Py-Galaxies - Utilising Python and merger graphs to update semi-analytic models of galaxy evolution.

Candidate Number: 183735
Supervisor: Prof. Peter Thomas
Word Count: 14100
Last Updated: May 19, 2021

University of Sussex

Abstract

When taking a semi-analytical approach to galaxy formation it is often necessary to use a fast and efficient programming language. Generally compiled, these languages make it difficult to change runtime parameters and sacrifice readability for speed. Here, we present Py-Galaxies; a hybrid language approach that combines Python and C for a user-friendly experience that doesn't lack the computational speed needed for complex calculations. Using L-Galaxies (Henriques et al., 2019) as the basis, several C-routines that model baryonic infall, cooling gas, and star formation were extracted and adapted to work in conjunction with Python through the module Cython. In addition, the merger graph (Roper, Thomas, and Srisawat, 2020), a new structure that tracks the evolution of dark matter haloes was implemented within a galaxy formation model for the first time. Merger graphs eliminate the occurrence of catastrophic failures and lead to smoother mass growth than the traditional merger tree. All results within the body of this paper are therefore new and original works.

Alongside laying the foundations of a hybrid-language galaxy simulation, it was within the scope of this project to compare and contrast different algorithmic structures to read and process merger graphs. It was found that using the core Python structures, such as lists and classes, produced the fastest version of the code. However, using NumPy structured arrays to store merger graph information was more memory efficient. In general, the class based method was more user-friendly and, at the scale of the current merger graphs, acceptable for them to be more memory inefficient.

Contents

List of Tables

List of Figures

Preface

Chapter 1 is an introduction to semi-analytic models of galaxy formation and an overview of their current structure. For the introduction, independent research was carried out, however, my supervisor was able to suggest source material regarding dark matter structure based on previous work done by them and their PhD student. The model parameters in Chapter 2 were taken from referenced materials suggested by my supervisor. All code written for this chapter is original work. The background information within Chapter 3 is a mixture of independent research and previous work carried out by my supervisor and their PhD student. The halo graph data described within this chapter and the following chapters, were provided by my supervisor's PhD student. All code written to structure and analyse the data in this chapter is entirely my own. Chapter 3 contains theory taken from referenced material as suggested by my supervisor, with all code my own. To implement the calculations, I adapted C code originally written for L-Galaxies and consequently wrapped it within my own Python code. The same is true for both Chapter 5 and Chapter 6. Chapter 7 is an original analysis between the two versions of code I have written. Chapter 8 features documentation that is entirely my own work.

Chapter 1

Introduction

Galaxy formation is the result of innumerable astrophysical processes woven together, encompassing the intricacies of stellar synthesis to the evolution of the cosmic web. Through recent advancements in observational astronomy and computer science, our understanding of these processes is accelerating (Vogelsberger et al., 2020). They can be simulated on larger scales, in greater detail, and tuned to a better accuracy than ever before owing to the sheer volume of data astronomers now collect about our skies (Gudivada, Baeza-Yates, and Raghavan, 2015). As simulations become more complex, it is integral to have a firm understanding of the underlying theory alongside strong programming skills.

Generally, galaxy simulations can be structured in two unique ways. The more complex of the two, hydro-dynamical, attempts to numerically solve the non-linear, N-body equations that govern galaxy formation. Due to the tricky nature of these calculations only small numbers of particles can be simulated within a small volume. In contrast, semi-analytical methods are able to simulate large volumes of space through the use of rough approximations based on empirical observations, albeit at a lower level of precision. The work presented within this research is focused on improving accessibility and performance of the semi-analytical simulation L-Galaxies (Henriques et al., 2019). Other notable hydro-dynamical and semi-analytical cosmology simulations have been listed in Table 1.1.

Hydro-dynamical		Semi-analytical	
Name	Paper	Name	Paper
EAGLE	Schaye et al. (2014)	GalICS 2.1	Cattaneo et al. (2020)
ENZO	Bryan et al. (2014)	GALFORM	Baugh et al. (2018)
FLASH	Calder et al. (2002)	Millennium Run	Springel et al. (2005)
GADGET-4	Springel et al. (2020)	MORGANA	Monaco et al. (2007)
GASOLINE2	Wadsley et al. (2017)	Santa Cruz	Somerville et al. (2008)

Table 1.1: Compilation of notable galaxy and dark matter halo evolution models.

L-Galaxies, also known as the Munich Galaxy Formation Model, is an international collaboration which has been developed over the past two decades. Conceived in 2001 L-Galaxies, much like other semi-analytical models (hereafter, SAMs), use dark matter halo and sub-halo merger histories as the basis for galaxy formation (Springel, White, et al., 2001). These dark matter haloes are essential within ΛCDM cosmology as without them, baryonic matter in the early universe could not have cooled sufficiently to form gravitationally bound objects such as galaxies (Peebles, 1982). Additionally, the speed at which present

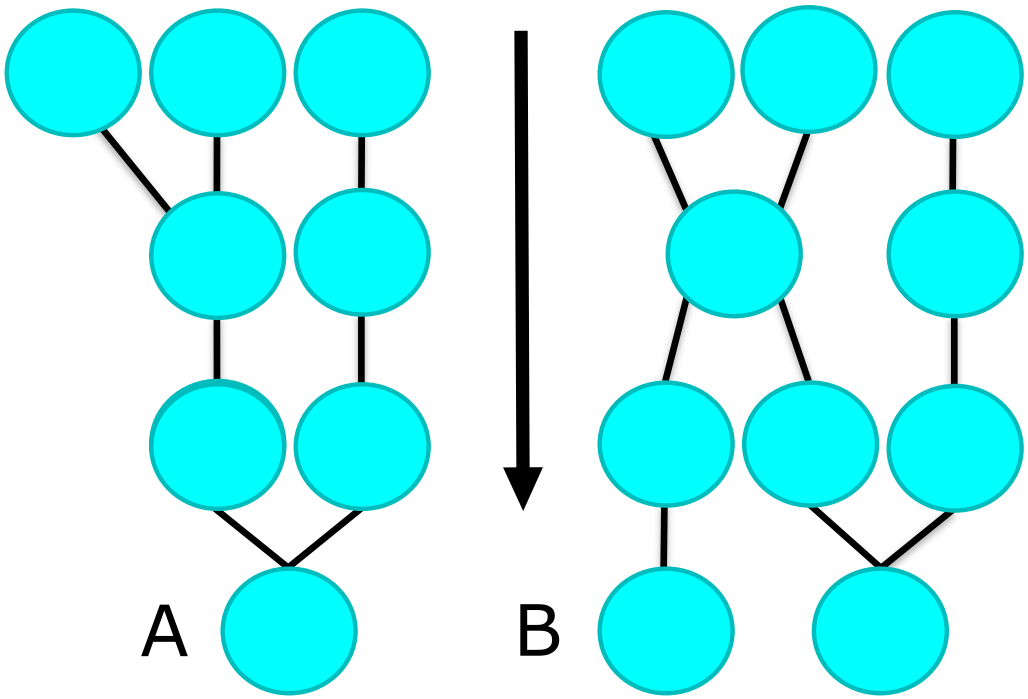

Figure 1.1: A simple diagram highlighting the differences between a merger tree (A) and a merger graph (B). The arrow denotes the direction of time.

day galaxies rotate cannot be explained by their baryonic matter alone (Rubin, Ford, and Thonnard, 1978). Owning to this, L-Galaxies utilise dark matter haloes (heron haloes) as containers for infalling baryons and dark matter subhaloes (heron subhaloes), which in turn, are containers for cooling baryons and galaxy formation. All information on the evolution of this hierarchical structure throughout time is mapped and stored using merger trees.

Merger trees, popularised by Lacey and Cole (1993), are simple graphical representations of how small, primordial haloes grow and merge eventually leading to the creation of a single, much larger halo at the present day. Each step in this graph is called a snapshot and current implementations of this structure enforce that a single subhalo be selected as the 'main' subhalo, neglecting all others as satellites; furthermore, halo mergers are deemed irreversible. Though the algorithms used to create merger trees from halo catalogues are varied, the structure itself has remained largely similar for 30 years leading to several widespread issues. Firstly, extreme mass fluctuations called 'flip-flops' can appear in the merger history of a halo. This is caused by difficulty pinpointing the centre about which to grow a halo and generally leads to mass jumping from one substructure to another (Roper, Thomas, and Srisawat, 2020). Secondly, the tree structure can generate 'catastrophic failures' where a halo appears from nothing, or completely disappears. This occurs when haloes get classified as a merged object when in reality, they are just passing through one another. As the merger tree structure does not allow for objects to split apart once merged, a new branch is then created terminating the history of one of the objects.

To solve these issues Roper, Thomas, and Srisawat (ibid.) have proposed the merger graph - an improvement over the merger tree as it allows for merged objects to separate, reducing the number of catastrophic failures (see a graphical representation in Figure 1.1). Furthermore, within the merger graph all substructures are treated equally meaning growth can be tracked for all objects and not just a single, enforced central halo. They have shown

that their implementation of this method is more effective and robust than the 10 leading merger tree algorithms compared in Srisawat et al. (2013). **The first key aim of this research is to begin building and benchmarking algorithmic structures to read and process merger graphs**, so that in future projects, the accuracy of the L-Galaxies simulation can be improved.

Having accurate simulations has become increasingly important. In the past, observational astronomers were feeding information into simulations and not getting much in return but now, simulations are becoming advanced enough to guide astronomers and help predict new physics. For example, simulations have predicted that galaxies form stars much slower than expected and that early galaxies are elongated, rather than spheroids (Ceverino, Primack, and Dekel, 2015). Not only do modern simulations have to be accurate, but as larger, more detailed spaces have been modelled they have had to become more robust and efficient. The issue then arises of which language to program them in. Simple, interpreted languages like Python and Ruby are more user-friendly but much slower than compiled languages such as C, Java, and Rust. L-Galaxies was written in C for this very reason. **The second key aim of this research is to find a method that enables Python to interface with existing L-Galaxies C-routines.** In doing this, the code will retain its efficiency while becoming more user-friendly through easy to access run-time options, enhanced scalability, and basic syntax.

The implementation of the two key aims described above have produced Py-Galaxies - the beginnings of a hybrid-language simulation that aims to build upon the successes of L-Galaxies. To demonstrate the merger graph structure paired with C integration, several physics routines from the L-Galaxies model have been implemented to further lay the foundations for future projects. The rest of this dissertation is structured as follows. Chapter 2 first explains how C has been integrated with Python and how the code has been structured. Chapter 3 then describes how the cosmic web was formed, how it is simulated, and how Py-Galaxies builds upon it. Chapter 4 illustrates how baryons become involved with the cosmic web and Chapter 5 explains how these baryons cool and accrete onto a halo's central subhalo. Chapter 6 first elaborates upon the formation of stars from a cold gas disk before introducing an empirical model by which results can be compared. In chapter 10 the model is summarised and conclusions drawn, while in Chapter 8, two algorithmic structures are compared and contrasted. Finally, Chapter 9 explains how the code has been documented.

Chapter 2

Code Structure & Setup

As the code for the Py-Galaxies model is a complex amalgamation of both Python and C, it is first necessary to discuss its structure before delving into the theory behind galaxy formation. A good starting point are the user-defined global parameters that need to be utilised by both Python and C. They contain information about the ΛCDM Universe such as the Hubble constant and density parameters which are essential for modelling galaxy formation. L-Galaxies and Py-Galaxies use Plank cosmology (Planck Collaboration, 2020) leading to the constants found in Table 2.1.

Name	Symbol	Value
Hubble Constant	H_0	67.30 kms^{-1}
Mass Density Parameter	Ω_m	0.315
Vacuum Density Parameter	Ω_Λ	0.685
Universal Baryon Fraction	f_b	0.155

Table 2.1: Cosmology parameters used within Py-Galaxies.

2.1 Py-Galaxies & Global Cosmology Parameters

The cosmology parameters above, and various other parameters that will be introduced throughout this paper, are stored within a single YAML Ain't Markup Language (YAML) file with short descriptions. This structured, readable format can then be read into Python as a dictionary, providing a convenient way to access global constants. A short snippet of the YAML file is shown below in "parameters.yml".

parameters.yml

```yaml
cosmology:
 omega_m:
  Description: 'Matter density parameter'
  Value:       0.315
  Units:       'None'
 H0:
  Description: 'The hubble parameter evaluated at t=0 (from Plank)'
  Value:       67.3
  Units:       '(km/s)/Mpc'
```

As C routines will be kept for complex calculations, essential constants will also need to be available within the C code once they have been read in by Python.

2.2 Merging Python and C

Python is an interpreted language which means that it is not fully translated to machine code prior to run time as is C or Java. A program called the interpreter reads the code line-by-line, executing each one as it goes. This causes Python code to be much slower than its compiled counterparts, which can be an issue when performing thousands of complex calculations. Luckily, Python's default interpreter was written in C so there are plenty of ways to combine both languages to get the best of both worlds. First, there is the original Python/C API which offers low level control including the ability to manipulate Python objects in C. However, this has a steep learning curve and requires C code to be written in a specific manner of which the L-Galaxies code is not. Moving on, PyBind11 offers higher-level methods but is mainly geared towards C++ and only has a short list of supported compilers. In the end Cython was chosen as not only does it provide a wrapper to compile Python code into C, but it can also be used to include existing pure C modules into a Python script - making it the best method to use. It is a highly mature library with lots of useful features and simple Python-like syntax. For these reasons it was used throughout the code to import and use L-Galaxies routines as well as control the global model parameters in C.

2.3 Using Cython To Control Global Constants

As Python will be reading in the constants from the YAML file after the C code has been compiled, a data structure with a companion function to update it is needed. The data structure was first created using a C header file that can be included accross all C modules alongside the definition of a function that will update it. The code shown below in "model_params.h" is a minimal example of this (any code included in the main body of the research will always be minimal working examples, with the full code either in the Appendix and referenced, or on GitHub).

model_params.h

```
1 struct allModelParameters{
2 double OmegaM;
3 double H0;
4 };
5 struct allModelParameters modelParams;
6 void update_c_model_params(double OmegaM, double H0);
```

The header file can then be included within a C file and a function created to take inputs and assign them to the newly defined global data structure `modelParams`. In this example, the function is created within in the file "c_model_params.c" seen below.

c_model_params.c

```c
#include <model_params.h>
void update_c_model_params(double OmegaM, double H0){
modelParams.OmegaM = OmegaM;
modelParams.H0 = H0;
};
```

The function must now be linked to Cython and defined again so it can be imported via Python. This is first done by creating a .pxd file, here named "L_Galaxies.pxd", which is the Cython equivalent of a header file in C.

L_Galaxies.pxd

```
cdef extern from "model_params.h":
    void update_c_model_params(double Omega, double H0);
```

The Cython header file can then be imported and used within a Cython file (.pyx). This is the file that will be compiled, with any functions within available to Python, so the C functions must be wrapped in Python syntax as seen below in "L_Galaxies.pyx".

L_Galaxies.pyx

```
cimport L_galaxies
def C_update_c_model_params(double Omega, double H0):
    L_galaxies.update_c_model_params(Omega, H0)
    return None
```

In order to compile and run this code as a module Python requires a setup script, an example of which can be seen below, creatively named "setup.py".

setup.py

```
from distutils.core import setup
from Cython.Build import cythonize
from distutils.extension import Extension

sourcefiles = ['L_Galaxies.pyx','c_model_params.c']
extensions = [Extension("L_Galaxies", sourcefiles)]

setup(ext_modules = cythonize(extensions))
```

Needed within the list `sourcefiles`, are the names of any C routines used, including the name of the main Cython file that is to be compiled. The `Extension` function then uses this list to construct an "Extension module" with the name L_Galaxies. The setup function, with the help of Cython, then produces a module file (L_Galaxies .so for Linux, .pyd for Windows) from the Extension modules. The setup.py file should be run from the command line using the code below.

Run in command line

```
python build setup.py build_ext --inplace
```

The C module parameters function can then be called from Python by first importing L_Galaxies, `import L_Galaxies`, and then accessed as `L_Galaxies.C_update_c_model_params`. The full version of this code which handles all of the global parameters can be found on GitHub. Any additional parameters needed in the future can be added to the data structure and consequently included as arguments to the function. In saying this, if the number of global parameters becomes large, it may be best to start passing in a single list or dictionary as an argument, though this would require the careful navigation of deconstructing complex Python types to simple C types. The calls to this C routine are all routed through the Parameters module and class written in Python. This class is also responsible for reading, and statically storing, tables of data used by other C routines at later stages in the model.

2.4 Reading In Data Tables With C

Within the L-Galaxies code, several C routines perform calculations using values extracted from tables provided by previous research, for example, gas cooling rates taken from Sutherland and Dopita (1993). These tables need to be available for the C code but are not needed with respect to any Python code. Additionally, they need to be read in only once and kept as a static variable so the data can be accessed without multiple disk reads. The solution to this was to read in the tables alongside the parameters; however, it was found the tables persist in memory even after the Python code has finished running. In some cases, when the code was run again from an iPython console, the tables would be read in and subsequently add their values onto those already found in memory from the previous run. To prevent this, the kernel had to be constantly restarted which is not ideal therefore, a Boolean flag was implemented into the model parameters C routine (c_model_params.c) to monitor whether or not the model had been run from the current kernel. Now that the C code contains the necessary information, physics routines can be implemented and used to calculate various properties.

2.5 Implementing Physics Routines With Cython

The implementation of the module parameters passed information one way: from Python to C. When a single value needs to be passed back from C to Python, i.e. the result of a calculation, the code presented in Section 2.3 is trivial to adapt. The update model parameters function in the C file would just need to return a value, changing void to the value's type, i.e. float. The same changes would then be made in the Cython files and the task completed. However, there are cases where multiple values need to be returned from

C to Python such as when star formation routines need to return the stellar mass formed as well as the mass of reheated gas. This isn't simple as a complex type such as a tuple or list cannot just be returned to Python. To get around this, a custom C data structure can be created and returned. First, it must be defined in a C header file alongside the function which utilises it. As an example, the structure has been added into the "model_params.h" file beside a function using dummy variables.

model_params.h

```
struct ReturnTuple{
    double ReturnStars, ReturnReheatedMass;
};
ReturnTuple starformation(double x, double y, double z);
```

The header file is then included in a C routine as in the previous example and the data structure used as the return variable from the function. Again, for the example, adding this as a new function in "c_model_params.c".

c_model_params.c

```
#include <model_params.h>
struct ReturnTuple starformation(double x, double y, double z)
{
    double Stars, ReheatedMass;
    // Do the calculations
    // Define the structure to return
    struct ReturnTuple ToReturn = {Stars, ReheatedMass};
    return ToReturn;
}
```

In the same fashion as the example in Section 2.3, the function needs to be defined in the Cython header file but in addition, the data structure also needs to be defined. These definitions are therefore added to the example file "L_Galaxies.pxd" as seen below.

L_Galaxies.pxd

```
cdef extern from "model_params.h":
    cdef struct ReturnTuple:
        double Returnstars
        double Return_reheated_mass

    ReturnTuple starformation(double x, double y, double z)
```

Again, importing the Cython header file into the Cython file "L_Galaxies.pyx".

L_Galaxies.pyx

```
cimport L_galaxies
def C_star_formation(double x, double y, double z):
    return L_galaxies.starformation(x, y, z)
```

The compilation step from Section 2.3 is then repeated and the function usable in the same manner. This C function can now return the custom data structure to Python, converting it to a dictionary where each variable name within the data structure becomes a key. In the model, these processes were repeated for each function needed from the original L-Galaxies code, with adaptions applied to each one to fit the Python use cases. All of the functions were made available to the Python code through a single module named "L_Galaxies" as seen in the examples. The Python code itself was then structured as a package containing four individual modules each of which, handled different sections of the model.

2.6 Py-Galaxies Python Package

To control the model, a single driver script is used to call Python routines from the Py-Galaxies packages which, in turn, calls C routines from the L-Galaxies Cython module. There are four modules within the Python package: "Halos.py", used to house all code pertaining to halo and subhalo data and calculations; "HDF_graph_properties.py", used to read, write, and store data from HDF5 input files; "Parameters.py", used to read in parameters for both Python and C routines; and "Monitor.py", used to monitor the memory and time consumption of the model. Figure 2.1 illustrates how the modules are stored alongside the C routines and an "__init__.py" file which tells Python to treat the Py-Galaxies modules as a package. Now the code structure has been documented, the foundations of the model can now be explained.

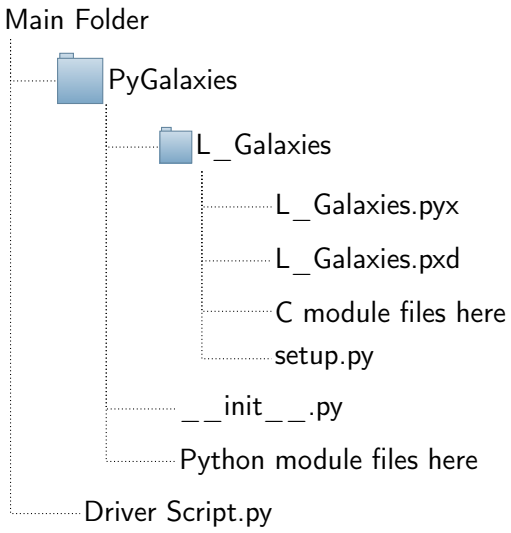

Figure 2.1: File tree representing the structure of the Py-Galaxies code.

Chapter 3

Dark Matter: The Simulation's Skeleton

3.1 Background Information

The starting point for galaxy formation is with dark matter haloes and the cosmic web. Early in the universe, it is thought quantum fluctuations caused perturbations to the density of matter which have since been exponentially increased due to the inflation of the Universe (Guth, 1981). This is evident from the tiny perturbations in the Cosmic Microwave Background Radiation (CMBR) growing to moderate perturbations in high redshift field surveys, and finally into large perturbations seen at the present day (i.e galaxies). At these early times, the universe was dominated by radiation meaning that baryonic matter was unable to collapse into structure. However, as far as we know, dark matter does not

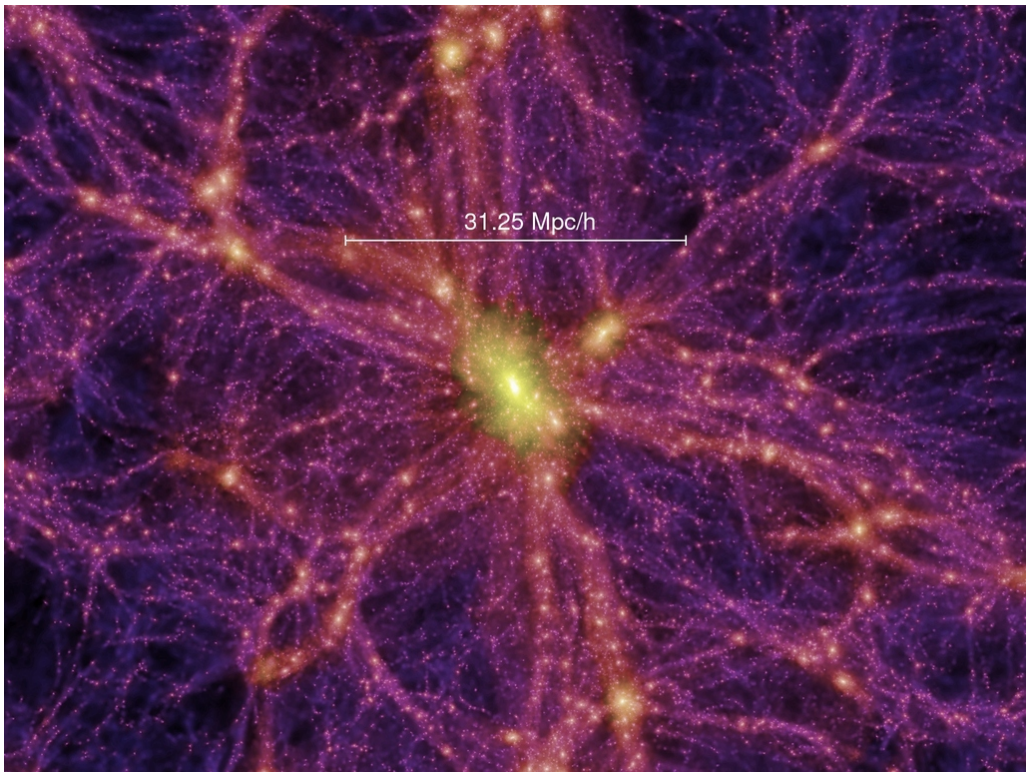

Figure 3.1: Simulated image of the cosmic web from the Millennium Simulation (Springel et al., 2005).

interact with any force besides gravity and therefore was able to collapse into dark matter

haloes. This leads to a vast network of interconnected haloes called the cosmic web, setting the stage for the formation of galaxies. A simulated image of this can be seen in Figure 3.1.

In order to implement this structure into galaxy simulations, haloes and subhaloes must first be identified and extracted from dark matter simulations. The algorithms that perform this task are called halo finders, with L-galaxies and Py-Galaxies slightly differing in the preferred method. L-Galaxies first identifies haloes through a friends-of-friends (FOF; Davis et al., 1985) algorithm, linking dark matter particles that have a separation less than 20% of the mean interpartcle spacing. The radius of this FOF group (halo) is then given by the largest sphere, centred on the potential minimum, that contains an overdensity 200 times the critical value. Useful properties such as the mass (M_{200c}) and circular velocity (V_{200c}) of the halo can then be derived:

$$M_{200c} = \rho_{200c}(z)\frac{4\pi}{3}R^3_{200c} = \frac{100}{G}H^2(z)R^3_{200c} = \frac{V^2_{200c}}{10GH(z)}, \tag{3.1}$$

where ρ_{200c} is the bounding radius of the overdensity and ρ_{200c} is the Universe's critical density multiplied by 200. The SUBFIND (Springel, White, et al., 2001) algorithm is then deployed to find self-bound substructure in the FOF group that contains more than 20 particles.

As Py-Galaxies uses the merger graph structure, the MEGA method (Roper, Thomas, and Srisawat, 2020) is used to extract haloes. This method again links particles that have a separation less than 20% of the mean interpartcle spacing; however, haloes then need to meet the criteria of a binding energy check,

$$E \leq 0, \tag{3.2}$$

where,

$$\begin{aligned} E &= \mathrm{KE} + \mathrm{GE} \\ &= \sum_{i=1}^{N_p} \frac{1}{2} m_i |\boldsymbol{v_i} - \langle \boldsymbol{v} \rangle| - \sum_{i=1}^{N_p} \sum_{j=1}^{i-1} \frac{G m_i m_j}{(r_{ij}^2 + s^2)^{\frac{1}{2}}}. \end{aligned} \tag{3.3}$$

This check (Equation 3.3) simply sums over each particle in the halo, $i = 1...N_p$, where m_i is the particle mass, M the total halo mass, $\boldsymbol{v_i}$ the particle velocity, $\boldsymbol{v}$ the mean halo velocity, r_{ij} distance between particle i and j, and s the softening of the simulation. As using a constant linking length paired with a binding energy check can lead to a range of overdensities, a 6-D phase space is created using an adaptive linking length derived from a constant overdensity of 200 times the critical value. The phase space also enables the filtering of 'fluff' around the edges of haloes caused by this algorithm not checking that individual particles are bound to the halo (see Roper, Thomas, and Srisawat (ibid.) for more detail). Haloes with as few particles as 10 can be included within the graph if a later halo relies on its existence.

The SUBFIND (L-Galaxies) and MEGA (Py-Galaxies) algorithms are then applied to, the Millennium I & II simulation catalogues (Springel et al., 2005; Boylan-Kolchin et al., 2009) and the GADGET-2 simulation catalogues (Springel, 2005) respectively to build the merger trees and graphs.

As mentioned in Chapter 1, merger graphs are a more advanced structure allowing for haloes and subhaloes to split apart after they have merged, resulting in smoother mass growth and fewer catastrophic failures. This is an entirely new structure that has not yet

been implemented into a galaxy simulation. Before explaining how they're implemented in the code, some key terms are first needed.

1. Snapshot
    - Simulations can only model discrete time and therefore occur in 'snapshots'. Each snapshot contains information about all haloes and subhaloes that were present at that time.
2. Progenitor
    - A progenitor is a halo in a previous snapshot from which the one in the current snapshot has evolved. Generally, only the direct progenitor is important, which is the halo from the preceding snapshot.
3. Descendent
    - A descendent is a halo in future snapshots from which the one in the current snapshot will evolve into. Again, only the direct descendent is important, which is the halo in the following snapshot.

3.2 Merger Graphs: The Input For Py-Galaxies

Merger graphs are the input for the Py-Galaxies simulation. For the purposes of code development, 975 graphs are stored as separate groups within a Hierarchical Data Format (HDF5) file to ensure both a small size on disk and quick read times. Graphs can contain just haloes, or haloes and subhaloes. In the case of haloes, each graph has 22 datasets stored within the HDF5 group detailing the attributes of each halo. Generally, these graphs are ignored as subhaloes are where galaxy formation occurs. When subhaloes are present within the graph, 47 datasets describe the subhalo and halo attributes. A full list of attributes can be found in Appendix A. All haloes and subhaloes are subsequently given unique identification numbers (IDs) ranging from 0, to $N_{halo/subhalo} - 1$. As opening datasets and groups from an HDF5 file is computationally expensive, attributes are stored as single arrays which can be read in and then indexed using the specific halo's ID. For example, if halo 10's redshift was needed, it can be extracted using the code shown below in "Extract Halo Data".

Extract Halo Data

```
halo_ID = 10
halo_example_redshift = all_halo_redshifts[halo_ID].
```

However, some attributes contain data about halo (or subhalo) progenitors or descendants, for example, the amount of mass in common the current halo has with a descendant halo. In this case, the arrays cannot simply be indexed using the Halo ID as haloes can have a variable number of progenitors/descendants. Instead, the halo ID is used to index and extract two pointers from companion arrays. The first is the start index which contains the array location of where a certain halo's progenitors/descendants begin. The second is then the number of progenitors/descendants a halo has. So for example, if the mass in common between halo 10 and its 3 descendants is needed, the data could be extracted

using the code seen below in "Extract Prog/Desc Data".

Extract Prog/Desc Data

```
halo_ID = 10
halo_desc_start_index = all_halo_desc_IDs[halo_ID]
halo_ndesc = all_halo_ndesc[halo_ID]
halo_desc_cont = all_halo_desc_cont[halo_desc_start_index:
                                    halo_desc_start_index +
                                    halo_ndesc ]
```

The same method is also implemented to store which halo is in which snapshot of the graph. A snapshot or 'generation' ID is stored alongside the number of haloes in the snapshot and its start index. The haloes in a certain snapshot are then extracted in the same manner as the code above. This method of data storage was developed by my supervisor and their PhD student and was seen as the most efficient way to store and access the data. Alongside attribute data for each halo, the HDF5 file also contains useful metadata such as the number of haloes in any given graph, the number of graphs in the file, and the mass of a dark matter particle in $M_{\odot}$ (see Appendix Table A.3 for a full list). In future versions of the code, units may also be included for each dataset to ensure clarity between input and internal units.

3.3 Internal Data Structures for Merger Graphs

Once the HDF5 file is opened and read in using the Python module H5PY, the merger graphs then need to be transformed into a data structure suitable for modelling galaxies halo by halo. Two versions of the Py-Galaxies code were developed to contrast and compare two different data structures, the first using Python classes and lists, and the second using NumPY structured arrays.

3.3.1 Class-based Py-Galaxies

The class based version utilises three main classes to store and structure data from the input graphs: `GraphProperties`, `HaloProperties`, and `SubhaloProperties`. Initially, the driver script loops over each graph in the HDF5 file and opens the datasets within. Each array is then stored as an attribute of the `GraphProperties` class. A minimal example is shown below in "Graph Properties Class" (see Appendix Section B.1 for full code).

Graph Properties Class

```
class GraphProperties:
    def __init__(self, graph_ID, open_HDF_file, part_mass):
        open_HDF_group = open_HDF_file[str(graph_ID)]
        self.graph_ID = graph_ID
        self.ndesc = open_HDF_group["ndesc"][:]
        self.mass  = open_HDF_group["nparts"][:] * part_mass
        self.nprog = open_HDF_group["nprog"][:]
```

Once these attributes have been assigned, the data is stored in a memory location that will not change for the rest of the iteration. If any of the arrays stored within these attributes are sliced to make a new array, only references to the original memory location will be created, consuming less memory than a copy.

Now the data has been read in and stored in memory, it needs to be split up into individual haloes and subhaloes. To do this, the `HaloProperties` and `SubhaloProperties` classes are used. These classes contain the intrinsic properties from the HDF5 file alongside any modelled properties that the code will introduce, e.g. stellar mass. A minimal example of the halo class' constructor is shown below in "Halo Properties". A full and commented version for both haloes and subhaloes is included within Appendix Sections B.2 & B.3.

Halo Properties

```
class HaloProperties:
    def __init__(self, graph_ID, halo_ID, graph_properties):

    self.graph_ID = graph_ID
    self.halo_ID = halo_ID
    self.catalog_ID = graph_properties.halo_catalog_halo_ids[halo_ID]
    self.mass = graph_properties.mass[halo_ID]
    self.prog_ids = graph_properties.direct_prog_ids[
        self.prog_start : self.prog_end
    ]
```

List comprehensions are used to initialise these classes for all the haloes/subhaloes, resulting in two lists, indexed by halo or subhalo ID. The driver script can then begin to loop over the snapshots, extracting the IDs of which haloes/subhaloes are in it. Those ID's can then be used to extract halo and subhalo information from the list.

3.3.2 Array-based Py-Galaxies

Instead of a class based data structure, the array-based version takes advantage of NumPy structured arrays. Each graph is again read in from the HDF5 file, but this time, data is stored within five separate 2-D structured arrays. Five are necessary as an array must be of a fixed length and there are five unique lengths that the datasets can be. Any halo properties or subhalo properties are of length N_{halo} or N_{subhalo} respectively. The descendent and progenitor information for haloes and subhaloes then have lengths $N_{\text{prog/desc}}$ and $N_{\text{sub\_prog/desc}}$ with the final length coming from the snapshot information. The code automatically handles the grouping of the different length properties, and uses the HDF5 dataset key and variable type to create the custom NumPy dtype and column label for the structured array. Everything is determined by the contents of the input file meaning that new properties could be added to the HDF5 file and the code would automatically adapt. Within the class based layout, the additional properties (attributes) would need to be added by hand to the class. An additional two arrays are then created to hold any properties that may be added over the course of the model, e.g. stellar mass, using a list of properties and types defined at run time to label the columns and dtypes. This results in 7 arrays where the row number is the halo or subhalo ID and the column is the property. It can be accessed with the simple code shown below in "Extract Halo Property (Array

Based)".

Extract Halo Data (Array Based)

```
halo_ID = 10
halo_property = "redshift"
halo_example_redshift = halo_data[halo_property][halo_ID]
```

The array is indexed as column and then row throughout the code as this is the most efficient way to do it (see Appendix Figure C.1). Indexing row first will return an array with a variable dtype which then gets sliced, whereas indexing the column first will limit the array to a single dtype.

3.4 How the graph is simulated

The haloes and subhaloes in each graph belong to a certain snapshot. The driver script loops over each of the 61 snapshots chronologically and extracts the IDs of any haloes within. Subsequently, the driver script then loops over the haloes and applies the routines that will be explained in the following chapters. Once this processing is done, any baryonic material in the halo must be passed down to its descendants. Differing from merger trees, haloes in merger graphs share baryons between descendants in proportion to their mass in common. In the situation were mass goes missing between progenitors and descendants, say, if one of the descendent haloes was less than 20 particles, then the remaining contribution is also passed down in proportion. The example calculation is shown in "Halo Proportional Contributions" below.

Halo Proportional Contributions

```
# Calculate mass in common with descendants.
contribution_fractions = descendent_masses  / current_halo_mass
# Calculate any extra contributions that need to be accounted for.
extra_contributions = (current_halo_mass - np.sum(descendent_masses)) *
↪  (contribution_fractions / sum(contribution_fractions))
# Add the extra contributions on to the original ones.
proportional_contribution = (descendent_masses + extra_contributions) /
↪  current_halo_mass
# An example proportional_contribution = [0.4, 0.2, 0.4]
# Give descendants the correct proportion of baryons.
descendants_baryons = current_halo_baryons * proportional_contribution.
```

However, subhaloes are a much denser structure than haloes (Springel, Wang, et al., 2008) meaning they are unable to split apart and consequently pass all their baryons on to a single descendent which has the most particles in common, the same as in a merger tree. If the subhalo has no descendent, all of it's baryons are given back to its host halo and classified as intracluster light. To control these processes, lists of properties that are to be

passed down to descendants are looped over, with the class-based code using getters and setters, and the array-based code simply indexing the structured arrays.

When using merger graphs, the central subhalo needs also needs to be selected. This is done using a 6-D phase space where velocity and position are added together to ensure that a halo which may simply be passing by is not selected as the central one. If the subhalo is then sufficiently close to the centre of the halo, it is deemed to be the central subhalo. Currently, Py-Galaxies just selects the closest possible subhalo where its total phase space distance is,

$$U_{\rm total} = \frac{\sqrt{\sum_{i=1}^{3}\left(p_{\rm halo,\ i} - p_{\rm subhalo,\ i}\right)^2}}{r_{\rm halo} + r_{\rm subhalo}} + \frac{\sqrt{\sum_{i=1}^{3}\left(v_{\rm halo,\ i} - v_{\rm subhalo,\ i}\right)^2}}{v_{\rm rms,halo} + v_{\rm rms,subhalo}}, \tag{3.4}$$

where p are the positions of the halo and subhalo, and v are the velocities. The central subhalo ID is then stored as an attribute within the halo class, or as a variable in the structured array.

3.5 The Input Haloes

Figure 3.2 displays the number of haloes present at different masses for three different snapshots in the model. As expected, the largest haloes are present at the current day, $z = 0$, due to the haloes having time to grow and evolve throughout the graph. The same is also true for subhaloes; however, proportionally there are fewer subhaloes at high redshifts demonstrating that it takes time for substructure to form within haloes.

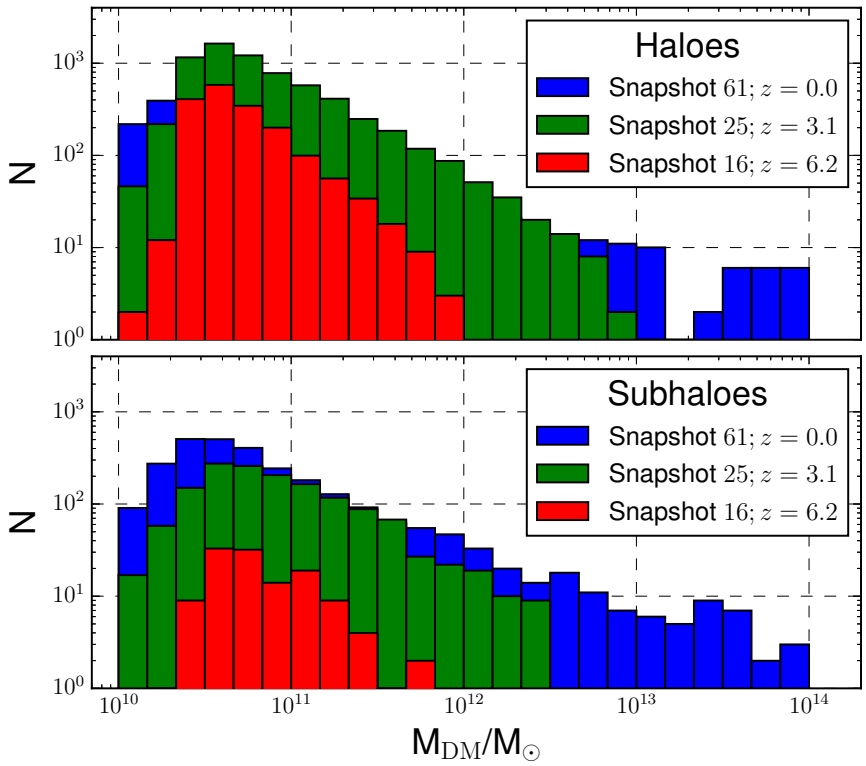

Figure 3.2: Distributions of halo and subhalo dark matter masses for different redshifts.

Chapter 4

Baryonic Infall and Reionization

4.1 Background Information

As the Universe continued to expand and cool, Hydrogen began to form through the recombination of electrons and protons. This cooling enabled primordial gas to begin falling into the gravitational potential wells of dark matter haloes. This means every halo has an associated mass of baryons equal to the cosmic mean. From Plank cosmology, the mean baryon fraction is calculated as $f_b^{cos} = 0.155$ though in reality, galaxy clusters exhibit a lower baryon to dark matter ratio (Guo et al., 2011). This is especially true for dwarf galaxies where the UV background heats pregalactic gas preventing condensation within the halo (Couchman and Rees, 1986; Efstathiou, 1992). This effect, called reionization, can be accounted for with a simple model proposed by Gnedin (2000) that adapts the baryon fraction based on a characteristic halo mass $M_F(z)$:

$$f_b(z, M_{200c}) = f_b^{cos}\left(1 + (2^{\frac{\alpha}{3}} - 1)\left[\frac{M_{200c}}{M_F(z)}\right]^{-\alpha}\right)^{-3/\alpha}. \tag{4.1}$$

Here, M_{200c} is our halo mass, α controls how rapidly f_b^{cos} decreases for low mass haloes,

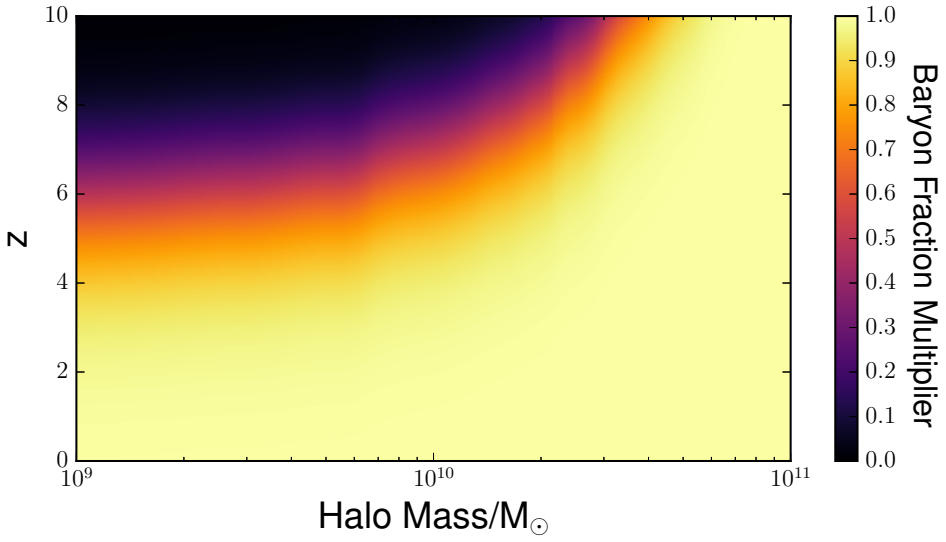

Figure 4.1: The baryon fraction multiplier (within the large brackets of equation 4.1) as a function of halo mass and redshift. The less massive the halo and the higher its redshift, the more reduction there is in the baryon fraction due to UV photo-heating.

and $M_F(z)$ controls the mass at which this decrease occurs for a given redshift. Okamoto, Gao, and Theuns (2008) found $\alpha = 2$ to provide the best fit and is therefore adopted here. They also provide a table of $M_F(z)$ ranging from $\sim 6.5 \times 10^9$ $M_\odot$ at z=0 to $\sim 10^7$ $M_\odot$ at $z \sim 8$. This means the baryon fraction will be equal to the cosmic mean for $M_{200c} \gg M_F$ but tend to zero $\propto (M_{200c}/M_F)^3$ for $M_{200c} \ll M_F$. Figure 4.1 displays the baryon fraction multiplier (within the large brackets of Equation 4.1) as a function of halo mass and redshift. As can be seen, the lower the halo mass and the higher the redshift, the more reduction there is in the baryon fraction.

4.2 Py-Galaxies Implementation

To implement the infall and reionization process, Py-Galaxies follows the same recipe as L-Galaxies. At each snapshot in the simulation, every halo is given an amount of baryons equal to the adjusted cosmic mean, which is simply the dark matter mass of the halo multiplied by the adjusted baryon fraction given in Equation 4.1. However, this process does not take place if the halo has already inherited baryons from a progenitor and surpasses this threshold. If the halo has inherited less, the infalling baryons top the halo's total baryons up to the cosmic mean. All baryons added to the halo through the process of infalling are added to the halo's hot gas reservoir. This is the first property of interest that the halo is given.

The calculations for Equation 4.1 are completed using a C routine from L-Galaxies. A table of $M_F(z)$ is read in with the global parameters runtime and the halo mass and redshift passed as arguments to the function. If halo masses found in the graph are above or below that found in the table, the minimum or maximum value is taken from the table. For masses in between, the table is linearly interpolated to get a result. The calculation is then carried out and the result returned to Python. This process is carried out with the same code structure as demonstrated in Section 2.3.

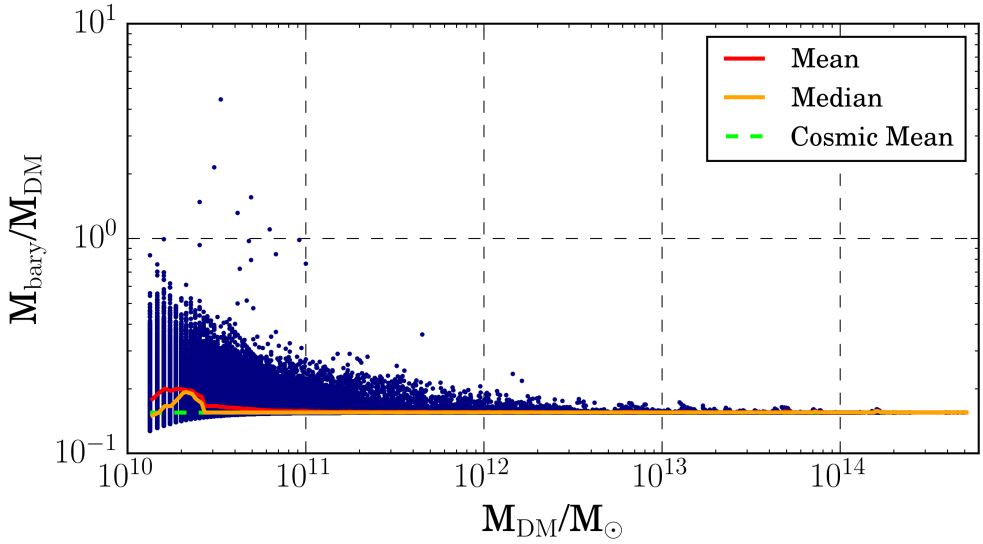

Figure 4.2: The baryon mass to dark matter mass ratio for each halo, plotted against their dark matter masses. The dashed green line denotes the cosmic mean fraction of 0.155, with the red and yellow lines denoting the population mean and median.

Figure 4.2 displays the baryon fraction of each halo plotted against its dark matter mass. These results were created when further processes, such as star formation, had been implemented. As can be seen, the vast majority of haloes are close to the cosmic mean (plotted in green). For the least massive haloes, a number fall below the cosmic mean due to reionization process, but many, far exceed the cosmic average. This is likely due to a small halo inheriting a large amount of baryons from a more massive halo.

Chapter 5

Cooling of Hot Gas

5.1 Background Information

The process of infalling converts vast amounts of potential energy into kinetic energy, rapidly heating the gas in a process called accretion shock. Due to this shock, hot gas reservoirs first form in the halo and then cool according to two separate regimes. In haloes of order $10^{10} - 10^{12}$ $M_\odot$ the gas has a free-fall time longer than that of it's cooling time and therefore undergoes its accretion shock close to the centre of the halo. This means that the shock is radiative, the cooling efficient, and the gas can collapse unimpeded. However, for larger haloes, the free-fall time is shorter than that of the cooling time and the accretion shock occurs at roughly the virial radius. This causes the shock to be non-radiative, the cooling inefficient, and the collapse arrested, creating a quasi-static hot gas atmosphere heated to the virial temperature (Rees and Ostriker, 1977). Over time, this atmosphere then cools and condenses onto the central subhalo through cooling flows (Birnboim and Dekel, 2003).

To differentiate between these two cooling regimes, SAMs utilise the difference in radius at which accretion shock occurs. Normally adopted as the cooling radius ($r_{\rm cool}$), it can be calculated as the point where the the cooling time of the gas is equal to the dynamical time of the halo. To calculate the dynamical time, we take the ratio of R_{200c} and V_{200c} (De Lucia et al., 2006) from equation 3.1:

$$t_{\rm dyn} = \frac{R_{200c}}{V_{200c}} = 0.1H(z)^{-1} = 0.1H_0^{-1}\left(\Omega_m(1+z)^3 + \Omega_\Lambda\right)^{-\frac{1}{2}}. \tag{5.1}$$

The cooling time however, is less simple to calculate. We need to model the intrinsic cooling processes of the gas, in addition to knowing its temperature and density. The main processes responsible for cooling are the two-body radiative reactions listed in Table 5.1. Each reaction requires free electrons and assumes that the gas is both in equilibrium and optically thin, neglecting radiative ionization effects.

Process	Reaction
Bremsstrahlung	$X^+ + e^- \Rightarrow X^+ + e^- + \gamma$
Recombination	$X^+ + e^- \Rightarrow X^+ + \gamma$
Collisional Ionisation	$X \quad + e^- \Rightarrow X^+ + 2e^-$
Collisional Excitation	$X \quad + e^- \Rightarrow X' + e^- \Rightarrow X + e^- + \gamma$

Table 5.1: Radiative processes responsible for cooling the quasi-static hot atmosphere.

These reactions can then be combined into a model called the Collisional Ionization Equi-

librium (CIE) cooling function (denoted Λ; Sutherland and Dopita, 1993). It is only dependent upon the temperature and metalicity of the gas and returns a cooling rate in ergs cm^3 s^{-1}, where metalicity is defined as the fraction of gas that is not hydrogen or helium,

$$Z = \sum_{i>\mathrm{He}} \frac{m_i}{M_{\mathrm{tot}}}. \tag{5.2}$$

Examples of cooling rates for gases of different metallicities have been plotted in Figure 5.1.

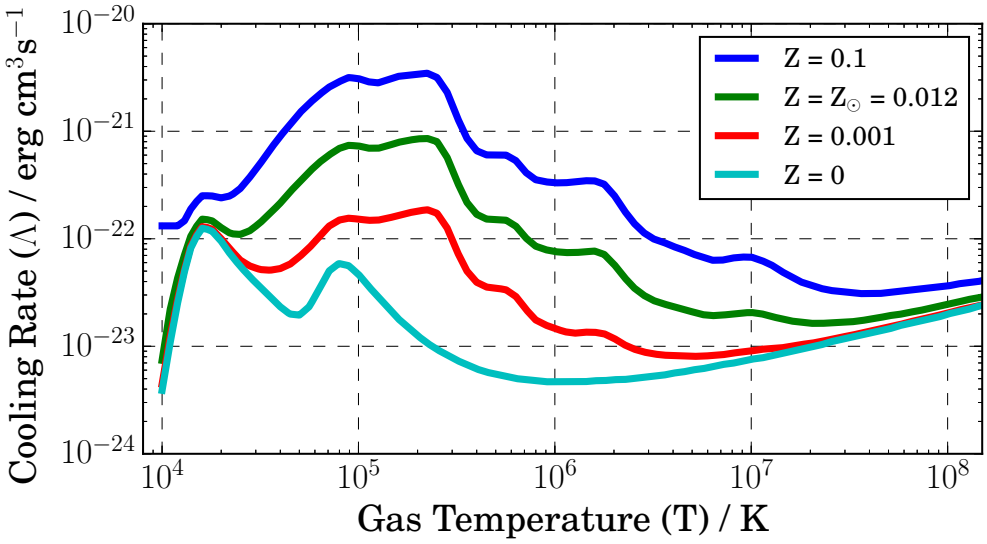

Figure 5.1: The CIE cooling rate plotted for gases of different metallicities.

The temperature of the hot gas can then obtained from the virial theorem assuming a truncated, singular isothermal sphere (SIS, $\zeta = 3/2$) filled with a monotonic gas ($\gamma = 5/3$):

$$0 = 2K + V = \frac{1}{\gamma - 1}\frac{k_B T_{200c} M_{\mathrm{hot}}}{\mu m_H} + \zeta \frac{G M_{\mathrm{hot}} M_{200c}}{r_{200c}} = \frac{k_B T_{200c}}{\mu m_H} + \frac{1}{2}\frac{G M_{200c}}{r_{200c}}. \tag{5.3}$$

Then rearrange for temperature and simplify using Equation 3.1,

$$T_{200c} = \frac{\mu m_H}{2 k_B} V_{200c}^2, \tag{5.4}$$

where μ is the mean molecular mass (≈ 0.59), m_H the mass of hydrogen, and k_B the Boltzmann constant. Here, L-galaxies and Py-Galaxies diverge, using different methods to calculate the density of the gas and therefore the cooling time.

5.1.1 L-Galaxies

L-Galaxies continues to use a SIS enabling the hot gas density to be described as,

$$\rho_{\mathrm{hot}}(r) = \frac{M_{\mathrm{hot}}}{4\pi R_{200c} r^2}. \tag{5.5}$$

leading to the definition of cooling time to be a simple ratio between the thermal energy of the gas and its cooling rate per unit volume,

$$t_{\mathrm{cool}} = \frac{\epsilon}{\rho_{\mathrm{hot}}(r)\Lambda(T_{\mathrm{hot}}, Z_{\mathrm{hot}})} = \frac{6\pi R_{200c} T_{200c} r^2 \mu m_H k_B}{M_{\mathrm{hot}}\Lambda(T_{\mathrm{hot}}, Z_{\mathrm{hot}})}. \tag{5.6}$$

Finally, this can be equated to the halo's dynamical time (Equation 5.1) and the cooling radius calculated,

$$r_{\rm cool} = \left(\frac{t_{\rm dyn} M_{\rm hot} \Lambda(T_{\rm hot}, Z_{\rm hot})}{6\pi R_{\rm 200c} T_{\rm 200c} \mu m_H k_B} \right)^{\frac{1}{2}}. \tag{5.7}$$

The two cooling regimes are then modelled as follows. When $r_{\rm cool} > R_{\rm 200c}$, gas is accreted in cold mode and rapidly infalls onto the central subhalo in the halo dynamical time (Guo et al., 2011):

$$\frac{dM_{\rm cool}}{dt} = \frac{M_{\rm hot}}{t_{\rm dyn}}. \tag{5.8}$$

However, if $r_{\rm cool} < R_{\rm 200c}$, gas is accreted onto the central subhalo in hot mode via cooling flows:

$$\frac{dM_{\rm cool}}{dt} = M_{\rm hot} \frac{r_{\rm cool}}{R_{\rm 200c} t_{\rm dyn}}. \tag{5.9}$$

This derivation also demonstrates that cooling is more efficient at high redshifts as, working backwards, it can be found that $R_{\rm 200c} \propto z$ but $r_{\rm cool} \propto z^{\frac{5}{4}}$.

Though this method is standard across several SAMs, it has drawbacks. Firstly, the density is infinite at the centre of the halo ($r = 0$, Equation 5.5). Secondly, gas outside the cooling radius has no effect on the rate of cooling, though in reality it would. Finally, the model cools too much gas when using large time steps.

5.1.2 Py-Galaxies

To solve these issues, Fournier, Thomas, and Henriques (In Preparation) have developed a new model that utilises an isothermal beta profile. This allows for a finite density at the centre of the halo, in turn, providing a finite luminosity from which to derive the rate of cooling gas. Within this method, there is no need for a cooling radius as the luminosity can be calculated out to infinity, so the two regimes are separated using the cooling time. The profile is presented as,

$$\rho_g = \frac{\rho_0}{1 + (\frac{r}{a})^2}, \tag{5.10}$$

where a is the constant core radius, r is the radius, and ρ_0 is the initial density. The density profile can then be integrated out to the halo radius $R_{\rm 200c}$ to obtain the mass of hot gas within:

$$\begin{aligned} M_g = 4\pi^2 \rho_0 \int_0^{R_{\rm 200c}} \frac{r^2}{1 + (\frac{r}{a})^2} \, {\rm d}r &= 4\pi \rho_0 a^3 \left[\frac{R_{\rm 200c}}{a} - \arctan\left(\frac{R_{\rm 200c}}{a} \right) \right] \\ &= 4\pi \rho_0 a^3 \big(Y - \arctan(Y) \big) \end{aligned} \tag{5.11}$$

The luminosity of the hot gas can also be derived,

$$\begin{aligned} L = \int_0^\infty \Lambda(T_{\rm hot}, Z_{\rm hot}) 4\pi n_g^2 r^2 \, {\rm d}r &= \frac{\Lambda(T_{\rm hot}, Z_{\rm hot}) 4\pi \rho_0^2}{\mu^2 m_H^2} \int_0^\infty \frac{r^2}{\left(1 + (r/a)^2 \right)^2} \, {\rm d}r \\ &= \frac{\Lambda(T_{\rm hot}, Z_{\rm hot}) \rho_0^2 \pi^2 a^3}{\mu^2 m_H^2} \end{aligned} \tag{5.12}$$

The cooling time can then be calculated as the time taken for the gas to radiate away it's energy. As the gas stays in equilibrium with its surroundings, pressure causes work to be done on it, meaning that specific enthalpy is more suitable. The cooling time, adapted from Thomas (1988) is then given as,

$$t_{\rm cool} = \frac{H \times M_g}{L}, \tag{5.13}$$

where H is the specific enthalpy,

$$H = \frac{5k_bT}{2\mu M_H}. \tag{5.14}$$

$t_{\rm cool}$ is therefore,

$$t_{\rm cool} = \frac{\frac{5k_bT}{2\mu M_H^2} \times M_g}{\frac{\Lambda(T_{\rm hot},Z_{\rm hot})\rho_0^2\pi^2a^3}{\mu^2 m_H^2}} = \frac{5k_bT\mu M_H M_g}{2\rho_0^2\Lambda(T_{\rm hot},Z_{\rm hot})\pi^2a^3}. \tag{5.15}$$

Now eliminating ρ_0 using Equation 5.11,

$$t_{\rm cool} = \frac{40a^3k_BT\mu M_H(Y-\arctan Y)^2}{M_g\Lambda(T_{\rm hot},Z_{\rm hot})} = \frac{40R_{200c}^3k_BT\mu M_H}{M_g\Lambda(T_{\rm hot},Z_{\rm hot})}\frac{(Y-\arctan Y)^2}{Y^3}, \tag{5.16}$$

which is very similar to the L-Galaxies cooling time in Equation 5.6. In the same approach as the L-Galaxies, the cooling is separated into two regimes, catastrophic infall when the cooling time is less than or equal to the halo's dynamical time, or cooling flows when the cooling time is longer.

5.2 Py-Galaxies Implementation

Using Python, Py-Galaxies initially calculates R_{200c} using equation 3.1. Consequently, Equations 5.1 and 5.4 are used to calculate V_{200c} and T_{200c}. T_{200c} is then the taken as the temperature of the entire reservoir of hot gas. Py-Galaxies currently assumes the metalicity of the hot gas to be constant throughout the simulation at a value of 0.0001 (primordial). In reality, star formation would lead to the production of more metals, which in turn would speed up cooling. A C-routine from L-Galaxies containing the CIE cooling function described in Section 5.1, is then called. It takes the metalicity and temperature of the hot gas as a parameter and returns the cooling rate. The function makes use of the original data tables from Sutherland and Dopita, 1993, linearly interpolating between the data points when needed. If the range of the tables are exceeded, the minimum/maximum is used (as seen at low temperatures in the 0.1 metalicity line within Figure 5.1). The mass of gas cooled is then calculated using Python, and subtracted from the hot gas reservoir. Traditionally, L-Galaxies uses 'mini' time steps, around one twentieth of the time between snapshots to prevent over-cooling; however, the new cooling method implemented in Py-Galaxies cools the same amount of gas irrespective of the time step. All gas cooled is given to the halo's central subhalo, becoming the first property of interest for subhaloes consequently, star formation can begin.

Chapter 6

Star Formation & Feedback

6.1 Background Information

6.1.1 Star Formation

As the gas continues to cool and condense onto the main subhalo, a disk of stars form due to the angular momenta of the infalling gas. Within L-Galaxies, the size of this disk is carefully modelled taking into account star formation, accretion, and halo merger events as in Guo et al. (2011). However, due to limited scope, Py-Galaxies will assume the disk radius is 10 times smaller than the halo radius R_{200c}. Once the gas has fallen onto this disk and cooled sufficiently, stars are able to form. Both L-Galaxies and Py-Galaxies model star formation as a simplified version of the Kennicutt-Schmidt law (Kennicutt, 1998), whereby stars only form efficiently in areas when their surface mass density is greater than a critical value. Below this critical value, star formation rates greatly drop due to low gas densities causing gravitational instability (Toomre, 1964). Kauffmann (1996) approximates this critical surface density as,

$$\Sigma_{\text{crit}}(R) = 12 \times \left(\frac{V_{200c}}{200 \text{ kms}^{-1}}\right)\left(\frac{R}{10 \text{ kpc}}\right) \text{M}_{\odot}\text{pc}^{-2}. \tag{6.1}$$

Integrating this out to the radius of the cold gas ($0.1R_{200c}$) yields,

$$M_{\text{crit}} = 3.8 \times 10^{8}\left(\frac{V_{200c}}{200 \text{ kms}^{-1}}\right)\left(\frac{R_{200c}}{10 \text{ kpc}}\right) \text{M}_{\odot}. \tag{6.2}$$

Using this critical value, stars are then formed using the cold gas within the disk with the star formation rate given as,

$$\frac{\text{d}M_*}{\text{dt}} = \alpha_{SF}\frac{(M_{\text{cool}} - M_{\text{crit}})}{t_{\text{dyn, disk}}}, \tag{6.3}$$

where $t_{\text{dyn, disk}} = {}^{R_{200c}}/_{10V_{200c}}$ and α_{SF} is a constant that controls the efficiency of star formation, in this case, 0.025.

6.1.2 Supernova feedback

Supernovae (SN) are incredibly powerful phenomena that occur when either, a massive star runs out of light elements to fuse, leaving an iron and nickel core to collapse; or, when a white dwarf in a binary system accretes mass past its critical limit. In seconds, they can release 10^{44} J of energy (Reinecke, Hillebrandt, and Niemeyer, 2002), permanently altering the formation of galaxies through the heating and expulsion of gas. These effects were first shown by Larson (1974) who demonstrated, particularly for small galaxies, that SN

explosions cause early gas loss which, in turn, lowers metal abundances below average and leads to less dense galaxies. Subsequent studies have explored this in more detail (Benson et al., 2003; Dekel and Silk, 1986) and established SN feedback as an essential competent for modelling galaxy formation. Large stars also add to this feedback as stellar winds can also reheat and eject gas from the system.

L-Galaxies uses SN energy to reheat a portion of cooled gas into hot gas, and eject a portion of hot gas into an external reservoir. Py-Galaxies is yet to have this external reservoir implemented, so for now simply reheats the cooled gas into hot gas. The mass of reheated gas is assumed to be directly proportional to the mass of stars formed,

$$\Delta M_{\text{reheat}} = \epsilon_{\text{disk}} \Delta M_* \tag{6.4}$$

where ϵ_{disk} is,

$$\epsilon_{\text{disk}} = \epsilon \times \left[0.5 + \left(\frac{V_{200c}}{V_{\text{reheat}}}\right)^{-\beta}\right], \tag{6.5}$$

and $\epsilon = 2.6$ (Martin, 1999), $V_{\text{reheat}} = 200$ kms^{-1} (Guo et al., 2011), and $\beta = 0.72$ (Henriques et al., 2019).

6.1.3 An Alternate Model Of Star Formation

As a secondary method to compare results with, a model based on empirical observations was also implemented into the Py-Galaxies code. Behroozi, Wechsler, and Conroy (2013) offer a single formula, derived from real-world observations, to relate the redshift and dark matter mass of a halo, to the expected stellar mass within. The equation is as follows,

$$\log_{10}\left(M_*(M_h)\right) = \log_{10}(\epsilon M_1) + f\left(\log_{10}\left(\frac{M_h}{M_1}\right)\right) - f(0), \tag{6.6}$$

where M_h is the dark matter mass of the halo, M_* the stellar mass, and $f(x)$ the function,

$$f(x) = -\log_{10}(10^{\alpha x} + 1) + \delta \frac{(\log_{10}(1 + \exp(x)))^{\gamma}}{1 + \exp(10^{-x})}. \tag{6.7}$$

The parameters are then,

$$\nu = \exp(-4a^2), \tag{6.8}$$

$$\log_{10}(\epsilon) = -1.777 + (-0.006(a-1))\nu - 0.119(a-1), \tag{6.9}$$

$$\log_{10}(M_1) = 11.514 + (-1.793(a-1) + (-0.251)z)\nu, \tag{6.10}$$

$$\alpha = -1.412 + (0.731(a-1))\nu, \tag{6.11}$$

$$\delta = 3.508 + (2.608(a-1)) - 0.043z)\nu, \tag{6.12}$$

$$\gamma = 0.316 + (1.319(a-1) + 0.279z)\nu, \tag{6.13}$$

where z is the redshift, a the scale factor $(1+z)^{-1}$, and M_1 a characteristic halo mass which separates two regimes. For $M_h \ll M_1$ the overall function is a power law with the slope $-\alpha$, but for $M_h \gg M_1$, the function is a sub-power law with the index γ. The relationship has been plotted for various redshifts in Figure 6.1.

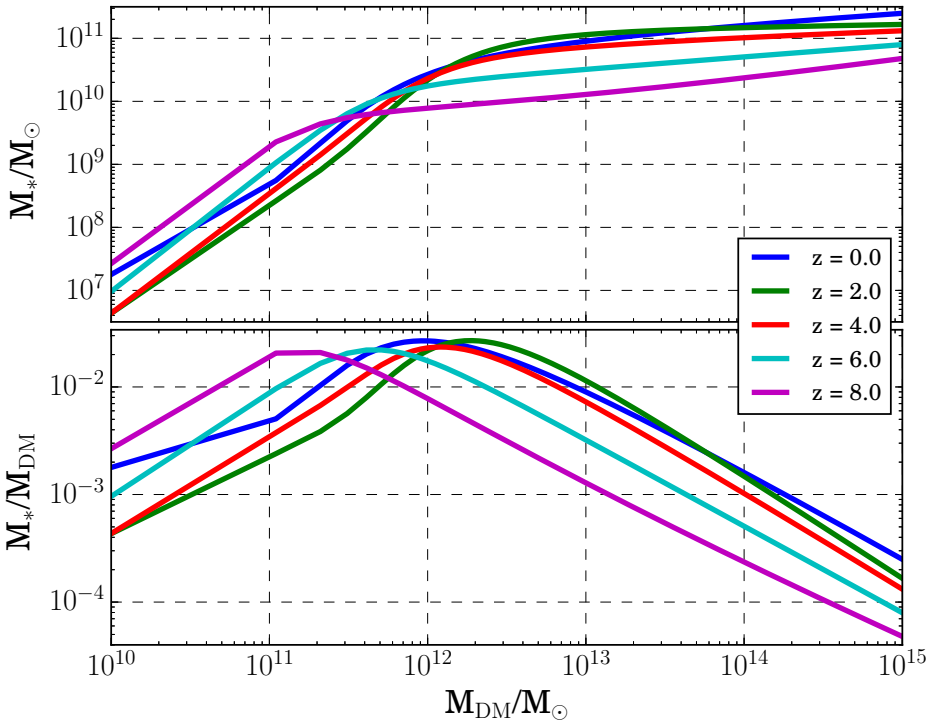

Figure 6.1: Behroozi, Wechsler, and Conroy (2013)'s relationship between a halo's dark matter mass and its stellar mass. The top plot shows the absolute stellar mass plotted against the halo's dark matter mass for various redshifts. The bottom plot then shows stellar mass to dark matter mass ratio for various redshifts.

6.2 Py-Galaxies implementation

The methods that perform these calculations belong to the `SubhaloProperties` class as both cold gas and stellar mass are properties of the subhalo. C-routines, adapted from L-Galaxies are called using the halo's V_{200c}, R_{200c} as parameters, alongside the subhalo's mass of cold gas and the time difference between the current and previous snapshot. The C-routine then returns the mass of stars formed and the mass that is reheated.

6.3 Results

Once each halo had been processed and the data saved, the modelled stellar mass of each halo accross the 975 graphs was plotted against it's dark matter mass, as seen in Figure 6.2. It's clear to see that the stellar mass is following an approximately linear relationship with the dark matter halo mass. As expected, lower mass haloes deviated from this linear relationship due to the large effect of SN feedback. To perceive how well the simulation is doing, the modelled stellar mass can be directly compared to the Behroozi stellar mass. Figure 6.3 shows this comparison for different redshifts. The brown dashed line indicates where the modelled mass would be equal to the Behroozi mass. As the Behroozi mass is an average, an even distribution around this line would be the perfect result. However, it can clearly be seen that the vast majority of haloes have a stellar mass above average. The coloured lines represent the average for a specific redshift regime and show that the most distant subhaloes are close to the expected average. However, the less distant the haloes, the further from the average they become. There are two reasons for this; the first, Activate

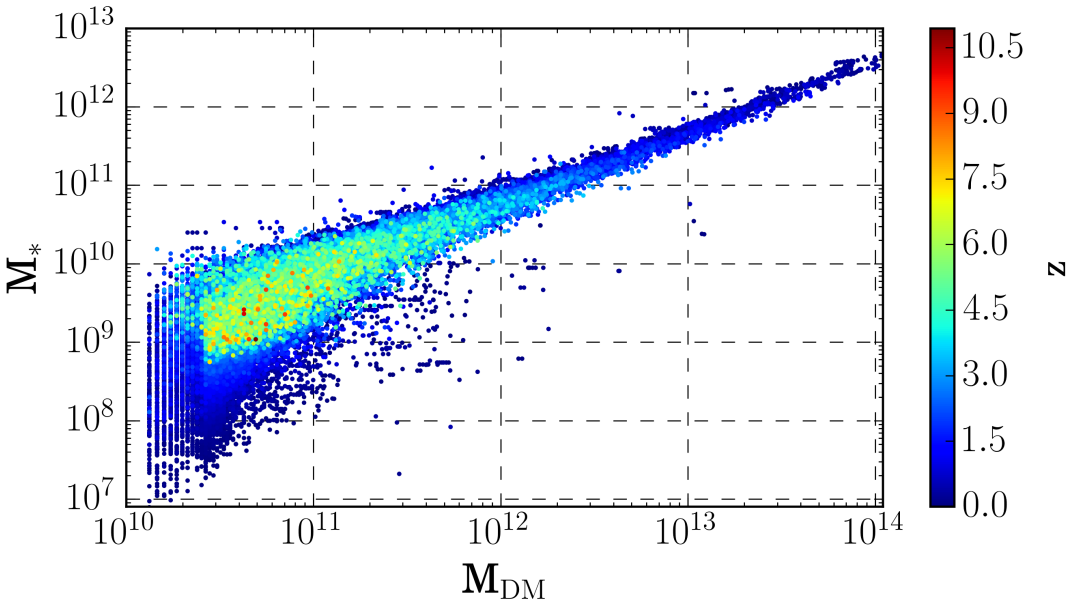

Figure 6.2: The stellar mass of each halo plotted against the dark matter mass of each halo. The colour of the scatter point denotes it's redshift.

Galactic Nuclei (AGN) feedback has not yet been implemented within L-Galaxies. AGNs essentially bring a stop to galaxy formation in massive haloes by heating large amounts of gas (Croton et al., 2006). This explains why the modelled stellar mass peaks at almost 10x that of the Behroozi stellar mass in Figure 6.3 and also why the modelled stellar mass in Figure 6.2 continues on a linear trajectory instead of being quenched as in the Behroozi formula in Figure 6.1. Secondly, ejected gas has not yet been implemented into Py-Galaxies. This means that when the cold gas is reheated, it can be cooled back down again in the next time step. If gas was ejected into an external reservoir, it would take much longer to cool down and therefore leave less available cool gas to form stars. Using the amount of stellar mass formed and the time between snapshots, it is also possible to derive the star formation rate (SFR, simple division). Figure 6.4 shows the SFR of each halo plotted against both its stellar mass and its dark matter mass, with coloured lines denoting the average SFR for different redshift regimes. Generally, the SFR is a little higher than it should be, which is to be expected as the raw amount of stars formed is above average. The most distant galaxies form stars at a faster rate than that of the closest galaxies which is as it should be. Generally, Figure 6.4 shows that the model is working as expected.

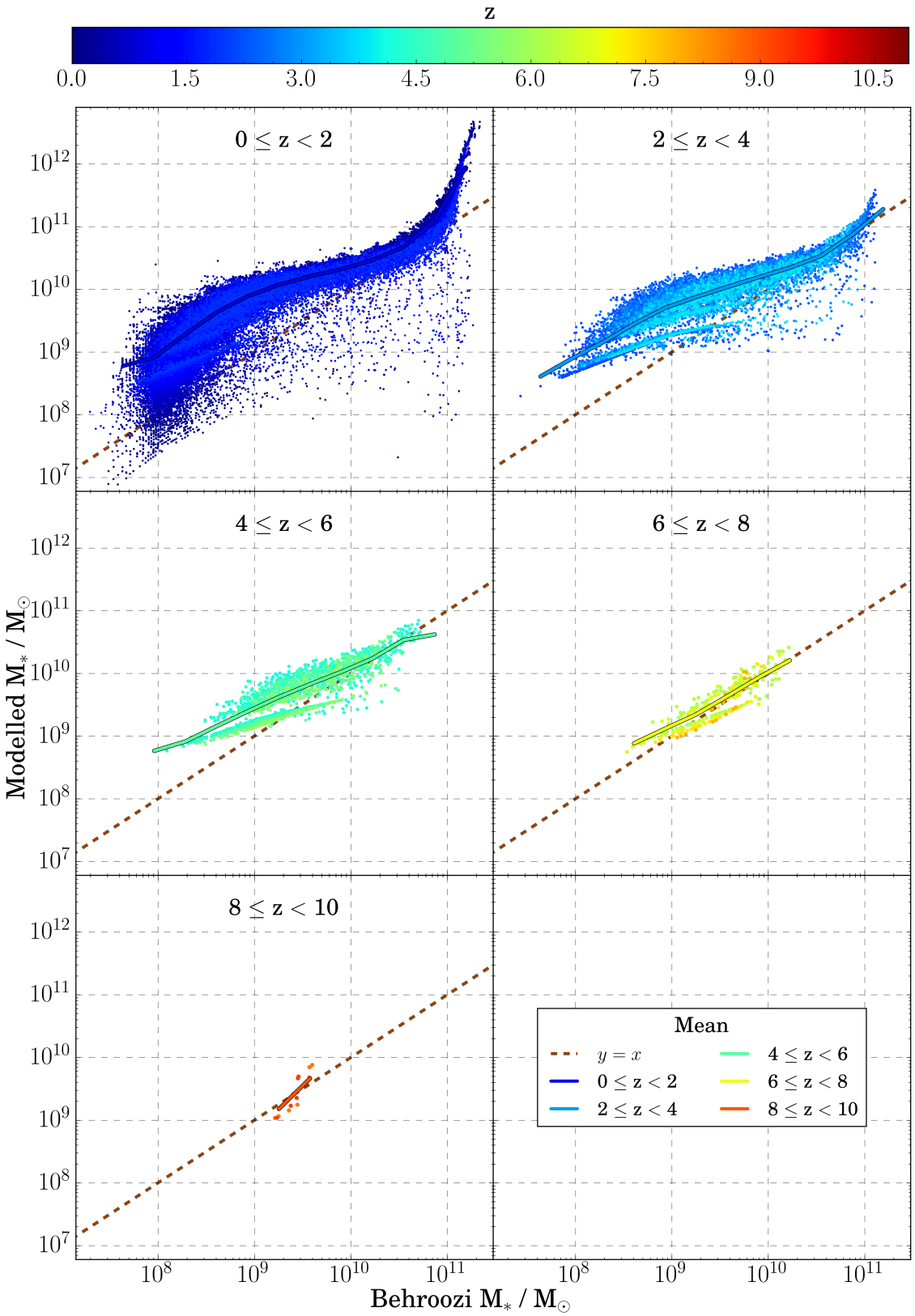

Figure 6.3: Modelled stellar mass plotted against the Behroozi stellar mass for each halo, separated into five different redshift regimes. The brown dashed line denotes where they would be equal and the solid coloured lines denote the average modelled stellar mass for the different redshift regimes.

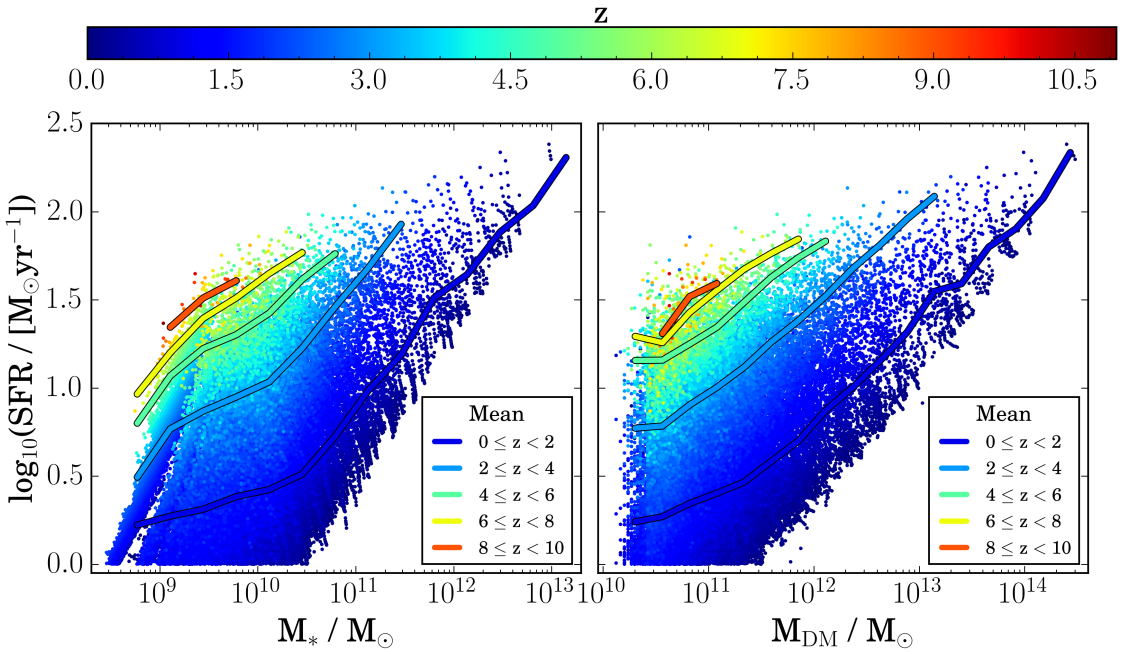

Figure 6.4: The star formation rate (SFR) for each halo plotted against it's stellar mass (left) and dark matter halo mass (right). Coloured lines denote the average SFR for different redshift regimes.

Chapter 7

Model Overview & Output

7.1 Overview

Starting with a merger graph as the chassis for the model, Py-Galaxies builds up the picture of galaxy formation step-by-step. Starting with baryonic infall and reionization, each dark matter halo in the merger graph is assigned a mass of hot gas equal to the universal mean. Over time, cooling begins to take place and the hot gas accretes onto a central subhalo within the main halo. After continuing to condense and cool, the cool gas starts to form stars which in turn reheat fragments of the cold gas. The full high-level summary of how Py-Galaxies does this, while combining both Python and C-routines has been displayed as a flowchart in Figure 7.1.

7.2 Output format

As the model runs, haloes and subhaloes gather different properties such as the mass of hot gas, the mass of cold gas, and stellar mass. As these properties are of interest, they need to be saved to disk, and to do this, HDF5 was chosen as the favoured format. Within the HDF5 file, data for haloes and subhaloes taken from all graphs were stored in two separate tables. To identify each halo, it's graph number, snapshot number, and halo ID were all stored. If the halo had a central subhalo, that subhalo's ID was also stored. The same was true for subhaloes, where the central subhalo ID was replaced with the host halo ID. A full list of properties saved has been included in Table 7.1.

Halo Properties		Subhalo properties	
Graph ID	Snapshot ID	Cold Gas Mass	Snapshot ID
Gas Metallicity	Catalogue ID	Mass Mean Position	Subhalo ID
Central subhalo ID	Halo ID	Dark Matter Mass	Host Halo ID
Mean Position	Redshift	Star Formation Rate	Graph ID
Hot Gas Cooling Rate	Baryon Mass	Modelled Stellar Mas	Redshift
Hot Gas Temperature	Viral Velocity	Behroozi Stellar Mass	
Dark Matter Mass	Hot Gas Mass		
Intracluster Light Mass			

Table 7.1: Output variables from the Py-Galaxies model.

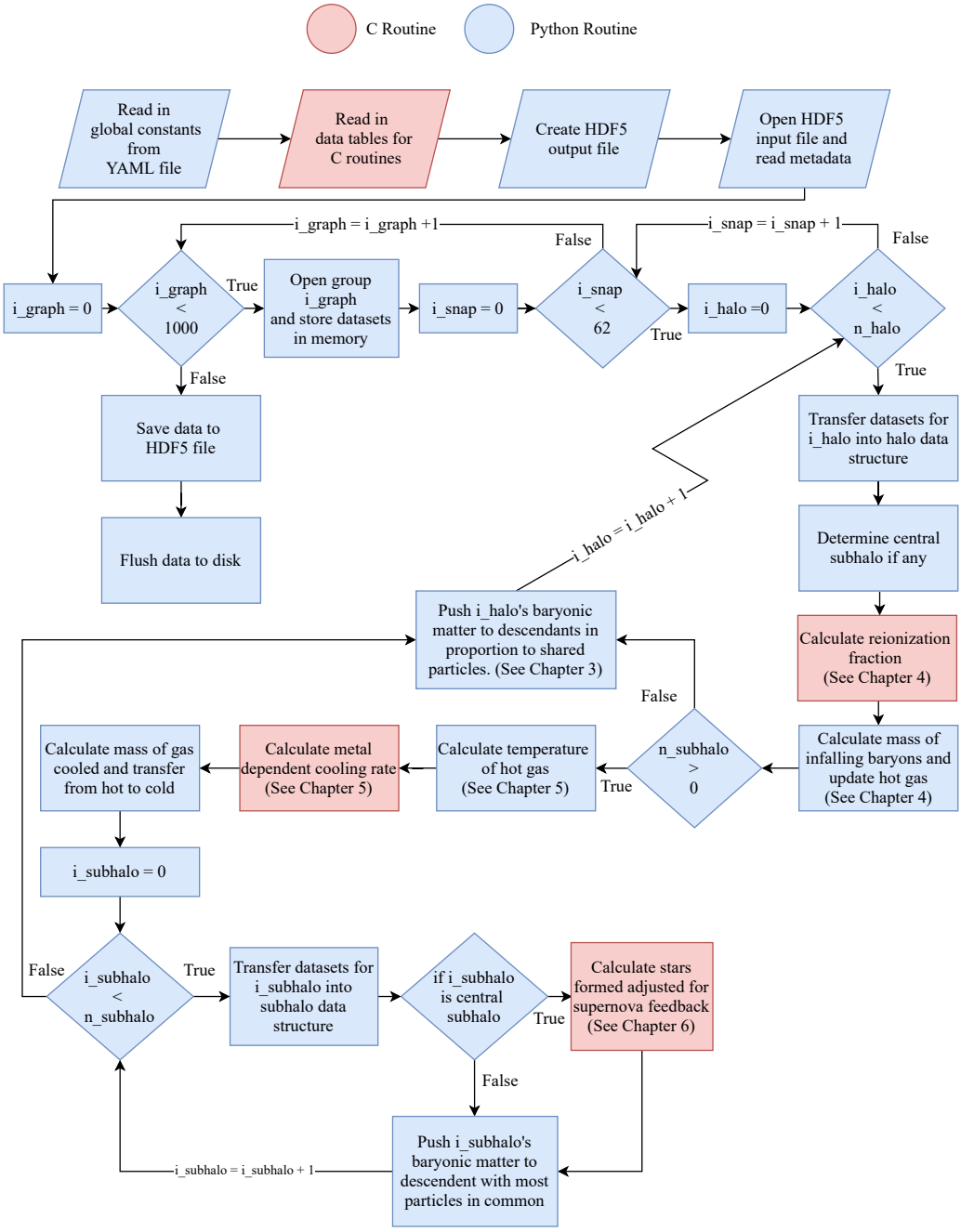

Figure 7.1: Flowchart displaying a high-level representation of the Py-Galaxies code. For more information on each specific process, see the corresponding chapter. Blue denotes a process that takes place in Python and red denotes processes that occur in C.

When outputting to HDF5 the data needs to be stored as an array and preferably, when storing data of multiple types into one table, a structured array. This was not issue for the array based code as the main halo and subhalo property arrays were simply flushed to disk once the model had finished running. For the class based code, data was extracted from the halo class, stored in NumPy structured arrays, and then flushed to disk. As this used additional memory, a user-defined parameter was introduced to control how large this intermediary NumPy array became before being flushed to disk.

Chapter 8

Class Based Vs. Array Based

A key aspect of this project was finding the most efficient way to structure the code, as it will be the foundation for future projects and research. As mentioned previously, both a class-based and an array-based code were created. In both versions of the code, a monitor module was also created that was used to measure the time taken and memory used by each graph and save the information to disk.

Before taking any measurements, both codes were standardised ensuring they contained the same calculations and the same level of optimisation. The only thing different between the two versions was the data structure.

8.1 Timing

Comparing the time complexity of the code was simple. A timing class defined in the monitor module was initialised behind a Boolean flag, assigned at runtime allowing the user to enable it when needed. The timer was started at the beginning of the graph loop and stopped at the end when all haloes had been processed. This enables the plotting of the processing time against the amount of haloes in each graph, as seen in Figure 8.1. From

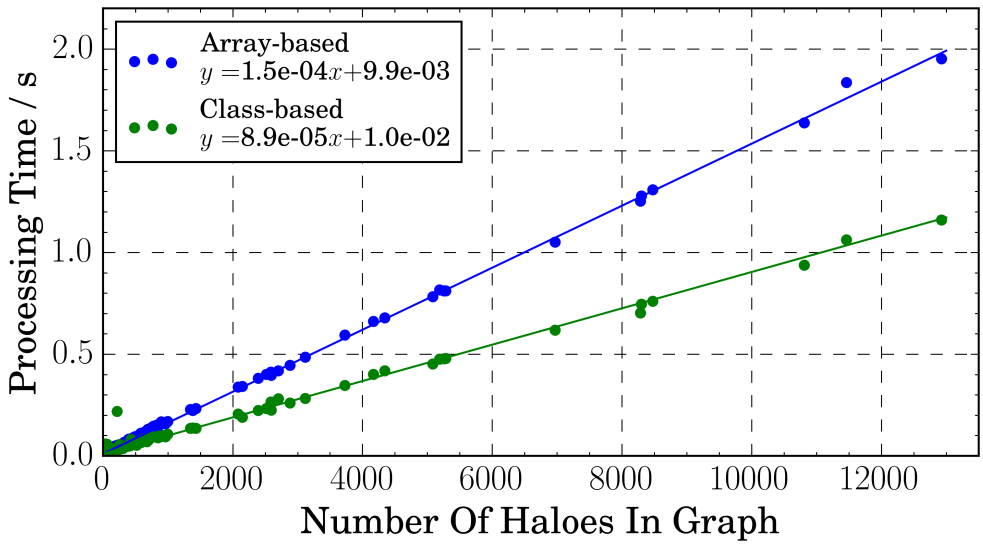

Figure 8.1: The time taken in seconds for each graph to be processed, plotted against the number of haloes in the graph. A linear fit has also been carried out to give an equation for how long a graph will take to process based on the amount of haloes within it.

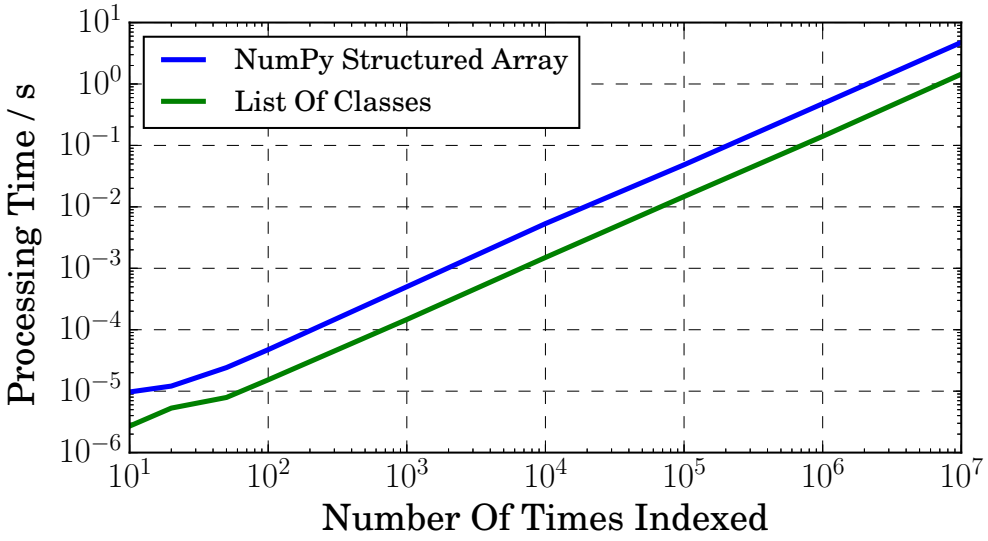

Figure 8.2: The time taken in seconds for Python to index and extract values from NumPy structured array and a list of classes.

these results, it is clear that the class based code is the better option as it is ~ 1.69 times faster. Both data structures demonstrated a linear complexity ($\mathcal{O}(n_{\text{halo}})$) which was then fitted to extract an equation to estimate the runtime for different use cases. For example, using the equations of best fit in Figure 8.1, we can extrapolate and estimate that a graph with 20,000 haloes would take ~ 1.78 s using the class-based code, but 3 seconds using the array-based code. This is likely due to array-based code having to slice and index one large array multiple times for each halo whereas the class based code simply extracts the correct halo from a list and accesses the attributes. NumPy is great for performing complex operations on large arrays, but when simply accessing the data, a list is much faster as shown in Figure 8.2. This is due to lists storing the memory location of each object instead of using a single contiguous memory block and calculating where the object is stored, as does NumPy.

8.2 Memory usage

Memory usage was a much trickier statistic to analyse. Initially, the cumulative memory used throughout processing the 975 graphs was tracked for both data structures. This was done by monitoring the Python process however, the very small amounts of memory used by the individual graphs were lost in noise from both the integrated development environment (IDE) and Python's garbage collector, invalidating the results. Instead, a more theoretical approach was taken by comparing the raw size of structured NumPy arrays with a list of classes. Python does not make this simple as the default function for getting the size of an object, `sys.getsizeof()`, only takes into account the size of the parent object. In the case of the list of classes, the size of the list will be returned but not the size of the class objects within; therefore, a recursive algorithm is needed.

To set up this test a simple class was created with 5 attributes: 4 floating point decimals and 1 NumPy array containing 10,000 values. This is to simulate the halo properties class that has a mixture of floats (e.g. redshift, mass), alongside NumPy arrays (e.g. progenitor IDs, position). The equivalent structure was then created within a NumPy structured array as seen below in "Memory Comparison".

Memory Comparison

```
class mem_test_class:
    def __init__(self):
        self.val1 = 1.0
        self.val2 = 1.0
        self.val3 = 1.0
        self.val4 = 1.0
        self.val5 = np.empty(10000,dtype=float)

mem_test_list = [mem_test_class() for i in range(0,length)]
mem_test_array  = np.empty(length, dtype=np.dtype([("Val1", float),
                                                   ("Val2", float),
                                                   ("Val3", float),
                                                   ("Val4", float),
                                                   ("Val5", float,
                                                   ↪  (10000,))]))
```

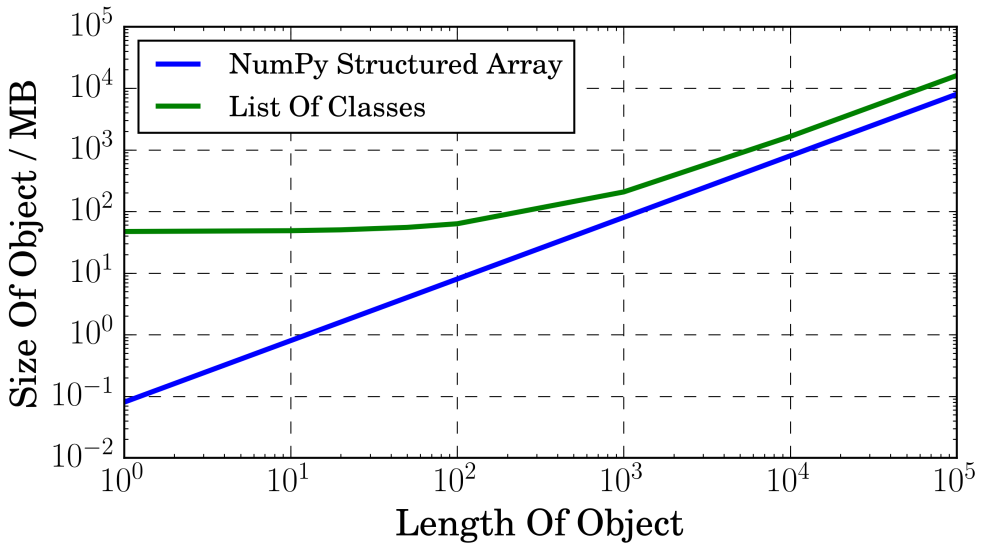

Figure 8.3: Memory used when storing a list of varying size and a NumPy structured array of varying size

The size of the two objects were then measured for several different lengths and the results plotted in Figure 8.3. From the graph, it is clear that the NumPy implementation uses the least memory. It also shows that there is significant overhead when using a list, as below 100 elements, the list stays at a constant size. Once around 1000 elements are reached, the list has linear complexity as does the NumPy array, with the list being around 2 times larger. In saying this, code to measure the list size produced slightly different results depending on whether it was run within an IDE, run from the command line, or run using iPython. This was believed to be caused by Python objects containing references to other objects that were not directly relevant to the list. For example, importing additional modules into the code caused the memory overhead of a Python class to increase. As the

current average size of the input merger graphs are around 0.1 MB (on disk), the majority of memory utilisation will be list overhead which, on a modern computer, is easily handled. However, if the graph sizes were to increase, the NumPy version of the code will be better suited to tackle them to ensure memory does not become a limiting factor in the model.

Chapter 9

Code Documentation

Throughout, NumPy documentation style has been used to document functions, methods, and classes. Below, in the code snippet labelled "Example Function Documentation" an example function has been shown. A short one line overview of the function is given, followed by a detailed description of what it does. Subsequently, the input arguments are described and their type stated, with the same carried out for output variables. This enables Read the Docs style HTML documentation to be automatically created, of which two example pages have been included in Figure 9.1. The full HTML pages are available on GitHub alongside a PDF version. Additional example codes where the documentation has been used to describe classes can be found in Appendix B.

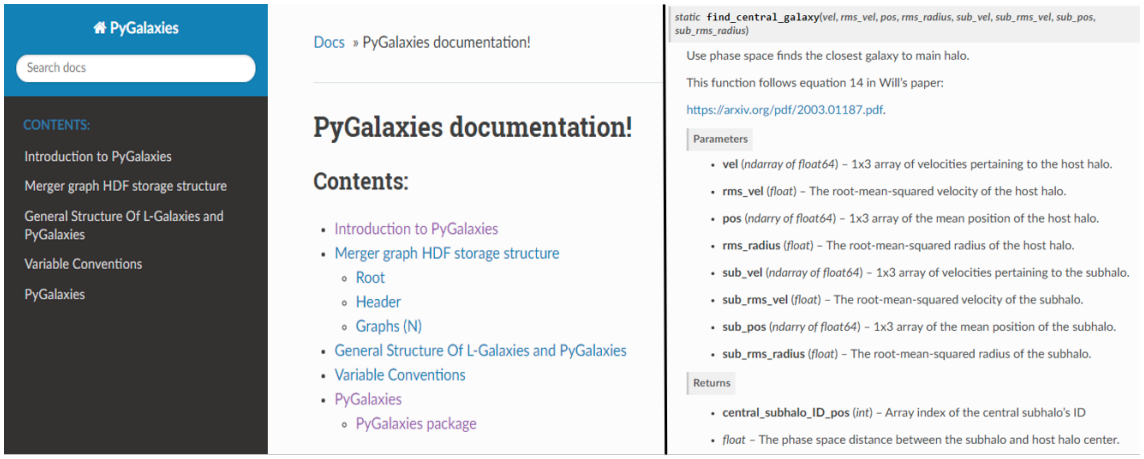

Figure 9.1: Screenshot taken from the HTML version of the documentation for Py-Galaxies.

Example Function Documentation

```python
def Behroozi_formula(a, z, halo_mass):
    """ Fitted equation from Behroozi et al. 2013 describing star formation.

    https://arxiv.org/abs/1207.6105
    This function takes in the halo mass, redshift, and scale factor,
    and then returns the expected stellar mass in the halo. This is an
    equation that has been fitted using real world data so the 'magic
    numbers' found below are arbitrary and can be found in the paper.

    Parameters
    ----------
    a : float
        Scale factor. 1/(1+z)
    z : float
        Redshift.
    halo_mass : float
        Dark matter mass of the halo in solar masses.

    Returns
    -------
    stellar_mass : float
        The expected stellar mass (in solar masses) for a halo of a
        halo_mass size
    """
    log_Mh = np.log10(halo_mass)
    nu = np.exp(-4 * (a ** 2))

    # Characteristic stellar/halo mass ratio
    log_epsilon = -1.777 + ((-0.006 * (a - 1) + (-0.000) * z) * nu) + (-0.119 * (a - 1))
    # Characteristic mass
    log_M1 = 11.514 + (-1.793 * (a - 1) + ((-0.251) * z)) * nu

    log_mass_frac = log_Mh - log_M1

    log_stellar_mass = (
        log_epsilon
        + log_M1
        + HaloProperties.powerlaw_Behroozi(log_mass_frac, nu, z, a)
        - HaloProperties.powerlaw_Behroozi(0, nu, z, a)
    )
    stellar_mass = 10 ** log_stellar_mass

    return stellar_mass
```

Chapter 10

Conclusions

The implementation of C-routines into Python using Cython has been highly effective. It has enabled Python to act as the front-end, setting run-time parameters without the need for recompilation, while maintaining performance during complex calculations. The simple framework for this laid out within Chapter 2 has laid the foundations for any projects building upon Py-Galaxies. A key focus of any future work would be understanding how lists or arrays of values can be passed to and from C in order to vectorize calculations and speed up the model.

Using Python to control the flow of the model and the data structures within has also been successful. The new merger graph structure, utilised as a class or an array, has enabled the accurate transition of baryonic properties from progenitor haloes to descendants. In a handful of cases, a halo's baryonic mass does exceed its dark matter mass (see Figure 4.2), which generally shouldn't occur. These cases are due to a small halo inheriting a large amount of baryonic matter from a much larger progenitor halo. The reasons for this are unclear; however, a possible cause could be the over-cooling of hot gas to cold gas. As there is no ejected reservoir within Py-Galaxies, gas heated by supernovae can cool down quicker than expected, leading to additional cold gas accreting onto subhaloes. In turn, as subhaloes give all their baryonic matter to a single descendent (instead of sharing it as haloes do), a small subhalo could inherit a vast amount of baryonic matter from a much larger progenitor.

The comparison of alternate methods to store and inherit halo and subhalo properties was also successful. The class based method is the most time efficient, though lacks the memory efficiency of NumPy arrays. Now and for the foreseeable future this trade off is worthwhile as a modest amount of RAM can handle the largest of graphs. Furthermore, the class based code is more readable and user friendly as the attributes of haloes and subhaloes are explicitly defined and documented. To fully understand the memory inefficiencies of the class based approach, a deeper investigation into Python's memory utilisation is necessary.

Star formation within the simulation, though not true to real life, is working as expected for the routines that have been implemented. To prevent the excess formation of stellar mass seen throughout Section 6.3 and therefore improve the model's accuracy, there are there are several steps that could be taken. Firstly, correctly modelling the size of galactic disks, instead of estimating based on the halo radius, would enable a more accurate calculation of the critical surface density used to approximate the star formation rate. Secondly, the previously mentioned ejected gas reservoir would prevent the over-cooling of gas which in turn, leads to excess star formation. And lastly, the inclusion of AGN feedback would quench star formation in larger galaxies. Each of these proposed improvements could be

easily worked into the current code structure and enable Py-Galaxies to begin producing result of a quality similar to L-Galaxies. In saying this, the initial aims of the project were to explore and test methods of implementing C-routines and to find the most efficient data structure to implement merger graphs with. Py-Galaxies has surpassed these initial aims and begun to take the form of a fully functional semi-analytic model of galaxy formation.

All Py-Galaxies source code, including analysis and plotting scripts, can be found on GitHub.

Acknowledgements

Thank you to my supervisor who has continuously given feedback on quality of my code and the validity of its results throughout. Thank you to Will Roper for providing the data and for their help whilst getting started on the project. I'd also like to mention how useful the open source modules Cython, Numpy, Matplotlib, and H5PY have been throughout the development of the Py-Galaxies code.

Bibliography

Baugh, C M et al. (2018). "Galaxy formation in the Planck Millennium: the atomic hydrogen content of dark matter haloes". In: *Monthly Notices of the Royal Astronomical Society* 483.4, pp. 4922–4937. ISSN: 1365-2966. DOI: 10.1093/mnras/sty3427.

Behroozi, Peter S., Risa H. Wechsler, and Charlie Conroy (2013). "The Average Star Formation Histories of Galaxies in Dark Matter Halos from z = 0-8". In: *The Astrophysical Journal* 770.1, 57, p. 57. DOI: 10.1088/0004-637X/770/1/57.

Benson, A. J. et al. (2003). "What Shapes the Luminosity Function of Galaxies?" In: *The Astrophysical Journal* 599.1, pp. 38–49. DOI: 10.1086/379160.

Birnboim, Yuval and Avishai Dekel (2003). "Virial shocks in galactic haloes?" In: *Monthly Notices of the Royal Astronomical Society* 345.1, pp. 349–364. DOI: 10.1046/j.1365-8711.2003.06955.x.

Boylan-Kolchin, Michael et al. (2009). "Resolving cosmic structure formation with the Millennium-II Simulation". In: *Monthly Notices of the Royal Astronomical Society* 398.3, pp. 1150–1164. DOI: 10.1111/j.1365-2966.2009.15191.x.

Bryan, Greg L. et al. (2014). "ENZO: AN ADAPTIVE MESH REFINEMENT CODE FOR ASTROPHYSICS". In: *The Astrophysical Journal Supplement Series* 211.2, p. 19. DOI: 10.1088/0067-0049/211/2/19.

Calder, A. C. et al. (2002). "On Validating an Astrophysical Simulation Code". In: *The Astrophysical Journal Supplement Series* 143.1, pp. 201–229. DOI: 10.1086/342267.

Cattaneo, A et al. (2020). "GalICS 2.1: a new semianalytic model for cold accretion, cooling, feedback, and their roles in galaxy formation". In: *Monthly Notices of the Royal Astronomical Society* 497.1, pp. 279–301. ISSN: 1365-2966. DOI: 10.1093/mnras/staa1832.

Ceverino, Daniel, Joel Primack, and Avishai Dekel (2015). "Formation of elongated galaxies with low masses at high redshift". In: *Monthly Notices of the Royal Astronomical Society* 453.1, pp. 408–413. ISSN: 0035-8711. DOI: 10.1093/mnras/stv1603.

Couchman, H. M. P. and M. J. Rees (1986). "Pregalactic evolution in cosmologies with cold dark matter". In: *Monthly Notices of the Royal Astronomical Society* 221, pp. 53–62. DOI: 10.1093/mnras/221.1.53.

Croton, Darren J. et al. (2006). "The many lives of active galactic nuclei: cooling flows, black holes and the luminosities and colours of galaxies". In: *Monthly Notices of the Royal Astronomical Society* 365.1, pp. 11–28. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2005.09675.x.

Davis, M. et al. (1985). "The evolution of large-scale structure in a universe dominated by cold dark matter". In: *The Astrophysical Journal* 292, pp. 371–394. DOI: 10.1086/163168.

De Lucia, Gabriella et al. (2006). "The formation history of elliptical galaxies". In: *Monthly Notices of the Royal Astronomical Society* 366.2, pp. 499–509. DOI: 10.1111/j.1365-2966.2005.09879.x.

Dekel, A. and J. Silk (1986). "The Origin of Dwarf Galaxies, Cold Dark Matter, and Biased Galaxy Formation". In: *The Astrophysical Journal* 303, p. 39. DOI: 10.1086/164050.

Efstathiou, G. (1992). "Suppressing the formation of dwarf galaxies via photoionization". In: *Monthly Notices of the Royal Astronomical Society* 256.2, 43P–47P. DOI: 10.1093/mnras/256.1.43P.

Fournier, Benoit, Peter A. Thomas, and Bruno M. Henriques (In Preparation). "A modified cooling algorithm for semi-analytic modelling". In: *Manuscript in preparation*.

Gnedin, Nickolay Y. (2000). "Effect of Reionization on Structure Formation in the Universe". In: *The Astrophysical Journal* 542.2, pp. 535–541. DOI: 10.1086/317042.

Gudivada, V. N., R. Baeza-Yates, and V. V. Raghavan (2015). "Big Data: Promises and Problems". In: *Computer* 48.3, pp. 20–23. DOI: 10.1109/MC.2015.62.

Guo, Qi et al. (2011). "From dwarf spheroidals to cD galaxies: simulating the galaxy population in a ΛCDM cosmology". In: *Monthly Notices of the Royal Astronomical Society* 413.1, pp. 101–131. DOI: 10.1111/j.1365-2966.2010.18114.x.

Guth, Alan H. (Jan. 1981). "Inflationary universe: A possible solution to the horizon and flatness problems". In: *Phys. Rev. D* 23 (2), pp. 347–356. DOI: 10.1103/PhysRevD.23.347.

Henriques, Bruno M B et al. (2019). "L-GALAXIES 2020: Spatially resolved cold gas phases, star formation, and chemical enrichment in galactic discs". In: *Monthly Notices of the Royal Astronomical Society* 491.4, pp. 5795–5814. ISSN: 1365-2966. DOI: 10.1093/mnras/stz3233.

Kauffmann, Guinevere (1996). "Disc galaxies at z=0 and at high redshift: an explanation of the observed evolution of damped Lyalpha absorption systems". In: *Monthly Notices of the Royal Astronomical Society* 281.2, pp. 475–486. DOI: 10.1093/mnras/281.2.475.

Kennicutt Robert C., Jr. (1998). "The Global Schmidt Law in Star-forming Galaxies". In: *The Astrophysical Journal* 498.2, pp. 541–552. DOI: 10.1086/305588.

Lacey, Cedric and Shaun Cole (1993). "Merger rates in hierarchical models of galaxy formation". In: *Monthly Notices of the Royal Astronomical Society* 262.3, pp. 627–649. ISSN: 0035-8711. DOI: 10.1093/mnras/262.3.627.

Larson, Richard B. (1974). "Effects of supernovae on the early evolution of galaxies". In: *Monthly Notices of the Royal Astronomical Society* 169, pp. 229–246. DOI: 10.1093/mnras/169.2.229.

Martin, Crystal L. (1999). "Properties of Galactic Outflows: Measurements of the Feedback from Star Formation". In: *The Astrophysical Journal* 513, pp. 156–160. DOI: 10.1086/379160.

Monaco, Pierluigi, Fabio Fontanot, and Giuliano Taffoni (2007). "The morgana model for the rise of galaxies and active nuclei". In: *Monthly Notices of the Royal Astronomical Society* 375.4, pp. 1189–1219. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2006.11253.x.

Okamoto, Takashi, Liang Gao, and Tom Theuns (2008). "Mass loss of galaxies due to an ultraviolet background". In: *Monthly Notices of the Royal Astronomical Society* 390.3, pp. 920–928. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2008.13830.x.

Peebles, P. J. E. (Dec. 1982). "Large-scale background temperature and mass fluctuations due to scale-invariant primeval perturbations". In: *The Astrophysical Journal* 263, pp. L1–L5. DOI: 10.1086/183911.

Planck Collaboration (2020). "Planck 2018 results". In: *Astronomy & Astrophysics* 641, A6. ISSN: 1432-0746. DOI: 10.1051/0004-6361/201833910.

Rees, M. J. and J. P. Ostriker (1977). "Cooling, dynamics and fragmentation of massive gas clouds: clues to the masses and radii of galaxies and clusters." In: *Monthly Notices of the Royal Astronomical Society* 179, pp. 541–559. DOI: 10.1093/mnras/179.4.541.

Reinecke, M., W. Hillebrandt, and J. C. Niemeyer (2002). "Refined numerical models for multidimensional type Ia supernova simulations". In: *A&A* 386.3, pp. 936–943. DOI: 10.1051/0004-6361:20020323.

Roper, William J, Peter A Thomas, and Chaichalit Srisawat (2020). "MEGA: Merger graphs of structure formation". In: *Monthly Notices of the Royal Astronomical Society* 494.3, pp. 4509–4524. ISSN: 0035-8711. DOI: 10.1093/mnras/staa982.

Rubin, V. C., Jr. Ford W. K., and N. Thonnard (Nov. 1978). "Extended rotation curves of high-luminosity spiral galaxies. IV. Systematic dynamical properties, Sa -> Sc." In: *The Astrophysical Journal* 225, pp. L107–L111. DOI: 10.1086/182804.

Schaye, Joop et al. (2014). "The EAGLE project: simulating the evolution and assembly of galaxies and their environments". In: *Monthly Notices of the Royal Astronomical Society* 446.1, pp. 521–554. ISSN: 0035-8711. DOI: 10.1093/mnras/stu2058.

Somerville, Rachel S. et al. (2008). "A semi-analytic model for the co-evolution of galaxies, black holes and active galactic nuclei". In: *Monthly Notices of the Royal Astronomical Society* 391.2, pp. 481–506. DOI: 10.1111/j.1365-2966.2008.13805.x.

Springel, V., J. Wang, et al. (2008). "The Aquarius Project: the subhaloes of galactic haloes". In: *Monthly Notices of the Royal Astronomical Society* 391.4, pp. 1685–1711. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2008.14066.x.

Springel, Volker (2005). "The cosmological simulation code GADGET-2". In: *Monthly Notices of the Royal Astronomical Society* 364.4, pp. 1105–1134. DOI: 10.1111/j.1365-2966.2005.09655.x.

Springel, Volker, Simon D. M. White, et al. (2001). "Populating a cluster of galaxies - I. Results at z=0". In: *Monthly Notices of the Royal Astronomical Society* 328.3, pp. 726–750. DOI: 10.1046/j.1365-8711.2001.04912.x.

Springel et al., Volker (2005). "Simulations of the formation, evolution and clustering of galaxies and quasars". In: *Nature* 435.7042, pp. 629–636. DOI: 10.1038/nature03597.

— (2020). *Simulating cosmic structure formation with the GADGET-4 code*. arXiv: 2010.03567.

Srisawat, Chaichalit et al. (2013). "Sussing Merger Trees: The Merger Trees Comparison Project". In: *Monthly Notices of the Royal Astronomical Society* 436.1, pp. 150–162. ISSN: 0035-8711. DOI: 10.1093/mnras/stt1545.

Sutherland, Ralph S. and M. A. Dopita (1993). "Cooling Functions for Low-Density Astrophysical Plasmas". In: *The Astrophysical Journals* 88, p. 253. DOI: 10.1086/191823.

Thomas, Peter (1988). "Multiphase cooling flows; a Lagrangian approach". In: *Monthly Notices of the Royal Astronomical Society* 235.2, pp. 315–341. ISSN: 0035-8711. DOI: 10.1093/mnras/235.2.315.

Toomre, A. (1964). "On the gravitational stability of a disk of stars." In: *The Astrophysical Journal* 139, pp. 1217–1238. DOI: 10.1086/147861.

Vogelsberger, Mark et al. (2020). "Cosmological simulations of galaxy formation". In: *Nature Reviews Physics* 2.1, pp. 42–66. ISSN: 2522-5820. DOI: 10.1038/s42254-019-0127-2.

Wadsley, James W., Benjamin W. Keller, and Thomas R. Quinn (2017). "Gasoline2: a modern smoothed particle hydrodynamics code". In: *Monthly Notices of the Royal Astronomical Society* 471.2, pp. 2357–2369. ISSN: 0035-8711. DOI: 10.1093/mnras/stx1643.

Appendix A

Additional Information On Input Format

This appendix contains additional tables of information regarding the input format of the HDF5 stored merger graphs.

Halo Attributes	**Units**	**Type**	**Shape**
3-D Velocity Dispersion	kms^{-1}	float	(N_{halo})
Catalogue Halo IDs	None	int	(N_{halo})
Descendent Start Index	None	int	(N_{halo})
Direct Descendent Contribution	None	int	(N_{desc})
Direct Descendent IDs	None	int	(N_{desc})
Direct Progenitor Contribution	None	int	(N_{prog})
Direct Progenitor IDs	None	int	(N_{prog})
Half Mass Radius	Mpc	float	(N_{halo})
Half Mass Velocity Radius	Mpc	float	(N_{halo})
Max Velocity	kms^{-1}	float	(N_{halo})
Mean Position	None	float	$(\mathrm{N}_{halo},\ 3)$
Mean Velocity	kms^{-1}	float	$(\mathrm{N}_{halo},\ 3)$
No. of Dark Matter Particles	None	int	(N_{halo})
No. of Direct Descendants	None	int	(N_{halo})
No. of Direct Progenitors	None	int	(N_{halo})
No. of Subhaloes	None	int	(N_{halo})
Progenitor Start Index	None	int	(N_{halo})
Redshifts	None	float	(N_{halo})
Root Mean Squared Radius	Mpc	float	(N_{halo})
Snapshots	None	int	(N_{halo})
Snapshot ID	None	int	(N_{halo})
Snapshot Length	None	int	(N_{halo})
Snapshot Start Index	None	int	(N_{halo})

Table A.1: Merger graph halo input attributes.

Subhalo Attributes	**Units**	**Type**	**Shape**
3-D Velocity Dispersion	kms^{-1}	float	$(\mathrm{N}_{subhalo})$
Catalogue Halo IDs	None	int	$(\mathrm{N}_{subhalo})$
Descendent Start Index	None	int	$(\mathrm{N}_{subhalo})$
Direct Descendent Contribution	None	int	$(\mathrm{N}_{subdesc})$
Direct Descendent IDs	None	int	$(\mathrm{N}_{subdesc})$
Direct Progenitor Contribution	None	int	$(\mathrm{N}_{subprog})$
Direct Progenitor IDs	None	int	$(\mathrm{N}_{subprog})$
Half Mass Radius	Mpc	float	$(\mathrm{N}_{subhalo})$
Half Mass Velocity Radius	Mpc	float	$(\mathrm{N}_{subhalo})$
Host Halo ID	None	int	$(\mathrm{N}_{subhalo})$
Max Velocity	kms^{-1}	float	$(\mathrm{N}_{subhalo})$
Mean Position	None	float	$(\mathrm{N}_{subhalo},\ 3)$
Mean Velocity	kms^{-1}	float	$(\mathrm{N}_{subhalo},\ 3)$
No. of Dark Matter Particles	None	int	$(\mathrm{N}_{subhalo})$
No. of Direct Descendants	None	int	$(\mathrm{N}_{subhalo})$
No. of Direct Progenitors	None	int	$(\mathrm{N}_{subhalo})$
Progenitor Start Index	None	int	$(\mathrm{N}_{subhalo})$
Redshifts	None	float	$(\mathrm{N}_{subhalo})$
Root Mean Squared Radius	Mpc	float	$(\mathrm{N}_{subhalo})$
Snapshots	None	int	$(\mathrm{N}_{subhalo})$
Snapshot ID	None	int	$(\mathrm{N}_{subhalo})$
Snapshot Length	None	int	$(\mathrm{N}_{subhalo})$
Snapshot Start Index	None	int	$(\mathrm{N}_{subhalo})$

Table A.2: Merger graph subhalo input attributes.

HDF5 Attributes	**Value**	**Type**	**Shape**
NO DATA FLOAT	NaN	float	-
NO DATA INT	1073741824	int	-
Dark Matter Particle Mass	$1.33 \times 10^9\ \mathrm{M}_\odot$	float	-
Halo Graph Lengths	-	int	(N_{graph})
Number Of Haloes In Graph	-	int	(N_{graph})
Subhalo Graph Length	-	int	(N_{graph})
Number Of Subhaloes In Graph	-	int	(N_{graph})

Table A.3: Additional information stored as attributes in the HDF5 file.

Appendix B

Source Code Snippets

Within this Appendix, snippets from the source code have been included to document the structure of GraphProperties class, which reads in the data for the graph; HaloProperties class, which truncates the graph data to each halo and performs calculations on the halo; and SubhaloProperties class, which stores the subhalo attributes and performs calculations on the subhalo.

Again, the full code can be found on GitHub, alongside the Array based code branch at: https://github.com/AnBowell/Final-Year-Project-py-galaxies.

B.1 Graph Properties Class

The code below is the entire code for the GraphProperties class. It contains all of the attributes that are used within the Py-Galaxies model.

```
class GraphProperties:
    """A container for all data within a single graph.

    This class consists of data read in from an HDF5 group (graph). The
    documentation provided here was aided by Will Roper as they provided the
    input data.

    Attributes
    ----------
    desc_start_index : ndarray of type 'int'
        The starting index (pointer) for each halo'sentries in all descendant
        halo arrays (i.e.direct_desc_ids,direct_desc_contribution, etc.).
        Entries containing 2**30 have no descendants.
    direct_desc_contribution : ndarray of type 'int'
        The number of dark matter particles contributed to each direct
        descendent from the halo.
    direct_desc_ids : ndarray of type 'int'
        The descendent halo IDs, extracted using desc_start_index and ndesc
    direct_prog_contribution : ndarray of type 'int'
        The number of dark matter particles contributed by each direct
        progenitor to thehalo.
    direct_prog_ids : ndarray of type 'int'
        The progenitor halo IDs, extracted using prog_start_index and nprog.
    generation_id : ndarray of type 'int'
        The ID (or number) associated with each generation. Counting starts
        from the earliest snapshot.
```

```
    generation_length : ndarray of type 'int'
        The number of halos in each generation
    generation_start_index : ndarray of type 'int'
        The starting index (pointer) for each host halo generation.
    graph_halo_ids : ndarray of type 'int'
        The internal graph halo ID assigned to ahalo. NOTE: These differ from
        the halo catalog. These run from 0 - Nhost with the index equal to the
        value.
    halo_catalog_halo_ids : ndarray of type 'int'
        The halo catalog ID assigned to each halo.
    mean_pos : ndarray of type 'float'
        The mean position of the particles in the halo.
    ndesc : ndarray of type 'int'
        The number of descendants for each halo.
    nparts : ndarray of type 'int'
        The number of dark matter particles in each halo.Well,
    nprog : ndarray of type 'int'
        The number of progenitors for each halo.
    prog_start_index : ndarray of type 'int'
        The starting index (pointer) for each halo'sentries in all progenitor
        halo arrays (i.e.direct_prog_ids,direct_prog_contribution, etc.).
        Entries containing 2**30 have no descendants.
    redshifts : ndarray of type 'float'
        The redshift for each halo in the graph.
    snapshots : ndarray of type 'int'
        The index of the snapshot in the snapshottext file dictated in the
        param file, for eachhalo.
    sub_desc_start_index : ndarray of type 'int'
        The starting index (pointer) for eachsubhalo's entries in all
        descendant subhaloarrays (i.e. sub_direct_desc_ids,
        sub_direct_desc_contribution, etc.).Entries containing 2**30 have
        no descendants.
    sub_direct_desc_contribution : ndarray of type 'int'
        The number of dark matter particles contributed to each direct
        descendent fromthe subhalo.
    sub_direct_desc_ids : ndarray of type 'int'
        The descendent subhalo IDs, extracted using sub_desc_start_index
        and sub_ndesc.
    sub_direct_prog_contribution : ndarray of type 'int'
        The number of dark matter particles contributed by each direct
        progenitor  to  the subhalo.
    sub_direct_prog_ids : ndarray of type 'int'
        The progenitor subhalo IDs, extracted using sub_prog_start_index and
        sub_nprog.
    sub_generation_id : ndarray of type 'int'
        The ID (or number) associated with each generation. Counting starts
        from the earliest snapshot.
    sub_generation_length : ndarray of type 'int'
        The number of subhalos in each generation.
    sub_generation_start_index : indarray of type 'int'nt
        The starting index (pointer) for each subhalo generation.
    sub_graph_halo_ids : ndarray of type 'int'
```

```
        The internal graph subhalo ID assigned to a subhalo. NOTE: These differ
        from the halo catalogue. These run from 0 - Nsut with the index equal to
        the value.
    sub_mean_pos : ndarray of type 'float'
        The mean position of the particles in the subhalo.
    sub_ndesc : ndarray of type 'int'
        The number of descendents for each subhalo.
    sub_nparts : ndarray of type 'int'
        The number of dark matter particles in each halo.
    sub_nprog : ndarray of type 'int'
        The number of progenitors for each halo.
    sub_prog_start_index : ndarray of type 'int'
        The starting index (pointer) for each subhalo's entries in all
        progenitor subhalo arrays (i.e. sub_direct_prog_ids,
        sub_direct_prog_contribution, etc.).Entries containing 2**30 have
        no descendants.
    sub_redshifts : ndarray of type 'float'
        The redshift for each subhalo in the graph.
    sub_snapshots : ndarray of type 'int'
        The index of the snapshot in the snapshot text file dictated in the
        param file, for each subhalo.
    subhalo_catalog_halo_ids : ndarray of type 'int'
        The subhalo catalog ID assigned to each subhalo.
    velocity_dispersion_3D : ndarray of type 'float'
        The velocity dispersion for each halo in the graph.
    half_mass_radius : ndaarray of type 'float'
        The half mass radius for each halo in the graph.
    half_mass_velocity_radius : ndaarray of type 'float'
        The half mass velocity radius for each halo in the graph.
    mean_vel : ndaarray of type 'float'
        An array 3 by N_halos long. It contains the x,y,z velocities for each
        halo in the graph.
    rms_radius : ndarray of type 'float'
        The rms radius of each halo within the graph.
    v_max : ndarray of type 'float'
        The maximum velocity of the halo within the graph.
    sub_halo : bool
        From yml input file. Sub halos or no sub halos basically.
    sub_velocity_dispersion_3D : ndarray of type 'float'
        The velocity dispersion for each subhalo in the graph.
    sub_half_mass_radius : ndaarray of type 'float'
        The half mass radius for each subhalo in the graph.
    sub_half_mass_velocity_radius : ndaarray of type 'float'
        The half mass velocity radius for each subhalo in the graph.
    sub_mean_vel : ndaarray of type 'float'
        An array 3 by N_halos long. It contains the x,y,z velocities for each
        subhalo in the graph.
    sub_rms_radius : ndarray of type 'float'
        The rms radius of each subhalo within the graph.
    sub_v_max : ndarray of type 'float'
        The maximum velocity of the subhalo within the graph.
    subhalo_start_index : ndarray of type 'int'
```

```
        Starting index for the subhalo within the graph. Indexed with the halo
        ID.
    host_halos : ndarray of type 'int'
        The host halo ID for the sub halos. n_subhalos long,indexed with sub-
        halo ID.

    """

    def __init__(self, graph_ID, open_HDF_file, model_params, part_mass):
        """Opening HDF datasets for a graph and saving them to the class.

        Parameters
        ----------
        graph_ID : int
            The ID of the graph to be opened.
        open_HDF_file : :obj: 'File'
            Open HDF5 file that is to be read from.
        model_params : obj: 'Class'
            ModelParams class object.
        part_mass : float
            The mass of the dark matter particles. Attribute in HDF.

        """
        self.graph_ID = graph_ID

        open_HDF_group = open_HDF_file[str(graph_ID)]

        self.desc_start_index = open_HDF_group["desc_start_index"][:]
        self.direct_desc_contribution = (
            open_HDF_group["direct_desc_contribution"][:] * part_mass
        )
        self.direct_desc_ids = open_HDF_group["direct_desc_ids"][:]
        self.direct_prog_contribution = (
            open_HDF_group["direct_prog_contribution"][:] * part_mass
        )
        self.direct_prog_ids = open_HDF_group["direct_prog_ids"][:]
        self.generation_length = open_HDF_group["generation_length"][:]
        self.generation_id = np.arange(
            len(self.generation_length), dtype=int
        )
        self.generation_start_index = open_HDF_group["generation_start_index"][:]
        self.halo_catalog_halo_ids = open_HDF_group["halo_catalog_halo_ids"][:]
        self.mean_pos = open_HDF_group["mean_pos"][:]
        self.ndesc = open_HDF_group["ndesc"][:]
        self.mass = open_HDF_group["nparts"][:] * part_mass
        self.nprog = open_HDF_group["nprog"][:]
        self.prog_start_index = open_HDF_group["prog_start_index"][:]
        self.redshifts = open_HDF_group["redshifts"][:]
        self.snapshots = open_HDF_group["snapshots"][:]
        self.velocity_dispersion_3D = open_HDF_group["3D_velocity_dispersion"][:]
        self.half_mass_radius = open_HDF_group["half_mass_radius"][:]
        self.half_mass_velocity_radius = open_HDF_group["half_mass_velocity_radius"][:]
```

```
        self.mean_vel = open_HDF_group["mean_vel"][:]
        self.rms_radius = open_HDF_group["rms_radius"][:]
        self.v_max = open_HDF_group["v_max"][:]
        self.graph_halo_ids = np.arange(len(self.mass))
        self.n_halos_in_graph = open_HDF_group.attrs["nhalos_in_graph"]
        self.n_subhalos = open_HDF_group["nsubhalos"][:]
        self.sub_desc_start_index = open_HDF_group["sub_desc_start_index"][:]
        self.sub_direct_desc_contribution = (
            open_HDF_group["sub_direct_desc_contribution"][:] * part_mass
        )
        self.sub_direct_desc_ids = open_HDF_group["sub_direct_desc_ids"][:]
        self.sub_direct_prog_contribution = (
            open_HDF_group["sub_direct_prog_contribution"][:] * part_mass
        )
        self.sub_direct_prog_ids = open_HDF_group["sub_direct_prog_ids"][:]
        self.sub_generation_id = open_HDF_group["sub_generation_id"][:]
        self.sub_generation_length = open_HDF_group["sub_generation_length"][:]
        self.sub_generation_start_index = open_HDF_group[
            "sub_generation_start_index"
        ][:]
        self.sub_mean_pos = open_HDF_group["sub_mean_pos"][:]
        self.sub_ndesc = open_HDF_group["sub_ndesc"][:]
        self.sub_nparts = open_HDF_group["sub_nparts"][:]
        self.sub_mass = self.sub_nparts * part_mass
        self.sub_nprog = open_HDF_group["sub_nprog"][:]
        self.sub_prog_start_index = open_HDF_group["sub_prog_start_index"][:]
        self.sub_redshifts = open_HDF_group["sub_redshifts"][:]
        self.sub_snapshots = open_HDF_group["sub_snapshots"][:]
        self.subhalo_catalog_halo_ids = open_HDF_group[
            "subhalo_catalog_halo_ids"
        ][:]
        self.sub_velocity_dispersion_3D = open_HDF_group[
            "sub_3D_velocity_dispersion"
        ][:]
        self.sub_half_mass_radius = open_HDF_group["sub_half_mass_radius"][:]
        self.sub_half_mass_velocity_radius = open_HDF_group[
            "sub_half_mass_velocity_radius"
        ][:]
        self.sub_mean_vel = open_HDF_group["sub_mean_vel"][:]
        self.sub_rms_radius = open_HDF_group["sub_rms_radius"][:]
        self.sub_v_max = open_HDF_group["sub_v_max"][:]
        self.subhalo_start_index = open_HDF_group["subhalo_start_index"][:]
        self.host_halos = open_HDF_group["host_halos"][:]
        self.sub_graph_halo_ids = np.arange(len(self.sub_nparts))
```

B.2 Halo Properties Class

The code below contains the constructor for the HaloProperties which is called to instantiate the class. Included, are two additional physics methods which are used by the constructor method.

```
class HaloProperties:
    """A container for the properties and methods needed for each halo.

    This class extracts the data needed for a single halo from the
    GraphProperties class. It also contains the methods needed to process
    the haloes such as infall and cooling routines. Each method within has
    its own documentation.

    Attributes
    ----------
    graph_ID : str
        The graph_ID (from HDF5 group).
    snap_ID : int
         The snapshot ID currently being processed.
    halo_ID : int
        The halo ID of currently being processed.
    catalog_ID : int
        The ID of the halo corresponding to the original catalogue.
    mass : int
        Mass of halo. Amount of dark matter particles * mass of particle.
    nprog : int
        The amount of progenitors the halo has.
    prog_start : int
        The index at which this halo's progenitors start.
    prog_end : int
        The index at which this halo's progenitors end.
    prog_ids : narray
        Numpy array of the progenitor IDs for the halo.
    prog_mass : ndarry
        Numpy array of the progenitor mass contributions.
    ndesc : int
        Amount of descendent halos.
    desc_start : int
        The index at which this halo's descendants start.
    desc_end : int
        The index at which this halo's descendants end.
    desc_ids : ndarray of type 'int'
        Numpy array of the halo's descendent IDs of type.
    desc_mass : ndarray of type 'int'
        Numpy array of the descendent mass contributions.
    mass_baryon : float
        Mass of Baryons within the halo.
    mass_from_progenitors : float
        Total mass of all the progenitor halos.
    mass_baryon_from_progenitors : float
        Total mass of all the Baryons contained within the progenitor halos.
    done : bool
        Whether or not the halo has been processed.
    subhalo_start_index : int
        The index at which the subhalos, corresponding to the main halo,
        start.
    n_subhalo : int
```

```
        The amount of subhalos in the host halo.
    sub_graph_halo_ids : ndarray of type 'int'
        The subhalo IDs that are contained within the main halo.
    mean_pos : ndarray of type 'float'
        The mean position (x,y,z) of the halo.
    velocity_dispersion_3D : float
        The 3D velocity dispersion of the halo.
    mean_vel : ndarry of type 'float'
        The mean velocity (V_x,V_y,V_z) of the halo.
    rms_radius : float
        The root mean squared radius of the halo
    redshift : float
        The redshift of the halo.
    total_halo_baryon_mass : float
        The total amount of baryons within the halo. E.g. hot gas +
        stellar and cold gas mass from any subhaloes within.
    hot_gas_mass : float
        The mass of hot gas within the halo.
    hot_gas_temp : float
        The temperature of the hot gas within the halo.
    gas_metalicity : float
        The metalicity of the gas. Currently constant at 1e-4.
    cold_gas : float
        The mass of cold gas
    metal_dependent_cooling_rate : float
        The cooling rate of the hot gas in ergs cm^3 / s.
    intracluster_stellar_mass : float
        The stellar mass within the halo that does not belong to any
        subhaloes. This occurs when a subhalo has no descendants.
    R_200 : float
        The radius of the halo's sphere which encloses an overdensity
        of 200 times that of the critical density.
    central_galaxy_ID : int
        The ID of the central subhalo if there is one. If not,
        it is filled with 2**30, the fill value for the model.
    """

    def __init__(
        self,
        graph_ID,
        halo_ID,
        graph_properties,
        model_params
    ):
        """Extract halo properties from GraphProperties for halo halo_ID

        Parameters
        ----------
        graph_ID : str
            The graph_ID (from HDF5 group).
        halo_ID : int
            The halo ID of currently being processed.
```

```
        graph_properties : :obj: 'Class'
            An instance of GraphProperties.
        model_params : Instance of ModelParams Class
            An instance of the model params class which contains all of the
            global parameters.
        """
        self.graph_ID = graph_ID
        self.snap_ID = None             # Not filled in initialisation
        self.halo_ID = halo_ID

        self.catalog_ID = graph_properties.halo_catalog_halo_ids[halo_ID]
        self.mass = graph_properties.mass[halo_ID]
        self.nprog = graph_properties.nprog[halo_ID]
        self.prog_start = graph_properties.prog_start_index[halo_ID]

        self.prog_end = self.prog_start + self.nprog
        self.prog_ids = graph_properties.direct_prog_ids[
            self.prog_start : self.prog_end
        ]
        self.prog_mass = graph_properties.direct_prog_contribution[
            self.prog_start : self.prog_end
        ]
        self.ndesc = graph_properties.ndesc[halo_ID]
        self.desc_start = graph_properties.desc_start_index[halo_ID]
        self.desc_end = self.desc_start + self.ndesc

        self.desc_ids = graph_properties.direct_desc_ids[
            self.desc_start : self.desc_end
        ]
        self.desc_mass = graph_properties.direct_desc_contribution[
            self.desc_start : self.desc_end
        ]
        self.mean_pos = graph_properties.mean_pos[halo_ID]
        self.velocity_dispersion_3D = graph_properties.velocity_dispersion_3D[halo_ID]
        self.mean_vel = graph_properties.mean_vel[halo_ID]

        self.rms_radius = graph_properties.rms_radius[halo_ID]
        self.redshift = graph_properties.redshifts[halo_ID]

        self.total_halo_baryon_mass = 0.0
        self.hot_gas_mass = 0.0
        self.hot_gas_temp = 0.0
        self.gas_metalicity = 1e-4

        self.metal_dependent_cooling_rate = 0.0
        self.intracluster_stellar_mass = 0.0

        self.subhalo_start_index = graph_properties.subhalo_start_index[halo_ID]
        self.n_subhalo = graph_properties.n_subhalos[halo_ID]
        self.sub_graph_halo_ids = graph_properties.sub_graph_halo_ids[
            self.subhalo_start_index : self.subhalo_start_index + self.n_subhalo
        ]
```

```
        self.done = False

        # Find the radius of the haloes overdensity.
        self.calc_radius_200(self.redshift, model_params)

        # If there are subhaloes, find the central one
        if self.n_subhalo > 0:

            central_sub_halo_ID_pos, central_sub_halo_dist = self.find_central_galaxy(
            self.mean_vel,
            self.velocity_dispersion_3D,
            self.mean_pos,
            self.rms_radius,
            graph_properties.sub_mean_vel[self.sub_graph_halo_ids],
            graph_properties.sub_velocity_dispersion_3D[
                self.sub_graph_halo_ids
            ],
            graph_properties.sub_mean_pos[self.sub_graph_halo_ids],
            graph_properties.sub_rms_radius[self.sub_graph_halo_ids],
            )
            # The central sub-halo dist will be used as a variable
            # in future versions of the code.
            # If close enough declare it a central halo -
            # currently 2**30 as a fill

            if central_sub_halo_dist < 2**30:

                self.central_galaxy_ID = self.sub_graph_halo_ids[
                        central_sub_halo_ID_pos]
            else:
                self.central_galaxy_ID = 2**30

        else:
            self.central_galaxy_ID = 2**30

    @staticmethod
    def find_central_galaxy(
        vel, rms_vel, pos, rms_radius, sub_vel, sub_rms_vel, sub_pos, sub_rms_radius
    ):
        """Use phase space finds the closest galaxy to main halo.

        This function follows equation 14 in Roper et al. Paper:
        https://arxiv.org/pdf/2003.01187.pdf.

        Parameters
        ----------
        vel : ndarray of float64
            1x3 array of velocities pertaining to the host halo.
        rms_vel : float
            The root-mean-squared velocity of the host halo.
        pos : ndarry of float64
            1x3 array of the mean position of the host halo.
```

```
        rms_radius : float
            The root-mean-squared radius of the host halo.
        sub_vel : ndarray of float64
            1x3 array of velocities pertaining to the subhalo.
        sub_rms_vel : float
            The root-mean-squared velocity of the subhalo.
        sub_pos : ndarry of float64
            1x3 array of the mean position of the subhalo.
        sub_rms_radius : float
            The root-mean-squared radius of the subhalo.

        Returns
        -------
        central_subhalo_ID_pos : int
            Array index of the central subhalo's ID
        float
            The phase space distance between the subhalo and host halo center.

        """
        pos_dists = np.sqrt(np.sum((pos - sub_pos) ** 2, axis=1)) / (
            rms_radius + sub_rms_radius
        )

        vel_dists = np.sqrt(np.sum((vel - sub_vel) ** 2, axis=1)) / (
            rms_vel + sub_rms_vel
        )

        total = pos_dists + vel_dists

        central_subhalo_ID_pos = np.argmin(total)

        return central_subhalo_ID_pos, total[central_subhalo_ID_pos]

    def calc_radius_200(self, z, model_params):
        """ Calculate the spherical overdensity radius of the halo

        Using equation S1 from the L-Galaxies 2015 model description,
        https://lgalaxiespublicrelease.github.io/Hen15_doc.pdf ,
        calculate the radius of the halo.

        Parameters
        ----------
        z : float
            Redshift of the halo.
        model_params : Instance of class ModelParams
            An instance of the class ModelParams where the global cosmology
            parameters have been stored.

        Returns
        -------
        None.

```

```
        """
        Hz = model_params.H0 * (((model_params.omega_m * ((1+z) ** 3)) +
                                 model_params.omega_lambda) ** (1/2))


        self.R_200 = ((self.mass * model_params.SolMass_Mpc_G) /
                      (100 * (Hz**2))) ** (1 / 3)

        return None
```

B.3 Subhalo Properties Class

The code below contains the entire SubhaloProperties class. It is only short as not many properties are needed from the graph to calculate the star formation within the central subhalo. The class contains two methods, one to pass down baryonic matter to descendants and a second to calculate the stellar mass formed.

```
class SubhaloProperties:
    """ A container for any subhalo properties

    A short class containing the attributes of subhalos as well as
    any methods that act on the class. For example, star formation.

    Attributes
    ----------

    graph_ID : int
        The graph ID the subhalo belongs to.
    snap_ID : int
        The snapshot the subhalo belongs to.
    host_halo_ID : int
        The host halo the subhalo is within.
    subhalo_ID : int
        The identification number of the subhalo.
    mean_pos : ndarray of type 'float'
        An array containing the (x,y,z) position of the subhalo.
    redshift : float
        The redshift of the subhalo.
    DM_mass : float
        The dark matter mass of the subhalo
    ndesc : int
        The number of descendants the subhalo has.
    desc_start_index : int
        The index at which this subhalo's descendants start.
    desc_ids : ndarry of type 'int'
        The IDs of any descendants of this subhalo.
    C_stellar_mass : float
        The stellar mass within the subhalo as modelled using the
        L-Galaxies C-routines.
    SFR : float
        The start formation rate of the subhalo. Amount of stars
```

```
        formed / the time between snapshots.
    stellar_mass : float
        The stellar mass from the Behroozi calculations.
    cold_gas_mass : float
        The mass of cold gas which has been cooled onto the subhalo.

    """
    def __init__(self, graph_ID, host_halo_ID, subhalo_ID,
                 graph_properties):
        """ Extract the relevant subhalo data from graph properties.

        Parameters
        ----------
        graph_ID : int
            The current graph ID.
        host_halo_ID : int
            The ID of the halo the current subhalo resides in.
        subhalo_ID : int
            The ID of the current subhalo.
        graph_properties : Instance of class GraphProperties
            An instance of the graph properties class.

        Returns
        -------
        None.

        """
        self.graph_ID = graph_ID
        self.snap_ID = None # Not in initialisation
        self.host_halo_ID = host_halo_ID
        self.subhalo_ID = subhalo_ID
        self.mean_pos = graph_properties.sub_mean_pos[subhalo_ID]
        self.redshift = graph_properties.sub_redshifts[subhalo_ID]
        self.DM_mass = graph_properties.sub_mass[subhalo_ID]
        self.ndesc = graph_properties.sub_ndesc[subhalo_ID]
        self.desc_start_index = graph_properties.sub_desc_start_index[subhalo_ID]
        self.desc_ids = graph_properties.sub_direct_desc_ids[
             self.desc_start_index : self.desc_start_index +  self.ndesc
        ]
        self.C_stellar_mass  = 0.0
        self.SFR = 0.0
        self.stellar_mass = 0.0
        self.cold_gas_mass = 0.0

    def calc_subhalo_props_descend(self,list_of_subhalo_properties,
                                   subhalo_descend_attrs,
                                   list_of_halo_properties):
        """ Pass on all baryons to a single descendent subhalo.

        This function iterates over a list of properties that need to
        be passed down to descendants. It utilises getters and setters
        to extract values from class attributes while only using a
```

```
        list. This means only the list has to be changed when new
        properties are added.

        Parameters
        ----------
        list_of_subhalo_properties : list of Subhalo Classes
            The main list of classes for subhalo properties.
        subhalo_descend_attrs : list of type 'str'
            List of baryonic properties (attribute names) that are to
            be passed down to descendants.
        list_of_halo_properties : list of Halo Classes
            The main list of classes for halo properties.

        Returns
        -------
        None.

        """
        if self.ndesc > 0 :

            largest_desc_ID = self.desc_ids[0]

            descendent_subhalo = list_of_subhalo_properties[largest_desc_ID]

            for subhalo_property in subhalo_descend_attrs:

                to_descend = getattr(self, subhalo_property)

                setattr(descendent_subhalo, subhalo_property,
                        getattr(descendent_subhalo, subhalo_property)
                                + to_descend)

        # If no descendants stars go to host halo as intracluster light.
        elif self.ndesc ==0:

            list_of_halo_properties[self.host_halo_ID].intracluster_stellar_mass += \
                self.C_stellar_mass




    def calculate_stars_formed(self, halo, dt):
        """ Calculate the mass of stars formed using C routine.

        Utilises the Cython C routine to calculate the mass of stars
        formed in the central subhalo.

        Parameters
        ----------
        halo : Instance of class HaloProperties.
            The halo the host halo resides in. This is currently
            needed as the galactic disk is an approximation based on
```

```
            halo size.
        dt : float
            Time between the last snapshot and the current one.

        Returns
        -------
        None.

        """
        # Calculate the mass of cold gas which has been reheated, and
        # the mass that has been used to form stars.
        return_dict = C_star_formation(halo.V_200, halo.R_200/10, self.cold_gas_mass,
                                       0, dt ,1)
        stars_formed = return_dict['Returnstars']
        reheated_mass = return_dict['Return_reheated_mass']

        self.C_stellar_mass += stars_formed

        # Take away the stellar mass from the cold gas.
        self.cold_gas_mass -= (stars_formed + reheated_mass)

        if self.cold_gas_mass < 0:
            print('Subhalo {} has used up too much cold gas'.format(self.subhalo_ID))

        self.SFR = stars_formed / dt

        return None
```

Appendix C

Additional Diagnostic Plots

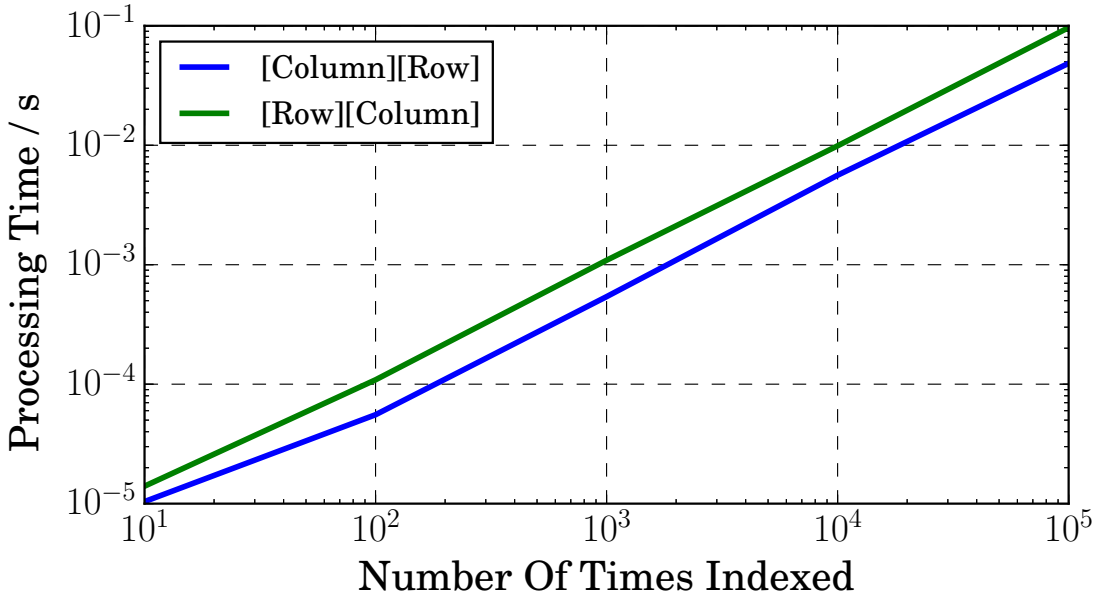

Figure C.1: The time taken in seconds for Python to extract values from a NumPy 2-D structured array. This test was performed on an array with 4 columns of different dtypes and a length of 10000 (a scaled down version of the array based code).