Galaxy Modelling and Simulations Using GALTHON: Manual and Documentation

By Riley Peterson

Gemini Research Intern September-December 2017

Supervisors:
Inger Jorgensen and Kristin Chiboucas

Table of Contents

Introduction

The goal of this document is to explain how to use two Python scripts created during my 4 month internship at the Gemini Observatory in 2017. One of the scripts is used to create galaxy simulations, while the other is used to run several programs (SExtractor and GALFIT) to model structural/photometric properties of galaxies within an image. The primary motivation for creating the later is that although there were tools in place to run SExtractor and GALFIT, their inner workings were complicated and not always performing in the way a user suspected they would. Additionally, I was motivated to create the latter to more easily evaluate the simulations, which was a major goal of my internship.

The first part of this document will explain the script used to produce galaxy models with GALFIT. This script will henceforth be referred to as **do_everything.py**. This script essentially follows the process outlined in GCP_2D_Photometry.pdf, which is a great reference. I will describe the preparation for GALFIT in section 1 and the running of GALFIT in section 2.

In the second part of this document (section 3), I will explain the script used to create simulated galaxies. This script will henceforth be referred to as **make_sims.py**.

Both scripts are written in Python 2.7[1] and this manual implicitly assumes a very basic knowledge of the Python programming language. In addition, some working knowledge of IRAF/Pyraf is assumed.

Background

Across many research studies in astronomy it is often an important scientific goal to evaluate the properties of galaxies within a given image. These properties include attributes such as a galaxies position, brightness, and size. To facilitate this process two programs have been developed (among others) SExtractor[2] and GALFIT[3]. In the capacity of this manual, SExtractor is used to quickly generate estimates of galaxy properties and GALFIT is used to fit two-dimensional surface brightness profiles to galaxies. The most common profile used by GALFIT is the Sérsic model given by:

$$\Sigma(r) = \Sigma_e \exp[-\kappa((\frac{r}{r_e})^{\frac{1}{n}} - 1)]$$

where *Σ(r)* and Σ_e are the pixel surface brightness at an arbitrary radius and at the effective radius of the galaxy, respectively. The effective radius is r_e, *n* is the sersic index and κ is a constant related to *n* (see Ciotti and Bertin[4], 1999). This formula is equation 2 from the GALFIT manual[5]. Disk galaxies typically have n ≈ 1, while ellipticals have n ≈ 4 [references]. When a galaxy is fit with this model (or any model for that matter) the goal of GALFIT is to minimize the reduced chi squared:

$$\chi_v^2 = \frac{1}{N_{DOF}} \sum_{x=1,y=1}^{nx,ny} \frac{(f_{data}(x,y) - f_{model}(x,y))^2}{\sigma(x,y)^2}$$

Where N_{DOF} is the degree of freedom and nx and ny are the x and y image dimensions, respectively. Again, this is from the GALFIT manual[5].

The image inputs for GALFIT are an image to be fit, a mask of faint objects in the image (not to be fit), a sigma image, and a point spread function (PSF). The most important input for GALFIT is the "feedme" or input file, this file provides paths to input images, information about the image, and initial guesses for galaxy parameters. GALFIT produces three output images: the original image, a model of the profile fit, and the residual (model - original) given in extensions [1], [2], and [3] of the output. The parameters of the model (e.g. x, y, magnitude, effective-radius, sersic index, axis ratio, and position angle for a "sersic" fit) are given in the header of the model extension and can be accessed by the following using iraf:

```
cl>imhead your_output_name_here.fits[2] l+
```

The closer that a galaxy is to truly representing the model fit the lower its reduced chi squared will be and the smaller its residual will be (generally). Galaxy modelling is a very complicated subject and GALFIT has many nuisances, even after roughly a year total using GALFIT I still am learning new things every time I use it. I will surely explain more as this manual progress. Hopefully, this has served as the most basic of introductions for someone who is wholly unfamiliar. I direct the reader to the GALFIT manual[5] for more information/background.

One of the outstanding questions often associated with using GALFIT is concerned with the errors produced by its output. To quote the GALFIT FAQ[6]: "Getting realistic errorbars from image fitting is often a tricky business." Personally, I find that when an error is too high or too low it usually indicates a fit went wrong. In order to better evaluate the errors on GALFIT measurements I have been tasked with creating a Python wrapper to create simulated data. Using these simulated galaxies, it is possible to determine the errors between the GALFIT models and simulated models. It is expected that simulated model errors from GALFIT will underestimate the error for real galaxies. This is because simulated models are single component perfect analytical models (with noise), whereas more complicated structures/irregularities exist in real galaxies which would result in larger errors. Perhaps, creating two component models or more complicated models will be a future feature added to make_sims.py. For now, the goal of make_sims.py is to create one component models and see how close the output parameters from GALFIT are to the actual value (i.e. find the error on GALFIT derived parameters). It is also a goal to test how the proximity to neighboring galaxies of varying size and luminosity affects the fitted parameters.

Note about Noise

At this point (December 19th, 2017), it is not clear if noise is being accounted for in the correct way for both the galaxy fitting processing and, by extension, the simulation software. The dark current is ignored in both these scripts, though this may prove to be a significant source of noise. Once a more thorough understanding of the noise sources and their significance is developed, these scripts should be modified to correctly account for all the sources of noise. I believe the full equation for sigma ought to be:

sigma = sqrt((1/IVM) + SCI/EXP)
where:
IVM-Inverse Variance Map [s/e-]^2 (seems this can be output from astrodrizzle as well)
SCI-Drizzled image in [e-/s]
EXP-Exposure time map output from astrodrizzle [s]
where:
IVM = (f*t)^2 / ((D + f*B) + σread^2)
where f is the inverse flatfield (as defined in the conventions for the flatfield reference files used in calibration), t is the exposure time (in seconds), D is the total accumulated dark current signal during the exposure, B is the total accumulated background level during the exposure, and σread is the read-out noise, with all three of the latter quantities being in units of electrons. This IVM equation is from Koekemoer et. al 2011 [7] (Section 5.8.1).

Quick Walkthrough

I realize this manual may be intimidating at first glance so here I put forth a quick walkthrough of how these scripts can be used. In general, the script has user options which are to be "yes" when the desired task is to be run. For example:

```
make_EXPTIME_1 = "yes"
make_ncomb="yes"
make_weight="no"
make_sigma="no"
.
.
.
```

will change the EXPTIME keyword for images to be 1 and create the ncomb image. The tasks are listed chronologically. As a good rule of thumb, to run a specific task the preceding tasks should, <u>at some point</u>, have been set to `"yes"`. As an example, it would not be possible to run GALFIT (`do_run_galfit`) without having made ran SExtractor (`run_sex1`) because the SExtractor output is used to create feedme files for GALFIT. However, the scripts are not strictly dependent on previous tasks being "yes". I mean that the following is possible:
First run:

```
    make_EXPTIME_1 = "yes"
    make_ncomb="yes"
    make_weight="no"
    make_sigma="no"
```

Second run:

```
    make_EXPTIME_1 = "no"
    make_ncomb="no"
    make_weight="yes"
    make_sigma="yes"
```

Because the preceding tasks have occurred and their products exist (the can be set to `"no"` and subsequent tasks can be run). Furthermore, most tasks have certain parameters that need to be specified. <u>Before running a task, you should check if the task requires input parameters and edit them accordingly</u>. The input parameters are listed below the possible tasks (Scroll down within the script). Example the input parameters for `make_assoc` are:

```
    ### Making association file inputs
    assoc_mag_cut = 24.2 ### MAG cut for association file
    assoc_noise_cut = 0.04 ### MAG_ERR cut for association file
    remember MAG_ERR=0.04 corresponds to S/N~25
    assoc_class_star_cut = 0.5 ### CLASS_STAR cut for association
    file
```

do_everything.py

1. Obtain your science image (drz image), weight map (wht image), and context image (ctx image).
2. Make a directory (base_folder) and put the images in this directory.
3. Set the variable base_folder to this path and rename the drz, wht, and ctx images.
   Examples:
   `base_folder = "/path/to/your/base_folder/`**`fieldname`**`"`
   `drz_image = "`**`fieldname`**`_drz.fits"`

```
wht_image = "fieldname_wht.fits"
ctx_image = "fieldname_ctx.fits"

base_folder="/rpeterson/FP/GALFIT_Simulations1/galfitsim1"
drz_image = "galfitsim1_drz.fits"
wht_image = "galfitsim1_wht.fits"
ctx_image = "galfitsim1_ctx.fits"
```

4. **`Fieldname`** or **`galfitsim1`** will prefix almost every output from do_everything.py. In this guide, it is referred to as the base name. *The base name part* ***cannot*** *contain any "_" characters. The base_folder should not have a trailing slash "/".* In the future these will be automatically corrected.
5. Now we are ready to run the script. You will probably want to make the exposure time of your images 1. This is because GALFIT and SExtractor use this to determine the magnitude. If so, set `make_EXPTIME_1 = "yes"`. It is possible to have all subsequent tasks set to `"yes"`, but I would not advise this since you probably do not yet know parameters like `assoc_mag_cut` or `mask_maglim`.
6. Then you will want to make the weight and sigma image. To do this set:

```
make_ncomb='yes'
make_weight='yes'
make_sigma='yes'
```

You can set `make_EXPTIME_1 = "no"` since this has already occurred (though if it is still yes the script will still run fine). The first of these three tasks will make an image (ncomb.fits) using the context image. This image represents the number of stacked images which were combined for each individual pixel.

The second of these makes the weight image. The weight image uses the ncomb image and the wht image. Here, you will want to set the parameters `nmin` to an appropriate value. This is the minimum number of stacked images needed for a pixel to be considered (a typically value might be 2 or 3, but it depends on how many images are stacked). If a pixel has `nmin` or more images contributing to it, it takes on the value of the wht image, otherwise it takes on the value zero (has no weight).

The third of these makes the sigma image. The sigma image uses all the previous images. It is made according to the following formula:

$$\sigma_{norm} = F_A^{1/2}\left[N_{comb}\left(\frac{ron}{normalization}\right)^2 + \frac{signal_{output}}{normalization}\right]^{1/2}$$

Here F_A is the variance reduction factor (see Casertano[8] et al. 2000) and *ron* is the read noise. To determine these, you need to access the image header information in a similar fashion:

```
--> imhead galfitsim1_wht.fits l+ | grep "READNSE"
READNSEA=        2.0200001E+01 / calibrated read noise for amplifier A
```

You should set `read_noise` to this value. [Recent findings suggest this may not be correct. The sigma image formula may be more complicated or the read noise from the image header may need to be converted to an effective read noise].

```
--> imhead galfitsim1_wht.fits l+ | grep "ISCL"
D001ISCL=             0.12825 / Drizzle, default IDCTAB pixel size(arcsec)
--> imhead galfitsim1_wht.fits l+ | grep "SCAL"
D001SCAL=                0.06 / Drizzle, pixel size (arcsec) of output image
--> imhead galfitsim1_wht.fits l+ | grep "PIXF"
D001PIXF=                 0.8 / Drizzle, linear size of drop
```

Using this information, you can find the variance reduction factor. See page 22 of the GCP_Photometry guide. Set `sqrt_FA` to this value.

```
#real_pixelscale = 0.06/0.12825 = 0.467836
#F_A^0.5 = (0.467836/0.8)*(1-(1/3)(0.467836/0.8)) = 0.4708
```

Lastly, you will need to set the value for `norm_factor`. This is typically the number of stacked images. This number is multiplied by the weight image to use as the normalization in the equation above. This is necessary if the values of the weight image generated in the step before are **not** approximately equal to the original EXPTIME (stored in important_parameters.txt).

```
--> imhead galfitsim1_wht.fits l+ | grep "NDRIZ"
NDRIZIM =                   4 / Drizzle, No. images drizzled onto output
```

Now you should run the script (with all subsequent other user options set to "no") and check the output. A good check is that the standard deviation of an area of sky in the original image should be roughly the same as the sigma image values (their mean).

7. The next step is to run SExtractor. This is accomplished by setting `run_sex1='yes'` (you can now set previous user options to "no" if you would like). You will want to go through the SExtractor variables:
```
ANALYSIS_THRESH = "2"
BACKPHOTO_THICK = "24"
BACKPHOTO_TYPE = "GLOBAL"
```
Etc.
Gemini interns see Section 5.3 of the GCP_Photometry_Guide Table 4 (the .param file should contain attributes in Table 6 of Section 9.1, but use MAG_AUTO not MAG_BEST). For reference here is where the default.sex file is on my system:

/Users/rpeterson/anaconda/pkgs/sextractor-2.19.5-0/share/sextractor/default.sex

"ASSOC" parameters should not be in the param file or defined as variables since SExtractor is not run in association mode in this context. Other SExtractor parameters may be added such as MAG_APER, but the bare minimum are the parameters listed in Table 6 Section 9.1 (also in § 1.4). If you are unsure about which parameters to put in the param file, I suggest only using the ones in Table 6 Section 9.1. The GAIN should be EXPTIME*CCDGAIN instead of EXPTIME.

If you have a SExtractor file which you would like to go off of, but feel it is a hassle to write the variables into the do_everything.py script you can have the variables printed in the appropriate format by setting `print_vars="yes"` and supplying `path_to_sex_config`, which is the path to the configuration file you will base your parameters on. This is purely meant to help the user, so they do not have to manually enter all the parameters in, instead the user can paste the printed values into the script and edit them accordingly. It will cause the script to exit such that subsequent tasks won't be run. SExtractor variables must be capitalized in order for them to be recognized by the script, no other defined variables should be capitalized. When you are ready to run SExtractor you can then set `print_vars="no"` and proceed to run the script.

8. Now set `make_assoc='yes'`. This will create an "association file" which tells GALFIT which galaxies to fit. These galaxies will have a non-zero id number in the output cat file. When you run this, the aperture image from SExtractor will display and two graphs will be shown. Looking at these graphs you need to determine the following for your sample:
   ```
   assoc_mag_cut = 24.5
   assoc_noise_cut = 0.04
   assoc_class_star_cut = 0.5
   ```
   The assoc_noise_cut should probably stay 0.04 (mag_err), this corresponds to a S/N of ~ 25 (Gemini interns see section 5.6.1 of Ryan Cole's write up). The value of assoc_mag_cut should be where the assoc_noise_cut first intersects your galaxy sample (such that you only select object with S/N > 25). The assoc_class_star_cut should be placed in between stars and galaxies. Everything with mag<assoc_mag_cut and class_star>assoc_class_star_cut should be a star. This discrepancy is typically blurred at faint magnitudes but it normally does not matter because only magnitudes with mag<assoc_mag_cut are considered.

   Everything with mag<assoc_mag_cut and class_star<assoc_class_star_cut and mag_err< assoc_noise_cut is considered a target galaxy and will be fit by GALFIT. When you have decided adequate cutoff for these values, you should run `make_assoc` again. The resulting output is an _ALLsex.cat file where target galaxies have non-zero ids (first column). If the user wants to fit a specific sample, this would be a good place to match coordinates and the user could set non-sample members to have id of zero.n

9. Now set `make_mask='yes'`. This will make the mask image. At first, `make_mask_name` and `final_mask_for_galfit` should be `"default"`. You should also determine `mask_maglim`.

   ```
   mask_maglim = min(magbright+6.2, magfaint+0.5)
   ```

   where magfaint and magbright are the max and min magnitudes of your SExtractor sample respectively. This suggested mask_maglim is printed at the end of make_assoc.

10. Now set `make_sex_sky='yes'` and `make_sky_plots="yes"`. This will make plots of the sky values and produce a list of iterstat values (_itersky.cat). Typically, I close the plots made and just look at sky_results.pdf. Go through each id (for which there are two graphs; see GCP_Photometry_Guide 9.4) and decide which action to take with the sky. An example:

    ```
    change_to_iterstat_list=[12,43,56]
    ```

```
take_average_list=[34,65,782]
remove_entirely_list=[]
```

Depending on the action you want to take (whether it be changing to the iterstat value, taking the average, or removing the galaxy entirely from the catalog), add the galaxy id to the appropriate list. Now set `make_sky_plots="no"` and run again. This will ensure the changes in lists above will be made. This _ALLsex_final.cat output will be used to create the input files for GALFIT.

11. I typically do not make a constraint file. I might recommend using your own devices to make the PSF (point spread function), however a basic PSF can be made by setting `make_psf='yes'`. This roughly follows the routine put forth here:
    http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?psf
    You can set the necessary iraf digiphot parameters in the PSF inputs section of the script. See the comments within the script to understand these variables. It is difficult to automate the PSF process. Your PSF(s) should be in the 2D folder within a folder named "PSF". Their filename should be psf-x-y.fits, where x and y represent coordinates for your PSFs. If you only have one PSF these coordinates don't really matter. Otherwise, the closest PSF is selected for each object.

12. Now it is time to run GALFIT. Set `do_run_galfit="yes"`. Currently, you ought to have the following:
    ```
        changes_type="local"
        use_output_as_input="no"
        method="stop_and_go"
    ```
    These may be expanded upon if time permits; they represent the different ways to run GALFIT. The first thing to do is to make the cutouts, this is done by setting:
    ```
        make_cutouts="yes"
        cutouts_list_to_make="default"
    ```
    Running this will now create all the cutouts within the galfit-input-cutouts folder. Once cutouts have been made the next step is to create the feedme files for GALFIT. Set `make_feedmes_sersic1="yes"` and `n_start1=1.5` (traditionally 1.5 is the starting Sersic index, though the user may select any value they wish).
13. Now feedme files can be sent to GALFIT, set `run_sersic1="yes"`. If the user would like to view residuals as they are made set `display_blocks_sersic1="yes"`. This feature will display the outputs at an interval (in seconds) given by `timer` (20 is a reasonable value). By opening the fit log and using the display blocks method, the user can get a head start on the refitting process.
14. The user needs to evaluate which objects should be refit. By using:
    ```
    check_residuals_sersic1="yes"
    check_residuals_sersic1_list="default"
    ```
    the user can view all the residuals. If the `check_residuals_sersic1_list` is provided only those id's will be checked, if "default" all will be checked. `plot_stats_sersic1="yes"` can likewise be used to spot outliers and compare galaxy parameters. When compiling the list of refit galaxies, it is good practice to take note of why it needs refitting.

15. Once a refit list has been established, the user can provide this list as input to `run_refits_sersic1_list`, these galaxies will be copied to a new folder where the refitting process will occur. Set `run_refits_sersic1="yes"` to initiate this process. Now the user should change directory (via the terminal) to this refit folder. Then run GALFIT at the command line making changes to the obj.in file as needed until the object is successfully refit. Once this is done press "n" to move on to the next galaxy. Read the next step.
16. As part of the refit process it is important to document the changes made to refit objects. This documentation takes place in the refits_summary.txt file. The typical format for this file is:
    1. A comment with the galaxy number and reason for refitting. Examples:
       "#21, bad residual"
       "#217, got nan values for errors"
    2. Action taken. Options are:
        a. Change starting value of a parameter (id parameter old new):
            i. "25 mag1 22.59098816 21"
            ii. "294 type3 sersic psf"
            iii. "565 q7 0.9876 0.5"
        b. Add a new component (similar to longfeedme format):
           "id8 75
           row8 1000
           type8 sersic
           x_global8 1012.54656982
           y_global8 2378.69506836
           x_start8 887
           x_end8 1136
           y_start8 2253
           y_end8 2502
           x_cutout8 133
           y_cutout8 117
           mag8 24
           re8 3
           n8 1.5
           q8 0.61750931
           pa8 -152.98328781"
           Make sure that if a cutout has seven components in the longfeedme (i.e. its last component is id7, row7, type7, etc.) then the added component should be plus one (i.e. id8, row8, type8, etc.), else you would overwrite a previous component instead of adding.
        c. Deleting a component or object ("delete" id component_number):
            i. "delete 297 1"
            ii. "delete 405 3"
            Deletions within the same id should have descending component numbers:
            "delete 300 4"
            "delete 300 3"
            Deletions should always appear after changes and additions.

3. A comment about the resulting fit. Examples:
   "#Residual still not great, but errors make sense"
   "#Error is reduced, residual looks better"

   If the user needs to mask an object see § 1.7, § 1.7.1 (if object not in original segmentation map), and § 2.7.1. After remaking the **necessary** cutouts, these galaxies can be refit with the new masked cutout.

17. Once changes have been documented in the summary file and all the needed galaxies are adequately refit, these changes need to be enacted on the longfeedme file. Changes are made locally so as not to affect other cutouts. This is done by setting `enact_refit_edits_sersic1="yes"`. This will create a new longfeedme file (longfeedme_w_refits.txt) with the appropriate changes, please double check to make sure your changes (especially if complicated) were enacted correctly.
18. Lastly, original fits and refits are combined in a final folder. Set: `combine_refits_and_original_to_final_folder_sersic1_and_make_table="yes"`
    This will place original and refitted galaxies into the final folder and make a table of outputs (fits and txt format). It will also copy the refit feedme and rename it final feedme.
19. Currently, the final feedme is used from the sersic1 fit as input for the devauc1 fit. For the devauc1 fit, start by making the feedmes (you need not remake the cutouts) and then repeat the steps 13-18, similarly.
20. Sersic2 will refit galaxies with a better devauc reduced chi square compared to the sersic1 fit and a sersic1 output n value > 3 with a starting n value of 4 instead of 1.5 (typically). If the fit improves from the sersic1 fit, the galaxy id should be added to `use_sersic2_list`. If a galaxy should not appear in the final catalog (typically when a reasonable fit cannot be produced by the Sersic type) then it should be added to `remove_from_final_catalog_sersic2_list`. Once `combine_refits_and_original_to_final_folder_sersic2_and_make_table` is run, the entire process is complete.

make_sims.py

1. This portion is likely to be outdated over the next few weeks, but in general this process is as follows:
2. Decide how many simulations to create, the places where noise should be added (to individual cutouts, before adding to the full image, or after), if convolution should be performed, how galaxy parameters are selected (interpolated or randomly), etc. I believe the flow charts I have made are helpful in understanding all the options.
3. If I were to create a large field with 800 galaxies I would do the following:
   a. Give a name for the `base_folder`, set your `login_cl_path`, supply a `psf_image` if convolution is to be performed, give the `galfit_binary` (if you can run galfit at the terminal with "$galfit feedme" then this should be "galfit")
   b. Set `make_feedmes = 'yes'` and set the ranges of parameters:
      ```
      large_field_xdim,large_field_ydim = 3133,3038
      mag_range = [18,26]
      re_range = [2,40]
      n_range = [0.5,5.5]
      q_range = [0.05,1.0]
      ```

```
pa_range = [-90,90]
number_of_sims = 800
cutout_size = 400
sky_range = [0,0]
mag_zp = 25.9463
universal_seed = 6061944
```

c. I would set `make_interpolate_chain = "yes"` and use the following for my chain data:

```
    chain_type1 = "cubic"
    chain_plot1 = "yes"
    chain_bins1 = 40
    chain_samples1 = 10000
    chain_data1 =
"/Users/rpeterson/FP/do_everything/UV105842F160W/UV105
842F160W_phot/UV105842F160W_tab_all.fits"
    chain_column1 = "MAG_AUTO"
    chain_obj1 = "mag"

    do_chain2 = "yes"
    if do_chain2=="yes":
        chain_type2 = "linear"
        chain_plot2 = "yes"
        chain_data2 =
"/Users/rpeterson/FP/do_everything/UV105842F160W/UV105
842F160W_phot/UV105842F160W_tab_all.fits"
        chain_column2 = "FLUX_RADIUS"
        chain_obj2 = "re"
```

Run this and see if the graphs produced are realistic to your real data.

4. Set:

```
make_galaxies = 'yes'
convolve = 'yes'
mknoise_cutouts = 'no'
display_5 = 'yes'
create_large_field = 'yes'
image_to_sim = "/path/to/image/you/want/to/simulate.fits"
image_to_add_to= "default"
sky_value = 0
add_mknoise_large1 = "no"
add_models = "yes"
label = "yes"
save = "no"
which_to_add="_convolved"
how_many_to_add = 800
add_mknoise_large2="yes"
if add_mknoise_large2=="yes":
    background2 = 0.136392
    gain2=6529.37744
    rdnoise2=2
    poisson2="yes"
compare_sky="yes"
```

Here `image_to_sim` is used in order to determine the possible coordinates for galaxies (my images were rotated so not all coordinates could contain galaxies). If you would like a regular rectangular image you can set `image_to_sim="default"` and it will make an image with size given by `large_field_xdim, large_field_ydim`.

To add simulated galaxies to an existing image simply change `image_to_add_to` to the path of your image (a copy will be created within `base_folder` and galaxies will be added to the copy). I would also change `how_many_to_add`.

If comparing sky, originally set the coordinates to `None` and you will be prompted to supply coordinates once the image has been created, then its easiest to find a sky region which is common to `image_to_sim` and the final simulated image, type in these coordinates and, when prompted, make the simulated coordinates the same by typing "y". If you know the coordinates beforehand you can change `None` values to the coordinates.

Do_Everything.py

What does do_everything.py do?

This script takes a drizzled image (drz_image), weight image (wht_image), and context (ctx_image) and uses them to produce a table of GALFIT model parameters. Currently, this script is designed for WFC3 F160W images and single component fits, however in theory it should work for many more types of images and with some modifications accommodate double component models.

What is required to run do_everything.py?

If you are a Gemini intern you already have everything that is required. Currently, do_everything.py runs on Mac/OSx (it may work for Linux systems, but it will not for Windows). You need Python 2.7 installed on your system. I use the Spyder IDE which comes with the conda distribution. You need the following imports many of which come with astroconda:

Astroconda: http://astroconda.readthedocs.io/en/latest/

```
import os
from pyraf import iraf
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from astropy.table import Table
from astropy.io import fits, ascii
import time
import numpy as np
import random
import subprocess
import sys
```

You should also be able to load `iraf.stsdas()`.

From pyraf import iraf: Before anything I would make sure that you can successfully run `from pyraf import iraf` with the IDE or distribution of Python which you are using. If you cannot, this most likely has to do with your Python path (where Python looks for modules to load). I have a virtual machine (Mac OS) on my personal computer and after entering the astroconda environment ($source activate astroconda) and then entering Python ($python) I had no problems (>from pyraf import iraf). For reference the path to iraf on my system is:
/Users/rpeterson/anaconda/envs/astroconda/lib/python2.7/site-packages/pyraf/iraf.py
You may need to add this *folder* to your Python path. The idea in the script is that it changes directory to your login.cl folder and then loads iraf, so that whatever iraf settings you may have are also loaded.

You also need to install GALFIT 3.0+ and SExtractor (Gemini interns already have these programs installed):

GALFIT: https://users.obs.carnegiescience.edu/peng/work/galfit/galfit.html

SExtractor: https://www.astromatic.net/software/sextractor

Where is do_everything.py?

The most current stable versions of do_everything.py is here:

```
/Users/rpeterson/FP/GALFIT_Simulations1/galfitsim1_python_scripts/most_curren
t/do_everything_v0.py
```

The version I used for my trial field which may serve as a helpful example is here:

```
/Users/rpeterson/FP/UV105842F160W/do_everything_v0.py
```

The official documentation/README for how I fit my trial field is here:

```
/Users/rpeterson/FP/UV105842F160W/A_Complete_Guide_to_fitting_UV105842F160W.d
ocx
```

Known Limitations

Errors/Bugs

This script has not been thoroughly tested for errors and bugs. With this in mind, please do not purposefully try to break the script. It is designed for the uses put forth within this manual. If you run into errors (especially colossal ones) feel free to contact me. Fortunately, Python is relatively easy to debug and hopefully most errors can be easily repaired by the user.

PSF

While do_everything.py has the option to create a PSF using the IRAF package daophot, this feature is not very robust. Often there is significant manual work that goes into making a PSF that is difficult to automate within a script. However, if a user already has a PSF or several PSFs they can be used just the same (see Inputs and Conventions).

Fit types

Currently do_everything.py only recognizes 2* GALFIT fit types: "sersic" and "devauc". Any other fit is not guaranteed to work. As stated above, it currently only handles 1 component fits.
*stars within an image can be fit using the "psf" function.

Inputs and Conventions

General Naming Conventions

When I refer to an object its variable or output name will typically be placed in parenthesis immediately following the object reference. For example, "the base folder (*base_folder*) is the place where do_everything.py creates all outputs." This is particular important for the weight images for which there are the (*wht_image*) and (*weight_image*) variable names and inputs/outputs (wht) and (weight). To distinguish between variables and outputs, italics are used for variable names. Besides this, I have tried to give appropriate names and use clear Python standards. When I refer to code in particular I will use a `special font`.

As stated above the base folder (*base_folder*) is the folder where all output is created. For example:

```
base_folder="rpeterson/here/is/the/path/UV105842"
```

The most important part of this variable name is the part after the last slash. This part is known as the base name (*base_name*) and will be referred to as such throughout this document. The base_name for the example above is UV105842. The base name (*base_name*) cannot contain an underscore "_" symbol. This is no longer a huge deal, because the code will fix this automatically. This is because the python method `string.split("_")` is used during the GALFIT run. The base name (*base_name*) will precede the name of almost every output (e.g. UV105842_mask.fits). Therefore, it is common for me to use (mask) or the appropriate shortening as I did above with (wht) and (weight) when referring to an output object.

Inputs
Initially the base folder should contain only 3 objects: drizzled image (*drz_image*), weight image (*wht_image*), and the context image (*ctx_image*). These variables should be defined as strings in the following way:

```
drz_image =”UV105842_drz.fits”
wht_image =”UV105842_wht.fits”
ctx_image =”UV105842_ctx.fits”
```

Their file names should be named likewise.

Subdirectories
One of the first things that do_everything.py does is create the subdirectory structure. Three folders will be created within the base folder (*base_folder*) “UV105842_imaging” (*imaging*), “UV105842_phot” (*phot*), and “UV105842_2D” (*two_D*). You don’t need to make these folders. Appropriate outputs will be stored in each of these folders (see the Flow Chart for more insight into what goes where).

Running a particular task
At the very top of do_everything.py are all of the automated tasks which can be performed. These tasks start with a verb like “make” or “run” such as (*make_mask*). To run a task, you must first give do_everything.py the appropriate input variables. For example, to make the mask for an image you should scroll down to:

```
 ### Mask inputs
mask_maglim = 24.61 ### maglim = min(magbright+6.2,magfaint+0.5), very important
add_to_mask_list=[227,230,233,403,594]
remove_from_mask_list=[]
```

There you will find several important parameters which need to be used as inputs to run the task. Most of these parameters have comments associated with them to give more information about what they are. You will need to edit these parameters for your individual situation, however this manual should offer some guidance in selecting the appropriate values. Finally, to create the mask image simply set:

```
make_mask=”yes”
```

then run the script. It is important to note that tasks are ordered in chronological order, such that running (*make_sigma*) will not execute properly unless (*make_weight*) has already been executed. However, if you want to remake the sigma image and already have a perfectly good weight image and ncomb image, then it is totally fine to have:

```
make_ncomb='no'
make_weight='no'
make_sigma='yes'
```

In this sense, the individual tasks have been designed to be as independent as possible. In addition, several tasks can be set to “yes” and they will run sequentially.

Section 1: Preparing for GALFIT

This section will describe going from the original drizzled, weight, and context image all the way up until the user is ready to run GALFIT. This section will end with the construction of ALLsex_final.cat. It may be helpful to study this process using the flow chart below before proceeding.

Section 1: Flow Chart

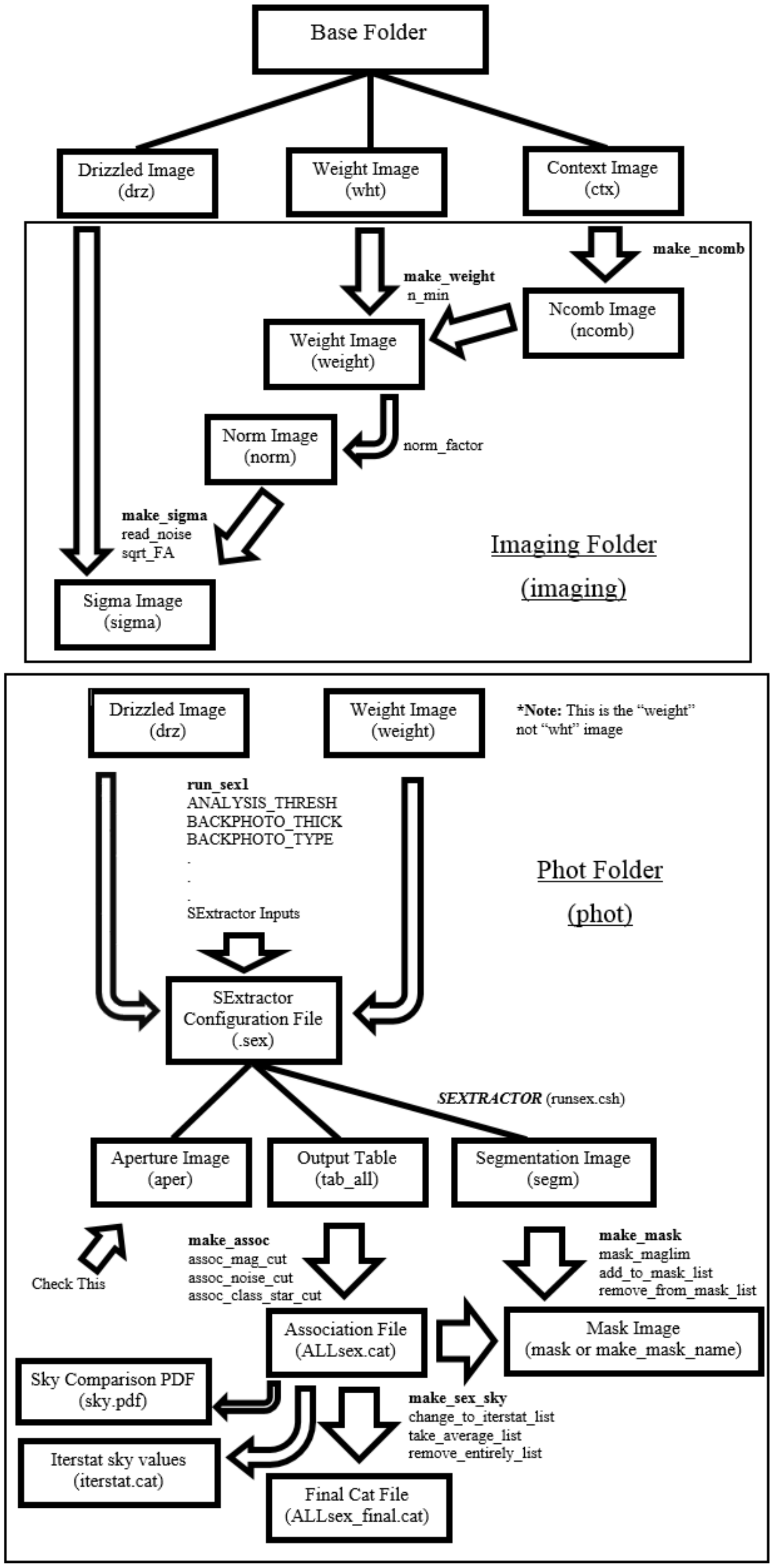


Figure 1 Diagram of GALFIT preparation within the imaging and phot folders.

§ 1.0 : make_EXPTIME_1

Input(s): fits files in base folder (should be drz, wht, ctx only)
Output(s): See summary
Variable(s):

Summary:

If yes, edits the EXPTIME keyword within the image header to be 1. Also, saves the original EXPTIME to the file "important_parameters.txt" which can be found in the base folder.

Motivation:

If units of your drz image are in counts/s or e^-/s then EXPTIME within the image header should be 1 (per the GALFIT manual[5]). This is so the magnitudes are correct. See equation 5 in the GALFIT manual[5].

How it works:

If the file "important_parameters.txt" doesn't exist yet, it creates this file. Then it goes through the image headers of the drz, wht, and ctx images and looks for lines that start with "FILENAME" and "EXPTIME". If it finds these then it writes these two to "important_parameters.txt".

Then it looks through all 3 headers and checks to see if the sum of their EXPTIME keywords is 3, if it is 3 nothing happens because this has (likely) already been done. If this sum isn't 3 it runs iraf hedit task to change EXPTIME to 1 on all three images.

Possible improvements:

This is primarily why it is necessary to only have 3 fits files in the base folder (*base_folder*). This could be written more robustly

§ 1.1 : make_ncomb

Input(s): ctx image
Output(s): ncomb image (ncomb)
Variable(s):
Summary:

If yes, takes the context image and converts it to an image whose pixel value represent how many stacked images contributed to that pixel.

Motivation:

In order for make_weight to determine which pixels to consider, it needs to know how many images contributed to that specific pixel. This information is encoded in the context image but is in binary, therefore this task simply converts this information.

How it works:

Makes a copy of the context image (ctx) and names it base_name+"_ncomb.fits". Then opens this and finds the maximum pixel value. Then goes through all numbers between 0 and this maximum value, if the image has that value it replaces that value which the sum of its digits when written in binary. For example, if a pixel value is 9 (=0101 in binary) then it converts this value to 2 (=0+1+0+1). Then displays the ncomb image.

§ 1.2 : make_weight

Input(s): ncomb image, wht image
Output(s): weight image (weight)
Variable(s): n_min
Summary:

If yes, makes the weight image by selecting pixels which have *n_min* or greater stacked images contributing to them, if a pixel does have *n_min* or greater stacked images contributing to it, then it gets the corresponding value of the pixel in the wht image, else it gets the value zero.

Motivation:

Often times at the edge of an image or in a particular bad patch on the CCD you will get a group of pixels with only a few stacked images which contributed, you want these pixels to have very little weight. Below some threshold (*n_min*) a pixel can be considered unreliable, such that it shouldn't have any weight.

Needed for SExtractor.

How it works:

Uses the iraf task imexpr. If a pixel in (ncomb) is ≥ (*n_min*) the output weight image (weight) maintains the corresponding value in (wht), else it gets the value zero.

Possible improvements:

Use astropy.fits as I did for ncomb (probably faster).

Notes:

In the original version mkweight.cl, instead of changing pixel values to zero, changed values with less than *n_min* to one. This gave me problems running SExtractor and the aperture image looked ugly.

§ 1.3 : make_sigma

Input(s): weight image, ncomb image, drz image
Output(s): sigma image
Variable(s): norm_factor, read_noise, sqrt_FA
Summary:

If yes, first creates a norm image (norm) which is simply (*norm_factor*) * (weight). The norm factor (*norm_factor*) in my case was simply the number of stacked images. Typically determined from:

```
--> imhead UV105842F160W_drz.fits l+ | grep "NDRIZIM"
NDRIZIM =                    4 / Drizzle, No. images drizzled onto output
```

Essentially, the weight image (*wht_image*) can be used as *normalization* in the equation below so long as it represents the real exposure time map of the drizzled image. For me the weight image (*wht_image*) was averaged (?) over the stacked images so in order to represent the real exposure time map this image needed to be multiplied by the number of stacked images. A good heuristic is to check the values in your (*wht_image*) and if they are not approximately equal to the original (*EXPTIME*) which can be found in important_parameters.txt, then you need a norm factor to make this so, otherwise just set *norm_factor* = 1.

Then make_sigma uses the following equation to create the sigma image:

sqrt((a)/c+b*(d/c)**2)*e

where "a" is (*drz*), "b" is (*ncomb*),"c" is (*norm*), "d" is (*read_noise*), "e" is (*sqrt_FA*). See the GCP photometry guide for more background on this equation (page 21 and 22). The particularly relevant equation is:

$$\sigma_{norm} = F_A^{1/2}\left[N_{comb}\left(\frac{ron}{normalization}\right)^2 + \frac{signal_{output}}{normalization}\right]^{1/2}$$

Here the Poisson noise, which should dominate, is represented by the second term, while the first represents the read noise contribution. Here σ_{norm} is our desired sigma image, $F_A^{1/2}$ is the square root variance reduction factor F_A (Casertano et al. 2000[8]). For me this was determined in the following way:

```
#from imhead:
#D003SCAL=                 0.06 / Drizzle, pixel size (arcsec) of output image
#D004ISCL=              0.12825 / Drizzle, default IDCTAB pixel size(arcsec)
#D004PIXF=                  0.8 / Drizzle, linear size of drop

#pixelscale = 0.06/0.12825 = 0.467836
#F_A^0.5 = (0.467836/0.8)*(1-(1/3)(0.467836/0.8)) = 0.4708
```

Your particular case may be different and I refer you to page 22 of the GCP photometry guide.

Similarly, read noise was obtained in a similar way:

```
#--> imhead uv-105842-d6p-01-284.0-f160w_sky_drz_wht.fits l+ | grep "READ"
#READNSEA=        2.0200001E+01 / calibrated read noise for amplifier A
#READNSEB=        1.9799999E+01 / calibrated read noise for amplifier B
#READNSEC=        1.9900000E+01 / calibrated read noise for amplifier C
#READNSED=        2.0100000E+01 / calibrated read noise for amplifier D
```

Since this was roughly 2 for all amplifiers I simply set it to 2 (Poisson noise should dominate anyway).

Motivation:

Needed for GALFIT.

How it works:

Uses the iraf task imexpr to execute the equation above.

Possible improvements:

Don't know if astropy.fits implementation would be feasible/worthwhile. Try extracting or recommending values for norm_factor, read_noise, and sqrt_FA by extracting information from the image header, would be easy to implement but not sure it would always be correct (at least for different types of images).

Notes:

A good check is that the standard deviation of an area of sky in the original image should be roughly the same as the sigma image values. Also, you should be able to see sources in your sigma image.

One thing that puzzles me is that in my images there are areas where the drizzled image and more notably weight image are zero (see Notes under make_weight), this should probably throw a divide by zero error. I guess it doesn't because N_{comb} is also zero in these places and the drizzled image is also zero for these pixels which would still be $\frac{0}{0}$ in the Poisson noise term (IRAF perhaps just says this is 0 instead of undefined).

§ 1.4 : run_sex1

Input(s): weight image, drizzled image, SExtractor Inputs (see variable(s))

Output(s): Aperture image (aper), Segmentation image (segm), Output table (tab_all), (.sex configuration file)

Variable(s): print_vars, path_to_sex_config

ANALYSIS_THRESH, BACKPHOTO_THICK, BACKPHOTO_TYPE, etc…

Summary:

If yes, runs SExtractor by creating a .sex file from the SExtractor input variables supplied.

If *print_vars*=="yes", then it prints all the values within *path_to_sex_config*. *path_to_sex_config* is supposed to be a previously known .sex configuration file with similar inputs. You can then paste this under "#SExtractor variables" and edit them as needed. Then when you are finally ready to run set *print_vars*=="no".

Important Notes:

All SExtractor variables need to be capitalized if they are to be recognized and written to the .sex file. They should also be the only variables that are entirely capitalized.

No ASSOC variables should be in the param file or list of defined variables, as SExtractor is not being run in association mode here.

Your .param file (PARAMETERS_NAME) should include the following:

NUMBER
FLUX_AUTO
FLUXERR_AUTO
MAG_AUTO
MAGERR_AUTO
KRON_RADIUS
BACKGROUND
ISOAREA_IMAGE
FLUX_RADIUS
X_IMAGE
Y_IMAGE
ALPHA_J2000
DELTA_J2000
THETA_IMAGE
ELLIPTICITY
FWHM_IMAGE
FLAGS
CLASS_STAR

Else you should receive an error during make_assoc. In fact, you could just paste this into a text file and make that your .param file. This list is the bare minimum required for the script, however the user can add additional parameters such as MAG_APER, MAGERR_APER, etc.

<u>It is best to include the full filename path for FILTER_NAME, PARAMETERS_NAME, STARNNW_NAME</u>.

You have no bearing over the following as they will get defined (overwritten) later in the script:

```
CHECKIMAGE_TYPE = "SEGMENTATION,APERTURES,BACKGROUND"
CHECKIMAGE_NAME=
base_name+"_segm.fits,"+base_name+"_aper.fits,"+base_name+"_background.
fits"
CATALOG_NAME = base_name+"_tab_all.fits"
WEIGHT_IMAGE = imaging+base_name+"_weight.fits"
WEIGHT_TYPE = "MAP_WEIGHT"
```

Motivation:

Supplies several useful outputs for GALFIT, including initial guesses for parameters

How it works:

Print_vars:

Reads through lines and finds ones which do not start with “#”, aren’t newlines, and whose first letter is uppercase. If “#” is in the line it splits the line and prints the name, entry, and comment. Else, in the case that you are using one without comments (e.g. one of my configuration files) it prints name and entry, but no comment because there is none. This is just designed to be helpful, so you don't painstakingly need to format and redefine variables. You can then paste this printed output directly into the script.

When print_vars is "no":
It proceeds to create the .sex file which is done by looking through all the upper case variables that have been defined and writing the name of the variable (item) and the value of the variable via eval(item). It writes these to the .sex file. Then it writes runsex.csh which is effectively just a command line statement to run SExtractor. Then it runs SExtractor by running runsex.csh. This is a little clunky but is an artifact of trying to write everything within scripts instead of just running from command line.

Possible improvements:
Print_vars is perhaps a little bit hard coded and tailored to my own situation, could be made more universal perhaps. Clunkiness described above could be improved. Could output the verbosity of SExtractor to an outfile.

§ 1.5 : auto_review_sex_output

If yes, just displays SExtractor output (tab_all) as an excel file and display it for you to review. I find it typically easier to just look at the .cat file.

§ 1.6 : make_assoc

Input(s): SExtractor Output (tab_all.fits)
Output(s): Original cat file or association file (ALLsex.cat)
Variable(s): assoc_mag_cut, assoc_noise_cut, assoc_class_star_cut

Summary:
This name "make_assoc" is somewhat of a misnomer as SExtractor is never run in association mode, but nevertheless this will give the same result: a cat file with appropriate ids.

If yes, creates the original cat file (ALLsex.cat) from the output fits table from SExtractor. Entries which do not meet the criteria below get a zero as their id (indicating they won't be fit). Also produces two graphs: CLASS_STAR vs. MAG_AUTO and MAGERR_AUTO vs. MAG_AUTO. The first of these is used to determine the cutoff between stars and galaxies. The second helps you decide what the magnitude cutoff should be. Galaxies which meet <u>all</u> the following conditions are going to be fit by GALFIT* (non-zero id):

1. MAGERR_AUTO<assoc_noise_cut
2. MAG_AUTO<assoc_mag_cut
3. CLASS_STAR<assoc_class_star_cut

The first condition comes about because the magnitude error is a good proxy for signal to noise (Gemini Interns see Ryan Cole's write up section 5.6.1). An appropriate value for *assoc_noise_cut* is 0.04 which as Ryan states corresponds to a S/N of approximately 25. You need to decide *assoc_mag_cut* based on where *assoc_noise_cut* intersects galaxies in the MAGERR_AUTO vs. MAG_AUTO plot. The idea being that you want to <u>exclude</u>

galaxies with MAGERR_AUTO>assoc_noise_cut (see Figure 2). In addition, you don't want to fit stars, so objects above a certain CLASS_STAR threshold.

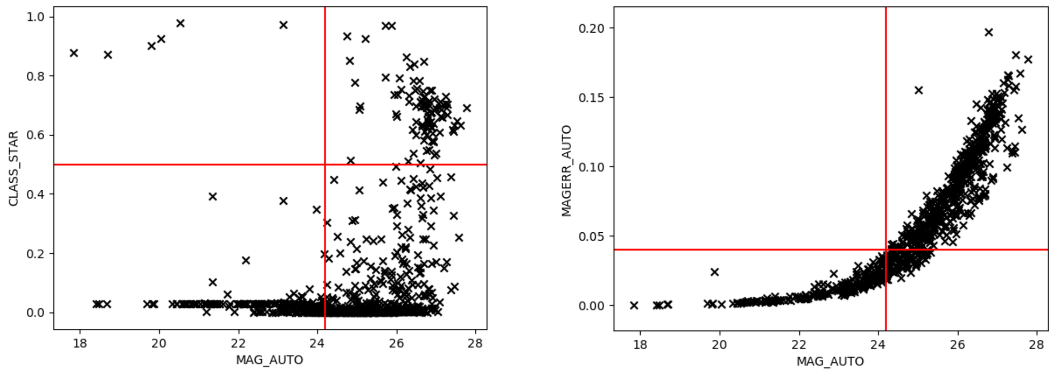


Figure 2 Left:CLASS_STAR vs. MAG_AUTO plot obtained from make_assoc task. Right: MAGERR_AUTO vs. MAG_AUTO plot obtained from make_assoc task.

Lastly, outputs the aperture image (aper) from SExtractor with labels on each galaxy for you to make sure everything is in order.

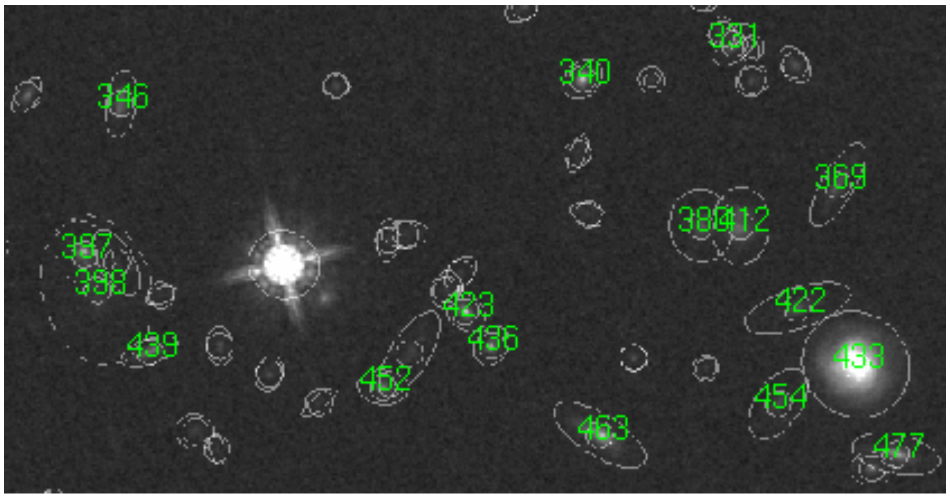


Figure 3 Close up for aperture image produced by make_assoc task.

The .cat file will looks something like this:

.

.

.

```
0  0.26841819  0.02718418  27.37427139  0.10998517  3.5  0.13688207 10  1.94723105
       1429.62207031  2723.578125  150.26197353  2.04016347 -41.5202446  0.50493842
       5.4318738 0  0.07554447
21  49.21134567  0.17937124  21.71613693  0.00395838  3.5  0.1367951 479  3.59537864
       1545.36083984  2681.58056641  150.26004332  2.03946351  35.25988388  0.3020699
       5.8005209 0  0.06231659
0  1.95676601  0.06588497  25.217453  0.03656599  3.5  0.13710082 69  2.98537612
       1009.41479492  2689.97460938  150.26898142  2.03960339 -63.67418289  0.16306216
       6.66914463 0  0.00891602
```

```
23  103.81716156  0.20607348  20.9056282  0.00215567  3.5  0.1371191 973  4.55846024
      1065.93896484  2665.48583984  150.26803875  2.03919525 -63.47914886  0.12056607
      5.8432374 2  0.0287193
0  0.44930986  0.05791214  26.81493568  0.13997598  6.80331707  0.13694094 11  2.70811772
      1344.74243164  2674.32202148  150.26338908  2.03934253  79.3816452  0.53123033
      5.79243374 0  0.06495944
25  12.40153885  0.16457039  23.21261215  0.01441139  4.69349337  0.13712652 288  5.32439566
      1058.53674316  2638.92138672  150.2681622  2.03875251  4.40433645  0.36724997
      7.13682795 3  0.0267529
.
.
.
```

*in the sense that it will have a cutout and officially be part of the catalog (i.e. something we care about). GALFIT will also technically fit fainter neighbors and this is set by (*mask_maglim*) which will be explained in section § 1.7.

Motivation:

This is an important step as it decides the magnitude below which you will be actually fitting galaxies with GALFIT.

How it works:

If yes, opens the table and plots the relationships described above. Then looks through the table and if a galaxy meets the parameters its details get written to the (ALLsex.cat) file with the corresponding id it had for "NUMBER" in the tab_all.fits file. Else its information gets written with a zero for the corresponding id. Makes a file region_file.reg which can be useful later on if you want to see which galaxies you are actually making fits for. Displays aperture image (aper) and overlays the region file.

Should throw an error if not all the parameters within `columns` are present.

Possible improvements:

Determine assoc_class_star_cut, etc. automatically?

§ 1.7 : make_mask

Input(s): Original cat file or association file (ALLsex.cat), segmentation image (segm)
Output(s): Mask image (mask)
Variable(s): mask_maglim, add_to_mask_list, remove_from_mask_list

Summary:

If yes, creates the mask image by "unmasking" the segmentation image based on the specified *mask_maglim*. If you want to add an id to the mask, then its id, as given by the ALLsex.cat file, should be in *add_to_mask_list*. *Remove_from_mask_list* works in a similar way, only those ids will be removed from the mask. Displays created mask.

If you want to create another mask and not overwrite the original you can change *make_mask_name* from "default" to "newmask" or some other appending characters. For example, if your original mask had the name UV105842F160W_mask.fits and you run make_mask again but this time with *make_mask_name*="final" it will make another mask UV105842F160W_mask_final.fits.

If you need to add something to the mask which was not properly deblended or found by SExtractor see §1.7.1.

Important Note:
The *final_mask_for_galfit* variable indicates the actual mask which will be used by GALFIT. This is due to §1.7.1

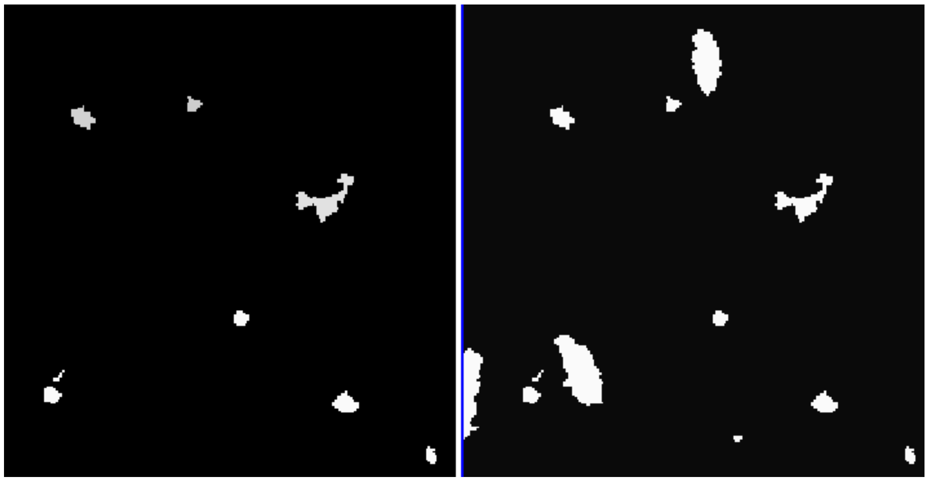


Figure 4 Mask image from the task make_mask. Left: mask image created by "unmasking" the segmentation map (right).

Motivation:
The mask is important for GALFIT so that it does not try to fit objects which are too faint.

How it works:
If yes, goes through ALLsex.cat and gets the third position in line.split(), which ought to be MAG_AUTO from SExtractor, and if this value is ≤ *mask_maglim* then it is added to an "unmask" list. Ids in *add_to_mask_list* are removed from the unmask list (sorting the double negatives this is "adding" to the "masking" list for anyone confused), ids in *remove_from_mask_list* are added to the unmask list. Iterates over the unmask list and looks for values in the segmentation image with that id value and if it finds one then it "unmasks" this object by making it zero. The segmentation image is output from SExtractor and shows how SExtractor split objects within an image. It also shows which pixels constitute a given object and therefore it is a useful tool for making the mask. The mask is saved and displayed according to the name provided. Also writes a text file with id numbers and coordinates (I thought that might be important) that were masked. Note that if you execute §1.7.1 this is no longer an accurate list of the mask.

Possible improvements:
Implement §1.7.1 in this task?

§ 1.7.1 : SExtractor Undersplitting

Occasionally SExtractor will not split an object enough. This becomes an issue if you want to mask an object adjacent to a galaxy you want to fit, but they are both considered the same object by the segmentation mask.

To address this, I wrote the following function:

```
make_not_deblended_additions_to_mask(initial_mask,
further_deblended_segm,
output_mask,
ids_in_deblended_mask,
start_value_for_ids_in_mask=1000)
```

In this section I will explain how this function works and what you need to do in order to add these unsplit or unrecognized objects to your mask. To retain information about how this task is run an infile which lists the inputs provided is created in the same folder as the output_mask. In the future, this may be written into the script like the other tasks, but this is somewhat trickier since a fair amount of user discretion is required (i.e. selecting which objects should be masked, determining DEBLEND params, etc.). For now, the following steps should suffice to generate a mask which includes these undetected objects:

1. Create a new folder within your base_folder perhaps call it "final_mask", it's really not important though.
2. Copy the following into this folder:
   -drizzled image (drz_image)
   -the .sex configuration file which should be in your (phot) folder
3. Open the .sex configuration file and decrease DEBLEND_MINCONT. It may also help to increase the DEBLEND_NTHRESH.
4. Once you have changed these parameters **cd** to your new folder then run the following:

   ```
   sex your_drz_image_name -c your_configuration_file_name
   ```
*If you did not use the whole path to _NAME variables this may throw an error
5. Now check the segmentation image, if the galaxies in question are still not split then repeat steps 3 & 4 until they are. When you have all the galaxies split, take note of their current ids in this segmentation image.
6. Go into your python shell are run the make_not_deblended_additions_to_mask function at the prompt where:
   - initial_mask is the full path to the mask which you want these objects to be added to (will not overwrite)
   - further_deblended_segm is the path to the new segmentation image you just created in the steps above
   - output_mask is the path name of the output mask to be created
   - ids_in_deblended is a list of the ids you took note of in step 5
   - start_value_for_ids_in_mask=1000 this is an optional parameter, essentially the ids of these newly masked objects in the output_mask will be given an id of

start_value_for_ids_in_mask + their_id_in_step_5. If you are unsure about this just make it larger than the total number of galaxies in your sample. Essentially this is here to prevent two individual mask objects from having the same id.

7. If it is important for you then you can add these new entries to the mask list created in §1.7
8. If you are happy with the result then you have the final mask_image ready for GALFIT. You should then set the variable final_mask_for_galfit to your new mask:

```
final_mask_for_galfit="/path/to/mask/you/just/made.fits"
```

When GALFIT runs it will look for the nonzero pixel values in this image and this is the official mask list (therefore step 7 is not vital).

§ 1.8 : make_sex_sky

Input(s): ALLsex.cat, itersky.cat
Output(s): Final Cat File (ALLsex_final.cat), itersky.cat, sky_results.pdf
Variable(s): make_sky_plots, change_to_iterstat_list, take_average_list, remove_entirely_list

Summary:

If yes, checks to see if itersky.cat exists if it doesn't and *make_sky_plots*=="yes" (this is should be "yes" if running for the first time), then it proceeds to run iterstat and compare with SExtractor sky values. This takes some time to run (due to iterstat), but it will make plots in the manner of the original *mksexsky* (please see the GCP_photometry_guide). It then saves these graphs to sky_results.pdf. In my opinion, it is just best to let this run and then go through the entire pdf and figure out the best action to take with the sky value. When you figure out which ones you want to change to the iterstat value, simply add the ids to the *change_to_iterstat_list*. Likewise, if you want to take the average you can do so by adding it the *take_average_list*. If you add something to the *remove_entirely_list*, then its <u>entire</u> entry in the cat file will be removed (you are probably better off just adding to the mask), nevertheless the option is there. Finally, you can run again with *make_sky_plots*=="no" and it will implement the changes. **Other custom changes to the cat file before it enters GALFIT can be made here, but know that running make_sex_sky again will overwrite the changes.**

Tips for interns:

When deciding what action to take with sky values, know that most of the time there is little difference between iterstat and SExtractor (if things are running correctly). Use your best judgement and keep in mind what each of the two graphs represent. You can still change the sky value later on if something goes wrong in the fit, but it is obviously best to start with the most accurate values. Sky values are very important for GALFIT to run correctly.

Important Notes:

<u>The original input for GALFIT is Final Cat File (ALLsex_final.cat) and this task makes that file so it is important to run this. If you just want to use SExtractor values, you can rename (ALLsex.cat) to (ALLsex_final.cat) and just not run this task.</u>

You need the following iraf packages for this to work:

```
iraf.stsdas(_doprint=0)
iraf.hst_calib(_doprint=0)
iraf.nicmos(_doprint=0)
```

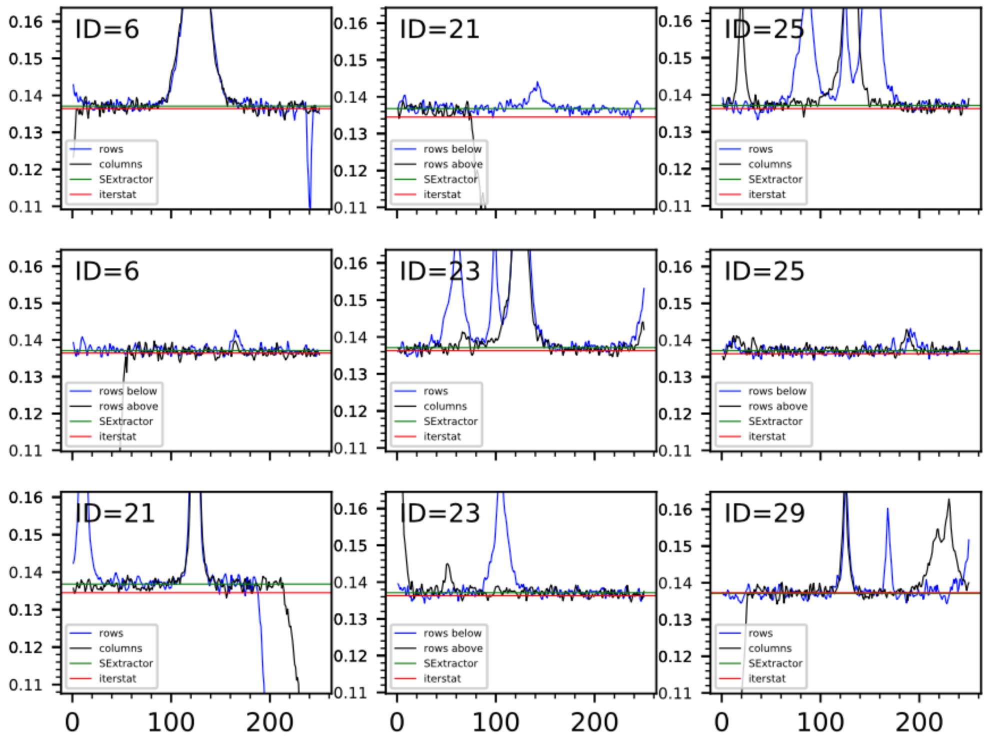


Figure 5 Output from make_sex_sky.

Motivation:

SExtractor sky values can sometimes not be accurate enough (per the GCP_Photometry_Guide) therefore iterstat is run as a sort of second opinion to make sure they are accurate.

How it works:

If yes, first sees if itersky.cat exists if it does it implements the changes in the lists mentioned above. If itersky.cat doesn't exist and make_sky_plots==”yes” then it makes plots. The plots it makes are a mean of 50 rows (black) and 50 columns (blue) centered on the galaxy and the average of 50 rows centered 85 pixels above and below the galaxy. It was somewhat tricky/annoying to get the subplotting to work. Essentially what goes on here is it gets IDs, sky_values, x_values, and y_values from the cat file then in a while loop goes through all the sky_values if the ID is not zero and it gets the x and y for that ID. Then it does the row and column averages, plots these, runs iterstat, plots the SExtractor and iterstat values as horizontal lines. Then it does the same 85 pixels above and below. The weird if statements are due to needing to change `axs[i,l]` to the correct indexes. This task works for me and I hope it will work for everyone else, but it very well may throw an error somewhere.

Possible improvements:

Make more stable. See bugs

Bugs:

I imagine that if your galaxy is close to an edge it may throw an error trying the following:
`image_data[y-125:y+125,x-25:x+25]`
if this happens to you can write an if statement to restrict x, y. Could also replace hardcoded 50 with image_data[y-125:y+125,x-25:x+25].shape[1]. Or something similar…

Notes:

This task was my least favorite to write, because it was painstaking to get the graphs in the same way as the original *mksexsky* task. If in doubt about something regarding the results of this task or it doesn't work then just use the original *mksexsky.cl*. Though you will need to make manual changes to the sky values

§ 1.9 : make_psf (optional)

Input(s): Drizzled image (drz), SExtractor output table (tab_all)
Output(s): PSF image
Variable(s): psf_apertures, psf_zmag, psf_scale, etc…

Summary:

You need (at least) one PSF image to run GALFIT so in that sense this section is not optional. However, if the user desires to make several local PSFs for their drizzled image then this is not really possible within this task. This task does the following:

If yes, loads the DAOPHOT package from IRAF sets the variables (typically "psf_*") to their IRAF equivalents, for example:

```
 .
 .
 .
iraf.photpars.zmag=psf_zmag
iraf.datapars.scale=psf_scale
iraf.datapars.fwhmpsf=psf_fwhmpsf
 .
 .
 .
```

Proceeds to the following:

1. daofind-to get coordinates of objects within the image
2. phot-to do photometry on said objects
3. pstselect-to select PSF candidate stars
4. Refines the list provided by pstselect by selecting stars based on CLASS_STAR (*star_thresh*) and MAG_AUTO (*limiting_mag*) within (tab_all.fits)

5. psf-to create the psf (though this really makes the residual)
6. nstar-to fit PSF to groups of stars simultaneously
7. substar-to create residual left by nstar
8. seepsf-to get the final psf model
9. Plots the average of rows zoomed in at the base of the PSF, average of PSF rows, and the central row of the PSF. Displays drz/substar output with circles on PSF stars. If *psf_comparator* is not "default" it will plot this psf as well.

This is roughly the routine outlined here:
http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?psf

If link is broken do `>>help psf` in an IRAF window and scroll to "GUIDE TO COMPUTING A PSF IN A CROWDED FIELD"

However, steps 8 and on are omitted because they were not helpful for me in removing faint companions (actually made the PSF worse, due to over subtraction of faint objects).

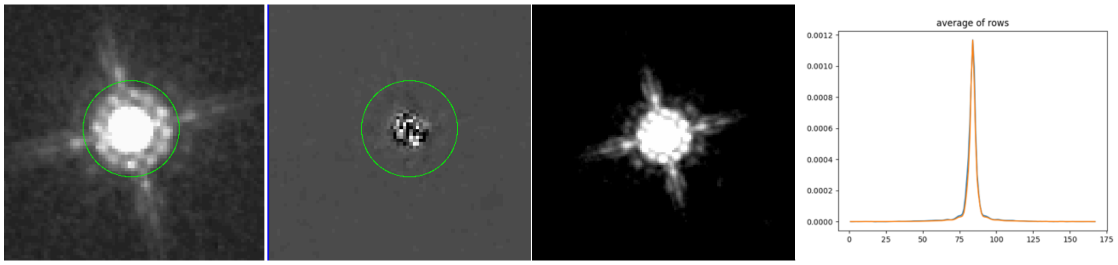


Figure 6 Outputs from make_psf.

Motivation:
PSF is important for GALFIT, should make sure the PSF output from this doesn't have any problems.

How it works:
See summary.

Bugs/Improvements:
Plots currently assume PSF is 167 pixels in x and y. Can easily be fixed. This has been fixed. Inger suggests to define the PSF in arcseconds.

§ 1.10 : make_constraint (optional)

Input(s):
Output(s): constraint.txt
Variable(s): n_constraint, re_constraint…

Summary:

If yes, creates a GALFIT constraint file based on the parameters listed under "### Constraint inputs". You can add more constraints if needed. To quote the GALFIT Manual: "When constraints are imposed it is unclear what the errorbars mean, if anything. Furthermore, it may prematurely force the solution to wander off into a corner of the parameter space from which it is difficult to wander out. So use constraints at own risk!"

Section 2: Running GALFIT

This section will describe running GALFIT and the refitting process. This section will end with the construction of final output tables of galaxy parameters. It may be helpful to study this process using the flow chart below before proceeding.

The following implies that *do_run_galfit*=="yes".

Section 2: Flow Chart

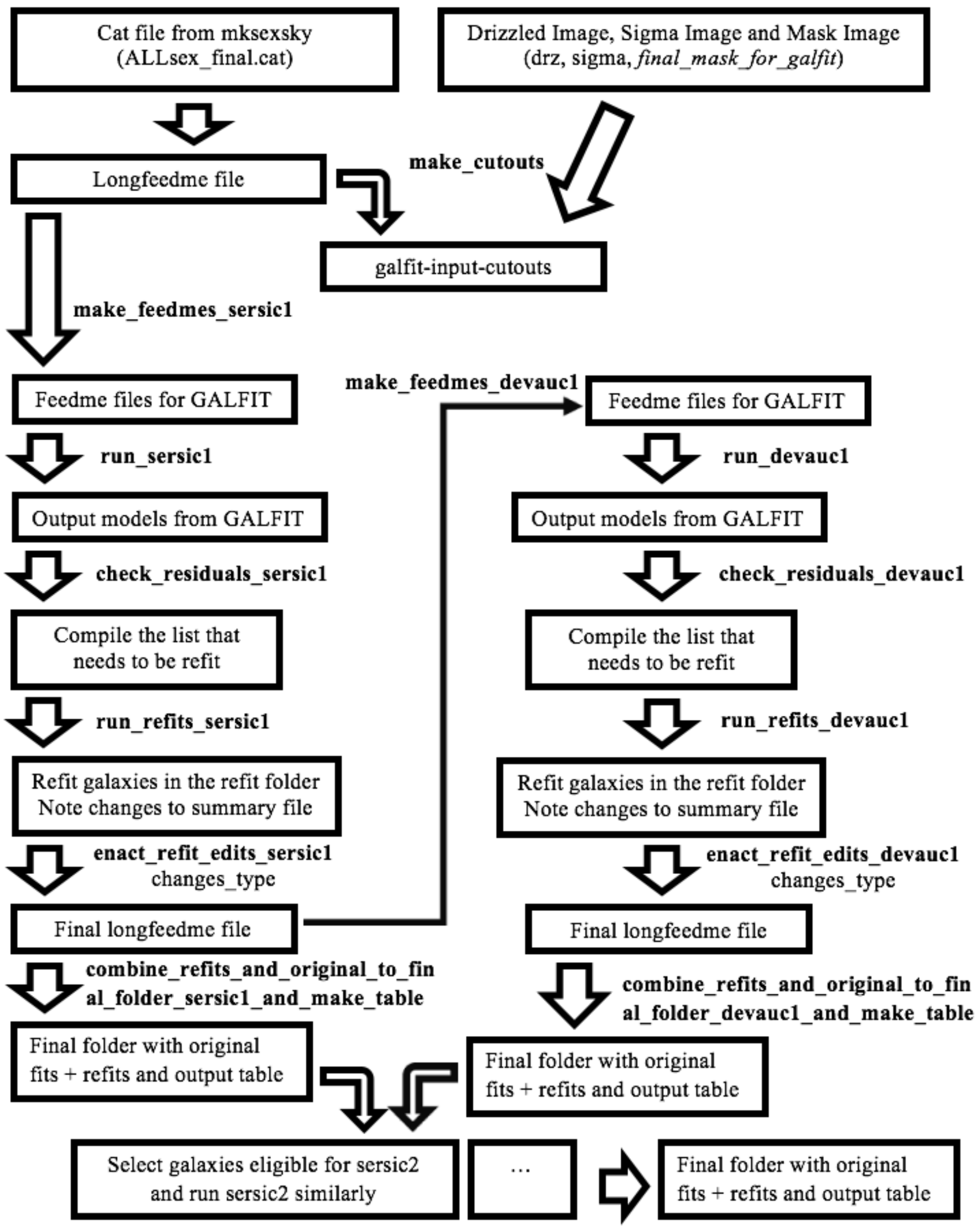

Cat file from mksexsky (ALLsex_final.cat)
Drizzled Image, Sigma Image and Mask Image (drz, sigma, final_mask_for_galfit)
Longfeedme file
make_cutouts
galfit-input-cutouts
make_feedmes_sersic1
Feedme files for GALFIT
make_feedmes_devauc1
Feedme files for GALFIT
run_sersic1
run_devauc1
Output models from GALFIT
Output models from GALFIT
check_residuals_sersic1
check_residuals_devauc1
Compile the list that needs to be refit
Compile the list that needs to be refit
run_refits_sersic1
run_refits_devauc1
Refit galaxies in the refit folder Note changes to summary file
Refit galaxies in the refit folder Note changes to summary file
enact_refit_edits_sersic1 changes_type
enact_refit_edits_devauc1 changes_type
Final longfeedme file
Final longfeedme file
combine_refits_and_original_to_final_folder_sersic1_and_make_table
combine_refits_and_original_to_final_folder_devauc1_and_make_table
Final folder with original fits + refits and output table
Final folder with original fits + refits and output table
Select galaxies eligible for sersic2 and run sersic2 similarly
…
Final folder with original fits + refits and output table

Figure 7 GALFIT flow chart.

§ 2.0 : Important Remarks & Preliminary Actions

There are several actions that take place before anything, this section is to describe those.

The first thing that happens when *do_run_galfit*==”yes” is that a final mask list is generated. As alluded to earlier, the mask list used for GALFIT is created by the nonzero pixels within *final_mask_for_galfit*. If *final_mask_for_galfit*==”default”, then this is the output from *make_mask*. The nonzero pixels are expected to be natural numbers whose value represents the id of the galaxy being masked.

The second thing that happens when *do_run_galfit*==”yes” is the *ALLsex_final.cat* file is copied into the *two_D* directory. If a dictionary version of this file does not exist yet, then a dictionary of this information is created. This Python dictionary and its text file version are known as the longfeedme format. The longfeedme format contains the following information about a galaxy:

id1-The SExtractor identification number of the galaxy
mag1-Magnitude of the first entry (primary galaxy)
n1-Sersic index
pa1-Position angle
q1-Axis ratio
re1-Half-light radius
row1-The row that the galaxy appears in *ALLsex_final.cat* (starts from row 1)
sky-Sky value
type1-Fit type (“sersic”, ”devauc”, etc.)
x_cutout1-x position of galaxy within the cutout
x_end1-x position which marks the end of the cutout in the drizzled image
x_global1-x position of the galaxy as given by SExtractor in the drizzled image
x_start1-x position which marks the beginning of the cutout in the drizzled image
y_cutout1-y position of galaxy within the cutout
y_end1-y position which marks the end of the cutout in the drizzled image
y_global1-y position of the galaxy as given by SExtractor in the drizzled image
y_start1-y position which marks the beginning of the cutout in the drizzled image

It is important to note that a longfeedme entry will have all the objects for a given cutout. For example, a galaxy (with id #36) has 4 objects within its cutout. If you wanted to know magnitude of the primary galaxy, the axis ratio of the 3rd object, the row of the 2nd component, and the x position in the cutout for the 4th object you would access this information via the following, respectively:

```
longfeedme_dictionary[36][“mag1”]
longfeedme_dictionary[36][“q3”]
longfeedme_dictionary[36][“row2”]
longfeedme_dictionary[36][“x_cutout4”]
```

The longfeedme format is the backbone of running GALFIT. It is the data structure through which almost all tasks are run. The primary advantage of this format is that changes to cutouts can be made locally instead of globally. Personally, I believe this to be of great benefit.

Automatically, the cat file is converted into this longfeedme format by calling the function `cat2longfeedme`. This function creates both a dictionary and text file, these are respectively named:

base_name_ALLsex_final_longfeedme.npy
base_name_ALLsex_final_longfeedme.txt

This works by going through lines in *ALLsex_final.cat* finding objects which aren't in the mask and whose vector association (id) is nonzero. The values listed above are made into the dictionary format, and neighboring galaxies (ones not in the mask) are added if they fall within the bounds of the cutout (determined by *min_cutout_radius*). *Min_cutout_radius* is somewhat of a misnomer since the cutout is a square. The side length of a cutout is twice *min_cutout_radius* (pixels). Axis ratio q is taken to be (1-ELLIPTICITY), the half-light radius is (FLUX_RADIUS/(q^0.5), position angle is determined by (THETA_IMAGE - 90°). Galaxies are given a default fit type of "sersic" (since we start with sersic fits, though this could be easily changed if the user wills it) and a Sersic index according to *n_start*. A text file is also made so the user can easily view the output.

I want to briefly comment about the general process of running galaxies through GALFIT. This is an ostensibly simple topic. Originally, the user runs three different fits: sersic1, devauc, and sersic2. The current script follows this tradition, though could be modified to support any ordering. In the original perl scripts, galaxies were run sequentially based on their position in the cat file. As models finished, the outputs for neighboring galaxies within the cutout were used as inputs for neighbors in subsequent cutouts. While on first consideration this seems like a good idea to speed up the convergence of neighbors, in practice it can lead to problems. It was particularly difficult to figure out why a fit went wrong because this could depend on the outputted guesses for neighbors and even the ordering of the cat file. This dilemma is the motivation for creating this script.

There still remains the question: what is the best way to generate guesses for feedme files? In this script GALFIT is run in the following way:

First use the guesses from SExtractor to run the sersic1 fit as the original scripts did. However, the feedme files are made prior to any fitting is done and their guesses are taken directly from the cat file (in longfeedme format). Then the user looks through the output and decides a list that needs to be refit. These are refit in a new folder and the changes are summarized in a text file (in a certain format). These changes are then enacted on the total longfeedme file and the final fits (original + refits) are moved to a final folder. This completes the sersic1 fit.

For the devauc (sometimes referred to as devauc1) fits, the guesses are taken from the final sersic1 longfeedme file (with appropriate changes for fit type and n). The logic here is that if a fit crashed or didn't come out right using the SExtractor guesses in sersic1 (i.e. it needed to be refit), then there is no point using those same inputs again in the devauc fit because it will likely fail again. Instead the changes that were implemented during the sersic1 refitting are already in place when you fit devauc, the idea being that you (hopefully) won't need to refit those galaxies again because their initial guesses have been revised. These feedmes are created, ran, and analyzed for refitting.

In my experience, the refitting for devauc is much quicker/easier. A summary file of changes is made, enacted on the longfeedme, and the final fits are merged.

Finally, if the output sersic index of the sersic1 fit was greater than 3 and the devauc fit had a lower (better) reduced chi square value, then this galaxy is considered for a sersic2 fit. Its feedme is created using the original values from the sersic1 final longfeedme except it has a starting sersic index of 4 instead of 1.5 (traditionally). The user analyzes the output and decides if the fit improved, if it did then these sersic2 galaxies are combined with the final sersic1 fits in the sersic2 final folder. Output text/fits tables are produced in for all final outputs. You can easily plot the results of any output folder by calling `plot_stats(path_to_outputs)`. Much more is explained in the pages that follow but this is the overview of the process.

§ 2.0.1 : Discussion: Ways to Run GALFIT

Local vs. Global Changes

As far as I can tell there are two sensible ways to implement refitting changes. The first is a local change where only the cutout which is being refit is affected by the changes. The other is a global change where the change you make to a galaxy will occur in every cutout which that galaxy is in. The benefit of making changes locally is that the user will know how the changes will take effect. If the user makes changes globally they should really refit all of the affected cutouts as well to make sure that the final values are the same or improved.

Use Parameters as Inputs for Next Fit

Another possible way to run GALFIT is to use the output parameters from the previous fit as inputs to the next fit. For example, if my original magnitude guess from SExtractor is 23.20 for a particular galaxy and after the sersic1 fit the GALFIT value is 23.60. Then you could start the devauc1 fit using 23.60 as the initial guess instead of 23.20. This is similar to the idea of the original Perl scripts to "auto-update" initial guesses. The challenge here is when the final parameters don't make sense, because in that case you do not want to use them as the input guesses. A common shortcoming of the original Perl scripts was when a star within the image was fit. The initial half-light radius for the star was typically small ~3 pixels, but after being fit once the final value would be something like 0.01. Starting with such a low half-light radius is a problem for GALFIT and can cause it to crash, so this would be an example where the program should always take the initial guess ~3.

In its current version do_everything.py is run using local changes and no auto-updating (i.e. not using outputs as inputs). When I have the time, I will implement these other options.

GALFIT Peculiarities

This section is to highlight why I'm in favor of local changes and no auto-updating. GALFIT does not always act the way one would expect. For example, I had a galaxy (#477) which crashed during its sersic1 fit. The initial inputs were:

```
Initial parameters:
 sersic    : (  126.71,   126.29)   22.89      9.33    1.50    0.44    78.47
 sersic    : (   63.59,   220.19)   23.79     13.32    1.50    0.37   -71.97
 sersic    : (  100.91,   183.91)   19.70     10.72    1.50    0.94    51.82
 sersic    : (   51.17,   158.78)   24.11     10.41    1.50    0.56   -34.76
 sersic    : (  194.19,   147.75)   21.44      5.98    1.50    0.79   -86.15
 sersic    : (  111.38,   117.18)   24.25      4.73    1.50    0.74   -89.37
```

I noticed the highlighted neighbor was causing the crash (low_re value) so I changed its initial magnitude and nothing else. Now the input parameters were:

```
Initial parameters:
 sersic    : (  126.71,   126.29)   22.89      9.33    1.50    0.44    78.47
 sersic    : (   63.59,   220.19)   23.00     13.32    1.50    0.37   -71.97
 sersic    : (  100.91,   183.91)   19.70     10.72    1.50    0.94    51.82
 sersic    : (   51.17,   158.78)   24.11     10.41    1.50    0.56   -34.76
 sersic    : (  194.19,   147.75)   21.44      5.98    1.50    0.79   -86.15
 sersic    : (  111.38,   117.18)   24.25      4.73    1.50    0.74   -89.37
```

This converged and had a good residual, the fit had the following final values:

```
Iteration : 46    Chi2nu: 2.476e+00     dChi2/Chi2: -1.35e-08   alamda: 1e+04
 sersic    : (  127.40,   126.41)   22.95      6.51    1.37    0.17    76.71
 sersic    : (   62.96,   219.90)   24.23      5.80    1.16    0.20   -67.56
 sersic    : (  101.11,   184.53)   19.61     10.17    1.78    0.80    28.94
 sersic    : (   51.04,   158.57)   24.60      4.89    0.83    0.31   -37.93
 sersic    : (  194.17,   147.47)   21.32      4.46    1.53    0.75   -87.53
 sersic    : (  111.28,   117.25)   24.22      1.51    8.72    0.12   -83.80
```

Note here that the initial guess was actually closer to the final value than my new input guess! I should also note that values upwards of 23.79 also seemed to crash

Now I try changing the input parameters to be the output:

```
Initial parameters:
 sersic    : (  126.71,   126.29)   22.89      9.33    1.50    0.44    78.47
 sersic    : (   62.96,   219.90)   24.23      5.80    1.16    0.20   -67.56
 sersic    : (  100.91,   183.91)   19.70     10.72    1.50    0.94    51.82
 sersic    : (   51.17,   158.78)   24.11     10.41    1.50    0.56   -34.76
 sersic    : (  194.19,   147.75)   21.44      5.98    1.50    0.79   -86.15
 sersic    : (  111.38,   117.18)   24.25      4.73    1.50    0.74   -89.37
```

This yields:

```
Iteration : 27    Chi2nu: 2.482e+00     dChi2/Chi2: -4.59e-08   alamda: 1e+09
 sersic    : (  127.40,   126.40)   22.93      6.62    1.40    0.18    76.78
 sersic    : (   62.95,   219.90)   24.22      5.83    1.19    0.20   -67.55
 sersic    : (  101.10,   184.53)   19.61     10.17    1.78    0.80    28.92
 sersic    : (   51.04,   158.57)   24.60      4.88    0.83    0.31   -37.90
 sersic    : (  194.17,   147.47)   21.32      4.46    1.54    0.75   -87.53
 sersic    : (  111.34,   117.24)   24.27      0.01    2.92    0.41   -73.85
```

Note here the chi squared value is higher even though we told it what the parameters for the second component ought to be! Note that the last component is also suspect with such a low half-light radius.

So now we know we can get a good fit with the starting magnitude being 23. We already know the output makes sense, even though the initial input admittedly does not. If you make the change locally, you don't have to worry about this because other cutouts will still preserve the initial magnitude (23.78) and we can move on to the next cutout. However, if you make the change globally, you should be wary that the magnitude is so far off. Indeed, when making this change globally and running the devauc fit another galaxy (#433) crashed because of this same neighbor. I should note that #433 runs fine with an initial value of 23.78. I put here the original cutout drizzled image of #477 for reference (troublesome neighbor is in the upper left):

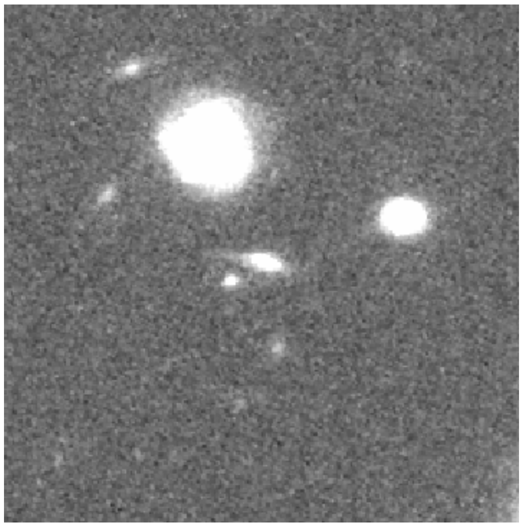


Figure 8 Cutout #477.

I'm not saying that global changes and auto-updating are incorrect or foolish ways to run GALFIT, but in my opinion the process is much simpler and more predictable if these methods are avoided. Hopefully, the above example demonstrates that.

§ 2.1 : make_cutouts

Input(s): drz_image, *final_mask_for_galfit*, sigma_image
Output(s): image stamps or cutouts of the inputs
Variable(s): *cutouts_list_to_make, min_cutout_radius*

Summary:

If yes, makes the cutouts by calling the function `do_make_cutouts`. Makes cutouts with a radius which is max(*min_cutout_radius,* FLUX_RADIUS). The "radius" here is half the side length of the cutout. Therefore if *min_cutout_radius*=125, the side length of a square cutout will be at least 250. Only makes cutouts for the ids in *cutouts_list_to_make,* if this is "default" it makes all of them. Cutouts are put in the *galfit-input-cutouts* folder within the *two_D* folder.

Motivation:

I imagine GALFIT would be overwhelmed fitting an entire field at once so we split it up into stamps/cutouts which are centered on the galaxy we want parameters for.

How it works:

Goes through the longfeedme dictionary (base_name_ALLsex_final_longfeedme.npy) and uses the start and end coordinates (longfeedme[id][“x_start”] etc.) to create cutouts of the appropriate size. Utilizes IRAF’s imcopy task. Overwrites cutout by deleting it before making it.

Possible improvements:

If a galaxy is near the edge of an image it may get hung up. I believe it should fix the start and end values appropriately, but I haven’t tested this since it wasn’t a problem with my images.

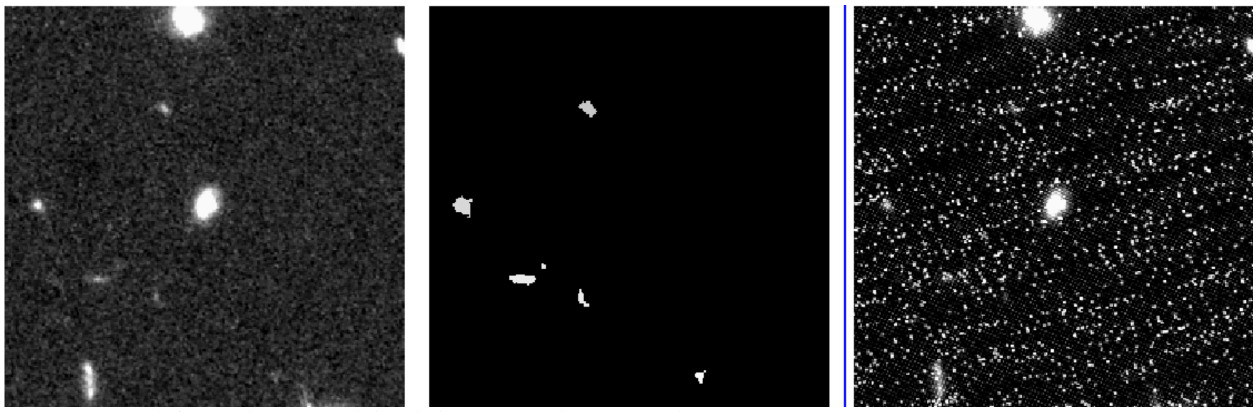


Figure 9 From left to right: Original drizzled image, mask image, and sigma image cutouts.

§ 2.2 : make_feedmes_sersic1

Input(s): base_name_ALLsex_final_longfeedme.npy
Output(s): feedme files within base_name_sersic1 folder
Variable(s): *psf_size, fit_sky*

Summary:

If yes, makes all the feedmes files from the information within base_name_ALLsex_final_longfeedme.npy and puts these in the base_name_sersic1 folder. Calls the function `longfeedme2feedmes`.

Also makes a copy of base_name_ALLsex_final_longfeedme.npy within the base_name_sersic1 folder and renames it base_name_longfeedme_sersic1.npy. Does the same for the text version of the longfeedme.

Motivation:

Converts longfeedme file format into usable instructions for GALFIT (feedme files).

How it works:

Iterates through the ids in the dictionary and creates an obj.in file for each id based on the information in the dictionary. Writes this information to the file in the appropriate format.

Possible improvements:

For the output path ("B) "), it is a lot cleaner if this is just the name of the output instead of the path. Nevertheless, this gets handled when refitting occurs, so this is not currently an issue. To explain this further, the feedme file looks like this:

```
## IMAGE PARAMETERS for 7
A) ../galfit-input-cutouts/galfitsim1_7_drz.fits    # Input data image (FITS file)
B) ../galfitsim1_sersic1/galfitsim1_7_out.fits    # Output data image block
C) ../galfit-input-cutouts/galfitsim1_7_sigma.fits    # Sigma image name (made from data if blank or 'none')
.
.
.
```

And what I'm saying is that it would be cleaner if it were always just:

```
## IMAGE PARAMETERS for 7
A) ../galfit-input-cutouts/galfitsim1_7_drz.fits    # Input data image (FITS file)
B) galfitsim1_7_out.fits    # Output data image block
C) ../galfit-input-cutouts/galfitsim1_7_sigma.fits    # Sigma image name (made from data if blank or 'none')
.
.
.
```

Then all that is important is that you run GALFIT from the folder where the output is supposed to go.

Currently, this is not an issue, but it might have been cleaner to code it this way. For the refitting, the obj.in files get copied into the refit folder. Then, this "B) " entry is changed in the manor described above so that when you run refits from the terminal in the refit folder the output lands in the refit folder (not the _sersic1 folder).

§ 2.3 : run_sersic1

Input(s): base_name_longfeedme_sersic1.npy, feedme files from the previous step
Output(s): output blocks from GALFIT, crash list, outfile
Variable(s): *run_sersic1_list*, *number_sent, display_blocks_sersic1, timer*

Summary:

If yes, runs the obj.in files created in the previous step by calling the function `run_galfit`. If *run_sersic1_list* is "default" then it runs all of them, else it runs the ids in this list. Most of the time I think this should be set to "default". If you want to refit something you should be running *run_refits_sersic1*. The *number_sent* parameter is the number of obj.in files that are sent to the terminal at once (in parallel). If *display_blocks_sersic1* is "yes" then the output blocks (original[1], model[2], residual[3], and mask) are displayed according to the time interval in seconds set by *timer*.

This task prints the elapsed time, number that have crashed, and other information about the run. The crashed fits are saved in crashes.txt and the verbose output from GALFIT is saved in outfile.txt (though it is not ordered due to the parallelism).

Motivation:

Produces GALFIT output.

How it works:

This is probably the most involved task. Works using a while loop. Initially sends obj.in files according to the *number_sent* to the terminal. Once one finishes and appears in the output folder it sends the next galaxy to the terminal. It figures out if something crashed by reading the outfile.txt for the word “crash”. If something crashes it also sends another galaxy to the shell.

The *number_sent* should probably be the number of cpu cores for your machine, however I have been using 10 on a 4 core machine and it works fine. Do not set this to be ridiculously high. Once a feedme file has been sent to the terminal it is no longer in your control (i.e. you cannot ctrl+c to quit it). This being said you can stop the run by aborting the Python script.

If you have *display_blocks_sersic1*==”yes” then you can start figuring out which will need to be refit while you are waiting for GALFIT to finish running. You can also open the fit.log for quantitative information while fitting.

Possible improvements:

Works in its current condition, but could be more stable. Should tell you which galaxies crashed while it is running, not just how many have crashed (though I tried and failed to implement this). Note that the list of crashed galaxies (including their ids) is printed at the end and also saved to the crashes.txt. It is difficult to identify which galaxy ids have crashed while run_sersic1 is actually running because the feedmes are sent to the shell in parallel. They also finish at very different times depending on how long the fit takes. For example, you may send 1, 2, and 3 to the shell initially. Let’s say that 2 finishes first, now 4 is sent to the shell. Suddenly, the word “crash” is counted in the outfile, so there has definitely been one crash, but it isn’t clear whether it is 1, 3, or 4, meanwhile since one crashed we can afford to send another to the shell (#5). The (only) way to tell which has crashed is to wait for 1, 3, and 4 to finish and use process of elimination. Let’s say 4 finishes, so we send #6 to the shell. Let’s say sometime later another “crash” is counted in the outfile. The two crashes could be any two of 1, 3, 5, or 6. If 1 finishes, we know one of the crashes is 3 (assuming we kept track of which had been sent at the time that 3 was running and how many had crashed so far). But I digress, the point is it is doable, but somewhat complicated. There is probably an easier, cleverer way…

§ 2.4 : check_residuals_sersic1

Input(s): output blocks from GALFIT
Output(s): display of output blocks
Variable(s): *check_residuals_sersic1_list*

Summary:

If yes, displays the three GALFIT extensions and the mask for each cutout and prints the output statistics for that specific galaxy. Does this for the ids in *check_residuals_sersic1_list*, if value is “default” goes through all the out files. Calls the function `check_residuals`.

Motivation:

Helps the user figure out which galaxies ought to be refit. Should be used if user did not use *display_blocks_sersic1*

How it works:

User is prompted with "Press any key to continue(q to quit, int to go to specific galaxy):". If the input to this is an integer it will display that id. If the input to this is “q” it will break/quit. Otherwise if the user pushes return or any other key it will move on to displaying the next galaxy (always ascending).

§ 2.5 : plot_stats_sersic1

Input(s): folder with outputs

Output(s): display of group statistics for galaxies

Variable(s):

Summary:

If yes, displays the values from the fit (magnitude, re, etc.) plotted against their error for all the outputs. This will give you a sense of which are outliers and which should be refitted. Also generates a refit list which recommends galaxies which should be refit and prints the reason why it should be refit. Calls the function `plot_stats`.

Motivation:

Helps the user figure out which galaxies ought to be refit.

How it works:

Takes information from the header of the model and plots it.

Possible improvements:

Depending on your Python IDE, you may need to add several plt.show() calls within the `plot_stats` function.

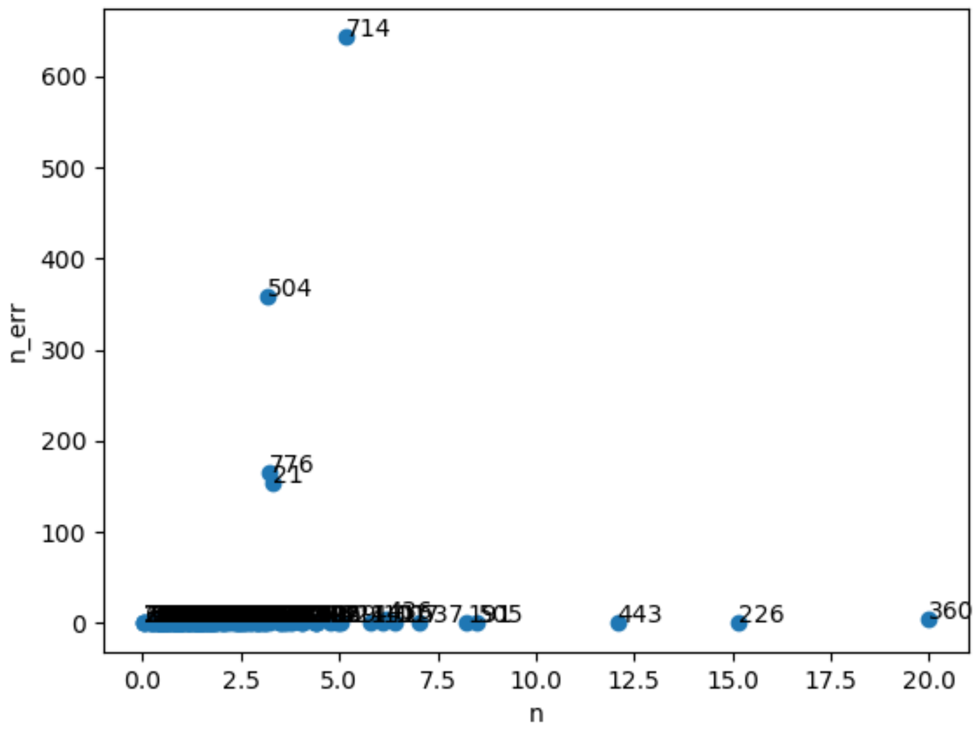


Figure 10 One such example of the plots produced by make_plots. Helps the user clearly see outliers which should be refit.

§ 2.6 : run_refits_sersic1

Input(s):
Output(s): folder of galaxies for refitting
Variable(s): *run_refits_sersic1_list*
Summary:

If yes, creates the refit folder and the initializes the summary of changes file. Then it calls the function `make_refits`. This function copies the obj.in files and out.fits files (if available) from the original sersic1 folder into the sersic1 refits folder. It does this for the ids in *run_refits_sersic1_list*. Opens the summary file (§ 2.6.1) and first obj.in file. Then prompts the user with "Press any key to continue(q to quit, int to go to specific galaxy, n for next):". The idea here is that you open a terminal and change directory to the refit folder. Then you can make changes to the obj.in file and run GALFIT in the terminal:

```
>galfit obj##.in
or
>path/to/your/galfit/binary obj##.in
```

You can then press any key (usually just hit return) to show the new residual and output parameters. If it still needs to be refit then change the parameters again. Continue until a satisfactory fit is achieved or you are pulling your hair output trying to get it to work, then press “n” to move on the next galaxy (make sure to update the obj.in file at the terminal prompt). Repeat this process until you have refit all the galaxies.

Note that when you are refitting galaxies the output from the refit will be considered the final version. This means the out.fits file ought to represent your best effort at refitting the galaxy.

For example, if I change a bunch of parameters, but ultimately realize the original feedme file is better I would need to undo these changes (cmd+z or you could copy the original obj.in file into the refit folder ←would need to change B) path) and then run the original again. This ensures the out.fits file is the best effort.

Motivation:

Helps the user figure out which galaxies ought to be refit. Should be used if user did not use *display_blocks_sersic1*.

How it works:

Works similar to check_residuals_sersic1.

Possible improvements:

Currently works by changing the output path by stripping the directory path. This is so your refit galaxies don’t end up in your original sersic folder, instead they land in the current directory (i.e. the refit folder). See § 2.1 improvements.

§ 2.6.1 : Formatting the Refit Summary File

At the top of the refit summary file are some examples of how to do refits with the appropriate format. I will expand on that information here. The format consists of a comment (begins with #) designating the id number being refit and a reason why it was refit. Examples:

#76, has low error (mag_err<=0.0001), should fit psf to stars

#191, has n >= 6

#217, got nan values for errors

#307, has low error will try for better fit, residual is lousy

Then a line describing the change to be made. Essentially you have three options:

1. Change the initial parameter of a galaxy.
   Format:
   Cutout_ID parameter_to_change old_value new_value

   Examples:

```
643 type4 sersic psf

75 x_cutout1 126.54656982 123
75 y_cutout1 126.69506836 133
75 re1 19.5580674131 4
75 mag1 22.49112701 24

107 sky 0.13667391 0.1382
```

2. Add another component to the cutout.
   Format/Example:

```
id8 75
row8 1000
type8 sersic
x_global8 1012.54656982
y_global8 2378.69506836
x_start8 887
x_end8 1136
y_start8 2253
y_end8 2502
x_cutout8 133
y_cutout8 117
mag8 24
re8 3
n8 1.5
q8 0.61750931
pa8 -152.98328781
```

3. Remove a component to the cutout.
   Format:
   Delete cutout_id component_to_be_removed

   Examples:

```
delete 217 7
delete 217 6
delete 217 5

delete 230 1
```

Lastly, write a comment (beginning with #) about the result of the refitting. Examples:

```
#SExtractor overspilt this object, looks better now, errors make more sense
#Completely fixes problem
#Second edit helped as well, residual looks great, no more nans
```

In the future this could be formalized to be some sort of output flag.

Important things to note:
-If adding a component all attributes need to be present, the easiest way to do this is to go to the longfeedme file and copy and paste the last entry for the cutout id into the summary file. Then increase the component number by one (e.g. mag7 → mag8) and edit the parameters as needed. You do not need to have x_global and y_global be accurate.
-If removing multiple components, the deletions should be done in descending order (as above).
-Deletions should be done last if multiple changes are going to take place (e.g. adding three components and deleting one).
-If the component number is 1 the entire dictionary entry for the id is removed. The object will still exist in other cutouts if *changes_type* is "local".

-When you enact the refits please double check that a few of the changes were made properly made (especially if complicated). You can do this by opening base_name_sersic1_longfeedme_w_refits.txt

§ 2.7 : enact_refit_edits_sersic1

Input(s): refit summary file
Output(s): longfeedme with edits (base_name_sersic1_longfeedme_w_refits)
Variable(s): *changes_type*
Summary:

If yes, enacts the refit changes set forth in the refits summary file on the sersic1 dictionary and outputs this new edited dictionary base_name_sersic1_longfeedme_w_refits. See § 2.5.1 for details on formatting. Calls the function `enact_summary_of_changes_to_longfeedme`.

Motivation:

Updates the sersic1 dictionary to use improved refit initial guesses.

How it works:

Reads the summary file and gets the sersic1 dictionary. Depending on the *changes_type* changes are made locally or globally. If the line starts with "delete" it will remove the designated component (or the entire entry if the component is 1). If the line starts with "id" it knows to add the component in. It expects exactly 16 attributes to be present. If the line starts with an integer it knows to replace the appropriate component with the new value for that id.

Possible improvements:

This needs more testing which is why I say you should **check** a few entries to make sure what you expected would happen actually did.

§ 2.7.1 : What if I want to mask something?

Before I begin, I should mention that it is best if additions or deletions from the mask are made prior to running sersic1 by editing *add_to_mask_list* and *remove_from_mask_list*. However, if you are refitting and decide some objects should be masked then you should first ask yourself if changes will be implemented globally or locally. As of right now, the answer is locally.

If **globally**, then you should simply "delete" the objects to be masked in the summary file, remake the mask adding the necessary ids or using § 1.7.1, and then remake **all** the cutouts.

If **locally** (as this script runs right now), you should "delete" the objects to be masked in the summary file, remake the mask adding the necessary ids or using § 1.7.1, and then **only remake the cutouts that need the new mask** (utilize *cutouts_list_to_make*). This is because if you remake all the cutouts and then go on to run the devauc fit the now masked object entries will still exist in other feedme files (because the changes/deletions made were locally).

§ 2.8 : combine_refits_and_original_to_final_folder_sersic1_and_make_table

Input(s): Refits and original sersic1 fits
Output(s): Final folder with the original and refits, output table, and final dictionary
Variable(s): *changes_type*
Summary:

If yes, copies the original fits and their obj.in files into the final folder (which it makes if it doesn't exist already). Then copies the refit obj.in and out.fits files into the final folder. Then makes fits and txt tables and copies the refit dictionary renaming it the final dictionary. Calls the functions `combine_refits`, `make_output_dict`, `output_dict2tab`, and `make_fits`.

Motivation:

Finalizes the sersic1 fit.

How it works:

By calling `combine_refits` it makes the final folder, then copies all the original fits and obj.in files (if they don't already exist in the final folder). Then copies the refits and their obj.in files to the folder. Only galaxies which have a dictionary entry will be copied in. So, if you deleted a postage stamp from the dictionary during the refits it won't be copied in. Then by calling `make_output_dict` a dictionary of all the output information for all the models in this folder is created. Then `output_dict2tab` makes a txt table and which is made into a fits table using `make_fits`. Tables produced are of the same format of the original Perl scripts. Columns are:

NUMBERI, MTOT_SER, E_MTOT_SER, AE_SER, E_AE_SER, RE_SER, E_RE_SER, LRE_SER, E_LRE_SER, MUE_SER, E_MUE_SER, N_SER, E_N_SER, ARATIO_SER, E_ARATIO_SER, EPS_SER, E_EPS_SER, PA_SER, E_PA_SER, X_SER, E_X_SER, Y_SER, E_Y_SER, SKY_SER, CHISQ_SER, NCOMP_SER, FLAG_SER, MAGin_SER, AEFFin_SER, ARATIOin_SER, PAin_SER, Xin_SER, Yin_SER

The refit dictionary is copied to the final folder and renamed (if it does not already exist in the final folder).

Possible improvements:

Perhaps should overwrite if the files already exist. In any case the user can just delete the folder and it will create the same output if overwriting were implemented.

§ 2.9 : devauc1

The process for running the devauc1 fit is very similar to steps § 2.2 - 2.8. In the following sections I will outline differences, if any.

§ 2.9.1 : make_feedmes_devauc1

The essential question here is what input guesses should be used for the devauc feedme files. Possible options include:

- Original SExtractor guesses
- SExtractor guesses with changes for refitting of sersic1 (global or local)
- the output from sersic1 fits

In the future, all of these will be implemented, but currently the SExtractor guesses with local edits from the sersic1 fit are used (see § 2.0.1 for this justification). Therefore, the sersic1 final dictionary is copied into the newly created devauc folder (appropriately renamed) and then edited (changing the type1 to "devauc" and n1 to "4.0"). Once this is done longfeedmes are created just as before.

§ 2.9.2 : run_devauc1

This does the same as before. Note I chose to make the parameters for *number_sent* and *timer* universal in the sense that there is no *number_sent_sersic1* and *number_sent_devauc1*, I figure this is unnecessary as you can just change the parameters before running.

§ 2.9.3 : check_residuals_devauc1

This does the same as before.

§ 2.9.4 : plot_stats_devauc1

This does the same as before.

§ 2.9.5 : run_refits_devauc1

This does the same as before. Hopefully there are fewer refits/crashes as we are using the "improved" dictionary (i.e. with refits) as initial guesses.

§ 2.9.6 : enact_refit_edits_devauc1

This does the same as before.

§ 2.9.5 : combine_refits_and_original_to_final_folder_devauc1_and_...

This does the same as before.

§ 2.10 : sersic2

The idea behind the sersic2 fits is that we start with a guess for the sersic index which is the same for all galaxies during sersic1 (typically 1.5). This is because SExtractor does not really supply any guess as to the sersic index, though it has been suggested this can be attained through other parameters. Due to this, our galaxies may converge to values closer to this arbitrary n value. To alleviate this concern, we consider galaxies which have a sersic1 output n value greater than 3 and a better reduced chi square value for the devauc fit. For these galaxies we use a starting n value of 4 and rerun the sersic fit (sersic2) on **these galaxies only**. Again, beside this step the process is similar so I outline differences below. The process for sersic2 is typically much quicker, I found that most galaxies converged to similar values as the sersic1 fit.

§ 2.10.1 : make_feedmes_sersic2

Before anything happens, we find the galaxies above which are "eligible" for being refit as sersic2. This is done by comparing the output from devauc and sersic1. Then a sersic2 dictionary is created which adopts the sersic1_final_dictionary guesses, but changes n1 to 4. Then feedme files are made from this sersic2 dictionary.

§ 2.10.2 : run_sersic2

This does the same as before.

§ 2.10.3 : check_residuals_sersic2

This does the same as before.

§ 2.10.4 : plot_stats_sersic2

This does the same as before.

§ 2.10.5 : run_refits_sersic2

This does the same as before. Hopefully there are no refits/crashes as we are using the "improved" dictionary (i.e. with refits) as initial guesses. Here you should look at the resulting output and decide if it has improved from the sersic1 fit. If it has you should take note of the id number and add it to *use_sersic2_list*.

§ 2.10.6 : enact_refit_edits_sersic2

This does the same as before.

§ 2.10.5 : combine_refits_and_original_to_final_folder_sersic2_and_...

Before combining you should have a list of the ones which you want to use the sersic2 results (*use_sersic2_list*). You also have one last chance to remove unruly objects/galaxies from the final catalog (*remove_from_final_catalog_sersic2_list*). Note here that not all the refits are used (as in devauc1 and sersic1) only the ones listed in *use_sersic2_list* otherwise the results come from the sersic1 final folder. Otherwise the steps are the same as before.

§ 2.11 : Closing Remarks

§ 2.11.1 : Flexibility of Running GALFIT

As I have explained in § 2.0.1 there are several ways to run GALFIT and the user may have some predilection about which will give the best results. To accommodate these different methods, I have tried to make this portion of the script more versatile. If you look at the actual code for this portion of the script it is essentially making a call to a function defined above for every if statement. This is unlike previous tasks, for which most of the functionality is written below each if statement. The idea behind this is that by writing functions the user can utilize these functions to change the process in order to suite their preferences. Really all that someone needs to do is figure out the appropriate inputs and outputs (for which I have provided a template in the script).

§ 2.11.2 : Future Improvements

Better measurement of the reduced chi square

One idea I hope to ultimately implement is a more accurate measure of the reduced chi square. This was inspired by a discussion with Inger. Occasionally, fits will have a large chi square value because a star or some neighbor has a bad residual. Therefore, the reduced chi square is not always very meaningful. A more meaningful number would be the reduced chi square calculated from the relevant pixels of the galaxy in question. I think the segmentation map would come in hand for this and a function could be written using the segmentation map, model, and original image to calculate this "zoomed" reduced chi square. It might be important to also consider close neighbors.

Predicting failed fits and automatically rerunning with different inputs

The refitting process is often the most time consuming and requires the most work on behalf of the user. However, often times specific indicators within a fit have relatively straight forward ways to be fixed. One example I experienced is that when the output half-light radius of a galaxy is too low (e.g. 0.01) it typically works to decrease the initial magnitude (this especially occurred during the devauc fitting). In the future, it would be great to have a tool which recognizes and makes these edits automatically, though this is perhaps more difficult then it sounds.

Implementing all different ways to run GALFIT

As has been stated repeatedly throughout this guide, the current implementation handles local changes and does not use auto-updating. If I can afford the time, I will enable all the methods described in §2.0.1.

Double components, accommodating all fit types, and improved PSF creation

I do not think these will be done during my time at Gemini, but it would not be terribly difficult to write code to do the aforementioned.

§ 2.11.3 : Tips for Refitting

I wanted to mention some tips for refitting:

1. If re winds up being too low (or axis ratio) try decreasing the initial magnitude.
2. Don't be afraid to delete or add an object as this can often help significantly.
3. Low errors < 0.0001 typically indicate something is amiss. Often a neighbor has ridiculously high error.
4. Fit stars with type "psf".
5. Increasing the sky value will decrease a large sersic value (e.g. n > 6.5), but make sure this is justified.

6. Make sure faint objects have correct centers. Sometimes faint objects with “migrate” to strange positions, if this occurs you can typically delete the object or mask it.

Make_sims.py

What does make_sims.py do?

In its most basic mode, this script takes ranges of possible parameters, randomly selects from those ranges and uses GALFIT to create simulated models. In its most advanced mode this script takes a PSF and a SExtractor output fits table of an image “to be simulated”, and this “to be simulated” image as inputs. It is then possible to either select randomly or more sophisticatedly interpolate the distribution of galaxy parameters present in the “to be simulated” image. Using these parameters it can then convolve with the supplied PSF and add the galaxy cutouts to randomly selected (possible) positions in the field based on the “to be simulated” image. Noise can be added along the way at many different stages. The sky can also be compared to the “to be simulated” image to make sure the noise is correct. Currently, this script is designed for WFC3 F160W images and single component galaxy profiles, however in it should work for many more types of images (there is no obvious dependence on the type of image) and with some modifications accommodate double component models. This script is very lightweight and quick to run taking about 2 minutes to create a field of 800 PSF convolved galaxies. I’m also particularly proud of the way parameters are interpolated to produce realistic resemblances to the “to be simulated” image (really the SExtractor output fits table). The original intention of this script is to add simulated galaxies to an image, so that their parameter values (as determined by GALFIT) can be compared with their actual values to derive errors on GALFIT measurements (since they sometimes aren’t meaningful). It may also be interesting to see correlations between these errors and angular distances between galaxies as well as how single-component fits of two-component models perform/compare. [As I implement the pairs approach this will change and so will other portions under this part of the manual.]

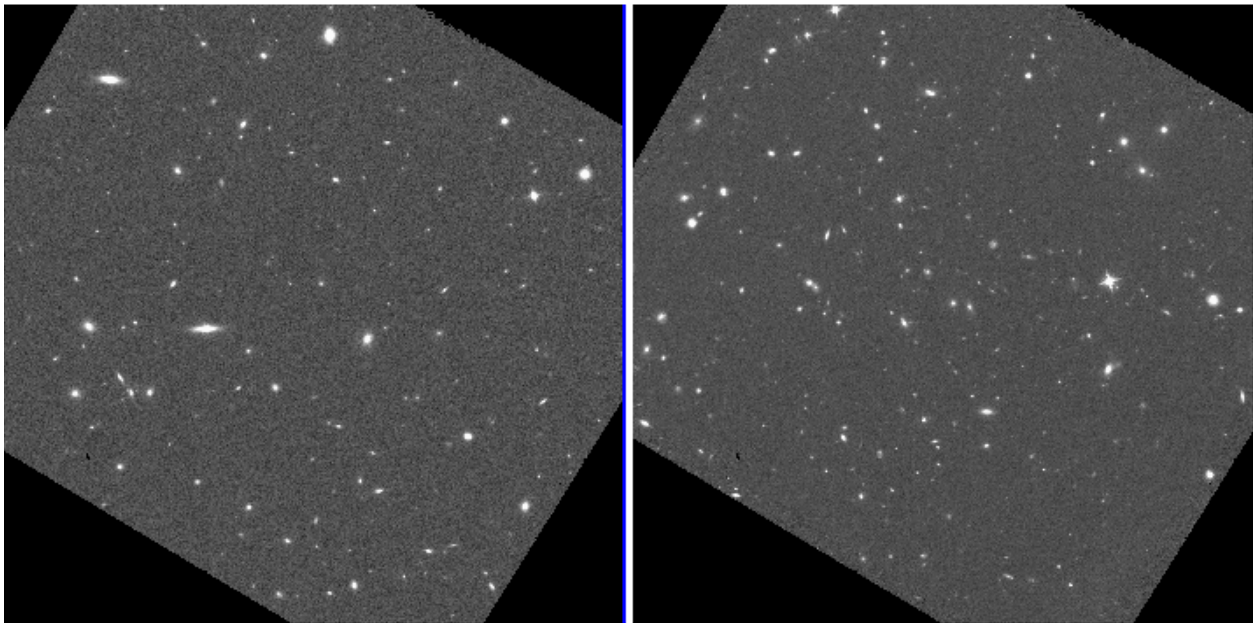


Figure 11 Right: Real drizzled image. Left: Simulated image

What is required to run make_sims.py?

If you are a Gemini intern you already have everything that is required. Currently, make_sims.py runs on Mac/OSx (it may work for Linux systems, but it will not for Windows, GALFIT is not made for Windows). You need Python 2.7 installed on your system, I use the Spyder IDE which comes with the conda distribution. You need the following imports (all of which come with astroconda or can be installed simply using `conda install package`):

Astroconda: http://astroconda.readthedocs.io/en/latest/

```
### Imports
import random
import os
import subprocess
import time
from astropy.io import fits
from astropy.convolution import convolve_fft
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate as interpolate
from drizzlepac import astrodrizzle, ablot
from stwcs import wcsutil
import glob
import pandas as pd
```

From pyraf import iraf: Before anything I would make sure that you can successfully run `from pyraf import iraf` with the IDE or distribution of Python which you are using. If you cannot, this most likely has to do with your Python path (where Python looks for modules to load).

I have a virtual machine (Mac OS) on my personal computer and after entering the astroconda environment ($source activate astroconda) and then entering Python ($python) I had no problems (>from pyraf import iraf). For reference the path to iraf on my system is:
/Users/rpeterson/anaconda/envs/astroconda/lib/python2.7/site-packages/pyraf/iraf.py
You may need to add this *folder* to your Python path. The idea in the script is that it changes directory to your login.cl folder and then loads iraf, so that whatever iraf settings you may have are also loaded.

You also need to install GALFIT (Gemini interns already have GALFIT installed):

GALFIT: https://users.obs.carnegiescience.edu/peng/work/galfit/galfit.html

Where is make_sims.py?

The most current stable versions of make_sims.py is here:

```
/Users/rpeterson/FP/GALFIT_Simulations1/galfitsim1_python_scripts/most_curren
t/make_sims.py
```

Known Limitations

Errors/Bugs

This script has not been thoroughly tested for errors and bugs. With this being said, I expect much fewer errors to arise than do_everything.py, this owing the fact that make_sims.py is in many ways much simpler and straightforward.

Profile types

Currently make_sims.py makes galaxies using the “sersic” function from GALFIT. It currently only makes one component models, although it is anticipated that two component models will be implemented by the end of my internship. It is foreseeable that option to add a certain number of stars will be implemented as well.

Inputs and Conventions

General Naming Conventions

Similar to do_everything.py, when I refer to an object its variable or output name will typically be placed in parenthesis immediately following the object reference. For example, “the base folder (*base_folder*) is the place where make_sims.py creates all outputs.” I believe the naming conventions are more straightforward than do_everything.py. I have tried to give appropriate names and use clear Python standards. When I refer to code in particular I will use a `special font`.

As stated above the base folder (*base_folder*) is the folder where all output is created. For example:

```
base_folder=”path/for/your/simulated/galaxies/”
```

Unlike do_everything.py, *base_folder* **should** have a slash at the end. This is no longer a huge deal, because the code will fix this. For better or worse the naming conventions for make_sims.py are more general than do_everything.py, there is no variable (*base_name*). Instead the names are straightforward and indistinct (beside the id number).

Inputs
If you want simulated galaxies to be convolved with a PSF then the variable (*psf_image*) should be the path to the PSF. If you want to perform the interpolation chain and create galaxies which represent actual galaxies you can supply *chain_data1 and chain_data2* to do this (see § 3.0.1). These are fits tables output from SExtractor. If the image you want to simulate is rotated (as mine was in the image below) you can supply this as the input for *image_to_sim* and it will also produce an image which masks these same zero regions. It also will only place galaxies at the possible coordinates so you still get the same number as *number_of_sims*. Otherwise, if you just want a regular box image you can set *image_to_sim*==”default” and it will produce an image of the size given by *large_field_xdim*, *large_field_ydim*.

Products
Outputs are created based on what is directed to be done by the user. All of the outputs go into the base folder (*base_folder*) including: objin files, models, convolved with the PSF models, noise added models/convolved with the PSF models with noise, and all full fields. Again, these objects are only made/modified if the user wills it.

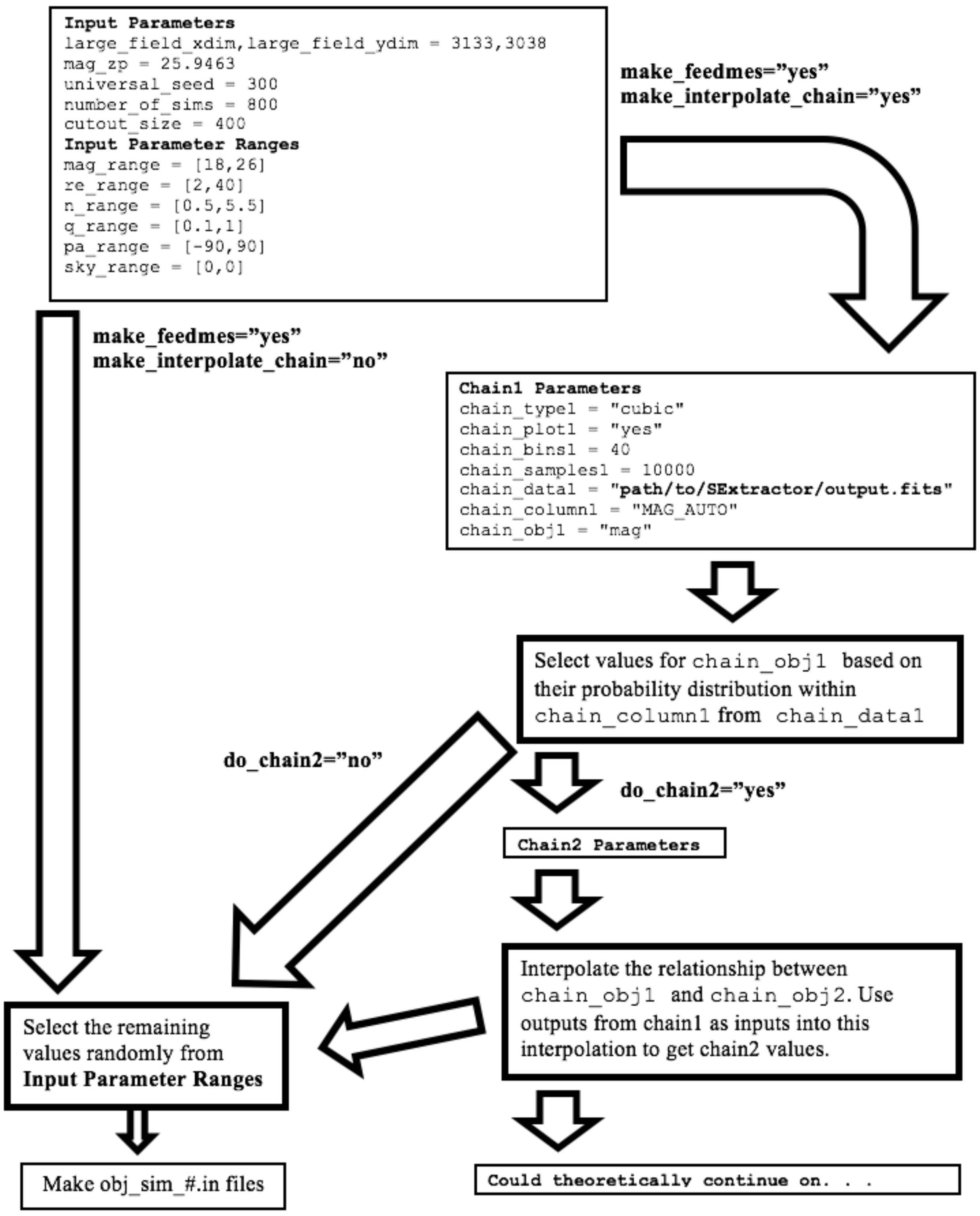
Universal Seed
There is a considerable amount of randomness used to make the simulations, to enable reproducibility I have implement a *universal_seed*. This value can either be an integer or `None`, if it is an integer and the script is run twice in a row the output will be the same, if this value is changed the products will be entirely different. Likewise, `None` will have (pseudo*) random output.

*for these purposes its essentially completely random, but I wouldn’t use Python’s random module for purposes where security was a priority.

Section 3: Running make_sims.py

§ 3.0 : make_feedmes

§ 3.0 : Flow Chart for make_feedmes



```
Input Parameters
large_field_xdim,large_field_ydim = 3133,3038
mag_zp = 25.9463
universal_seed = 300
number_of_sims = 800
cutout_size = 400
Input Parameter Ranges
mag_range = [18,26]
re_range = [2,40]
n_range = [0.5,5.5]
q_range = [0.1,1]
pa_range = [-90,90]
sky_range = [0,0]
```

```
Chain1 Parameters
chain_type1 = "cubic"
chain_plot1 = "yes"
chain_bins1 = 40
chain_samples1 = 10000
chain_data1 = "path/to/SExtractor/output.fits"
chain_column1 = "MAG_AUTO"
chain_obj1 = "mag"
```

make_feedmes=”yes”
make_interpolate_chain=”yes”
make_feedmes=”yes”
make_interpolate_chain=”no”
Select values for chain_obj1 based on their probability distribution within chain_column1 from chain_data1
do_chain2=”no”
do_chain2=”yes”
Chain2 Parameters
Interpolate the relationship between chain_obj1 and chain_obj2. Use outputs from chain1 as inputs into this interpolation to get chain2 values.
Select the remaining values randomly from Input Parameter Ranges
Make obj_sim_#.in files
Could theoretically continue on. . .

Figure 12 Flow chart detailing the method for chain interpolation.

§ 3.0.0 : make_feedmes: random

Input(s):
Output(s): Object in files (obj.in) (as many as *number_of_sims*)
Variable(s): feedme_verbose, mag_range, re_range, n_range, q_range, pa_range, number_of_sims, cutout_size, sky_range, mag_zp

Summary:

If yes and *make_interpolate_chain*==”no”, selects randomly from the ranges of parameters set by the variables above to create individual feedme files (e.g. obj_sim_46.in). Makes as many as *number_of_sims*, *cutout_size* is the “diameter” size of a square cutout (I normally make this plenty big ~400, must be even). If you don’t want any sky added set *sky_range*=[0,0]. The other variables are what they say they are.

If yes and *make_interpolate_chain*==”yes” see § 3.0.1.

Motivation:

Creates the “instructions” for GALFIT to create the simulated galaxies.

How it works:

Uses the definition:
```
comp1(file1,cutout_size,mag_range,re_range,n_range,q_range,
pa_range)
```
Which calls `random.uniform` which will uniformly select from the ranges provided.

§ 3.0.1 : make_feedmes : chain_interpolate

Input(s): SExtractor Output Table (tab_all)
Output(s): More accurate object in files (obj.in), Output Plots
Variable(s): chain_type1, chain_plot1, chain_bins1, chain_samples1, chain_data1, chain_column1, chain_obj1, (…same if chain2 is implemented), feedme_verbose, mag_range, re_range, n_range, q_range, pa_range, number_of_sims, cutout_size, sky_range, mag_zp

Summary:

If yes and *make_interpolate_chain*==”yes”, uses the “chain” variables, most notably the fits table provided by chain_data1, to make galaxies which reflect the sample which you are trying to simulate. The best way to explain this is through a walkthrough of the process. My current chain1 variables are defined as:
```
    chain_type1 = "cubic"
    chain_plot1 = "yes"
    chain_bins1 = 40
    chain_samples1 = 10000
    chain_data1 = "/Users/rpeterson/FP/UV-105842_id6p01/UV-
105842_id6p01_phot/uv-105842_d6p-01-284.0-
f160wtab_all.fits"
    chain_column1 = "MAG_AUTO"
    chain_obj1 = "mag"
```

The code will do the following with this information:

1. Get the data in *chain_column1* from *chain_data1* (I recommend using magnitude as your first chain object).
2. Make a histogram of this distribution using *chain_bins1* as the number of bins used. I recommend an intermediate number like 40, but it really depends on the number of data points in *chain_column1* and their range. The histogram for the original data is in purple/blue below.
3. Interpolate the histogram, which acts as a probability distribution function, using the interpolation method for *chain_type1*. The cubic interpolation for the original data is the green line below.
4. Create a range of values to select from based on min and max of the interpolation (e.g. for magnitude this might be 18.5-26), so we create a discrete array which would look something like [18.5002, 18.5004, … 25.998, 26]. In reality, the number of discrete options is currently 100,000 I see no reason to make this more precise. This is called `x_chain`.
5. Get values for our interpolation (PDF) using `x_chain`. This is called `p`.
6. Use np.random.choice to select from `x_chain` with the weights from `p`. A histogram of these values is plotted in red below. These are the magnitudes of our sample `chain_array.`

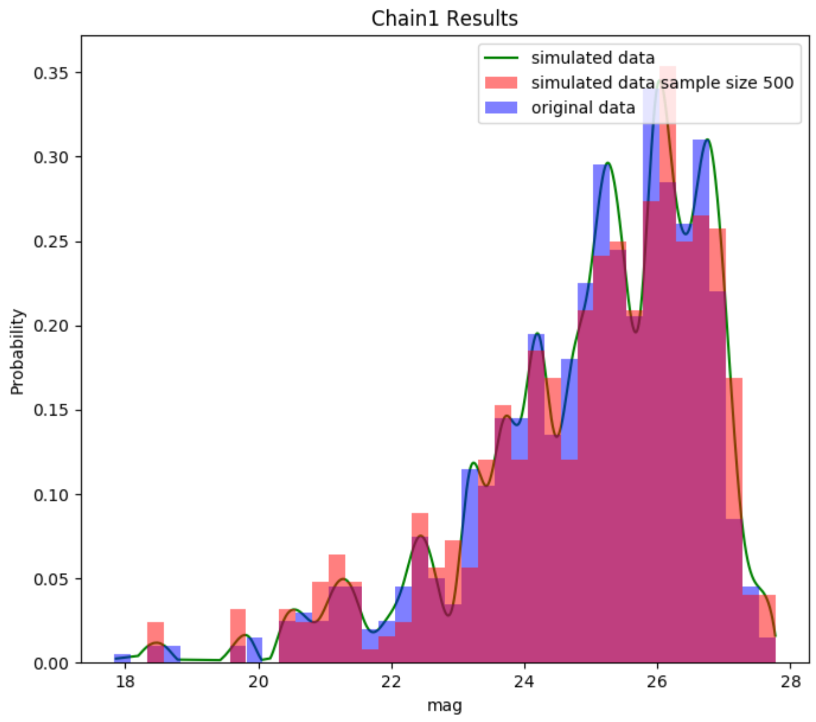


Figure 13 Output from Chain1 (probability distribution of magnitudes). In blue/purple is the real data from the SExtractor run on one of my trial fields. In red is the distribution of simulated data. In green is the cubic interpolation of real data.

Chain2 proceeds as follows:

```
    do_chain2 = "yes"
    if do_chain2=="yes":
        chain_type2 = "linear"
        chain_plot2 = "yes"
        chain_data2 = "/Users/rpeterson/FP/UV-
105842_id6p01/UV-105842_id6p01_phot/uv-105842_d6p-01-284.0-
f160wtab_all.fits"
        chain_column2 = "FLUX_RADIUS"
        chain_obj2 = "re"
```

7. Interpolate the relationship between *chain_column1* and *chain_column2* from *chain_data1* and *chain_data2*, respectively. I anticipate that most of the time *chain_data1* and *chain_data2* would be in the same output fits file from SExtractor. The interpolation between magnitude and effective radius is in green below.
8. Use the `chain_array1` values as inputs to the interpolation function from step 7. These are the effective radii shown in red below (original data is blue).

This process could potentially continue on to chain3 and so on, but the correlations and interpolations would probably begin to be as accurate as randomly selecting.

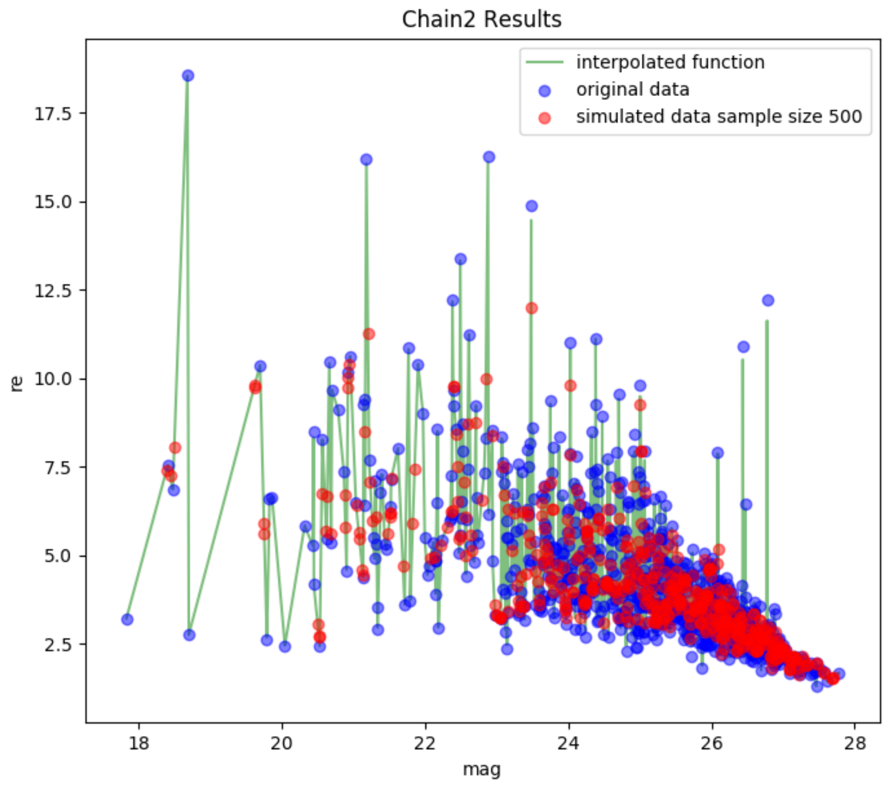


Figure 14 Chain2 results. In blue is the original re (FLUX_RADIUS) vs. mag distribution. In red are the simulated data. In green is the interpolation between mag and re for the original data.

Motivation:

Creates the “instructions” for GALFIT to create the simulated galaxies.

How it works:

See summary above. Uses interpolate.interp1d to interpolate the data.

§ 3.0.2 : make_pairs

Input(s):

Output(s): Object in files (obj.in) (as many as *number_of_sims*)

Variable(s): pixel_distance_steps, luminosity_ratio_steps, re_ratio_steps, n_steps, interp_re

Summary:

If yes, makes pairs of galaxies according to the ratios and values in pixel_distance_steps, luminosity_ratio_steps, re_ratio_steps, and n_steps. The first half of galaxies (parent galaxies) is made either randomly or being interpolated. The second half (child galaxies) is made by applying the ratios and values in the steps variables.

It should be that len(pixel_distance_steps)*len(luminosity_ratio_steps)*len(re_ratio_steps)*len(n_steps) equal ½*number_of_sims. If interp_re=”yes”, then the effective radius will be interpolated. In this case you can assume the length of re_ratio_steps to be 1.

Motivation:

Investigate how GALFIT errors are affected by nearest neighbor distance.

How it works:

A list of tuples named aptly “pairs” is created. When the parameter values are being selected, the random or interpolated values are overwritten by the pairs value for the second half of galaxies. This information is written to the input dictionary.

Placing pairs into the image is somewhat more complicated. The first half are placed randomly. The only constraint for parents is that they are placed at least max(pixel_distance_steps) pixels away from any other parent. If this cannot be achieved in a reasonable amount of time (1 second) the possible coordinate positions are reevaluated and parents continue to get placed in these revised possible coordinates. When there are no more possible positions (hopefully max(pixel_distance_steps) and/or number_to_add are/is small enough that this doesn’t happen), the child galaxies are placed.

The child galaxies are similarly placed randomly at the possible coordinates given by a specified distance from the parent. If it takes more than a second to randomly find a possible position, all the possible positions are analyzed. If there are no possible positions the child is skipped. Skipped galaxies are removed from the input dictionary.

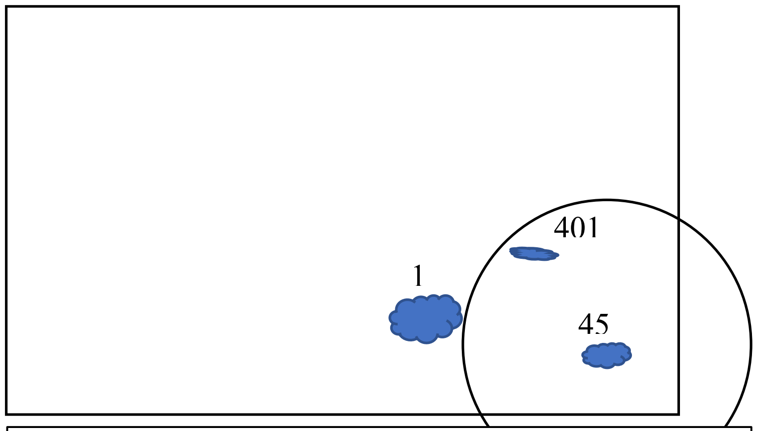


Figure 15 Example: Galaxies 1 and 401 are a parent and child galaxy pair. 45 is a parent galaxy whose child galaxy would be skipped because there are no possible positions to put place it. The circle around 45 represents the possible positions for 45's child galaxy such that the pixel_distance_steps value is satisfied. This should rarely happen if max(pixel_distance_steps) and/or number_to_add are/is small enough.

§ 3.1 : make_galaxies

Input(s): feedmes from § 3.0
Output(s): GALFIT output models (galout_#_model.fits)
Variable(s): galfit_binary, make_galaxies_verbose

Summary:

If yes, creates the model described in the feedme files by executing GALFIT. If you can run GALFIT from the terminal by just typing "galfit" then *galfit_binary* should be "galfit", else it should be the path to the executable (e.g. "/net/frigg/Users/inger/bin/galfit").

See upper left of Figure 15.

Motivation:

Create simulated galaxies.

How it works:

Uses the command `subprocess.Popen([command,name])` in a loop to execute feedmes with GALFIT.

§ 3.2 : make_convolve

Input(s): GALFIT output models from § 3.1, PSF
Output(s): Convolved output models (galout_#_model_convolved.fits)
Variable(s):

Summary:

If yes, convolves the model images with the PSF provided in *psf_image*. PSF need not be the same size as *cutout_size*.

See lower left of Figure 15.

Motivation:

Real image data is convoluted by the PSF so this step imparts this effect on the simulated data.

How it works:

Uses this function from astropy.convolution to convolve the simulated data with the PSF:

```
convolve_fft(sim[0].data,psf[0].data,boundary="wrap",normal
ize_kernel=True)
```

Possible improvements:

Investigate how this convolution function compares to the way GALFIT convolves images. Perhaps there is a way to also convolve using GALFIT (in the same sense that it makes a model for you if you don't supply an input image).

§ 3.3 : mknoise_cutouts

Input(s): Convolved or non-convolved output models
Output(s): Models with noise added to them (galout_#_model_convolved_noised.fits)
Variable(s): background, gain, rdnoise, poisson

Summary:

If yes, runs the IRAF task "mknoise" to add noise to the individual models.

See lower right of Figure 15.

Motivation:

This will be useful for when you would like to add galaxies to a real image.

How it works:

Uses IRAF task "mknoise":

```
iraf.mknoise(input_name,output="galout_"+str(i)+"_model_convolve
d_noised.fits",background=background,gain=gain,rdnoise=rdnoise,p
oisson=poisson,seed=seed)
```

Possible improvements:

Takes some time to run if *number_of_sims* is large, could maybe parallelize.

§ 3.4 : display_5

Summary:

If yes, displays 5 randomly selected galaxies and their feedme files.

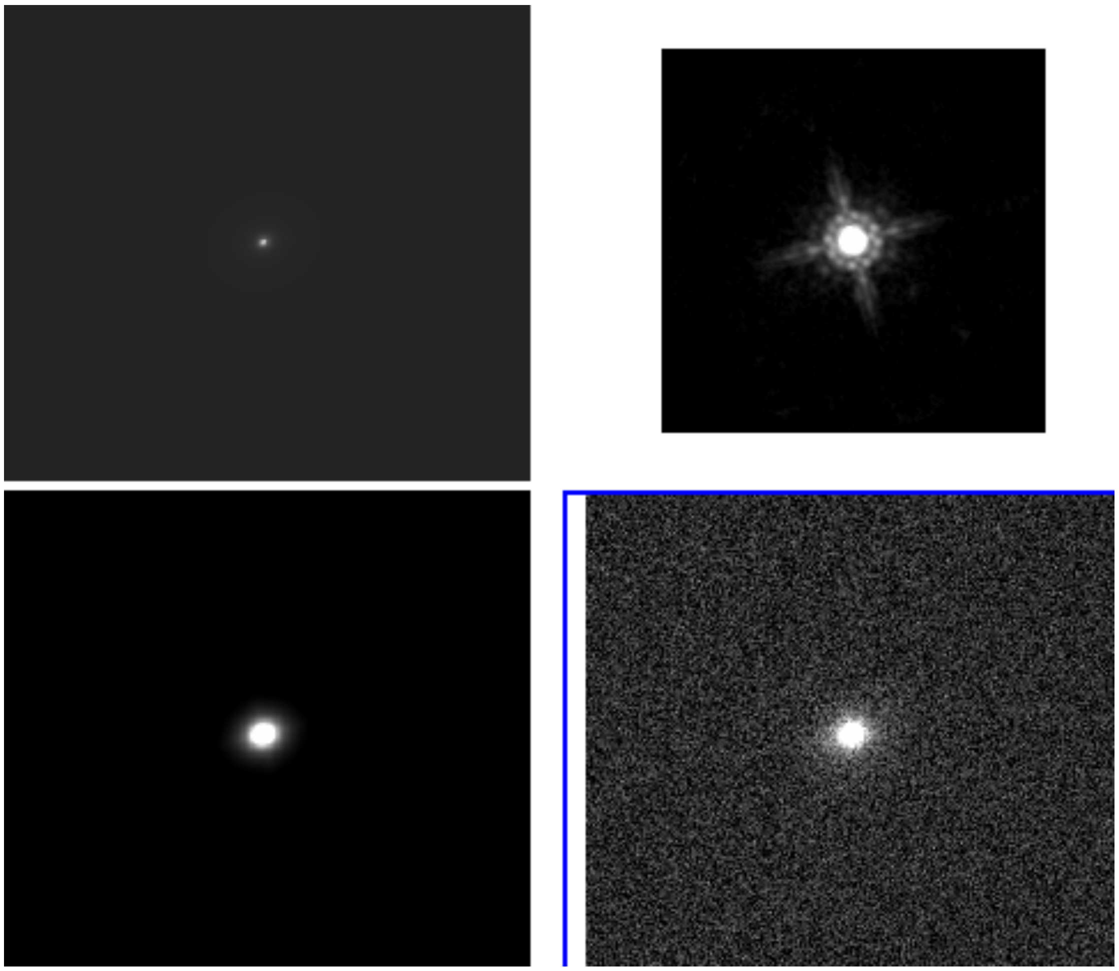


Figure 16 Upper left: model produced by make_galaxies. Upper right: my PSF model. Lower left: the convolution of the model and the PSF. Lower Right: the convolved model with noise added using IRAF MKNOISE.

Motivation:

Serves as a check to make sure simulated galaxies are adequate.

How it works:

os.system call to ds9 to display the images. For a given random number within the number of simulations, looks for the model, psf, convolved model, noised model, and the convolved noised model. The images that exist are displayed in a ds9 window for each of the 5 randomly selected galaxies.

§ 3.5 : create_large_field

Input(s): Output from § 3.1, 3.2, or 3.3, Image to simulate (optional).
Output(s): Full field image

Variable(s): display_large_field, image_to_sim, sky_value, add_mknoise_large1 [background1,etc.], add_models, add_models_verbose, which_to_add, how_many_to_add, label, label_verbose, save, add_mknoise_large2 [background1,etc.], compare_sky

Summary:

If yes, runs the options specified by variables to produce a full field image. In the following sections I will explain each of the steps in this process.

Motivation:

Create a field of galaxies rather than just individual cutouts.

How it works:

See create_large_field flow chart.

§ 3.5 : Flow Chart for create_large_field

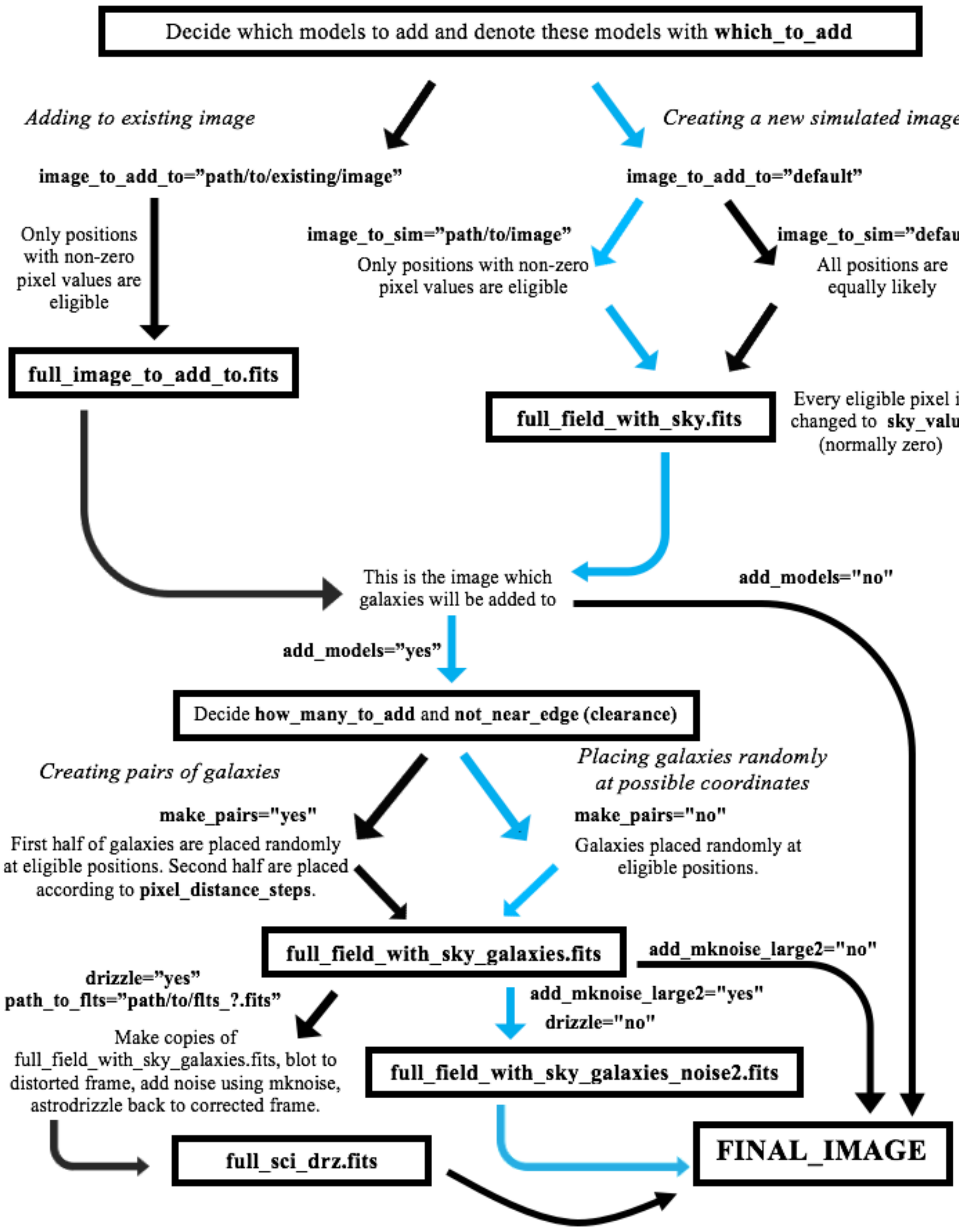

Decide which models to add and denote these models with which_to_add
Adding to existing image
Creating a new simulated image
image_to_add_to="path/to/existing/image"
image_to_add_to="default"
Only positions with non-zero pixel values are eligible
image_to_sim="path/to/image"
Only positions with non-zero pixel values are eligible
image_to_sim="default"
All positions are equally likely
full_image_to_add_to.fits
full_field_with_sky.fits
Every eligible pixel is changed to sky_value (normally zero)
This is the image which galaxies will be added to
add_models="no"
add_models="yes"
Decide how_many_to_add and not_near_edge (clearance)
Creating pairs of galaxies
Placing galaxies randomly at possible coordinates
make_pairs="yes"
First half of galaxies are placed randomly at eligible positions. Second half are placed according to pixel_distance_steps.
make_pairs="no"
Galaxies placed randomly at eligible positions.
full_field_with_sky_galaxies.fits
add_mknoise_large2="no"
drizzle="yes"
path_to_flts="path/to/flts_?.fits"
add_mknoise_large2="yes"
drizzle="no"
Make copies of full_field_with_sky_galaxies.fits, blot to distorted frame, add noise using mknoise, astrodrizzle back to corrected frame.
full_field_with_sky_galaxies_noise2.fits
full_sci_drz.fits
FINAL_IMAGE

Figure 17 Flow chart for how the full_field image is created.

§ 3.5.0 : full_field_with_sky.fits

Input(s): image_to_sim
Output(s): full_field_with_sky.fits

Variable(s): sky_value, large_field_xdim, large_field_ydim

Summary:

If yes and *image_to_sim*="default", creates an image of the size designated by *large_field_xdim, large_field_ydim* for which all the pixels are *sky_value*.

If yes and an *image_to_sim* is supplied, creates an image of the same size as *image_to_sim* where pixel values which are not zero within *image_to_sim* are made *sky_value* (typically zero).

Motivation:

Create a "blank canvas" for galaxies to be added to.

How it works:

First deletes "full*" and "*.reg" files within your *base_folder*. If you would like the output be saved see § 3.5.4. If *image_to_sim*="default" it creates an image where every pixel value is 1. Otherwise *image_to_sim* is copied to the *base_folder* and renamed full_field_with_sky.fits. Then it updates the non-zero pixels values within full_field_with_sky.fits. These non-zero pixels are given the constant value *sky_value* (typically zero).

Possible improvements:

Task isn't very useful as the same could be accomplished by add_mknoise_large1. Nevertheless, makes a "canvas" for galaxies to be added.

§ 3.5.1 : add_mknoise_large1

Input(s): full_field_with_sky.fits
Output(s): full_field_with_sky_noise.fits
Variable(s): background1, gain1, rdnoise1, poisson1

Summary:

If yes uses the IRAF task mknoise to add noise to full_field_with_sky.fits.

Motivation:

Add noise prior to galaxies being added to the image.

How it works:

Runs IRAF task mknoise.

§ 3.5.2 : add_models

Input(s): full_field_with_sky.fits **or** full_field_with_sky_noise.fits
Output(s): full_field_with_sky_galaxies.fits **or** full_field_with_sky_noise_galaxies.fits
Variable(s): add_models, add_models_verbose, label, label_verbose, which_to_add, how_many_to_add, not_near_edge, clearance

Summary:

Adds the models indicated by *which_to_add* to whichever is the working image so far. The number it adds is *how_many_to_add*. Only adds galaxies at non-zero pixel values within *image_to_sim*. If *not_near_edge*=”yes”, the galaxies will not be placed near the edge (determined by the value of *clearance* [pixels]). This makes an assumption that legitimate sky values within *image_to_sim* are not exactly zero. This was the simplest way I could think of to adequately place galaxies at reasonable positions within my rotated images. Because cutouts are square, this is also used to clip the cutouts from the edge of the image (see below). This has the side effect that saturated regions (if pixel values are zero) will also be present in the simulated image.

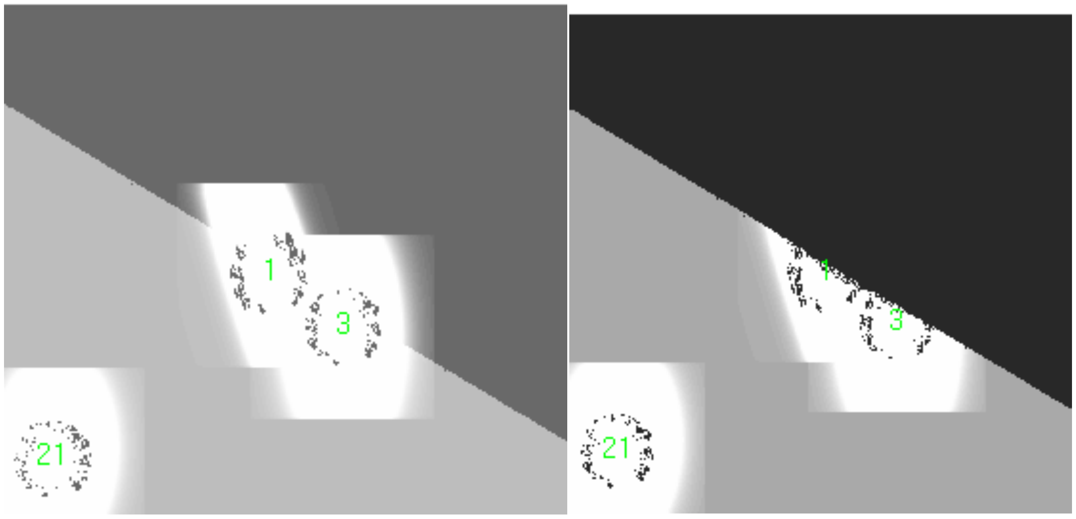


Figure 18 Clipping of cutouts according to zero-valued pixels in image_to_sim.

If *image_to_sim*=”default”, all positions within *large_field_xdim, large_field_ydim* are equally likely.

Motivation:

Add galaxies to a large field image.

How it works:

Finds indexes of non-zero elements with the image array of *image_to_sim*, these are the possible coordinate locations. Selects randomly from these coordinate tuples. Finds xmin, xmax, ymin, ymax for both the cutout and the large image array and trims these values if necessary (e.g. the galaxy cutout is off edge, this took a bit of thinking to do correctly). Adds the cutout array to the large field array. Labelling occurs during this stage, if desired by the user it will create a ds9 region file with ids/locations of galaxies added.

Possible improvements:

The list of possible coordinates is tremendously large, so there are possible speed considerations here. Don’t really need possible coordinates either in the case that *image_to_sim*=”default” and could just select random.randint(0, large_field_xdim)…

To address the case of a legitimate sky value being zero, it might be possible to define the diamond like region of possible coordinates rather than just the non-zero pixels. There are several other ways I can think of, but they require a fair amount of thought and work to implemented correctly.

§ 3.5.3 : display_large_field, label, save

Input(s): final_image
Output(s): ds9 display and/or saved file
Variable(s): label_verbose

Summary:

Doing these 3 in 1.

If *display_large_field*=”yes”, it will display the final_image as designated by the inputs. If *label*=”yes”, galaxies are labelled according to the region file made in § 3.5.2. If *save*=”yes” and a saved image already exists, prompts the user whether they want to overwrite or not, else saves the image with “_saved.fits” as the appended file name.

Motivation:

Provides useful outputs for displaying, labeling, and saving output of *create_large_field*.

How it works:

See summary.

§ 3.5.4 : add_mknoise_large2

Input(s): full_field_with_sky_galaxies.fits **or** full_field_with_sky_noise_galaxies.fits
Output(s): full_field_with_sky_galaxies_noise2.fits **or** full_field_with_sky_noise_galaxies_noise2.fits
Variable(s): background2, gain2, rdnoise2, poisson2

Summary:

If yes uses the IRAF task mknoise to add noise to the working image so far.

Motivation:

Add noise after galaxies have been added to the image.

How it works:

Runs IRAF task mknoise.

§ 3.5.5 : drizzle

Input(s): full_field_with_sky_galaxies.fits **or** full_field_with_sky_noise_galaxies.fits
Output(s): full_drz_sci.fits (drz), full_drz_wht.fits (wht), full_drz_ctx.fits (ctx)

Variable(s): background2, gain2, rdnoise2, poisson2, path_to_flts, display_blots, backgrounds (list of flt backgrounds), compare_noised_blots_sky, final_bits, final_pixfrac, final_fillval, final_scale, final_rot, resetbits, compare_final_drz, path_to_actual_driz

Summary:

This has very recently been added as a feature. If "yes", creates drizzled output. If "yes", then your image_to_sim should be the real final drizzled image. Prior to getting to the drizzle stage the simulated image is created to look like the real final drizzled image (i.e. image_to_sim).

1. This image is copied into individual images
(e.g. full_field_with_sky_galaxies_1.fits, full_field_with_sky_galaxies_2.fits, etc.).
2. Then the first extension of the real flt images are copied into simple fits files
(e.g. id6p01xyq_flt_simple.fits, id6p01y0q_flt_simple.fits, etc.)
3. Simulated images are blotted back to the distorted frame
(e.g. full_field_with_sky_galaxies_1_distorted.fits, etc.)
~~4. Obtain coordinates with TweakReg~~ (I don't think this is necessary)
5. Noise is added according to background2, gain2, rdnoise2, poisson2. Since the background values in the flt images may be slightly different from one another, the user may provide a list of background values (backgrounds), and these will be used. If it is OK to use background2 value for all the flt images, the user should set backgrounds to "default". Seed value is unique for each image so they don't have the same noise. I recommend compare_noised_blots_sky="yes", so the user can verify the values.
(e.g. full_field_with_sky_galaxies_1_distorted_noise2.fits)
6. Build the other extensions for astrodrizzle
    ext 0 – header taken from ext 0 of the real flt (no data array)
    ext 1 – data array is the noised simulated image, header is from ext 1 of the real flt
    ext 3 – data quality array is changed to all ones, header is from ext 3 of real flt
(e.g. sim_flt_1.fits)
7. Astrodrizzle the sim_flt_?.fits images
(e.g. full_drz_sci.fits)
I recommend comparing the sky of this final product with path_to_actual_driz (i.e. compare_final_drz="yes")

Motivation:

Simulate correlated noise.

How it works:

See summary.

§ 3.5.6 : compare_sky

Input(s): final_image
Output(s): plots comparing the sky with *image_to_sim*.
Variable(s):

Summary:

If yes, it will show the comparison of sky values for *image_to_sim* and your *final_image*. Essentially, plots histograms of the sky and returns sky statistics. I originally wrote this to use specific regions, but it was tedious to be entering specific coordinates in every time. Therefore, it simply analyzes the entire image. If you want to compare specific regions use the definition `sky_compare` and define x, y coordinates.

Motivation:

Allows you to compare the sky of a simulated large field to the actual sky of *image_to_sim*.

How it works:

Gets the array for the entire image, figures out the median. If clean="yes" it will neglect outliers (often cosmic rays) less than median/3 and greater than 1.66*median. It selects a range for the histogram (median - 3*std, median + 3*std).

References

1. Python 2.7: https://www.python.org/download/releases/2.7/, Guido van Rossum
2. SExtractor 2.19.5: https://www.astromatic.net/software/sextractor,
   Bertin, E., & Arnouts, S. 1996, A&AS, 117, 393
3. GALFIT 3.0+: https://users.obs.carnegiescience.edu/peng/work/galfit/galfit.html,
   Peng et al. 2010, AJ, 139, 2097
4. Ciotti, L., & Bertin, G. 1999, A&A, 352, 447
5. GALFIT manual: https://users.obs.carnegiescience.edu/peng/work/galfit/README.pdf
6. GALFIT FAQ: https://users.obs.carnegiescience.edu/peng/work/galfit/TFAQ.html
7. Koekemoer, A. M. et al. 2011, ApJS, 197, 36
8. Casertano, S., et al. 2000, AJ, 120, 2747