Running the Stacking Code: caustic_stack

Updated: June, 2014

“This is a very general overview of how to run the stacking code on any generic situation, be it 3D simulation, light cone, real data, etc.”

1. Introduction:

Here is a quick tutorial on how to get and run the Stacking code. It is dependent on Dan Gifford’s causticpy, as well as other personalized scripts. Having a basic understanding of causticpy will greatly help in understanding caustic_stack.

Within caustic_stack is __init__.py, which is a script that allows you to take the phase spaces of galaxy clusters (the cluster-centric radius vs. line-of-sight velocity of galaxies or DM particles) and overlay them on top of each other (aka “stacking”), creating a joined phase space called an ensemble phase space, or ensemble cluster. The code then utilizes causticpy to run the caustic technique over the ensemble to estimate a mass (Mcrit200), along with other dynamics of the ensemble cluster.

The basic premise of the code, is that the users feeds it an array of phase spaces, and the code then stacks them and runs the caustic technique.

2. Dependencies:

See causticpy for relevant dependencies. Along with the same software causticpy is dependent upon, this code is also dependent on DictEZ.py and AttrDict.py, which are included with caustic_stack.

3. Getting the Code:

The code can be downloaded from Github: caustic_stack can be found here, and causticpy can be found here. There are directions to set up causticpy in its README file. To install caustic_stack, you only need to **git clone** the URL, and then add the caustic_stack/ directory to your PATH or PYTHONPATH. In a bash terminal this would be a command in your ~/.bashrc or ~/.bash_profile file that looks something like this:

```
export PYTHONPATH=/where_ever_caustic_stack_lives/:$PYTHONPATH
```

4. Overview of the Code:

All of the necessary code is in one script called __init__.py. In order to use it, simply add this to the front of a python script:

```
from caustic_stack import *
```

This will not only load in the classes defined in __init__.py, but *will also load in all of the modules listed at the top of __init__.py*. So it might be good to check out what modules you are loading in and how they are loaded in.

There are three classes defined in __init__.py:

Data

This class is a class that defines an "empty container" so-to-speak, where one can attach data to it and it will stack or append that data for you. It has a few methods attached to it that allows one to do this. For example, if I have 3 arrays, a b and c, that I want to "stack" together into an array called 'final' (in a sense this is merely concatenating them) then I can do this inside a python script:

```
import numpy as np
from caustic_stack import *
a = np.array([1,2,3,4,5])
b = np.array([6,7,8,9,10])
c = np.array([11,12,13,14,15])
D = Data()
D.extend({'final':a})
D.extend({'final':b})
D.extend({'final':c})
print D.final
np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
```

Here I am feeding the D.extend() method a dictionary, with a **key** named "**final**" and a **value** that corresponds to **a**, **b**, and **c**, which are the arrays I want to stack. The stacked array, "final", then becomes an attribute of D, so to call it you would use: D.final

There are other methods, such as D.add(), D.replace(), D.clear() and D.append().

Stack

This is the class that contains the function **caustic_stack()**, which is the main function you need in order to run the stacking code.

Universal

This contains functions needed by Stack.caustic_stack() in order to complete its stacking routine. Other functions that may be of interest, but not necessary, are there, such as line_of_sight(), which is a way to make a mock projection of galaxies within a simulation, and print_varibs(), which is a nice function to call at the beginning of any script to print out the values of flags and variables defined in the run in a clean way.

Creating a caustic_params.py file:

In order to use this caustic_stack(), you need to define a few flags outside of the function. This is most easily done by creating a parameter file, which can be called caustic_params.py. As of now, this is a requirement; you must have a caustic_params.py file in the same directory where you run your python scripts (not in caustic_stack/). Inside of the parameter file, you should define the following flags:

gal_num: integer

Number of galaxies to take per individual cluster phase space (Ngal)

line_num: integer

Number of individual phase spaces to stack into one ensemble (Nclus)

scale_data: True, False

Scale radial data by R200 before stacking?

run_los: True, False

Run caustic technique over individual phase spaces (lines-of-sight)?

avg_meth: String. Either ‘median’ or ‘mean’

Uses either median or mean to estimate the properties of a Bin. Must be a string, and match exactly one of two strings listed above.

mirror: True, False

Used in causticpy’s estimation of the caustic surface: mirror the v data of the phase space before solving for the caustic surface?

edge_perc: float

Fractional percent (ex. 0.1=>10% or 0.25=>25%) of top galaxies in the phase space to use for edge detection surface finding. Note that the edge detection and normal caustic detection are different processes, affecting one won’t affect the other.

c: 2.9979e5

Speed of light in km/s

h: 1.0

Hubble constant / 100.0

H0: 100*h

Hubble constant

In order to properly load the caustic_params.py file, you should incorporate the following code block with your IMPORT statements:

```
import os, sys
sys.path.insert(0,os.getcwd())
__import__('caustic_params')
from caustic_params import *
print "Loaded caustic_params from",sys.modules['caustic_params']
```

This will take the file called “caustic_params.py” in your working directory and load any variables defined in it into the working session. To check this is the “correct” caustic_params.py, look at the print statement, it should print out the caustic_params.py file in your working directory.

5. Worked Example

Within a custom python script, called ***script.py*** for example, we can load caustic_stack and work with the function Stack.caustic_stack(*args,**kwargs) to run the stacking code. First we define a basic ***caustic_params.py*** file in the same directory as ***script.py*** in order to define certain variables that are needed.

```
------------ caustic_params.py --------------
run_los = True              # Run caustic over each individual cluster as well?
scale_data = False          # Scale radial data by cluster R200 before stacking?
init_clean = False          # Run shiftgapper on individual cluster before stacking?
mirror = True               # Mirror phase space over velocity axis before surface estimation?
ens_num = 1                 # Number of ensembles to solve for
gal_num = 50                # Number of galaxies to stack per line of sight (Ngal)
line_num = 5                # Number of clusters to stack into one ensemble (Nclus)
method_num = 0              # Method to choose galaxies from line of sight, 0=Top brightest
avg_meth = ‘median’         # Method to create bin properties, ‘mean’ or ‘median’ (fed as string)
c = 2.9979e5                # Speed of light in km/s
h = 1.0                     # hubble constant / 100
H0 = 100*h                  # hubble constant
------------- end caustic_params.py ------------
```

How to feed Rdata and Vdata arguments in caustic_stack() function

In this brief example, we can assume that we already have our phase spaces created: in simulation data this means taking the cluster centric galaxy coordinates, making a mock projection away from the cluster center and putting the galaxies in a cluster-centric radius vs. line-of-sight velocity diagram, which is called the phase space. Lets assume that we have 5 individual clusters we want to stack (line_num = 5) and we want to stack 50 galaxies from each cluster (gal_num = 50), and we only want to do this for 1 ensemble (ens_num = 1). We will assume that this data is provided in the form of a 2 dimensional numpy array called Rdata and Vdata, for example:

```
Rdata = np.array([
                [#,#,#,#,........,#]
                [#,#,#,#,........,#]
                [#,#,#,#,........,#]
                [#,#,#,#,........,#]
                [#,#,#,#,........,#] ])
```

The same goes for Vdata. The first index of Rdata--as in Rdata[0] or Rdata[1] etc.--differentiates between unique individual clusters, meaning that the second index--as in Rdata[0][0] or Rdata[0][1]--differentiates between unique galaxies within a unique individual cluster. *In other words, each horizontal [#,#,....#] array is an individual phase space with #,#,...# being its galaxies.*

Note that caustic_stack() will take care of limiting the phase space and building the ensemble according to either **method 0 - take top Ngal brightest galaxies** or **method 1- take a random Ngal galaxies**, but if you want to just stack the Rdata and Vdata fed arrays, you can feed caustic_stack() the keyword stack_raw = True. Therefore, we are not constrained to have every array within Rdata or Vdata be the same length; although we want a final gal_num of 50 galaxies to be stacked per individual cluster, we can feed Rdata as an array that has 300 elements in Rdata[0], and 257 elements in Rdata[1] and 791 elements in Rdata[2], caustic_stack() will handle limiting the sample down to gal_num galaxies within R200 so long as gal_reduce is kept set to True as it is by default.

In light of this, it is good to note that numpy does not like having a 2 dimensional array with its 1 dimensional components having different lengths. To work around this try appending the desired 1 dimensional arrays to a list (as arrays), and then change the final list to an array, as such:

```
Rdata = []
for i in range(10):
        Rdata.append(unique_cluster_rdata_as_an_array)
Rdata = np.array(Rdata)
```

Other arguments and keywords in caustic_stack() function

After having the basics, like our phase spaces in Rdata and Vdata arrays, we need the HaloIDs and HaloData at the very minimum.

HaloID is a 1 dimensional array that has the same length as the first axis of Rdata or Vdata, this means that it has the same length as the number of clusters we want to stack into an ensemble cluster (line_num), which in our case is 50. It is a unique identifier, usually a long integer, for each cluster.

HaloData should be fed as a 2 dimensional array, but only has three rows and line_num columns. The three rows should be the M200, R200 and HVD of each cluster, which are defined in simulations. If you don't have this data, i.e. real universe, you still need to feed the HaloData argument, but you should feed it as None. (Not yet incorporated…)

Lastly, you need to feed the 'stack_num' argument, which is the same as 'line_num' flag, it is the number of cluster that you plan on stacking into one ensemble cluster.

Additionally, if you plan on using method 0 to build an ensemble (take top Ngal brightest galaxies from phase space), you should also use the keyword feed_mags = True. If this is the case you need to feed all three keywords magnitude arrays, G_Mags, R_Mags and I_Mags. If you only have on magnitude array, then **be sure to feed it to R_Mags** and create [None] type arrays of the same shape and feed those to G_Mags and I_Mags.

Finally I'll give a brief example of what a script.py file looks like (the file you would run from the command line to run your code, you don't have to call it script.py, I call mine millennium_stack.py), how to call the Stack() class and use the caustic_stack() function.

```
------------------------- script.py ----------------------------
## Load Modules
from caustic_stack import *

## Load caustic_params.py file
sys.path.insert(0,os.getcwd())
__import__('caustic_params')
from caustic_params import *
print "Loaded caustic_params from",sys.modules['caustic_params']

## Make variables defined in caustic_params.py file into a dictionary using DictEZ.py loaded
from caustic_stack as ez
keys =
['run_los','scale_data','init_clean','mirror','ens_num','gal_num','line_num','method_num','avg_met
h','c','h','H0']
variables = ez.create(keys,locals())

## Initialize Stack() class as S, as well as other classes, passing the 'variables' dictionary
through (requirement)
S = Stack(variables)
U = Universal(variables)

## Having already created Rdata and Vdata arrays, run caustic_stack()
stack_data =
S.caustic_stack(Rdata,Vdata,HaloID,HaloData,50,feed_mags=True,G_Mags=G_Mags,R_Mags
=R_Mags,I_Mags=I_Mags)

----------------------- end script.py ----------------------------
```

'stack_data' is returned as a dictionary, containing variables within it that pertain to the caustic run over the ensemble, and individual phase spaces if run_los = True, such as:

ens_caumass: “Ensemble Caustic Mass”, float
This is the caustic mass estimate of the ensemble at hand.

ens_caumass_est: “Ensemble Caustic Mass using R200 estimated by code”, float
This is the caustic mass estimate using caustipy’s estimate of R200, rather than the fed R200.

ens_edgemass: “Ensemble Edge-detector Mass”, float
This is the mass estimated by causticpy’s edge detector function.

ens_edgemass_est: “Ensemble Edge-detector Mass using R200 estimate”, float
This is the edge detector mass, but using causticpy’s estimate of R200, rather than fed.

ens_causurf: “Ensemble Caustic Surface”, array
This is an array containing the caustic surface (A(r) in Diaferio 1999).

ens_nfwsurf: “Ensemble NFW Fit Surface”, array
This is the NFW that is fit to the caustic surface, mostly outdated and not necessary.

ens_r: “Ensemble R Data”, array
This is an array that contains rdata for ensemble phase space.

ens_v: “Ensemble V Data”, array
This is an array that contains vdata for ensemble phase space.

ens_gmags,ens_rmags,ens_imags: “Ensemble SDSS (G/R/I) Magnitudes”, array
These are arrays containing the SDSS G/R/I magnitudes of the ensemble galaxies.

ens_gal_id: “Ensemble Galaxies ID”, array
This is an array containing a unique identifier (index) for each galaxy within the ensemble, with respect to the natural cumulative ordering of galaxies in the fed Rdata and Vdata arrays.

ens_clus_id: “Ensemble Galaxies’ cluster ID”, array
This is an array containing the galaxies’ initial parent cluster HaloID for each galaxy in the ensemble.

Same arrays exist for individual phase spaces, but with “ens” replaced with “ind”.

x_range: “Range of x axis for Gaussian Kernel Density Estimation”, array
Steps in radius used for creation of density map. Matches up with ens_causurf and ind_causurf arrays, and used as x-axis in order to plot those arrays.

Accessing the data

You can access these variables by referencing it from the “stack_data” dictionary:

```
print stack_data[‘ens_caumass’]
```

An easier way, is to update the stack_data dictionary to your global dictionary. That way you can access the variables directly, as if they were assigned in your “__main__” namespace.

```
globals().update(stack_data)
print ens_caumass
print ens_r
print ens_causurf
```