Internship Documentation:
Transient Events Statistics for MACSJ1149.5+2223

Author: Emma Geoffray
Supervisors: Christoph Schäfer and Jean-Paul Kneib

September 2018

Contents

1 Motivation

The code we describe in this documentation has the following goal : to prove that clusters can be used as cosmic telescopes to observe individual far away stars (that would be impossible for us to detect otherwise at present). Most of these observations should take the form of transient events much like the one observed by [1].

In order to accomplish this, the code strives to make statistics of such transient events in the case of the cluster MACSJ1149.5+2223. Indeed, this cluster is one of the six HFF clusters [2], thus we have very deep fields of it. Furthemore, we are confident that the lens models derived for it using LENSTOOL are sufficiently accurate to make accurate statistics. *(No paper given for the LENSTOOL model.)*

However, the analysis of only these lens models is not sufficient to draw any conclusion. Indeed, for transient events, the micro-lensing due to cluster stars is important.[3] To study this, the code will use the micro-lensing tool GERLUMPH.[4] It will first generate GERLUMPH magnification μ maps using as input the values of dimensionless surface mass density κ and shear γ derived using LENSTOOL as well as values of the flux measured by HST treated so that only the red light, which traces best cluster stars, is taken into account.[5]

The ultimate goal is to predict whether or not we would be able (using HST monitoring of this cluster) to observe Pop III stars (the first stars to light up in our Universe). A good number of papers try, using different methods, to predict what these stars should look like. A good summary of where the research lies is given by [6]. *(Contact me directly if you need more details.)* Most of them agree on the fact that a lot of the Pop III stars should be very massive which is what gives us hope that we could observe them in this way. Still the research is not, as of yet, conclusive enough to make predictions about the Pop III IMF (Initial Mass Function). This is somewhat of a problem for this work, as good statistics about the number of events one can hope to observe depends on this parameter.

2 Code Organisation

2.1 General Introduction

The code developed to achieve this goal is divided into three files: Run.py, Make_data_cube.py and Definitions.py. The most important one for a person wishing to only use the code as simply as possible is the Run.py file. It is also the file one should modify to achieve the desired statistics.

We chose an oriented object approach to the problem meaning that we developed three classes `Simulation`, `Source` and `Cluster` to collect all the parameters needed to make the statistics. These are defined in the script Definitions.py as well all other functions needed for the statistics.

Let us insist on the fact that one needs a GERLUMPH executable file "lenser_gpu" in the same directory as the three python files to run the code. One also needs to store the LENSTOOL FITS files "kappa_map.fits", "gamma_map.fits" (named as such), and optionally the observational data "obs_map.fits", in the same directory as well for the code to give results. One should finally also include the python script "N2mu_bin2fits.py" to the same folder (in order to allow the conversion of DAT files created by the GERLUMPH executable to FITS file easier to use in our case).

2.2 Run.py file

The script Run.py works by taking three consecutive steps. The first one is the construction of the data that will be analysed later on (if such a file is not already provided). This is done by calling the script Make_data_cube.py (see §2.3 for more details). The second step is to create instances of three classes : `Simulation`, `Source`, `Cluster` (defined in the file Definitions.py). These will contain all the specific parameters needed to do transient events statistics (see §2.4 for more details). The last one is to call two `Cluster` methods: basic_stats and advanced_stats (defined in the file Definitions.py) to actually generate the maps microlensing GERLUMPH maps and carry out transient events statistics for the chosen parameters and plot the results in a folder named "Plots" (sorting them using the source's mass and redshift).

A step-by-step example of the "Run.py default" mode is presented in §3. Once one has mastered this particular example, one can feel free to modify the "Run.py perso" mode to achieve one's desired statistics.

2.3 Make_data_cube.py file

This script constructs the cube of data to be analysed: matching κ, γ and flux F maps for the target cluster MACSJ1149.5+2223.

First, it uses the FITS files "kappa_map.fits", "gamma_map.fits" and "obs_map.fits" (which have to be named as such and placed in the same folder as the script) in order to interpolate the observational data onto a grid which corresponds to the one from the κ and γ maps. More precisely (since the observational map is smaller in terms of WCS coordinates in our case), it finds for the observation map corners the closest κ map pixels (using their respective WCS coordinates). Then the interpolation process takes place (using the `scipy` function `interpolate.interp2d`), but only if the observation map is "straight". Meaning the bottom right and bottom left corners are on the same line of the κ map, while the bottom left and upper left corners are on the same column of the κ map. After that, the κ and γ maps are restricted to the size of this new interpolated map.

Finally, the observational data (given in units of e^-/s) is converted to a flux (given in units of $erg/s/cm^2$) via multiplication: first by the factor PHOTFLAM (found in the observational data's header, as explained in the HST data handbooks [7], and second by the bandwidth found in [8] (we here make the assumption that the filter is almost flat). The three maps are then saved to a FITS file named data_cube.fits, whose header corresponds to the one of the κ map to which we have made a few modifications for consistency.

An important remark is that the observational data used in our case is free of the blue background light (has it has been previously removed using the work of [5]). The second important remark is that, to compute the flux, we use only the data from the F814W band, as it is the one that best traces cluster galaxies' light in our case.

Let us also say that the script can also generate a cube of data with a default flux value, provided is is called as such:

```
python Make_data_cube.py default
```

This can be useful when one does not have the observational data on hand or wishes to test its influence

on statistics results.

This file could also be used to create data cubes for other target clusters provided a few modifications to deal with the specifics of the maps for each cases.

2.4 Definitions.py file

This file contains the definitions of the classes `Simulation`, `Source` and `Cluster` as well as additional functions needed for particular methods.

2.4.1 Simulation class

This class gathers all the parameters related to the simulation, i.e. the time step `dt` (if one wants to move the lenses and/or the source), the number of consecutive GERLUMPH maps to create `n_maps` (the lenses moving between each map), the number of average ray count per pixel `navg` and the resolution `res` of the GERLUMPH maps.

All these parameters are directly defined as private instance variables upon the creation of an instance `Simulation`. If no value is given for one or more arguments, they take the following default values: `dt=1`, `n_maps=1`, `navg=50`, `res=512`.

In addition to the `__init__` method, each instance variable has a get method: `get_dt`, `get_n_maps`, `get_navg` and `get_res`.

2.4.2 Source class

This class is the one that gathers all the parameters pertaining to the source: its `mass` (which should be given as a Quantity object in units of solar masses), its redshift `z` and the two coefficients defining its mass-luminosity relation `ml_exp` and `ml_fact` (see eq.1).

Once again, all these parameters are directly defined as private instance variables upon the creation of an instance `Source`. If no value is given for one or more arguments, they take the following default values: `mass=100*u.M_sun`, `z=10`, `ml_exp=3.5`, `ml_fact=1`. These values for the mass-luminosity relation coefficients correspond to an old stellar population, which is what is needed for MACSJ1149.5+2223 (need a paper to justify this choice).

In addition to the `__init__` method, the following get methods are available: `get_mass`, `get_z`, `get_navg` and `get_ml_coeff` (which returns the tuple (`ml_exp`, `ml_fact`)).

Other instance methods are provided:

- `luminosity`: which takes no parameters and returns the source's luminosity (as a Quantity object with units of solar luminosity) using the chosen mass-luminosity relation:

$$\frac{L}{L_\odot} = \texttt{ml\_fact} \cdot \left(\frac{M}{M_\odot}\right)^{\texttt{ml\_exp}} \tag{1}$$

- `app_mag_pre_mu`: which takes a parameter `cosmology` (with default value `astropy.cosmology.FlatLambdaCDM(H0=70, Om0=0.3)`) and returns the apparent magnitude of the source when there is no magnification happening along the way.

- `app_mag_post_mu`: which takes two parameters: a magnification value `mu` (with default value 10^4) and a `cosmology` (with default value `astropy.cosmology.FlatLambdaCDM(H0=70, Om0=0.3)`). It returns the apparent magnitude of the source when there is a magnification `mu` happening along the way.

- `limit_mu`: which takes a parameter `cosmology` (with default value `astropy.cosmology.FlatLambdaCDM(H0=70, Om0=0.3)`) and returns the minimum magnification needed (along the path travelled by the source's light) to see the source with the HST telescope.

Finally, there is an additional static method `trajectory` which takes as parameters a `simulation`, a number of points `n_pts` and a Boolean `lenses_moving` (with respective default values : `Simulation()`, `15` and `True`). This method creates x and y coordinate vectors for the source in the following way. First, it uses the `random_pos` function (see §2.4.4) to get the vectors first elements. It also determines a random velocity in x and y using the `random_vel` function (see §2.4.4). Then, it computes the following positions taking the velocity and initial positions into account. However, it distinguishes between two cases :

- `if lenses_moving is True`: the length of the the position vectors is given by the number of maps (i.e. the simulation attribute `n_maps`). This option is given for users who would like for the sources and the lenses to move simultaneously.

- `else`: the length of the position vectors is given by the parameter `n_pts`. This option is given for users who would like only for the sources to move.

Finally, as the x and y coordinates on a GERLUMPH map are restricted to $[0, 99] \times [0, 99]$ (given in Einstein radius units)[4], the function uses a `while` loop to make sure that the vectors' ending points are not outside this range.

Note that this last method could have easily been implemented as an independent function. We chose instead to make it a static method in anticipation of future improvements to the code (see 5).

2.4.3 Cluster class

This class is the one that gathers all the parameters related to the cluster. The parameters given to the `__init__` method (other than the `self`) are: the redshift `z`, the path to the data cube `cube_file` and the mass-luminosity relation (see eq.1) coefficients `ml_exp` and `ml_fact` with default values: `z=0.5440`, `cube_file='./data_cube.fits'`, `ml_exp=1` and `ml_fact=1.5`). These values for the mass-luminosity relation coefficients correspond to an young stellar population, which is what is needed for the far away sources we strive to study (need a paper to justify this choice).

The parameters `z`, `ml_exp` and `ml_fact` are then directly used as such to define private instance variables with the same names. Three other private instances variables: `_kappa`, `_gamma` and `_flux` are extracted from the FITS file `cube_file`. And the two final private instances variables `_theta_x` and `_theta_y` are extracted from the FITS file's header (they correspond to the angular dimension of a pixel on the sky in radian).

The following get methods are available: `get_z`, `get_kappa`, `get_gamma`, `get_flux`, `get_ml_coeff` (which returns the tuple (`ml_exp`, `ml_fact`)) and `get_pix_dim` (which returns the tuple (`_theta_x`,

_theta_y)).

The other methods available are:

- `compute_sigma_st(self, cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: first we use the small angle approximation to compute the pixel area:

$$A_{pix} = D_l^2 \cdot \texttt{\_theta\_x} \cdot \texttt{\_theta\_y} \tag{2}$$

where θ_x and θ_y are the pixels dimension on the sky (in rad) and D_l is the angular diameter distance to the lens in kpc (computed using a method from `FlatLambdaCDM`). From which we deduce the luminosity :

$$L = \texttt{\_flux} \cdot A_{pix} \tag{3}$$

where A_{pix} was converted to cm^2 beforehand, thus the luminosity L is given in erg/s. Finally, we use the inverted Mass-Luminosity relation :

$$\frac{M}{M_\odot} = \left(\frac{1}{\texttt{\_ml\_fact}} \frac{L}{L_\odot} \right)^{1/\texttt{\_ml\_exp}} \tag{4}$$

to deduce the mass surface density:

$$\Sigma^* = \frac{M}{A_{pix}} \tag{5}$$

which is converted to units of $\mathrm{M}_\odot/\mathrm{kpc}^2$ before the its value is returned.

- `compute_kappa_star(self, source=Source(), cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: first, we compute the critical surface mass density using the function `compute_sigma_cr` (see §2.4.4). Then we compute the surface mass density from the observational data using the `Cluster` method `compute_sigma_st`. Finally, we return the value of κ^* computed as follows:

$$\kappa^* = \frac{\Sigma^*}{\Sigma_{cr}} \tag{6}$$

(which is dimensionless). Note also that if there exist a point for which κ^* is bigger than $\kappa * r$ where r is given by eq.10, this function prints an error message.

- `basic_stats(self, source=Source(), simul=Simulation(), cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: this method will create a GERLUMPH map (using the function `create_maps`) for each set of parameters κ and γ such that the magnification μ given by:

$$\mu = \left| \frac{1}{(1-\kappa)^2 - \gamma^2)} \right| \tag{7}$$

is above a threshold computed as:

$$\mu_{thr} = 10^{\lfloor \log_{10}(\texttt{source.limit\_mu(cosmology)}) \rfloor - 1} \tag{8}$$

.

Then, for each of these GERLUMPH magnification maps, it will test whether or not the maximum magnification is above the HST limiting magnification for observing the specified source (using the function `mu_above_limit`, see §2.4.4).

After which it will print the number and percentage of pixels (of the data cube map) with magnification above the threshold. As well as the number and percentage of pixels (of the data

cube map) such that the return value of the function `mu_above_limit` on the GERLUMPH map was `True`.

Finally, it will plot the map with pixels above the limit magnification in blue and the rest in white, as in the "Plots" folder as an image named "above_mu_map.png". Then it returns an array storing these pixels coordinates.

- `adv_stats(self, pix_for_adv_stats, n_pts=15, source=Source(), simul=Simulation(), cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: for all the pixels with coordinates stored in `pix_for_adv_stats`, this method will generate a 100 different source trajectories (using the `Source` method `trajectory` in the case where the parameter `lenses_moving` is `False`, see §2.4.2). Then it will analyse each of these light curves produced and test whether or not they are "good" light curves, meaning whether or not they have two properties that we deem necessary to be able to observe them in reality:
  - 25% of the points on the source trajectory should have a magnification μ above the limiting one.
  - the magnitude difference between the trajectory's maximum magnitude and its minimum magnitude should exceed 1 mag.

  These conditions were not put to test in real conditions, so one should be careful, and eventually see what changes when a particular numbers change before making any definitive conclusion.

  The number of good light curve for each pixel is then recorded and in the end, the method makes a plot of a map where each pixel appears with a different color dependind on the percentage of "good" light curves observed. The plot is stored in the right "Plots" sub-folder as "good_light_curves_percentage.png".

  Finally, an example of the plot of a "good" light curve is shown stored in the same folder as "light_curve_example.png"

2.4.4 Functions

Here we present briefly all the functions defined in the script Definitions.py.

- `random_pos(scale=100)`: it uses the `numpy` function `random.choice` to generate two uniformly random values between 0 and `scale`.
- `random_vel(scale)`: it uses the `numpy` function `random.choice` to generate two uniformly random values between 0 and `scale` and also to generate a random sign (+1 or -1).
- `compute_sigma_cr(z_l, z_s, cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: it uses the `FlatLambdaCDM` methods `angular_diameter_distance_z1z2` and `angular_diameter_distance` to compute three angular diameter distances and from them deduce the critical surface mass density Σ_{cr} using:

$$\Sigma_{cr} = \frac{c^2}{4\pi G}\frac{D_s}{D_{ls}D_l} \tag{9}$$

  where D_l, D_s and D_{ls} denote the angular diameter distances to the lens, the source and between the lens and the source respectively. Note that Σ_{cr} is converted to units of $M_\odot/kpc^2$ before the it is returned.

- `distances_ratio(z_l, z_s, cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: it uses the `FlatLambdaCDM` methods `angular_diameter_distance_z1z2` and `angular_diameter_distance` to compute the distance ratio:
$$r = \frac{D_{ls}}{D_s} \tag{10}$$
- `read_pos_and_m(pos_path)`: this function extracts the values of the lenses' x and y positions as well as the lenses' masses from the file `pos_path+"lens_pos.dat"` and returns them (in this order).
- `mu_above_limit(map_path, source=Source(), cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: this function takes the absolute value of the data contained in the FITS file `map_path` and it returns a Boolean which is `True` if the maximum value is above the value given by the `Source` method `limit_mu` and which is `False` otherwise.
- `move_lenses(simulation, path)`: it uses `read_pos_and_m` to get the initial lenses' positions. Then, it uses `random_vel` to initiate random velocities in the x and y directions for all the lenses. Finally, it computes the lenses positions `n_maps` times (using the simulation time step `dt`) and writes them in a DAT file in the appropriate folder.
- `new_map(kappa, gamma, kappa_st, simulation, path, begin=True)`: this function is the one calling the executable GERLUMPH file `lenser_gpu` to make at directory `path`. If the Boolean `begin` is `True` the map is made with random lenses' positions. However, if `begin` is `False`, the map is made using pre-computed lenses' positions (see right below). Then, the files created using the GERLUMPH executable are transferred to the right folder and a FITS file is created from the DAT file using the python script "N2mu_bin2fits.py"
- `create_maps(kappa, gamma, kappa_st, simulation, path)`: this function first creates some needed folders. Then, it creates the first map using the function `new_map` with the parameters `kappa`, `gamma`, `kappa_st`, `simulation`, some relevant path and `begin=True` (meaning the lenses positions will be random). Finally, if the number of maps in the simulation is strictly above 1, it uses the function `move_lenses(simulation, path)` to create the files containing the lenses positions at each time step and then creates the relevant maps using `new_map` with the parameters `kappa`, `gamma`, `kappa_st`, `simulation`, some relevant path and `begin=False` (which means the lenses' positions will not be random but pre-computed).
- `plot_moving_lenses(kappa, gamma, kappa_st, source=Source(), simulation=Simulation(), cosmology=FlatLambdaCDM(H0=70, Om0=0.3))`: this function can be used to plot sources' light curves in the case that one wants to investigate moving lenses.
- `test_path(path_folder)`: if the path `path_folder` exists this function removes all the files it contains. If it does not exist, it creates the folder.

3 A Step-by-step Example: Run.py default

In this section we shall present in great details what goes on with the code when one runs the following command in the terminal : `python Run.py default`

In order to do this the Run.py script is reproduced below with additional comments (in blue-green) explaining what goes on.

```
# Step 1: see that the files needed to make the cube of data exist.
if os.path.isfile("./kappa_map.fits"):
    if os.path.isfile("./gamma_map.fits"):
        # Step 2: see whether or not the data cube already exists.
        if not(os.path.isfile("./data_cube.fits")):
            # If the cube does not exist, there are two ways to make one :
            # with a default flux value or with an observational map.

            # os.system("python Make_data_cube.py default")
            os.system("python Make_data_cube.py")
        else:
            # If the cube already exists, it is not computed again.
            # The algorithm then prints a warning.
            print("Using pre-computed data_cube.fits.")

        # Create the file if it does not exist
        # and cancel everything in it if it does.
        test_path("./Maps")

        # Step 3: choose between Run.py default and Run.py perso.
        if len(sys.argv) > 1 and sys.argv[1] == "default":
            # Create Cluster and Source instances with the desired parameters.
            cluster = Cluster()
            source = Source(mass=100*u.M_sun, z=2)
            n_pts = 15

            # Create the file if it does not exist
            # and cancel everything in it if it does.
            test_path("./Plots/M"+str(source.get_mass().value)+"_z"+str(source.get_z()))

            # Do the basic and advanced statistics on the instances created.
            # Note that here we make great use of the default values
            # for some of the methods parameters.
            pix_for_adv_stats = cluster.basic_stats(source)
            cluster.adv_stats(pix_for_adv_stats, n_pts, source)

        elif len(sys.argv) > 1 and sys.argv[1] == "perso":
            # Put your own program here.
    else:
        print("Error: No gamma map.")
else:
    print("Error: No kappa map.")
```

4 Remaining Issues

First, there seems to be a problem with the code still concerning the κ^* values created with the command: `python Make_data_cube.py` (so non-default mode for creating the data cube). They are several orders of magnitude below what we expect (see [9]). This error should be localised either to the script "Make_data_cube.py" or to the following `Cluster` methods (found in the script "Definitions.py"): `compute_sigma_st`, `compute_kappa_star`, `basic_stats`, `adv_stats`.

Once, this problem is solved, another one is likely to appear: it is probable that GERLUMPH will refuse to make some of the maps because the number of micro-lenses needed to achieve the desired fraction of micro-lensing due to compact object will be too high. We think that if this happens, one should go looking into the GERLUMPH code and change the upper limit set on the variable "IP.Nl<LENS_CHUNK".

Finally, before any conclusion can be drawn, one should also:

- verify the conditions set for what constitutes a "good" light curve in the `Cluster` method `adv_stats`. These should be tested and adjusted.
- review the default mass-luminosity relations set for the `Cluster` and `Source` instances. A paper to justify the choices made must be found.

5 Potential Improvements

The improvements considered are:

- to add the conversion of the distances of the GERLUMPH maps given in Einstein radius units to units of kpc for example. This will be useful to set the scale on light curves plots drawn by the `Cluster` methods `adv_stats` and `plot_moving_lenses`.
- to change the function `random_vel` so that the velocity of an object depends on its redshift.
- to make every parameter in the code which should have units of time, distance or velocity into a `Quantity`, so that there are no problems once the two improvements described above are implemented.

References

[1] Kelly, P. L. et al. 2018, Nat. Astron., https://doi.org/10.1038/s41550-018-0430-3.

[2] Lotz J. M., Koekemoer A., Coe D., et al. 2016, ApJ, in press (arXiv:1605.06567).

[3] Venumadhav T., Dai L. and Miralda-Escudé J. 2017, ApJ, 850, 49.

[4] Vernardos G., Fluke C. J., Bate N. F. and Croton D. 2014, ApJ, 211, 16.

[5] Millon M., Joseph R. and Courbin F. 2016, "Source/lens separation: application to the Hubble Frontier Fields", (Lausanne: EPFL).

[6] Norman M. L. 2010, AIP Conf. Proc., 1294, 17.

[7] Smith E., et al. 2011, "Introduction to the HST Data Handbooks", Version 8.0, (Baltimore: STScI).

[8] Pavlovsky C., et al. 2002, "ACS Instrument Handbook", Version 3.0, (Baltimore: STScI).

[9] Morishita, T., Abramson, L. E., Treu, T., et al. 2017, ApJ, 846, 139.