimParams`, `args` contains the parsed command line arguments. `config` is a dictionary. Users are free to add to the dictionary the keyword-value pairs that a `sexconfig` file understands. These will then override what was given in `--sexconfig` by giving them as command line parameters to SExtractor. This effectively allows users to do anything from the Balrog command line they could do from the SExtractor command line. While none of the keywords will cause an error in `SextractorConfigs`, the same issue discussed in Section 2.4 applies here. A dozen or so of the keywords will not actually overwrite anything. Namely, each of the SExtractor parameters in the right-hand column of Table 5.2 cannot explicitly be overridden in Balrog.

7. Output

Each Balrog run generates a number of output files. At first, this number may seem larger than was perhaps expected. However, we air on the side of writing to disk anything users might possibly want afterward. We intend for Balrog to be transparent and debuggable. All runs should be recreateable from their output files.

The output files are organized into a fixed directory structure. Users indicate the `--outdir` command line argument, and the remainder of the naming scheme occurs automatically, placing files in subdirectories under `--outdir`. Four subdirectories are written, labeled according to what type of files they contain. Images write to `--outdir/balrog_image/`, catalogs go to `--outdir/balrog_cat/`, SExtractor configurations are saved to `--outdir/balrog_sexconfig/`, and log files are directed to `--outdir/balrog_log/`. Table 7.3 lists the contents of each of these subdirectories, giving a brief description of each file. Depending on how Balrog was configured, not necessarily every file in Table 7.3 will be present in every run. The `*` symbol in Table 7.3 will be replaced with the base name of the input image file. By base name, we mean `--image`'s file name, stripping off the `.fits` extension, as well as any proceeding directories locating it on the filesystem. For example, if the input image is named `/Users/Balrog/example.fits`, `*` will be replaced with `example`. If the input file name does not end with the `.fits` extension, the file name itself is used as the base name.

The following sections will further examine the contents of all Balrog's output files, building off Table 7.3. Each section discusses one of the output's four subdirectories, equivalent to one of the headings in Table 7.3. Section 7.1 briefly addresses the output images, while Section 7.2 describes the catalog files Balrog saves. Section 7.3 concerns SExtractor files, and Section 7.4 details what exactly Balrog logs to file.

7.1 Output Images

Every successful Balrog run saves three, possibly four types of images to `balrog_image/`. The `*.nosim.fits` and `*.psf` files are copies of the input image (`--image`) and the input PSF (`--psf`) respectively. If no subsampling was done, `*.nosim.fits` and `*.psf` are actually symbolic links to the input files instead of hard copies. Making symbolic links avoids writing potentially large files to disk. SExtractor only operates over full files, so subsampling requires writing new files. Subsampling the PSF amounts to appropriately changing the `POLZERO1` and `POLZERO2` header keywords. The `*.sim.fits` file is the image the simulated objects have been written into. If the weight map lives in a separate file from the flux image, it is written to `*.weight.fits`. It will also be a symbolic link if no subsampling has occurred. A `.sim` or `.nosim` qualifier for the weight image is irrelevant because the simulation process does not change it. When users flag the `--clean` option, the images in `balrog_image/` are deleted upon completion of Balrog.

In dual image mode, Balrog does not change the detection image (`--detimage`) or weight map (`--detweight`), and as of the writing of this documentation, copies of them do not write out. However, a copy of the detection PSF is written, which would be a subsampled version if relevant. The inconsistency of this behavior should be changed in the future.

7.2 Output Catalogs

Balrog writes up to three catalog files: `*.measuredcat.nosim.fits`, `*.measuredcat.sim.fits`, and `*.truthcat.sim.fits`. The truth catalog, `*.truthcat.sim.fits`, is always written, unless `--nodraw` is set. The catalog contains a column for each of the attributes of listed in Table 6.1 Five of the columns are named exactly as they appear in Table 6.1: `x`, `y`, `g1`, `g2`, and `magnification`. `sersicindex`, The `halflightradius`, `magnitude`, `axisratio`, and `beta` attributes are allowed to exist as arrays in Balrog. The truth catalog includes columns for each of these attributes, whose names are indexed with respect to how many Sérsic profiles have been simulated. The indexing appears as a `'_%i'` appended to the attribute name. Indexing begins at zero. For example, when simulating a

Table 7.1: BALROG's output truth catalog.

Name	Units	Meaning
`balrog_index`		Unique identifying index for each object
`x`	pixel	Centroid x-coordinate
`y`	pixel	Centroid y-coordinate
`g1`		Reduced shear
`g2`		Reduced shear
`magnification`		Magnification, $1{+}\kappa$
`sersicindex_*`		Sérsic index
`halflightradius_*`	arcsec	Half light radius along major axis
`axisratio_*`		Minor to major axis ratio, $^b/_a$
`beta_*`	degree	Orientation angle, counterclockwise from $\hat{x}$-direction
`flux_*`	ADU	Flux
`flux_noiseless`	ADU	Sum of flux GALSIM drew into postage stamp before adding CCD noise
`flux_noised`	ADU	Sum of flux GALSIM drew into postage stamp after adding CCD noise
`not_drawn`		1 if GALSIM drawing failed (e.g. too large FFT); 0 otherwise
+ User-defined columns		

two component Sérsic object, the truth catalog would contain `sersicindex_0` and `sersicindex_1` columns. A single component model would only include `sersicindex_0`. In the truth catalog, `magnitude` exists as `flux` [ADU] instead of explicitly `magnitude` [mag] because flux is what is drawn into the image by GALSIM. Two additional columns of `*.truthcat.sim.fits` are `flux_noiseless` and `flux_noised`. While the `flux_%i` keywords are the simulation parameters that were generated for each Sérsic component, `flux_noised` is the total flux GALSIM actually drew into the postage stamp, and `flux_noiseless` is the same without any noise. Only one `flux_noiseless` and `flux_noised` exists per galaxy. The `balrog_index` column is an integer to uniquely identify each simulated object, beginning at `--indexstart`, and incrementing for each object. The truth catalog will also contain any user-defined columns defined from the `pyconfig` file.

`*.measuredcat.nosim.fits` and `*.measuredcat.sim.fits` are the two SEXTRACTOR catalogs. `*.measuredcat.sim.fits` includes the simulated galaxies and `*.measuredcat.nosim.fits` does not. If `--nonosim` is flagged, there will not be a `.nosim.fits` file. If `--imageonly` is flagged, neither file will exist. When using association mode, the catalogs are made up solely of simulated galaxies, modulo blending effects (c.f Section 2.4). Likely all the simulated galaxies will not be detected so there usually will not be as many rows as simulated galaxies. Furthermore, when association mode is used, `.sim.fits` includes the truth information in the `VECTOR_ASSOC` column. Each row of of `VECTOR_ASSOC` is an array of truth parameters. The catalog's header enumerates exactly what each element of the array means. There are keywords `V%i` and `VUNIT%i`, which will match the names of the truth catalog's columns. These are the ordered values of the `VECTOR_ASSOC` array along with their units. For example, we can print the relevant header information from the default example runs in Section 4:

```
% listhead "default_example/output/balrog_cat/example.measuredcat.sim.fits[2]" | grep V | grep
    " ="
TTYPE1 = 'VECTOR_ASSOC' / ASSOCiated parameter vector
V0 = 'not_drawn'
VUNIT0 = 'dimensionless'
V1 = 'g2 '
VUNIT1 = 'dimensionless'
```

```
V2 = 'g1 '
VUNIT2 = 'dimensionless'
V3 = 'flux_noiseless'
VUNIT3 = 'ADU '
V4 = 'magnification'
VUNIT4 = 'dimensionless'
V5 = 'balrog_index'
VUNIT5 = 'dimensionless'
V6 = 'y '
VUNIT6 = 'pix '
V7 = 'x '
VUNIT7 = 'pix '
V8 = 'flux_noised'
VUNIT8 = 'ADU '
V9 = 'halflightradius_0'
VUNIT9 = 'arcsec '
V10 = 'beta_0 '
VUNIT10 = 'deg '
V11 = 'sersicindex_0'
VUNIT11 = 'dimensionless'
V12 = 'axisratio_0'
VUNIT12 = 'dimensionless'
V13 = 'flux_0 '
VUNIT13 = 'ADU '
```

Think of '`V`' as shorthand for '`VECTOR`'. The indices then label the components of this vector. Since a truth catalog itself is written by Balrog it is not strictly necessary to include the truth vector in the measured catalog. However, it is convenient and essentially comes for free with Balrog's association mode setup.

7.3 SExtractor Configuration Files

Table 7.3 itself is clear enough for most of the details we would like to point out relevant to the files in the `balrog_sexconfig/` directory, so just a few additional comments are in order. To run SExtractor in association mode, users give SExtractor a list of coordinates to match to, which is stored in a `.txt` file. This is what `*.assoc.nosim.txt` and `*.assoc.sim.txt` are used for; they amount to `txt` file versions of the truth catalog. The files given by command line arguments `--sexconfig`, `--sexparam`, `--nosimsexparam`, `--sexnnw`, and `--sexconv` are copied to the output directory to facilitate reproducibility. Users may very well modify the SExtractor configurations files on their filesystem, and even if they do, they still have logged copies of how they existed when Balrog ran. Because Balrog automatically accounts for turning SExtractor association mode on and off, `--sexconfig` and `--sexparam`/`--nosimsexparam` may need to be slightly modified (c.f. Section 5). `*.measuredcat.nosim.sex.config`, `*.measuredcat.sim.sex.config`, `*.measuredcat.nosim.sex.params`, and `*.measuredcat.sim.sex.params` are the slightly modified versions and are written to the output directory as well. Any settings unrelated to association mode remain unchanged.

7.4 Log Files

Balrog saves four files to the `balrog_log/` directory. The first is an exact copy of the user's `pyconfig` file. Next, as its name would suggest, `run.log.txt` is a Balrog *run log*. Any errors or warnings Balrog raised would be directed to this file. The verbosity of the file's messages is dictated by `--logverbosity`; see Table 5.1 for the available options. For the most conservative logging, using `--logverbosity vv` will print everything Balrog has been programmed to possibly log. `args.log.txt` logs the command line arguments passed to Balrog and `sex.log.txt` logs exaclty how SExtractor was called, including all the command line arguments and any output SExtractor printed. `args.log.txt` and `sex.log.txt` include comment lines specifying what is being printed in that section.

The easiest way for users to familiarize themselves with the entirety of `args.log.txt`'s contents is to open the file and have a look themselves. Quickly summarizing, `args.log.txt` contains the exact call Balrog was instantiated with from the command line, the values Balrog initially assumed for each argument (filling in with defaults where necessary), the final parsed value for each argument, and some *pseudo arguments*–parameters Balrog deduces from the command line arguments. Strictly speaking users should not need to understand the pseudo arguments; nevertheless, they may be illustrative for understanding Balrog's workflow or helpful during debugging.

Table 7.3: File structure of the output written by Balrog. * is replaced with the base name of the input image.

- Output Images -

`--outdir/balrog_image/`

`*.nosim.fits`
Copy of (subsampled) input image. Present if `--nonosim` is unflagged.

`*.sim.fits`
Image containing simulated galaxies.

`*.weight.sim.fits`
Copy of (subsampled) weight map. Present if `--image`!=`--weight`

`*.psf`
Copies of (subsampled) PSFEx PSF image for measurement image (and detection image if relevant).

- Output Catalogs -

`--outdir/balrog_cat/`

`*.measuredcat.nosim.fits`
SExtractor catalog measured over original image, prior to simulation. Present if `--nonosim` is unflagged.

`*.measuredcat.sim.fits`
SExtractor catalog measured over image which includes simulated galaxies. Present if `--imageonly` is unflagged.

`*.truthcat.sim.fits`
Truth catalog of the simulated galaxies' properties. Present if `--nodraw` is unflagged.

- SExtractor Configuration Files -

`--outdir/balrog_sexconfig/`

`*.assoc.nosim.txt`
Association mode matching list for SExtractor run prior to adding simulated galaxies. Present if `--nonosim`, `--noassoc`, and `--imageonly` are unflagged.

`*.assoc.sim.txt`
Association mode matching list for SExtractor run with simulated galaxies. Only written if `--noassoc` and `--imageonly` are unflagged.

`${--sexconfig}`
Explicit copy of file given by `--sexconfig`.

`${--sexparam}`
Explicit copy of file given by `--sexparam`.

`${--nosimsexparam}`
Explicit copy of file given by `--nosimsexparam`.

`${--sexnnw}`
Explicit copy of file given by `--sexnnw`.

`${--sexconv}`
Explicit copy of file given by `--sexconv`.

`*.measuredcat.nosim.sex.config`
SExtractor `config` file actually with run prior to simulation. If this file exists, it is closely related, but not necessarily identical to `--sexconfig`. Present if `--nonosim` and `--imageonly` are unflagged.

`*.measuredcat.sim.sex.config`
SExtractor `config` file actually used with run post simulation. If this file exists, it is closely related, but not necessary identical to `--sexconfig`. Present if `--imageonly` is unflagged.

`*.measuredcat.nosim.sex.params`
SExtractor `param` file used during run prior to inserting simulated galaxies. Closely related, but not necessarily identical to `--sexparam`. Present if `--nonosim` and `--imageonly` are unflagged.

`*.measuredcat.sim.sex.params`
SExtractor `param` file used during run including simulated galaxies. Closely related, but not necessarily identical to `--sexparam`. Present if `--imageonly` is unflagged.

- Log Files -

`--outdir/balrog_log/`

`run.log.txt`
Any Balrog warnings, errors, or other messages generated a run. `--logverbosity` applies to this file

`args.log.txt`
Logs the command line parameters given for a Balrog run.

`sex.log.txt`
SExtractor command line input/output.

`${--pyconfig}`
Explicit copy of file given by `--pyconfig`

Bibliography

Bertin E., 2011, in Evans I. N., Accomazzi A., Mink D. J., Rots A. H., eds, Astronomical Society of the Pacific Conference Series Vol. 442, p. 435

Bertin E., Arnouts S., 1996, A&AS, 117, 393

Jouvel S. et al., 2009, A&A, 504, 359

Rowe B. et al., 2014, arXiv:1407.7676

A. Appendices

A.1 SExtractor Comments

Please be aware, that this treatment is not intended to be a comprehensive guide to running or understanding SExtractor. Rather, we simply offer a few comments which will hopefully help run SExtractor successfully within Balrog. For those who have never used SExtractor, we direct you to the official SExtractor user manual or alternatively the so-called Source Extractor for Dummies text.

SExtractor runs are configured via roughly a handful of files. In this scope, the two most relevant ones are given by command line arguments `--sexconfig` and `--sexparam`. `sexparam` files are often denoted with a `.param` extension in their file names, while `sexconfig` files are often denoted with a `.sex` extension in their file names. In the example files shipped with Balrog, the `sexconfig` file is suffixed with a `.config` extension for consistency with our terminology. `sexconfig` is allowed to specify any of the arguments which can also be given as command line arguments to SExtractor. These set conditions such as the detection thresholds, the aperture sizes for photometry, the magnitude zeropoint, and the background subtraction strategy. The `sexparam` file is a list of keywords. Each keyword is a measurement SExtractor will make for every extracted object, which will therefore appear as a column in the output catalog.

It is `sexparam` which controls whether or not SExtractor will perform model fits to the galaxies. How do to this is rather scarcely documented, but has been passed down by word of mouth in the Astronomy community. We consider it worthwhile to elaborate in writing. SExtractor includes up to two possible types of models to fit. One, denoted by the key `DISK`, is a model with fixed Sérsic index of $n = 1$, i.e. an exponential. The other, denoted by the key `SPHEROID`, is a model with a free Sérsic index. SExtractor can fit either of these independently or both simultaneously. Each model written into and uncommented from the `sexparam` file is fit, meaning if just the `DISK` key is present a disk only model with $n = 1$ is fit; if just the `SPHEROID` key is present a bulge only model with free n is fit; and if both `DISK` and `SPHEROID` keys are present both the disk and bulge are fit simultaneously, which is of course different than fitting them independently. Each model can fit for the flux, magnitude, axis ratio, and orientation angle of the model. The spheroid model also fits the sersic index and half light radius. The disk model fits the scale radius, as opposed to the half light radius. Open the `bulge+disk.param` file included with Balrog in the `astro_config` directory for the SExtractor names of all the parameters. The meanings of the `DISK` and `SPHEROID` names are human understandable.

A.2 OLD SECTION: What Is Balrog Good For?

No astronomical imaging survey is perfectly homogeneous. Variations in image quality and depth are a necessary result from any multi-epoch measurement, even without variations in weather, sky brightness, or telescope properties that can drive inhomogeneities within a single epoch [REF: e.g. Holmes, Hogg, & Rix 2012]. Effective use of imaging data requires a good model for what happens to the data during processing, but the process of creating a catalog from an image generally is generally nonlinear[1], and the relationship between the resulting data products and the world of forms can be quite difficult to model analytically or even heuristically [e.g., SDSS sky subtraction, see REF:Aihara paper, or any deblending algorithm ever].

Our preferred method for characterizing this relationship is the likelihood function:

$$L = p(\text{Catalog}|\text{world of forms}) \tag{A.1}$$

. for given data reduction and catalog-making recipes. This would be necessary for a fully Bayesian imaging analysis (which is probably presently impractical for any large imaging survey), but even for a catalog of point estimates of object properties, L is useful for characterizing the noise and bias of the estimators.

[1]Thresholding does this, and no matter what else you do, all catalog-making requires thresholding.

We provide three examples of contemporary relevance below. For each of these, we provide example pseudocode that shows how to compute the quantity of interest using BALROG.

A.2.1 Completeness

The most common first step in any procedure (REF: SExtractor, PHOTO, maybe THELI?) for creating a catalog from an image is a matched filtering of the image followed by a thresholding. Typically, the filter is chosen to be similar (in the broadest sense of similarity) to the expected image psf [REF: Photo-lite, SEXtractor, etc.]. The resulting detection map is thresholded, and the pixels that exceed the threshold become the focus of subsequent analysis.

For a surface-brightness profile of the same shape as the filter, the matched filter corresponds to the optimal[2] measurement of the flux of an object at that position, so thresholding the detection map corresponds to an actual flux limit. For other surface-brightness profiles, however, this is not true, and any catalog-making process that begins with this step cannot be said to be "magnitude-limited" or "flux-limited". This is a problem, because the completeness of most astronomical imaging is frequently characterized with a single number, corresponding to the magnitude of the faintest object in the catalog; occasionally, conscientious authors will record a significance level (corresponding to the threshold) or that the quoted limiting magnitude is appropriate for either point-sources or galaxies.

Our preferred method for describing the survey depth is to compute the probability of detection for the ensemble of sources that will be subjected to analysis. BALROG makes this step straightforward: by embedding a catalog of objects in the real images, and then re-running the catalog-making pipeline, it is easy to determine the detection probability for any objects of interest as a function of their underlying properties.

A.2.2 Clustering

Two-point clustering measures are among the most common intermediate science products of large astronomical surveys. Variations in the effective survey depth across the sky can produce large systematic errors in the clustering measures (e.g., [REF:]Ross et al 2012, [REF:]Huterer et al 2011). This is universally modeled in such studies by a window function $W(\bar{\theta})$, which is proportional to the probability that a galaxy in the sampled population would have been detected at the position $\bar{\theta}$.

The observed galaxy overdensity field $\delta_g^{\mathrm{obs}} = \frac{n_g}{\bar{n}} - 1$ is then related to the "true" density field (for the sample being selected) as:

$$\delta_g^{\mathrm{obs}}(\bar{\theta}) = \delta_g(\bar{\theta})W(\bar{\theta}) \tag{A.2}$$

and the galaxy autocorrelation function $w(\theta)$ is:

$$w(|\theta|) = \left\langle \delta_g(\bar{\theta}', \delta_g(\bar{\theta}' + \bar{\theta}) \right\rangle \tag{A.3}$$

Here, the expectation values are meant to signify averaging over independent realizations of the universe. Ergodicity allows us to replace this with a volume average. If we assume that W is independent of the choice of realization, then we can form an unbiased (though sub-optimal; see [REF:Landy-Szalay], though the choice is irrelevant for this argument) estimator for w, $\hat{w}$:

$$\hat{w}(\theta) = \frac{\left\langle \delta_g^{\mathrm{obs}}(\bar{\theta}), \delta_g^{\mathrm{obs}}(\bar{\theta}' + \bar{\theta}) \right\rangle}{|W|^2}. \tag{A.4}$$

For high-precision galaxy surveys, however, we cannot ignore the dependence of the detection probability W on δ_g. Others have presented evidence for that this is significant in Sloan Digital Sky Survey imaging [REF: Aihara et al; Ross et al; Huff & Graves], but because of the basic mechanics of detection and catalog-making, similar effects are virtually certain to be present in any large imaging survey.

[2] Really optimal, in the sense that it saturates the Cramer-Rao bound, and so a lower-variance unbiased estimator is mathematically impossible.

To see why this may be important, consider the case where W has a linear dependence on δ_g, $W = W_0 + W_1\delta_g$. If this is ignored, and only the static survey window function is known (i.e., we assume $W = W_0$) then the estimator $\hat{w}$ is biased:

$$\hat{w}(\theta) = \frac{\left\langle \delta_g^{\rm obs}(\bar{\theta}), \delta_g^{\rm obs}(\bar{\theta}' + \bar{\theta}) \right\rangle}{|W_0|^2} \tag{A.5}$$

$$= \left\langle \delta_g(\bar{\theta}), \delta_g(\bar{\theta}' + \bar{\theta}) \right\rangle + \frac{W_1}{W_0} \left\langle \delta_g^2(\bar{\theta}), \delta_g(\bar{\theta}' + \bar{\theta}) \right\rangle + \left(\frac{W_1}{W_0}\right)^2 \left\langle \delta_g^2(\bar{\theta}), \delta_g(\bar{\theta}' + \bar{\theta})^2 \right\rangle \tag{A.6}$$

There is no static window function W_0, and hence no way to use the standard technique (for correlation functions) of cross-correlating with random catalogs (i.e., DD/RR-1, or (DD-2DR+RR)/RR, etc.) to estimate $w(\theta)$.

We can, however, attempt to re-weight $\delta_g^{\rm obs}$ by an estimate of the δ-dependent detection probability, thus keeping the window function inside the ensemble-average, and forming a new estimator:

$$\hat{w}(\theta) = \left\langle \frac{\delta_g^{\rm obs}(\bar{\theta})}{W(\bar{\theta}|\delta_g)}, \frac{\delta_g^{\rm obs}(\bar{\theta}' + \bar{\theta})}{W(\bar{\theta}' + \bar{\theta}|\delta_g)} \right\rangle \tag{A.7}$$

We now show how to use BALROG to estimate $W(\bar{\theta}|\delta_g)$.

A.2.3 Optimization

For gaussian noise with covariance matrix C, an observable vector $\bar{p}$ (such as measured properties for a single entry in a galaxy catalog), and a signal $\bar{s} = \alpha\hat{s}$ (where $\hat{s}$ is a unit vector), the maximum-likelihood estimator for α is:

$$\alpha = \frac{\bar{x}^T C^{-1} \hat{s}}{\hat{s}^T C^{-1} \hat{s}} \tag{A.8}$$

TODO: Demonstrate what you can do if you know the measurement covariance matrix.