SAGE-analysis Documentation

Release 0.0.1

Jacob Seiler, Manodeep Sinha, Darren Croton

Apr 13, 2020

User Documentation

This is the documentation for the Semi-Analytic Galaxy Evolution (**SAGE**) analysis package. This package ingests, analyses, and plots the data products of the **SAGE** model, located here. Please refer to the to the **SAGE** repo for full documentation regarding how to run the base model.

CHAPTER 1

Installation

The recommended installing method is through pip:

```
$ pip install sage-analysis
```

CHAPTER 2

Maintainers

- Jacob Seiler (@jacobseiler)
- *User Documentation*
- *API Reference*

2.1 Introduction

The **SAGE**-analysis package was developed to handle the data products produced by the **SAGE** model, available here.

2.1.1 Why Did We Create A New Repo?

SAGE is an extremely modular and flexible semi-analytic model. The base model (presented in Croton et al., 2016) has been adjusted and altered to answer a number of science questions on a variety of topics including galactic HI and angular momentum properties in DARK SAGE, the Epoch of Reionization in RSAGE, and the star formation history and galactic dust in *DUSTY SAGE* _.

Due to the wide array of science that **SAGE** can cover and the number of models that spawned from its development, there has been a need to develop a framework to ingest in data from (ideally) any **SAGE** variant. This repo represents such an effort. It represents a series of modules intended to provide the easy ingestion and analysis of the base **SAGE** data, whilst serving as a template for analysing any other **SAGE** flavours.

2.1.2 Advantages of the Package

- Easy analysis and plotting of multiple different **SAGE** models. For example, comparing **SAGE** models with/without supernova feedback.
- Memory efficient analysis of **SAGE** output. All calculations are performed using only a single output file at a time, ensuring no extra memory overhead associated with opening many files.
- Support for the user to implement their own functions to analysis + plotting.

- Template for creating custom data classes to ingest any arbitrary **SAGE** data output. Useful if you're looking to develop using **SAGE**.

2.2 Setting up SAGE

This package ingests, analyses, and plots the data products produced by **SAGE**. Hence, the first step is to run the **SAGE** model and simulate some galaxies! We defer to the **SAGE** documentation for instructions on how run the **SAGE** model.

2.3 Analysing SAGE Output

On this page, we show how to analyse the **SAGE** output for a single snapshot. This full example is shown in the galaxy_properties module using the default parameter file.

We explain how to analyse **SAGE** output across multiple snapshots *here*.

2.3.1 Setting Things Up

First things first, we need to specify exactly what we want to analyse/plot, how the plots will be saved, and where the plots will be saved.

```
# Going to just plot the stellar mass function.
plot_toggles = {"SMF": 1}

plot_output_format = "png"
plot_output_path = "./plots"

if not os.path.exists(plot_output_path):
    os.makedirs(plot_output_path)
```

2.3.2 Model Dictionary

Each model that you wish to analyse is specifed through a dictionary. This defines properties such as the snapshot you wish to analyse, the location of the **SAGE** parameter file, etc.

```
millennium = { "snapshot": 63,   # Snapshot we're plotting properties at.
               "IMF": "Chabrier",  # Chabrier or Salpeter.
               "label": "Mini-Millennium",  # Legend label.
               "sage_file": "../input/millennium.par",
               "sage_output_format": "sage_hdf5",
               "first_file": 0,  # File range (or core range for HDF5) to plot.
               "last_file": 0,  # Closed interval, [first_file, last_file].
             }
```

NOTE: If the `sage_output_format` is `sage_binary` (i.e., **SAGE** wrote as binary output), then you must also specify the number of output files, `num_output_files`.

2.3.3 Setting up the Calculation and Plotting Dictionaries

To ensure that **sage-analysis** does not perform extraneous computations, the properties for each Model are calculated depending upon the `plot_toggles` specified. For example, the black hole mass of each galaxy will only be read if the black hole-bulge relationship plot toggle is set. We refer to this page <./custom_calculations for a full list of the default plot toggles.

To achieve this, we search for all functions in a module that are named `calc_<plot_toggle>`. We build these functions into a dictionary that are passed into *`calc_properties_all_files()`*.

```
from sage_analysis.utils import generate_func_dict

# Search for functions named "calc_<plot_toggle>" in the "example_calcs"
# module.
calculation_functions = generate_func_dict(
                            plot_toggles,
                            module_name="sage_analysis.example_calcs",
                            function_prefix="calc"
                            )
```

NOTE: All functions must have the signature `calc_<plot_toggle>(model, galaxies, **optional keyword arguments)`. We expand on this more in *Using Keyword Arguments*.

In a similar manner, we search for all the functions in a module that are named `plot_<plot_toggle>`. From this dictionary, we can then iterate over and make all the plots!

```
# Search for functions named "plot_<plot_toggles>" in the "example_plots"
# module.
plot_functions = generate_func_dict(
                    plot_toggles,
                    module_name="sage_analysis.example_plots",
                    function_prefix="plot_"
                    )
```

NOTE: All functions must have the signature `calc_<plot_toggle>(list of models, plot_output_path, **optional keyword arguments)`. We expand on this more in *Using Keyword Arguments*.

2.3.4 Initializing a Model

With the calculation functions prepped, we are now poised to perform the actual analysis. The analysis of **SAGE** models is done through a specialized *`Model`* class. **Importantly,** the Model class only handles the calculating properties. To actually read the **SAGE** output, each Model requires a data class. These are specific to the **SAGE** output format. For example, we include a data class for *`SageHdf5Data`* and *`SageBinaryData`*. Through this data class, the package can be easily extended to ingest any arbitrary **SAGE** output format. We show such an example *here*.

```
from sage_analysis.model import Model
from sage_analysis.sage_hdf5 import SageHdf5Data

model = Model()
model.plot_output_format = plot_output_format

model.data_class = SageHdf5Data(model, millennium["sage_file"])

# The data class has read the SAGE ini file.  Update the model with the parameters
# read and those specified by the user.
model.update_attributes(model_dict)
```

2.3.5 Storing Galaxy Properties

When performing calculations, **sage-analysis** stores all the calculating properties in the *properties* attribute of the Model instance. This attribute is a dictionary and can be used to access any of the properties pertaining to the Model; for example, `model.properties["SMF"]` stores the array representing the stellar mass function.

These properties must first be initialized. **sage-analysis** offers three ways to compute and store galaxy properties.

Binned Properties

These are properties that are binned on some value. For example: the stellar mass function is binned depending upon the galaxy stellar mass; the fraction of quiescent galaxies is binned upon the galaxy stellar mass; the mass of gas in each **SAGE** reservoir (cold gas/hot gas/stars/etc) is binned upon the friends-of-friends halo mass. The bins themselves are accessed through the *bins* attribute of the model instance. This attribute is a dictionary and can be used to access any of the bins for the Model; for example, `model.bins["stellar_mass_bins"]` would return the stellar mass bins used for the stellar mass function.

```
# Properties binned on stellar mass.
stellar_properties = ["SMF", "red_SMF", "blue_SMF"]
min_mass = 8.0  # log10(Msun).
max_mass = 12.0  # log10(Msun).
bin_width = 0.1  # log10(Msun).
bin_name = "stellar_mass_bins"
model.init_binned_properties(min_mass, max_mass, bin_width, bin_name,
                             stellar_properties)

# Properties binned on FoF halo mass.
component_properties = [f"halo_{component}_fraction_sum" for component in
                        ["baryon", "stars", "cold", "hot", "ejected", "ICS", "bh"]]
min_mass = 10.0  # log10(Msun)
max_mass = 14.0  # log10(Msun)
bin_width = 0.1  # log10(Msun)
bin_name = "halo_mass_bins"
model.init_binned_properties(min_mass, max_mass, bin_width, bin_name,
                             component_properties)
```

Scatter Properties

In many instances, we don't want to fit an exact line to the properties, but rather just get a sense of the typical data point values. For these, we want to compute lists of `(x, y)` coordinates that we will plot later. For example, the black hole bulge relationship will show a number of black hole masses and the corresponding bulge mass. The (maximum) number of data points shown on each plot can be set through the *sample_size* attribute.

```
# For each of these, we need a list for both x and y points. E.g., the
# black hole bulge needs both "bh_mass" and "bulge_mass".
scatter_properties = ["bh_mass", "bulge_mass", "BTF_mass", "BTF_vel"]
model.init_scatter_properties(scatter_properties)
```

Single Properties

Finally, often we want to use a single number to summarize a property for all galaxies across a single snapshot. This is most useful when analyzing galaxy properties over a range of snapshots through the history module. These properties are initialized with a value of `0.0`.

```
single_properties = ["SMFD", "SFRD"]
model.init_single_properties(single_properties)
```

2.3.6 Doing the Analysis and Plotting

We have set up the dictionary for the plotting functions in *Setting up the Calculation and Plotting Dictionaries*. Once all the properties have been calculated, we can finally do the plotting!

```
# Calculate all the properties.
model.calc_properties_all_files(calculations_functions)

# Now do the plotting.
for func_name in plot_functions.keys():
    func = plot_functions[func_name][0]
    func([model], plot_output_path, plot_output_format)
```

NOTE: The plotting scripts accept a list of Model classes as the first argument. For this scenario, we only have one model and so we cast it to a list first.

The above code snippets produce the glorious stellar mass function!

|SMF| .. |SMF| image:: ../figs/SMF.png

2.3.7 Using Keyword Arguments

`generate_func_dict()` accepts an optional argument to allow the calculation or plotting functions to handle keyword arugments. This argument is a dictionary with keys equal to the plot toggles. The value of each entry is another dictionary containing all of the keyword arguments and their corresponding value.

```
from sage_analysis.utils import generate_func_dict

# By default, the stellar mass function is not computed for the red and blue
# galaxy populations. Let's turn it on.
keyword_args = {"SMF": {"calc_sub_populations": True}}

calculation_functions = generate_func_dict(
                            plot_toggles,
                            module_name="sage_analysis.example_calcs",
                            function_prefix="calc",
                            keyword_args=keyword_args
                            )
model.calc_properties_all_files(calculations_functions)

# Then we can adjust "plot_SMF" to also plot these extra populations.
keyword_args = {"SMF": {"plot_sub_populations": True}}

plot_functions = generate_func_dict(
                    plot_toggles,
                    module_name="sage_analysis.example_plots",
                    function_prefix="plot_",
                    keyword_args=keyword_args
                    )

# Now do the plotting with the extra kwargs.
for func_name in plot_functions.keys():
```

(continues on next page)

(continued from previous page)

```
    func = plot_functions[func_name][0]
    keyword_args = plot_functions[func_name][1]
    func(models, plot_output_path, plot_output_format, **keyword_args)
```

|SMF_pop| .. |SMF_pop| image:: ../figs/SMF_pop.png

2.4 Analysing Across Multiple Snapshots

We show how to analyse the output of **SAGE** at a single snapshot *here*. On this page, we show how to analyse **SAGE** output across multiple snapshots. This is very useful if you wish to analyse the evolution of (e.g.,) the stellar mass function, the stellar mass density, etc.

This full example is shown in the history module using the default parameter file.

2.4.1 Setting Things Up

In a similar manner to analysing a *single snapshot*, we first specify which properties we wish to analyse and plot.

```
# Base specifications.
plot_output_format = "png"
plot_output_path = "./plots"  # Will be created if path doesn't exist.

plot_toggles = {"SMF_z" : 1,  # Stellar mass function across redshift.
                "SFRD_z" : 1,  # Star formation rate density across redshift.
                "SMD_z" : 1}  # Stellar mass density across redshift.
```

Then, we specify the **SAGE** output we wish to analyse and the redshifts at which we want the properties to be calculated at.

NOTE: For all the specified redshifts, **sage-analysis** searches for the snapshot closest to the redshift. Also set the entry to `"All"` to analyse across all snapshots.

```
millennium = {"SMF_z" : [0.0, 1.0, 2.0, 3.0],  # Redshifts you wish to plot the␣
↪stellar mass function at.
              "density_z": "All", # Redshifts to plot the stellar mass/star formation␣
↪density at.
              "IMF": "Chabrier",  # Chabrier or Salpeter.
              "label": "Mini-Millennium",  # Legend label.
              "sage_file": "../input/millennium.par",
              "sage_output_format": "sage_hdf5",
              "first_file": 0,  # File range (or core range for HDF5) to plot.
              "last_file": 0,  # Closed interval, [first_file, last_file].
             }
```

NOTE: If the `sage_output_format` is `sage_binary` (i.e., **SAGE** wrote as binary output), then you must also specify the number of output files, `num_output_files`.

2.4.2 Initializing the Model

Calculation and Plotting Dictionaries

Again, as outlined previously, we first generate the dictionaries necessary to analyse and plot properties.

```
from sage_analysis.utils import generate_func_dict

# Search for functions named "calc_<plot_toggle>" in the "example_calcs"
# module.
calculation_functions = generate_func_dict(
                            plot_toggles,
                            module_name="sage_analysis.example_calcs",
                            function_prefix="calc"
                            )

# Search for functions named "plot_<plot_toggles>" in the "example_plots"
# module.
plot_functions = generate_func_dict(
                    plot_toggles,
                    module_name="sage_analysis.example_plots",
                    function_prefix="plot_"
                    )
```

Setting up the Class

The analysis of **SAGE** models is done through a specialized *Model* class. **Importantly,** the Model class only handles the calculating properties. To actually read the **SAGE** output, each Model requires a data class. These are specific to the **SAGE** output format. For example, we include a data class for *SageHdf5Data* and *SageBinaryData*. Through this data class, the package can be easily extended to ingest any arbitrary **SAGE** output format. We show such an example *here*.

```
from sage_analysis.model import Model
from sage_analysis.sage_hdf5 import SageHdf5Data

model = Model()
model.plot_output_format = plot_output_format

model.data_class = SageHdf5Data(model, millennium["sage_file"])

# The data class has read the SAGE ini file.  Update the model with the parameters
# read and those specified by the user.
model.update_attributes(model_dict)
```

Specifying the Empty Property Containers

We also initialize the Model properties as outlined previously.

```
stellar_properties = ["SMF", "red_SMF", "blue_SMF"]
model.init_binned_properties(8.0, 12.0, 0.1, "stellar_mass_bins",
                             stellar_properties)

# Properties that are extended as lists.
scatter_properties = []
model.init_scatter_properties(scatter_properties)

# Properties that are stored as a single number.
single_properties = ["SMFD", "SFRD"]
model.init_single_properties(single_properties)
```

(continues on next page)

(continued from previous page)

```
# We will store the values of each snapshot in a dictionary.
model.properties["SMF_dict"] = {}
model.properties["SFRD_dict"] = {}
model.properties["SMD_dict"] = {}
```

2.4.3 Setting Up The Snapshot Loop

The key difference for this example is that we want to analyse properties over a number of redshifts. We hence must determine which snapshots in the model correspond to the requested redshifts.

```
# We may be plotting the density at all snapshots...
if model_dict["density_z"] == "All":
    model.density_redshifts = model.redshifts
else:
    model.density_redshifts = model_dict["density_z"]

# Same for SMF
if model_dict["SMF_z"] == "All":
    model.SMF_redshifts = model.redshifts
else:
    model.SMF_redshifts = model_dict["SMF_z"]

# Find the snapshots that most closely match the requested redshifts.
model.SMF_snaps = [(np.abs(model.redshifts - SMF_redshift)).argmin() for
                   SMF_redshift in model.SMF_redshifts]

model.density_snaps = [(np.abs(model.redshifts - density_redshift)).argmin() for
                       density_redshift in model.density_redshifts]

# Check which snapshots we uniquely need to loop through.
snaps_to_loop = np.unique(my_model.SMF_snaps + my_model.density_snaps)
```

2.4.4 Iterating Through Snapshots

Finally, we are poised to iterate through the snapshots and calculate all the properties required. Importantly, at the end of each snapshot, we must place the calculate properties into the appropriate dictionary and reset the property.

```
for snap in snap_iter:

    # Each snapshot is unique. So reset the tracking.
    model.properties["SMF"] = np.zeros(len(model.bins["stellar_mass_bins"])-1,
                                       dtype=np.float64)
    model.properties["SFRD"] = 0.0
    model.properties["SMD"] = 0.0

    # Update the snapshot we're reading from. Data Class specific.
    model.data_class.update_snapshot(model, snap)

    # Calculate all the properties. Since we're using a HDF5 file, we want to keep
    # the file open because we read other snapshots from that one file.
    model.calc_properties_all_files(calculation_functions, close_file=False)

    # We need to place the SMF inside the dictionary to carry through.
```

(continues on next page)

(continued from previous page)

```
    if snap in model.SMF_snaps:
        model.properties["SMF_dict"][snap] = model.properties["SMF"]

    # Same with the densities.
    if snap in model.density_snaps:

        model.properties["SFRD_dict"][snap] = model.properties["SFRD"]
        model.properties["SMD_dict"][snap] = model.properties["SMD"]

# Close the HDF5 file cause we're done with it.
model.data_class.close_file(model)
```

Finally, plot the properties!

```
# Similar to the calculation functions, all of the plotting functions are in the
# `example_plots.py` module and are labelled `plot_<toggle>`.
plot_functions = generate_func_dict(plot_toggles,
                                    module_name="sage_analysis.example_plots",
                                    function_prefix="plot_")

# Now do the plotting.
for func_name in plot_functions.keys():
    func = plot_functions[func_name][0]
    keyword_args = plot_functions[func_name][1]

    func(models, plot_output_path, plot_output_format, **keyword_args)
```

This produces the stellar mass function, star formation rate density, and stellar mass density over the various redshifts.

user/../figs/SMF_redshift.png user/../figs/SFRD.png user/../figs/SMD.png

2.5 Analysing and Plotting Multiple Models

A key feature of the **sage-analysis** package is its ability to easily and succinctly analyse and plotting a number of different **SAGE** Models. This makes it easy to compare different models very quickly.

We have covered the basics of plotting a single model *here*. In this example, we touch upon those things that must change to visualize multiple models.

2.5.1 Setting Things Up

Again, specify what you want to plot in an identical manner to previously.

```
# Going to just plot the stellar mass function.
plot_toggles = {"SMF": 1}

plot_output_format = "png"
plot_output_path = "./plots"
```

(continues on next page)

(continued from previous page)

```
if not os.path.exists(plot_output_path):
    os.makedirs(plot_output_path)
```

2.5.2 Defining the Model Dictionaries

Each Model requires its own dictionary to define its input files and other parameters. For this example, let's suppose we wish to compare the results of running **SAGE** on the default Mini-Millennium simulation and Mini-Millennium without supernova feedback.

```
millennium = { "snapshot": 63,   # Snapshot we're plotting properties at.
               "IMF": "Chabrier",  # Chabrier or Salpeter.
               "label": "Mini-Millennium",  # Legend label.
               "sage_file": "../input/millennium.par",
               "sage_output_format": "sage_hdf5",
               "first_file": 0,  # File range (or core range for HDF5) to plot.
               "last_file": 0,  # Closed interval, [first_file, last_file].
             }

millennium_noSN = { "snapshot": 63,   # Snapshot we're plotting properties at.
                    "IMF": "Chabrier",  # Chabrier or Salpeter.
                    "label": "Mini-Millennium-NoSN",  # Legend label.
                    "sage_file": "../input/millennium_noSN.par",
                    "sage_output_format": "sage_binary",
                    "first_file": 0,  # File range (or core range for HDF5) to plot.
                    "last_file": 0,  # Closed interval, [first_file, last_file].
                    "num_output_files": 1,
             }

sims_to_plot = [millennium, millennium_noSN]
models = []
```

The important line here is the very last line. We place all the models we wish to analyse into a single list, `sims_to_plot`. Then, in the following code block, we will iterate through each of these simulations and place the Model classes in the `model` list.

NOTE: For example purposes, we use the binary **SAGE** output option for the no supernovae simulation. Hence, we must specify the number of output files that **SAGE** wrote to, `num_output_files`.

2.5.3 Iterating Through Models

Analysing multiple models is down in an identical manner as shown *previously*. The only difference is that we wrap the code in a `for` loop and iterate through `models_to_plot`. We defer to the *previous page* for a full explanation of the following code block.

```
# Set up the dictionaries for the calculation and plotting functions.
from sage_analysis.utils import generate_func_dict

# Search for functions named "calc_<plot_toggle>" in the "example_calcs"
# module and "plot_<plot_toggle>" in the "example_plots" module.
calculation_functions = generate_func_dict(
                            plot_toggles,
                            module_name="sage_analysis.example_calcs",
```

(continues on next page)

(continued from previous page)

```python
                            function_prefix="calc"
                            )

plot_functions = generate_func_dict(
                    plot_toggles,
                    module_name="sage_analysis.example_plots",
                    function_prefix="plot_"
                    )

# Iterate through the simulations and set up a Model for each.
from sage_analysis.model import Model
from sage_analysis.sage_hdf5 import SageHdf5Data
from sage_analysis.sage_binary import SageBinaryData

for model_dict in sims_to_plot:
    model = Model()
    model.plot_output_format = plot_output_format

    # This switch case should be extended if you're defining your own
    # custom data class.
    if model_dict["sage_output_format"] == "sage_hdf5":
        model.data_class = SageHdf5Data(model, millennium["sage_file"])
    elif model_dict["sage_output_format"] == "sage_binary":
        model.data_class = SageBinaryData(model, model_dict["num_output_files"],
                                          model_dict["sage_file"],
                                          model_dict["snapshot"])
    else:
        raise ValueError

    # The data class has read the SAGE ini file.  Update the model with the parameters
    # read and those specified by the user.
    model.update_attributes(model_dict)

    # Initialize the properties for this model. Only plotting the SMF.
    stellar_properties = ["SMF", "red_SMF", "blue_SMF"]
    min_mass = 8.0   # log10(Msun).
    max_mass = 12.0   # log10(Msun).
    bin_width = 0.1   # log10(Msun).
    bin_name = "stellar_mass_bins"
    model.init_binned_properties(min_mass, max_mass, bin_width, bin_name,
                                 stellar_properties)

    # Calculate all the properties.
    model.calc_properties_all_files(calculations_functions)

    # Append this model to the list.
    models.append(model)
```

2.5.4 Plotting Multiple Models

All the hard work has already been done! The plotting functions defined in `plot_functions` accept a list of models as their first argument. Hence, lets just pass in `models`!

```python
for func_name in plot_functions.keys():
    func = plot_functions[func_name][0]
```

(continues on next page)

(continued from previous page)

```
    func(models, plot_output_path, plot_output_format)
```

This produces the stellar mass function for multiple models. From this, we can better understand the role that supernova feedback plays in regulating galaxy growth.

user/../figs/SMF_noSN.png

2.5.5 Using Keyword Arguments

Both the calculation and plotting functions support the use of optional keyword arguments. This allows finer control over plotting (e.g.) galaxy sub-populations. The procedure for using this option for multiple models identical to the single Model scenario which we show in *Using Keyword Arguments*.

2.5.6 Analysing Multiple Models Over Redshift

To analyse multiple models over a number of redshifts, one simply needs to wrap the code outlined *here* in the `for model in sim_to_plot` loop.

We defer to the history module for a full example of analysing multiple models over redshifts.

user/../figs/SMF_redshift_noSN.png user/../figs/SFRD_noSN.png user/../figs/SMD_noSN.png

2.6 Defining Custom Properties

We have previously given examples how to analyse **SAGE** properties at a *single snapshot* or *over multiple redshifts*. However, it is quite common that you may wish to calculate properties different to those included in the base version of **sage-analysis**. For example, you may wish to investigate the time since a last major merger as a function of galaxy stellar mass.

This page outlines how **sage-analysis** can be used to calculate any arbirtary properties.

2.6.1 Default Properties

The **sage-analysis** package includes a number of properties that can be analysed and plotted by default.

Property	Plot Toggle Name	Description	Property Type
Stellar mass function	SMF	Number of galaxies with a given stellar mass.	Binned.
Baryonic mass function	BMF	Number of galaxies with a given stellar plus cold gas mass.	Binned
Gas mass function	GMF	Number of galaxies with a given cold gas mass.	Binned.
Baryonic Tully-Fisher	BTF	Maximum velocity of a galaxy as a function of baryonic (stellar plus cold gas) mass.	Scatter.
Specific star formation rate	sSFR	Specific star formation rate as a function of stellar mass.	Scatter.
Gas fraction	gas_frac	Fraction of baryons (stellar plus cold gas) in the form of cold gas as a function of stellar mass.	Scatter.
Mass metallicity	metallicity	Metallicity as a function of stellar mass.	Scatter.
Black hole bulge	bh_bulge	Mass of galaxy black hole as a function of galaxy bulge mass.	Scatter.
Quiescent galaxy population	quiescent	Fraction of galaxies that are quiescent as a function of stellar mass.	Binned.
Bulge fraction	bulge_fraction	Fraction of stellar mass in the form of bulge/disk as a function of stellar mass.	Scatter.
Baryon fraction	baryon_fraction	Baryon fraction in each reservoir (cold, hot, stellar, ejected, intracluster, and black hole) as a function of FoF halo virial mass.	Binned.
Reservoir mass	reservoirs	Amount of mass in each reservoir (cold, hot, stellar, ejected, intracluster, and black hole) as a function of FoF halo virial mass.	Scatter.
Spatial distribution	spatial	Spatial distribution of galaxies across the simulation box.	Scatter.

There are also a handful of toggles available to analyse properties over a number of redshifts.

Property	Plot Toggle Name	Description	Binning Type
Stellar mass function	SMF_z	Number of galaxies with a given stellar mass over multiple redshifts for each model.	Binned.
Star formation rate density	SFRD_z	Total star formation rate density across entire simulation box as a function of redshift.	Single.
Stellar mass density	SMD_z	Total stellar mass density across entire simulation box as a function of redshift.	Single.

2.6.2 Adding Your Own Properties

There are

1. Set up Model.properties[“fff”] field.
2. Write the function to compute the property.
3. Add the function to the calculations dict.
4. Add the plotting function to the plot dict.

SAGE operates by allowing each processor to write to its own file as galaxies are evolved through cosmic time. **sage-analysis** processes galaxy properties of each of these files individually. After calculating each property, they are stored in the `properties` attribute and carried across files. The pseudo-code looks like this:

```
for file in num_files:

    compute stellar mass function for file
    add stellar mass function to Model.properties["SMF"] array.

    calculate black hole bulge relationship for file
    extend the Model.properties["bh_mass"] and Model.properties["bulge_mass"] lists

    ...complete for other properties...
```

To calculate each of these properties, a function named `calc_<property_name>` is called; for example, `calc_SMF()` is called to compute the stellar mass function of each **SAGE** file.

2.7 Ingesting Custom Data

2.8 sage_analysis.Model

This module contains the `Model` class. The `Model` class contains all the data paths, cosmology etc for calculating galaxy properties.

To read **SAGE** data, we make use of specialized Data Classes (e.g., `SageBinaryData` and:py:class:~*sage_analysis.sage_hdf5.SageHdf5Data*). We refer to ../user/data_class for more information about adding your own Data Class to ingest data.

To calculate (and plot) extra properties from the **SAGE** output, we refer to ../user/calc.rst and ../user/plotting.rst.

Author: Jacob Seiler.

class `sage_analysis.model.`**`Model`**(*sage_file: str, sage_output_format: Optional[str], label: Optional[str], first_file_to_analyze: int, last_file_to_analyze: int, num_sage_output_files: Optional[int], random_seed: Optional[int], IMF: str, plot_toggles: Dict[str, bool], plots_that_need_smf: List[str], sample_size: int = 1000, sSFRcut: float = -11.0*)

Handles all the galaxy data (including calculated properties) for a `SAGE` model.

The ingestion of data is handled by inidivudal Data Classes (e.g., `SageBinaryData` and `SageHdf5Data`). We refer to ../user/data_class for more information about adding your own Data Class to ingest data.

`calc_properties`(*calculation_functions*, *gals*, *snapshot: int*)

Calculates galaxy properties for a single file of galaxies.

Parameters

- **calculation_functions** (*dict [string, function]*) – Specifies the functions used to calculate the properties. All functions in this dictionary are called on the galaxies. The function signature is required to be `func(Model, gals)`
- **gals** (exact format given by the `Model` Data Class.) – The galaxies for this file.
- **snapshot** (*int*) – The snapshot that we're calculating properties for.

Notes

If *sage_output_format* is `sage_binary`, `gals` is a `numpy` structured array. If *sage_output_format*: is `sage_hdf5`, `gals` is an open HDF5 group. We refer to ../user/data_class for more information about adding your own Data Class to ingest data.

`calc_properties_all_files`(*calculation_functions*, *snapshot: int*, *close_file: bool = True*, *use_pbar: bool = True*, *debug: bool = False*)

Calculates galaxy properties for all files of a single *Model*.

Parameters

- **calculation_functions** (*dict [string, list(function, dict[string, variable])]*) – Specifies the functions used to calculate the properties of this *Model*. The key of this dictionary is the name of the plot toggle. The value is a list with the 0th element being the function and the 1st element being a dictionary of additional keyword arguments to be passed to the function. The inner dictionary is keyed by the keyword argument names with the value specifying the keyword argument value.

  All functions in this dictionary for called after the galaxies for each sub-file have been loaded. The function signature is required to be `func(Model, gals, <Extra Keyword Arguments>)`.
- **snapshot** (*int*) – The snapshot that we're calculating properties for.
- **close_file** (*boolean, optional*) – Some data formats have a single file data is read from rather than opening and closing the sub-files in `read_gals()`. Hence once the properties are calculated, the file must be closed. This variable flags whether the data class specific `close_file()` method should be called upon completion of this method.
- **use_pbar** (*Boolean, optional*) – If set, uses the `tqdm` package to create a progress bar.
- **debug** (*Boolean, optional*) – If set, prints out extra useful debug information.

`init_binned_properties`(*bin_low: float, bin_high: float, bin_width: float, bin_name: str, property_names: List[str], snapshot: int*)

Initializes the *properties* (and respective *bins*) that will binned on some variable. For example, the stellar mass function (SMF) will describe the number of galaxies within a stellar mass bin.

bins can be accessed via `Model.bins["bin_name"]` and are initialized as `ndarray`. *properties* can be accessed via `Model.properties["property_name"]` and are initialized using `numpy.zeros`.

Parameters

- **bin_low, bin_high, bin_width** (*floats*) – Values that define the minimum, maximum and width of the bins respectively. This defines the binning axis that the `property_names` properties will be binned on.
- **bin_name** (*string*) – Name of the binning axis, accessed by `Model.bins["bin_name"]`.
- **property_names** (*list of strings*) – Name of the properties that will be binned along the defined binning axis. Properties can be accessed using `Model.properties["property_name"]`; e.g., `Model.properties["SMF"]` would return the stellar mass function that is binned using the `bin_name` bins.
- **snapshot** (*int*) – The snapshot we're initialising the properties for.

`init_scatter_properties`(*property_names: List[str], snapshot: int*)

Initializes the *properties* that will be extended as `ndarray`. These are used to plot (e.g.,) a the star formation rate versus stellar mass for a subset of *sample_size* galaxies. Initializes as empty `ndarray`.

Parameters

- **property_names** (*list of strings*) – Name of the properties that will be extended as `ndarray`.
- **snapshot** (*int*) – The snapshot we're initialising the properties for.

`init_single_properties`(*property_names: List[str], snapshot: int*) → None

Initializes the *`properties`* that are described using a single number. This is used to plot (e.g.,) a the sum of stellar mass across all galaxies. Initializes as `0.0`.

Parameters

- **property_names** (*list of strings*) – Name of the properties that will be described using a single number.
- **snapshot** (*int*) – The snapshot we're initialising the properties for.

`IMF`

The initial mass function.

Type {`"Chabrier"`, `"Salpeter"`}

`base_sage_data_path`

Base path to the output data. This is the path without specifying any extra information about redshift or the file extension itself.

Type string

`bins`

The bins used to bin some *`properties`*. Bins are initialized through *`init_binned_properties()`*. Key is the name of the bin, (`bin_name` in *`init_binned_properties()`*).

Type dict [string, `ndarray`]

`box_size`

Size of the simulation box. Units are Mpc/h.

Type float

`calculation_functions`

A dictionary of functions that are used to compute the properties of galaxies. Here, the string is the name of the toggle (e.g., `"SMF"`), the value is a tuple containing the function itself (e.g., `calc_SMF()`), and another dictionary which specifies any optional keyword arguments to that function with keys as the name of variable (e.g., `"calc_sub_populations"`) and values as the variable value (e.g., `True`).

Type dict[str, tuple[func, dict[str, any]]]

`first_file_to_analyze`

The first *SAGE* sub-file to be read. If *`sage_output_format`* is `sage_binary`, files read must be labelled *`sage_data_path`*.XXX. If *`sage_output_format`* is `sage_hdf5`, the file read will be *`sage_data_path`* and the groups accessed will be Core_XXX. In both cases, `XXX` represents the numbers in the range [*`first_file_to_analyze`*, *`last_file_to_analyze`*] inclusive.

Type int

`hubble_h`

Value of the fractional Hubble parameter. That is, `H = 100*hubble_h`.

Type float

`label`

Label that will go on axis legends for this *`Model`*.

Type string

last_file_to_analyze
The last **SAGE** sub-file to be read. If *sage_output_format* is `sage_binary`, files read must be labelled *sage_data_path*.XXX. If *sage_output_format* is `sage_hdf5`, the file read will be *sage_data_path* and the groups accessed will be Core_XXX. In both cases, `XXX` represents the numbers in the range [*first_file_to_analyze*, *last_file_to_analyze*] inclusive.

Type int

num_gals_all_files
Number of galaxies across all files. For HDF5 data formats, this represents the number of galaxies across all *Core_XXX* sub-groups.

Type int

num_sage_output_files
The number of files that **SAGE** wrote. This will be equal to the number of processors the **SAGE** ran with.

Notes

If *sage_output_format* is `sage_hdf5`, this attribute is not required.

Type int

output_path
Path to where some plots will be saved. Used for `plot_spatial_3d()`.

Type string

parameter_dirpath
The directory path to where the **SAGE** paramter file is located. This is only the base directory path and does not include the name of the file itself.

Type str

plot_toggles
Specifies which plots should be created for this model. This will control which properties should be calculated; e.g., if no stellar mass function is to be plotted, the stellar mass function will not be computed.

Type dict[str, bool]

plots_that_need_smf
Specifies the plot toggles that require the stellar mass function to be properly computed and analyzed. For example, plotting the quiescent fraction of galaxies requires knowledge of the total number of galaxies. The strings here must **EXACTLY** match the keys in *plot_toggles*.

Type list of ints

properties
The galaxy properties stored across the input files and snapshots. These properties are updated within the respective `calc_<plot_toggle>` functions.

The outside key is `"snapshot_XX"` where `XX` is the snapshot number for the property. The inner key is the name of the proeprty (e.g., `"SMF"`).

Type dict [string, dict [string, `ndarray`]] or dict[string, dict[string, float]

random_seed
Specifies the seed used for the random number generator, used to select galaxies for plotting purposes. If `None`, then uses default call to `seed()`.

Type Optional[int]

redshifts
Redshifts for this simulation.

Type ndarray

sSFRcut
The specific star formation rate above which a galaxy is flagged as "star forming". Units are log10.

Type float

sage_data_path
Path to the output data. If *sage_output_format* is `sage_binary`, files read must be labelled *sage_data_path*.XXX. If *sage_output_format* is `sage_hdf5`, the file read will be *sage_data_path* and the groups accessed will be Core_XXX at snapshot *snapshot*. In both cases, `XXX` represents the numbers in the range [*first_file_to_analyze*, *last_file_to_analyze*] inclusive.

Type string

sage_file
The path to where the **SAGE** `.ini` file is located.

Type str

sage_output_format
The output format **SAGE** wrote in. A specific Data Class (e.g., *SageBinaryData* and *SageHdf5Data*) must be written and used for each *sage_output_format* option. We refer to ../user/data_class for more information about adding your own Data Class to ingest data.

Type {`"sage_binary"`, `"sage_binary"`}

sample_size
Specifies the length of the *properties* attributes stored as 1-dimensional `ndarray`. These *properties* are initialized using *init_scatter_properties()*.

Type int

snapshot
Specifies the snapshot to be read. If *sage_output_format* is `sage_hdf5`, this specifies the HDF5 group to be read. Otherwise, if *sage_output_format* is `sage_binary`, this attribute will be used to index *redshifts* and generate the suffix for *sage_data_path*.

Type int

volume
Volume spanned by the trees analyzed by this model. This depends upon the number of files processed, `[:py:attr:`~first_file_to_analyze`, :py:attr:`~last_file_to_analyze`]`, relative to the total number of files the simulation spans over, `num_sim_tree_files`.

Notes

This is **not** necessarily *box_size* cubed. It is possible that this model is only analysing a subset of files and hence the volume will be less.

Type volume

2.9 sage_analysis.sage_hdf5

This module defines the `SageHdf5Data` class. This class interfaces with the *`Model`* class to read in binary data written by **SAGE**. The value of *`sage_output_format`* is generally `sage_hdf5` if it is to be read with this class.

We refer to *Ingesting Custom Data* for more information about adding your own Data Class to ingest data.

Author: Jacob Seiler.

class `sage_analysis.sage_hdf5.`**`SageHdf5Data`**(*model: sage_analysis.model.Model, sage_file_to_read: str*)

Class intended to inteface with the *`Model`* class to ingest the data written by **SAGE**. It includes methods for reading the output galaxies, setting cosmology etc. It is specifically written for when *`sage_output_format`* is `sage_hdf5`.

`_check_model_compatibility`(*model: sage_analysis.model.Model, sage_dict: Optional[Dict[str, Any]]*) → None

Ensures that the attributes in the *`Model`* instance are compatible with the variables read from the **SAGE** parameter file (if read at all).

Parameters

- **model** (*`Model`* instance) – The model that this data class is associated with.
- **sage_dict** (*optional, dict[str, Any]*) – A dictionary containing all of the fields read from the **SAGE** parameter file.

Warning:

UserWarning Raised if the user initialized `Model` with a value of *`num_sage_output_files`* that is different to the value specified in the HDF5 file.

`_read_sage_params`(*sage_file_path: str*) → Dict[str, Any]

Read the **SAGE** parameter file.

Parameters **sage_file_path** (*string*) – Path to the **SAGE** parameter file.

Returns **model_dict** – Dictionary containing the parameter names and their values.

Return type dict [str, var]

`close_file`(*model*)

Closes the open HDF5 file.

`determine_num_gals`(*model: sage_analysis.model.Model, snapshot: int, *args*)

Determines the number of galaxies in all cores for this model at the specified snapshot.

Parameters

- **model** (*`Model`* class) – The *`Model`* we're reading data for.
- **snapshot** (*int*) – The snapshot we're analysing.
- ***args** (*Any*) – Extra arguments to allow other data class to pass extra arguments to their version of `determine_num_gals`.

`determine_volume_analyzed`(*model: sage_analysis.model.Model*) → float

Determines the volume analyzed. This can be smaller than the total simulation box.

Parameters **model** (*`Model`* instance) – The model that this data class is associated with.

Returns **volume** – The numeric volume being processed during this run of the code in (Mpc/h)^3.

Return type float

read_gals(*model*, *core_num*, *pbar=None*, *plot_galaxies=False*, *debug=False*)
Reads the galaxies of a single core at the specified `snapshot`.

Parameters

- **model** (`Model` class) – The `Model` we're reading data for.
- **core_num** (*Integer*) – The core group we're reading.
- **pbar** (`tqdm` class instance, optional) – Bar showing the progress of galaxy reading. If `None`, progress bar will not show.
- **plot_galaxies** (*Boolean, optional*) – If set, plots and saves the 3D distribution of galaxies for this file.
- **debug** (*Boolean, optional*) – If set, prints out extra useful debug information.

Returns **gals** – The galaxies for this file.

Return type `h5py` group

Notes

`tqdm` does not play nicely with printing to stdout. Hence we disable the `tqdm` progress bar if `debug=True`.

update_snapshot_and_data_path(*model: sage_analysis.model.Model*, *snapshot: int*)
Updates the `snapshot` attribute to `snapshot`. As the HDF5 file contains all snapshot information, we do **not** need to update the path to the output data. However, ensure that the file itself is still open.

2.10 sage_analysis.sage_binary

This module defines the `SageBinaryData` class. This class interfaces with the `Model` class to read in binary data written by **SAGE**. The value of `sage_output_format` is generally `sage_binary` if it is to be read with this class.

We refer to *Ingesting Custom Data* for more information about adding your own Data Class to ingest data.

Author: Jacob Seiler.

class `sage_analysis.sage_binary.`**SageBinaryData**(*model: sage_analysis.model.Model*, *sage_file_to_read: str*)
Class intended to inteface with the `Model` class to ingest the data written by **SAGE**. It includes methods for reading the output galaxies, setting cosmology etc. It is specifically written for when `sage_output_format` is `sage_binary`.

_read_sage_params(*sage_file_path: str*) → Dict[str, Any]
Read the **SAGE** parameter file.

Parameters **sage_file_path** (*string*) – Path to the **SAGE** parameter file.

Returns **model_dict** – Dictionary containing the parameter names and their values.

Return type dict [str, var]

determine_num_gals(*model: sage_analysis.model.Model*, **args*)
Determines the number of galaxies in all files for this `Model`.

Parameters

- **model** (`Model` class) – The `Model` we're reading data for.
- ***args** (*Any*) – Extra arguments to allow other data class to pass extra arguments to their version of `determine_num_gals`.

determine_volume_analyzed(*model: sage_analysis.model.Model*) → float
Determines the volume analyzed. This can be smaller than the total simulation box.

Parameters **model** (`Model` instance) – The model that this data class is associated with.

Returns **volume** – The numeric volume being processed during this run of the code in (Mpc/h)^3.

Return type float

get_galaxy_struct()
Sets the `numpy` structured array for holding the galaxy data.

read_gals(*model*, *file_num*, *pbar=None*, *plot_galaxies=False*, *debug=False*)
Reads the galaxies of a model file at snapshot specified by `snapshot`.

Parameters

- **model** (`Model` class) – The `Model` we're reading data for.
- **file_num** (*int*) – Suffix number of the file we're reading.
- **pbar** (`tqdm` class instance, optional) – Bar showing the progress of galaxy reading. If `None`, progress bar will not show.
- **plot_galaxies** (*bool, optional*) – If set, plots and saves the 3D distribution of galaxies for this file.
- **debug** (*bool, optional*) – If set, prints out extra useful debug information.

Returns **gals** – The galaxies for this file.

Return type `numpy` structured array with format given by `get_galaxy_struct()`

Notes

`tqdm` does not play nicely with printing to stdout. Hence we disable the `tqdm` progress bar if `debug=True`.

update_snapshot_and_data_path(*model: sage_analysis.model.Model*, *snapshot: int*)
Updates the `_sage_data_path` to point to a new redshift file. Uses the redshift array `redshifts`.

Parameters **snapshot** (*int*) – Snapshot we're updating `_sage_data_path` to point to.

2.11 sage_analysis.example_calcs

Here we show a myriad of functions that can be used to calculate properties from the **SAGE** output. By setting the correct plot toggles and calling `generate_func_dict()`, a dictionary containing these functions can be generated and passed to `calc_properties_all_files()` to calculate the properties.

The properties are stored (and updated) in the `properties` attribute.

We refer to *Analysing SAGE Output* for more information on how the calculations are handled.

Author: Jacob Seiler

`sage_analysis.example_calcs.`**`calc_BMF`**(*model*, *gals*, *snapshot: int*)

Calculates the baryon mass function of the given galaxies. That is, the number of galaxies at a given baryon (stellar + cold gas) mass.

The `Model.properties["snapshot_<snapshot>"]["BMF"]` array will be updated.

`sage_analysis.example_calcs.`**`calc_BTF`**(*model*, *gals*, *snapshot: int*)

Calculates the baryonic Tully-Fisher relation for spiral galaxies in the given set of galaxies.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["BTF_mass"]` and `Model.properties["snapshot_<snapshot>"]["BTF_vel"]` arrays is given by *`sample_size`* weighted by `number_spirals_passed / _num_gals_all_files`. If this value is greater than `number_spirals_passed`, then all spiral galaxies will be used.

`sage_analysis.example_calcs.`**`calc_GMF`**(*model*, *gals*, *snapshot: int*)

Calculates the gas mass function of the given galaxies. That is, the number of galaxies at a given cold gas mass.

The `Model.properties["snapshot_<snapshot>"]["GMF"]` array will be updated.

`sage_analysis.example_calcs.`**`calc_SFRD_history`**(*model*, *gals*, *snapshot: int*)

Calculates the sum of the star formation across all galaxies. This will be normalized by the simulation volume to determine the density. See `plot_SFRD()` for full implementation.

The `Model.properties["snapshot_<snapshot>"]["SFRD"]` value is updated.

`sage_analysis.example_calcs.`**`calc_SMD_history`**(*model*, *gals*, *snapshot: int*)

Calculates the sum of the stellar mass across all galaxies. This will be normalized by the simulation volume to determine the density. See `plot_SMD()` for full implementation.

The `Model.properties["snapshot_<snapshot>"]["SMD"]` value is updated.

`sage_analysis.example_calcs.`**`calc_SMF`**(*model: sage_analysis.model.Model*, *gals*, *snapshot: int*, *calc_sub_populations: bool = False*, *smf_property_name: str = 'SMF'*)

Calculates the stellar mass function of the given galaxies. That is, the number of galaxies at a given stellar mass.

The `Model.properties["snapshot_<snapshot>"]"SMF"]` array will be updated. We also split the galaxy population into "red" and "blue" based on the value of *`sSFRcut`* and update the `Model.properties["snapshot_<snapshot>"]["red_SMF"]` and `Model.properties["snapshot_<snapshot>"]["blue_SMF"]` arrays.

Parameters

- **snapshot** (*int*) – The snapshot the SMF is being calculated at.
- **plot_sub_populations** (*boolean, optional*) – If `True`, calculates the stellar mass function for red and blue sub-populations.
- **smf_property_name** (*string, optional*) – The name of the property used to store the stellar mass function. Useful if different calculations are computing the stellar mass function but saving it as a different property.

`sage_analysis.example_calcs.`**`calc_SMF_history`**(*model*, *gals*, *snapshot: int*)

Calculates the stellar mass function of the given galaxies. That is, the number of galaxies at a given stellar mass.

The `Model.properties["SMF"_history]` array will be updated.

`sage_analysis.example_calcs.`**`calc_baryon_fraction`**(*model*, *gals*, *snapshot: int*)

Calculates the `mass_baryons / halo_virial_mass` as a function of halo virial mass for each baryon reseroivr (stellar, cold, hot, ejected, intra-cluster stars and black hole). Also calculates the ratio for the total baryonic mass.

The `Model.properties["snapshot_<snapshot>"]["halo_<reservoir_name>_fraction_sum"]` arrays are updated for each reservoir. In addition, `Model.properties["snapshot_<snapshot>"]["halo_baryon_f` is updated.

Notes

The halo virial mass we use is the **background FoF halo**, not the immediate host halo of each galaxy.

We only **sum** the baryon mass in each stellar mass bin. When converting this to the mass fraction, one must divide by the number of halos in each halo mass bin, `Model.properties["snapshot_<snapshot>"]["fof_HMF"]`. See *`plot_baryon_fraction()`* for full implementation.

If the `Model.properties["snapshot_<snapshot>"]["fof_HMF"]` property, with associated bins `Model.bins["halo_mass"bin"]` have not been initialized, a `ValueError` is thrown.

`sage_analysis.example_calcs.`**`calc_bh_bulge`**(*model*, *gals*, *snapshot: int*)

Calculates the black hole mass as a function of bulge mass.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["BlackHoleMass"]` and `Model.propertiesp["snapshot_<snapshot>"]["BulgeMass"]` arrays is given by *`sample_size`* weighted by `number_galaxies_passed / _num_gals_all_files`. If this value is greater than `number_galaxies_passed`, then all galaxies will be used.

Notes

We only consider galaxies with bulge mass greater than 10^8 Msun/h and a black hole mass greater than 10^5 Msun/h.

`sage_analysis.example_calcs.`**`calc_bulge_fraction`**(*model*, *gals*, *snapshot: int*)

Calculates the `bulge_mass / stellar_mass` and `disk_mass / stellar_mass` ratios as a function of stellar mass.

The `Model.properties["snapshot_<snapshot>"]["fraction_bulge_sum"]`, `Model.properties["snapshot_<snapshot>"]["fraction_disk_sum"]`, `Model.properties["snapshot_<snapshot>"]["fraction_bulge_var"]`, `Model.properties["snapshot_<snapshot>"]["fraction_disk_var"]` arrays will be updated.

Notes

We only **sum** the bulge/disk mass in each stellar mass bin. When converting this to the mass fraction, one must divide by the number of galaxies in each stellar mass bin, the stellar mass function `Model.properties["snapshot_<snapshot>"]["SMF"]`. See *`plot_bulge_fraction()`* for full implementation.

`sage_analysis.example_calcs.`**`calc_gas_fraction`**(*model*, *gals*, *snapshot: int*)

Calculates the fraction of baryons that are in the cold gas reservoir as a function of stellar mass.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["gas_frac_mass"]` and `Model.properties["snapshot_<snapshot>"]["gas_frac"]` arrays is given by *`sample_size`* weighted by `number_spirals_passed / _num_gals_all_files`. If this value is greater than `number_spirals_passed`, then all spiral galaxies will be used.

`sage_analysis.example_calcs.`**`calc_metallicity`**(*model*, *gals*, *snapshot: int*)

Calculates the metallicity as a function of stellar mass.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["metallicity_mass"]` and `Model.properties["snapshot_<snapshot>"]["metallicity"]` arrays is given by *`sample_size`* weighted by `number_centrals_passed / _num_gals_all_files`. If this value is greater than `number_centrals_passed`, then all central galaxies will be used.

`sage_analysis.example_calcs.`**`calc_quiescent`**(*model*, *gals*, *snapshot: int*)

Calculates the quiescent galaxy fraction as a function of stellar mass. The galaxy population is also split into central and satellites and the quiescent fraction of these are calculated.

The `Model.properties["snapshot_<snapshot>"]["centrals_MF"]`, `Model.properties["snapshot_<snapshot>"]["satellites_MF"]`, `Model.properties["snapshot_<snapshot>"]["quiescent_galaxy_counts"]`, `Model.properties["snapshot_<snapshot>"]["quiescent_centrals_counts"]`, and `Model.properties["snapshot_<snapshot>"]["quiescent_satellites_counts"]` arrays will be updated.

Notes

We only **count** the number of quiescent galaxies in each stellar mass bin. When converting this to the quiescent fraction, one must divide by the number of galaxies in each stellar mass bin, the stellar mass function `Model.properties["snapshot_<snapshot>"]["SMF"]`. See *`plot_quiescent()`* for an example implementation.

`sage_analysis.example_calcs.`**`calc_reservoirs`**(*model*, *gals*, *snapshot: int*)

Calculates the mass in each reservoir as a function of halo virial mass.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["reservoir_mvir"]` and `Model.properties["snapshot_<snapshot>"]["reservoir_<reservoir_name>"]` arrays is given by *`sample_size`* weighted by `number_centrals_passed / _num_gals_all_files`. If this value is greater than `number_centrals_passed`, then all central galaxies will be used.

`sage_analysis.example_calcs.`**`calc_sSFR`**(*model*, *gals*, *snapshot: int*)

Calculates the specific star formation rate (star formation divided by the stellar mass of the galaxy) as a function of stellar mass.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["sSFR_mass"]` and `Model.properties["snapshot_<snapshot>"]["sSFR_sSFR"]` arrays is given by *`sample_size`* weighted by `number_gals_passed / _num_gals_all_files`. If this value is greater than `number_gals_passed`, then all galaxies with non-zero stellar mass will be used.

`sage_analysis.example_calcs.`**`calc_spatial`**(*model*, *gals*, *snapshot: int*)

Calculates the spatial position of the galaxies.

The number of galaxies added to `Model.properties["snapshot_<snapshot>"]["<x/y/z>_pos"]` arrays is given by *`sample_size`* weighted by `number_galaxies_passed / _num_gals_all_files`. If this value is greater than `number_galaxies_passed`, then all galaxies will be used.

2.12 sage_analysis.example_plots

Here we show a myriad of functions that can be used to plot properties calculated from the **SAGE** output.

We refer to ../user/plot for more information on how plotting is handled.

Authors: (Jacob Seiler, Manodeep Sinha)

`sage_analysis.example_plots.`**`adjust_legend`**(*ax*, *location='upper right'*, *scatter_plot=0*)

Adjusts the legend of a specified axis.

Parameters

- **ax** (`matplotlib` axes object) – The axis whose legend we're adjusting
- **location** (String, default "upper right". See `matplotlib` docs for full options) – Location for the legend to be placed.
- **scatter_plot** (*{0, 1}*) – For plots involved scattered-plotted data, we adjust the size and alpha of the legend points.

Returns

Return type None. The legend is placed directly onto the axis.

`sage_analysis.example_plots.`**`plot_BMF`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the baryonic mass function for the specified models. This is the mass function for the stellar mass + cold gas.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*List of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, optional*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>2.BaryonicMassFunction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_BTF`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the baryonic Tully-Fisher relationship for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>4.BaryonicTullyFisher.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_GMF`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the gas mass function for the specified models. This is the mass function for the cold gas.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>3.GasMassFunction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_SFRD_history`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the evolution of star formation rate density for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*List of ints*) – This is a dummy variable that is present to ensure the signature is identical to the other plot functions.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **snapshot** (*int*) – This is a dummy variable that is present to ensure the signature is identical to the other plot functions.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>B.SFRDensity.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_SMD_history`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the evolution of stellar mass density for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.

- **snapshots** (*List of ints*) – This is a dummy variable that is present to ensure the signature is identical to the other plot functions.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>C.StellarMassDensity.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_SMF`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png', plot_sub_populations: bool = False*) → matplotlib.figure.Figure

Plots the stellar mass function for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **plot_sub_populations** (*Boolean, default False*) – If `True`, plots the stellar mass function for red and blue sub-populations.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>1.StellarMassFunction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_SMF_history`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format='png'*) → matplotlib.figure.Figure

Plots the evolution of the stellar mass function for the specified models. This function loops over the value of `model.SMF_snaps` and plots and the SMFs at each snapshots.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – This is a dummy variable that is present to ensure the signature is identical to the other plot functions.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**

- **The plot will be saved as "<plot_output_path>A.StellarMassFunction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_baryon_fraction`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png', plot_sub_populations: bool = False*) → matplotlib.figure.Figure

Plots the total baryon fraction as afunction of halo mass for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **plot_sub_populations** (*Boolean, default False*) – If `True`, plots the baryon fraction for each reservoir. Otherwise, only plots the total baryon fraction.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>11.BaryonFraction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_bh_bulge`**(*models: List[sage_analysis.model.Model], snapshots: int, plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the black-hole bulge relationship for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>8.BlackHoleBulgeRelationship.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_bulge_fraction`**(*models: List[sage_analysis.model.Model], snapshots: int, plot_output_path: str, plot_output_format: str = 'png', plot_var: bool = False*) → matplotlib.figure.Figure

Plots the fraction of the stellar mass that is located in the bulge/disk as a function of stellar mass for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshot** (*int*) – Snapshot we're plotting at.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **plot_var** (*Boolean, default False*) – If `True`, plots the variance as shaded regions.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>10.BulgeMassFraction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_gas_fraction`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the fraction of baryons that are in the cold gas reservoir as a function of stellar mass for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>6.GasFraction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_metallicity`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the metallicity as a function of stellar mass for the speicifed models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.

- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>7.Metallicity.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_quiescent`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png', plot_sub_populations: bool = False*) → matplotlib.figure.Figure

Plots the fraction of galaxies that are quiescent as a function of stellar mass for the specified models. The quiescent cut is defined by *`sSFRcut`*.

Parameters

- **models** (List of *`Model`* class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each *`Model`* in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **plot_sub_populations** (*Boolean, default False*) – If `True`, plots the centrals and satellite sub-populations.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>9.QuiescentFraction.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_reservoirs`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → List[matplotlib.figure.Figure]

Plots the mass in each reservoir as a function of halo mass for the specified models.

Parameters

- **models** (List of *`Model`* class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each *`Model`* in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **A plot will be saved as ‘“”<plot_output_path>12.MassReservoirs<model.label>.<plot_output_format>”‘ for each mode.**

`sage_analysis.example_plots.`**`plot_sSFR`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the specific star formation rate as a function of stellar mass for the specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **The plot will be saved as "<plot_output_path>5.SpecificStarFormationRate.<plot_output_format>"**

`sage_analysis.example_plots.`**`plot_spatial`**(*models: List[sage_analysis.model.Model], snapshots: List[int], plot_output_path: str, plot_output_format: str = 'png'*) → matplotlib.figure.Figure

Plots the spatial distribution of the galaxies for specified models.

Parameters

- **models** (List of `Model` class instance) – Models that will be plotted. These instances contain the properties necessary to create the plot, accessed via `Model.properties["snapshot_<snapshot>"]["property_name"]`.
- **snapshots** (*list of ints*) – The snapshot to be plotted for each `Model` in `models`.
- **plot_output_path** (*string*) – Path to where the plot will be saved.
- **plot_output_format** (*string, default "png"*) – Format the plot will be saved in, includes the full stop.
- **Generates**
- **———**
- **A plot will be saved as '""<plot_output_path>13.SpatialDistribution<model.label>.<plot_output_format>"' for each**
- **model.**

`sage_analysis.example_plots.`**`plot_spatial_3d`**(*pos, output_file, box_size*) → matplotlib.figure.Figure

Plots the 3D spatial distribution of galaxies.

Parameters

- **pos** (`numpy` 3D array with length equal to the number of galaxies) – The position (in Mpc/h) of the galaxies.
- **output_file** (*String*) – Name of the file the plot will be saved as.

Returns

Return type None. A plot will be saved as `output_file`.

`sage_analysis.example_plots.`**`setup_matplotlib_options`**`()`

Set the default plotting parameters.

2.13 sage_analysis.utils

`sage_analysis.utils.`**`generate_func_dict`**`(`*plot_toggles*, *module_name*, *function_prefix*, *keyword_args={}*`)` → Dict[str, Tuple[Callable, Dict[str, Any]]]

Generates a dictionary where the keys are the function name and the value is a list containing the function itself (0th element) and keyword arguments as a dictionary (1st element). All functions in the returned dictionary are expected to have the same call signature for non-keyword arguments. Functions are only added when the `plot_toggles` value is non-zero.

Functions are required to be named `<module_name><function_prefix><plot_toggle_key>` For example, the default calculation function are kept in the `model.py` module and are named `calc_<toggle>`. E.g., `sage_analysis.model.calc_SMF()`, `sage_analysis.model.calc_BTF()`, `sage_analysis.model.calc_sSFR()` etc.

Parameters

- **plot_toggles** (*dict, [string, int]*) – Dictionary specifying the name of each property/plot and whether the values will be generated + plotted. A value of 1 denotes plotting, whilst a value of 0 denotes not plotting. Entries with a value of 1 will be added to the function dictionary.
- **module_name** (*string*) – Name of the module where the functions are located. If the functions are located in this module, pass an empty string "".
- **function_prefix** (*string*) – Prefix that is added to the start of each function.
- **keyword_args** (*dict [string, dict[string, variable]], optional*) – Allows the adding of keyword aguments to the functions associated with the specified plot toggle. The name of each keyword argument and associated value is specified in the inner dictionary.

Returns **func_dict** – The key of this dictionary is the name of the function. The value is a list with the 0th element being the function and the 1st element being a dictionary of additional keyword arguments to be passed to the function. The inner dictionary is keyed by the keyword argument names with the value specifying the keyword argument value.

Return type dict [string, tuple(function, dict[string, variable])]

Examples

```
>>> import sage_analysis.example_calcs
>>> import sage_analysis.example_plots
>>> plot_toggles = {"SMF": 1}
>>> module_name = "sage_analysis.example_calcs"
>>> function_prefix = "calc_"
>>> generate_func_dict(plot_toggles, module_name, function_prefix) #doctest:␣
↪+ELLIPSIS
{'calc_SMF': (<function calc_SMF at 0x...>, {})}
>>> module_name = "sage_analysis.example_plots"
>>> function_prefix = "plot_"
>>> generate_func_dict(plot_toggles, module_name, function_prefix) #doctest:␣
↪+ELLIPSIS
{'plot_SMF': (<function plot_SMF at 0x...>, {})}
```

```
>>> import sage_analysis.example_plots
>>> plot_toggles = {"SMF": 1}
>>> module_name = "sage_analysis.example_plots"
>>> function_prefix = "plot_"
>>> keyword_args = {"SMF": {"plot_sub_populations": True}}
>>> generate_func_dict(plot_toggles, module_name, function_prefix, keyword_args)
↪#doctest: +ELLIPSIS
{'plot_SMF': (<function plot_SMF at 0x...>, {'plot_sub_populations': True})}
```

```
>>> import sage_analysis.example_plots
>>> plot_toggles = {"SMF": 1, "quiescent": 1}
>>> module_name = "sage_analysis.example_plots"
>>> function_prefix = "plot_"
>>> keyword_args = {"SMF": {"plot_sub_populations": True},
...                 "quiescent": {"plot_output_format": "pdf", "plot_sub_
↪populations": True}}
>>> generate_func_dict(plot_toggles, module_name, function_prefix, keyword_args)
↪#doctest: +ELLIPSIS
{'plot_SMF': (<function plot_SMF at 0x...>, {'plot_sub_populations': True}),
↪'plot_quiescent': (<function plot_quiescent at 0x...>, {'plot_output_format':
↪'pdf', 'plot_sub_populations': True})}
```

`sage_analysis.utils.`**`read_generic_sage_params`**(*sage_file_path: str*) → Dict[str, Any]

Reads the **SAGE** parameter file values. This function is used for the default `sage_binary` and `sage_hdf5` formats. If you have a custom format, you will need to write a `read_sage_params` function in your own data class.

Parameters **sage_file_path** (*string*) – Path to the **SAGE** parameter file.

Returns

- **model_dict** (*dict [str, var]*) – Dictionary containing the parameter names and their values.
- *Errors*
- ——
- *FileNotFoundError* – Raised if the specified **SAGE** parameter file is not found.

`sage_analysis.utils.`**`select_random_indices`**(*inds: numpy.ndarray, global_num_inds_available: int, global_num_inds_requested: int, seed: Optional[int] = None*) → numpy.ndarray

Flag this with Manodeep to exactly use a descriptive docstring.

Parameters

- **vals** (`ndarray` of values) – Values that the random subset is selected from.
- **global_num_inds_available** (*int*) – The total number of indices available across all files.
- **global_num_inds_requested** (*int*) – The total number of indices requested across all files.
- **seed** (*int, optional*) – If specified, seeds the random number generator with the specified seed.

Returns **random_vals** – Values chosen.

Return type `ndarray` of values

Examples

```
>>> import numpy as np
>>> seed = 666
>>> inds = np.arange(10)
>>> global_num_inds_available = 100
>>> global_num_inds_requested = 50 # Request less than the number of inds␣
↪available
...                                # across all files, but more than is in this␣
↪file.
>>> select_random_indices(inds, global_num_inds_available, global_num_inds_
↪requested, seed) # Returns a random subset.
array([2, 6, 9, 4, 3])
```

```
>>> import numpy as np
>>> seed = 666
>>> inds = np.arange(30)
>>> global_num_inds_available = 100
>>> global_num_inds_requested = 10 # Request less than the number of inds␣
↪available
...                                # across all files, and also less than what is
...                                # available in this file.
>>> select_random_indices(inds, global_num_inds_available, global_num_inds_
↪requested, seed) # Returns a random subset.
array([12,  2, 13])
```

```
>>> import numpy as np
>>> inds = np.arange(10)
>>> global_num_inds_available = 100
>>> global_num_inds_requested = 500  # Request more than the number of inds␣
↪available
...                                  # across all file.
>>> select_random_indices(inds, global_num_inds_available, global_num_inds_
↪requested) # All input indices are returned.
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Python Module Index

Index