MilkyWay@home Python Package Documentation (*mwahpy* v1.0.0)

Tom Donlon

Department of Physics, Applied Physics and Astronomy
Rensselaer Polytechnic Institute

Contents

1 Introduction

1.1 What is MilkyWay@home?

At best, "MilkyWay@home" is a nebulous term that can refer to several different projects, softwares, and/or groups of people. At its core, MilkyWay@home is a crowd-sourced supercomputer that takes volunteer computing time to perform complex and time-consuming calculations that are designed to improve our understanding of the Milky Way Galaxy. At the moment, this includes both the MilkyWay@home Separation application and the MilkyWay@home N-body application. The MilkyWay@home project was developed at Rensselaer Polytechnic Institute under the direction of Dr. Heidi Jo Newberg, and has utilized countless hours of work from many graduate students and researchers. For more information regarding MilkyWay@home, I direct the user to the MilkyWay forums at `https://milkyway.cs.rpi.edu/milkyway/`. All of the MilkyWay@home code is publicly available at `https://github.com/Milkyway-at-home`.

While the Separation application is an important part of MilkyWay@home, the Separation application is difficult to apply to other projects. Additionally, the tools that I have developed for the Separation application are quite situation specific and not designed with user friendliness/adaptability in mind. For this reason, *mwahpy* does not include functionality for the Separation application. For Separation application tools and documentation, see the Separation application github page at `https://github.com/Milkyway-at-home/milkywayathome_client/tree/master/separation`.

What I refer to as "MilkyWay@home" in this document is actually the MilkyWay@home N-body software. This software was originally designed to generate model dwarf galaxies in the Milky Way gravitational potential and integrate them forwards in time. This evolved dwarf galaxy would then be compared to observations of stellar streams in the sky in order to determine a quality of fit. By optimizing over the parameters that were used to generate the dwarf galaxy, one could in theory determine the parameters of the progenitor dwarf galaxy. This could even be done for streams with no obvious progenitor (e.g. the Orphan Stream).

It was realized shortly thereafter that the N-body application could be used to integrate the orbits of any bodies in the Milky Way, not just those that were generated as part of a dwarf galaxy. The ability to insert list of bodies to integrate forwards in time was provided. Additionally, the ability to control the size of the timestep of the simulation, the orbit of a dwarf galaxy progenitor, the timescale of the simulation, the dwarf galaxy parameters, the underlying gravitational potential, and other MilkyWay@home functionality made it a versatile and robust tool for N-body integration.

While the MilkyWay@home N-body application is currently used by many graduate and undergraduate students at Rensselaer Polytechnic Institute, the MilkyWay@home team is making an effort to encourage more widespread usage of the MilkyWay@home N-body software. Our goal is to provide a user-friendly, comprehensive experience for N-body integration. This will decrease the time needed to get new students and researchers up to speed with N-body software, which has become commonplace in the dynamical astrophysics community.

We expect that the MilkyWay@home N-body application will become more widely used, and that proper documentation and accompanying tools will quickly become necessary. Additionally, new applications of the N-body software have expanded simulations for upwards of one million bodies. Analysis of these large simulations requires quick, robust software. It is for these reasons that *mwahpy* was developed.

1.2 What is *mwahpy*?

The MilkyWay@home N-body software has been used by the Galactic dynamics research group at Rensselaer Polytechnic Institute for years. As each new student comes into contact with the software, each one has had to develop their own tools for analyzing the outputs of the software. Each individual software has its own set of bugs, sets the student back a few weeks while developing their own tools, and by not using a standardized set of analysis software we open up the group to problems with sharing data and code. Additionally, MilkyWay@home is written in **C**, a language which can be daunting to the typical undergraduate student (and is often unecessarily unwieldly if you are just trying to cut data or make a plot). My goal to alleviate these issues was a compilation of the tools that I had built in python and tested over my tenure as an undergraduate (and improved during my time as a graduate student) in the MilkyWay@home dynamics group. This became the *mwahpy* package.

Before we talk about what *mwahpy* is, it is good to clarify what *mwahpy* is not. This package is not able to do N-body integration. Things like generating dwarf galaxies, running simulations, and performing routines on the MilkyWay@home supercomputer are to be delegated to the proper MilkyWay@home software. Additionally, I would like to express that because the Separation application is interacted with by many fewer people than the N-body application, *mwahpy* will not support Separation application functionality.

There is plenty that *mwahpy* is meant to be used for, though. At its very base, *mwahpy* is a python package that is designed to easily and quickly read in N-body data ouput from the MilkyWay@home N-body software. Once read in, it is fairly simple to cut the data, plot the data, and save the data in a variety of formats. This package is also able to output data in a format that is readable by the MilkyWay@home N-body software. The software will automatically calculate complicated values such as proper motion and energies as needed, instead of computing everything up front. The major benefit of these routines is that new users can be confident that the values that are produced by *mwahpy* have been tested over several years and are known to be accurate.

There are several associated auxiliary packages in *mwahpy* as well, such as the coordinate transformation subpackage and the orbit fitting subpackages. These are provided in the package because while working with N-body simulations, I often ran into situations where I used the functionality provided in the auxiliary subpackages. The auxiliary subpackages are less streamlined and complete compared to the main subpackages, but are still fairly well tested and can be trusted.

The code for *mwahpy* is publically available and can be found at `https://github.com/thomasdonlon/mwahpy`. Not only is collaboration on the code allowed, it is encouraged! If you would like functionality added to *mwahpy* or if you find any bugs in the code, you can either leave a comment on the github page or you can write the code yourself and make a pull request. This code is still being actively maintained, and I plan on eventually getting around to any bug fixes or desired functionality that are brought to my attention.

1.3 What can *mwahpy* do?

The following functionality is provided in *mwahpy*:

- Easily & quickly read in data from a MilkyWay@home N-body *.out* file
- MilkyWay@home N-body output only provides 3D Cartesian positions and velocities, as well as mass (in MilkyWay@home structural units). The *mwahpy* package converts this data into many other useful forms, such as R.A. & Dec., Galactic longitude and latitude, angular momenta about the Galactic center, line-of-sight velocity, proper motions, and others.
- Make cuts based on any of *mwahpy*'s supported values, cut the data based on individual components of the data, or subsample the data randomly or symmetrically.
- Quickly plot N-body data in any of the supported values
- A variety of coordinate transformations for typical coordinate systems used in Galactic astronomy
- Two separate orbit fitting routines that have been used in publications

The proper syntax, usage, and details of each of the above capabilities will be outlined in the following sections.

2 Installing *mwahpy*

For a typical user, installing *mwahpy* is as simple as installing any other python package. For linux machines, go to your terminal and type

Code Block 1: Installing *mwahpy*

```
>>> pip3 install mwahpy
```

Alternatively, you can also install *mwahpy* using

Code Block 2: Installing *mwahpy*

```
>>> python3 -m pip install mwahpy
```

It may be necessary to install *mwahpy* only in your user directory due to user access restrictions. **It is strongly recommended you do not use sudo to install python packages.** Using **sudo** makes it difficult to track who has the permissions to use what package on your machine, and on machines with multiple users, it can mean you have to install it for each user individually anyways. Misuse of **sudo** can also result in accidentally using a different version of *mwahpy* than intended, particularly after updates. To install *mwahpy* for only the active user, you can instead type

Code Block 3: Installing *mwahpy*

```
>>> pip3 install mwahpy --user
```

or

Code Block 4: Installing *mwahpy*

```
>>> python3 -m pip install mwahpy --user
```

Any of these lines of code will install *mwahpy* on your machine, as well as install all of the prerequisite packages needed for *mwahpy* to work.

In order to use *mwahpy* after it is installed, simply import the subpackages as you would any other other package in your python script. For example, the following lines are all syntactically acceptable imports:

Code Block 5: Importing *mwahpy*

```
import mwahpy.output_handler
from mwahpy.data import Data
from mwahpy.coords import *
```

It should be noted that *mwahpy* is only built for and maintained for python v3.2.3 and above. You should get an error when installing if your python installation is not recent enough for *mwahpy* . Depending on your installation, you may instead have to type `>>> python -m pip install mwahpy` instead of `python3`, or use `pip3` instead of `pip`. If you are experiencing errors, asking questions on the *mwahpy* github page or trying different permutations of these installation methods are recommended.

If you wish to install *mwahpy* for development, then you should clone the most recent *mwahpy* github repository (found at `https://github.com/thomasdonlon/mwahpy`). Feel free to make your own fork of the master branch if that's what you would prefer. Then, you should download the source code wherever you wish on your machine. If you want to import mwahpy into a python script after doing this, you will have to add a couple extra lines to the import portion of your script:

Code Block 6: Using *mwahpy* as a developer

```
import sys
sys.path.insert(1, '/home/path/where/init.py/is/located/')
import mwahpy_subpackage
```

In this case, you would replace the path with the path to your **init.py** file is located (it usually ends with `.../mwahpy/mwahpy/`), and you would replace `import mwahpy_subpackage` with whatever subpackage you wish to use (for example, `import output_handler as oh`).

If you then wish to add, remove, or change code in your distribution of *mwahpy*, that can be done easily in the folder where you cloned *mwahpy*. Any changes that you feel are beneficial to the package should be sent as a pull request to the master branch.

3 Core Functionality

WARNING:

MilkyWay@home uses a right-handed Galactic Cartesian coordinate system. This is defined as positive X being in the direction of the Sun towards the Galactic center, positive Y being in the direction of the disk spin at the location of the Sun, and positive Z being in the direction of the right-handed cross product X×Y (often called the "Galactic north"). In our coordinate system, the Sun is located at the position (X,Y,Z) = (-8,0,0).

This is in contrast to many other Galactic scientists, who prefer a left-handed coordinate system where the X-axis is flipped and the Sun is located at (X,Y,Z) = (8,0,0). In a left-handed coordinate system the physical interpretation of certain quantities are not always clear (such as the right-handed angular momentum cross product). Many coordinate transformations in *mwahpy* allow for left-handed coordinates. However, be aware that by default, *mwahpy* (and MilkyWay@home) output is right-handed.

3.1 The *Timestep* Class

The `Timestep` class is the heart of *mwahpy*, and it's where the majority of the important and useful calculations in the package are performed. An instance of the `Timestep` class represents a single timestep of an N-body simulation. In other words, a `Timestep` instance is the data from a single MilkyWay@home `.out` file. The code for the `Timestep` class can be found in *mwahpy*'s `timestep.py` file.

3.1.1 Initializing a *Timestep*

The most basic usage of `Timestep` object is a blank instance, to which you can then manually add data:

Code Block 7: Blank Timestep

```
import numpy as np
from mwahpy.timestep import Timestep

t = Timestep()

t.id = np.array([0, 1, 2])
t.x = np.array([10, 50, 100])
t.y = np.array([12, 15, 17])
t.z = np.array([1, 2, 3])
t.vx = np.array([13, 183, 102])
t.vy = np.array([0, 50, 180])
t.vz = np.array([23, 69, 12])
t.mass = np.array([1, 1, 1])
```

These 8 values are all that you need to specify for the `Timestep` class to do its job. These 8 values (ID, x, y, z, vx, vy, vz, and mass) will be referred to as *"provided values"*. This is in contrast to the "*supported values*", which are any values that *mwahpy* can calculate for you (a full list of the supported values is provided in Section 6.1). In the `Timestep` implementation, all of the particle IDs, x positions, etc. are stored in order as a `numpy` array of those values. As such, the *mwahpy* package heavily utilizes features of the `numpy` package, and some knowledge of `numpy` can be helpful for those working with *mwahpy*.

If you wish to print the information of a single particle, you can do so with `printParticle` by specifying the ID of the particle you are interested in:

Code Block 8: Printing data for a single particle

```
>>> t.printParticle(1)
Printing data for Particle 1:
(id:1, x:50, y:15, z:2, vx:183, vy:50, vz:1, mass:1, )
```

WARNING:

The provided values should **always** be `numpy` arrays of identical length. Attempting to calculate values, plot data, or write out data when the provided values have mismatched length or are not `numpy` arrays will typically result in an error. It is strongly recommended that the user uses the built-in `Timestep` methods for cutting or adding data instead of doing it manually.

From this point forward, the user can ask for any *mwahpy* supported value, and it will be calculated for them. For example, if you wanted the line-of-sight velocities of the particles, you could type:

Code Block 9: Calculating supported values

```
>>> t.vlos
array([  7.894394  , 235.8965911 , 128.90974589])
```

Note that if you call `t.printParticle` again (this time using the optional `dec` argument to shorten the output), it now shows more data for the particle!

Code Block 10: Printing data for a single particle

```
>>> t.printParticle(1, dec=2)
Printing data for Particle 1:
(id:1, x:50, y:15, z:2, vx:183, vy:50, vz:69, mass:1, msol:222288.47,
l:0.29, b:0.04, ra:266.54, dec:-28.67, dist:59.94, lx:935, ly:3084,
lz:-245, lperp:3222.62, ltot:3231.92, r:52.24, R:52.2, vlos:235.9,
vgsr:247.14, rad:192.15, rot:-4.69, distFromCOM:52.24, )
```

This was all calculated in the background when you asked the package to calculate `vlos` for this `Timestep`. Due to the overhead on some calculations, the more computationally complex calculations are avoided until the user requests those values. This is explained in more detail in Section **??**.

The `Timestep` class has one very unique property: attributes of a `Timestep` instance can be accessed via the usual method, or as the key to a dictionary. In fact, comparing the two methods shows that these two actions produce equivalent results.

Code Block 11: Accessing Timestep attributes

```
>>> t.x #accessing data as an attribute
array([ 10,  50, 100])
>>> t['x'] #accessing data as a dict key
array([ 10,  50, 100])
>>> np.all(t.x == t['x'])
True
>>> t.x[0] = 1 #changing the value for Particle 0's x position
>>> t.x[0] #the same value is accessed by both methods
1
>>> t['x'][0]
1
>>> t.x[0] == t['x'][0]
True
```

At first this property may seem confusing and not particularly useful. What is the point of being able to access the same data in two different ways? It turns out that adding this functionality to `Timestep` allows for some rather powerful behavior. Of the greatest importance to the typical user is the implementation of the iterator for `Timestep`, which iterates over the *keys* of the class. In this case, that has been implemented as the names of all of the supported values (that have been calculated so far!).

Code Block 12: Iterating over a Timestep

```
>>> outstr = ''
>>> for key in t:
...     outstr += (key + ',␣')
>>> print(outstr)
id, x, y, z, vx, vy, vz, mass, msol, l, b, ra, dec, dist, lx, ly,
lz, lperp, ltot, r, R, vlos, vgsr, rad, rot, distFromCOM,
>>> outstr = ''
>>> for key in t:
...     outstr += (str(round(t[key][0],2)) + ',␣')
>>> print(outstr)
0, 10, 12, 1, 13, 0, 23, 1, 222288.47, 0.88, 0.06, 266.86, -28.15, 21.66,
276, 217, -156, 351.09, 384.19, 15.65, 15.62, 7.89, 21.43, 9.77, -9.99,
15.65,
```

This may not seem like a big deal at first. However, if one tries to reproduce this behavior for a class that doesn't have the property `c.x == c['x']`, then you will quickly run into several problems. I expect that the typical reader would try to iterate over the class' built-in attribute dictionary, which is what I initially tried. What happens if you want to add attributes to the class that cannot be iterated over at the same time as your arrays, such as a single identifying string `c.name`? I suggest that the reader play around with this idea on their own if they are so inclined.

3.1.2 Reading In & Writing Out Data

So far, we know how to initialize a `Timestep`, how to add data to it manually, and how to access this data. This is all fine and good, but most of the time a user will be interested in using data that has already been generated by the MilkyWay@hoe N-body application. Conveniently, there is a *mwahpy* function built specifically for this in the `mwahpy.output_handler` subpackage:

Code Block 13: Reading in a Timestep

```
>>> import mwahpy.output_handler as oh
>>> t = oh.readOutput('<path/to/mwahpy>/test/test.out')
Reading in data from ../test/test.out...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> t.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562, 3.57179938])
```

The `oh.readOutput` function takes in the path to a MilkyWay@home `.out` file and outputs a `Timestep` instance of the data from that file. In the above code block, we read in the test `,out` file, which is provided in `.../mwahpy/test/`. The `oh.readOutput` function provides you with a progress bar, which is useful for large files. In this case, the progress bar finished almost instantly since the file was small. Typically, the progress bar will reach 100%. The `oh.readOutput` function also provides you with the number of particles that were read in from the file, and a brief update on when it is converting the data (again, useful for very large files). From this point forward, you can perform any normal `Timestep` operations on the new `Timestep` object.

The `oh.readOutput` function has been heavily optimized, and operates extremely quickly. Even for files with millions of values, reading in the data only takes a few seconds on most machines. If you are working with N-body output, it is strongly recommended that you use this function to initialize your `Timestep` instances.

The *mwahpy* package also offers a few different options for printing out `Timestep` data to a file. Notably, *mwahpy* can print out the data to a `.csv` file,

Code Block 14: Writing out a Timestep

```
>>> oh.makeCSV(t, '/path/to/my/file.csv')
Writing header...
Printing data...
Timestep output to /path/to/my/file.csv
```

Saving the data to a `.csv` saves all of the supported values that has been calculated so far. Alternatively, you can write out data from a `Timestep` to a MilkyWay@home `.in` file,

Code Block 15: Writing out a Timestep

```
>>> oh.makeNbodyInput(t, '/path/to/my/file.in')
Writing Timestep as N-body input to /path/to/my/file.in...
done
```

Note that a MilkyWay@home `.in` file will only include the data about the 8 provided values, and not any of the other supported values. This is due to the format that MilkyWay@home requires readable files to be in. Any file that you generate with `makeNbodyInput` is immediately ready to be used as the manual body input for a MilkyWay@home N-body simulation.

If a user wishes to suspend a `Timestep` object for later use, I recommend the `pickle` package (`https://docs.python.org/3/library/pickle.html`). The `pickle` package allows for object serialization and saving out/reading in arbitrary objects as binary code.

3.1.3 Manipulating Data

Aftet N-body data has been read in, it is useful to be able to only select the data that you are interested in. There are a few different routines for this, namely `subsetRect()` and `subsetCirc()`. These methods are n-dimensional cutting routines. For example, say that you took the test data and only wanted particles with a positive X value.

Code Block 16: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subsetRect(['x'], [(0,1000)]) #[axes], [(lower lim, upper lim)]
>>> tcopy.x
array([4.53813066, 0.11393187, 0.38487162, 0.35803147, 0.3565866 ,
       0.52268562, 3.57179938])
>>> tcopy.y
array([-0.05701174,  0.39540971, -0.43899208,  0.14577695,  0.0082146 ,
       -0.34261058,  0.53908404])
```

Note that `subsetRect()` has cut along **all** axes, not only the axis that we instructed it to cut along. As such, it is easy to see that `tcopy.y` has the same length as `tcopy.x`. It is also important to note that we made a copy of `t` before using `subsetRect()`, because the `subset` methods treat the `Timestep` object as directly mutable.

While cutting the data, we provided `1000` as an upper value for X. This was because `subsetRect()` requires both a lower and upper limit for each axis when cutting the data. To get around this, we just chose some arbitrarily large value that would not cut any data off at the high end of X values.

You can also cut on multiple axes with `subsetRect()`. Say that you wanted to only include particles with positive X and negative Y values:

Code Block 17: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subsetRect(['x','y'], [(0,1000),(-1000,0)])
>>> tcopy.x
array([4.53813066, 0.38487162, 0.52268562])
>>> tcopy.y
array([-0.05701174, -0.43899208, -0.34261058])
```

As all methods in `Timestep`, these routines can be performed on any supported values **after they have been calculated**. Attempting to cut on line-of-sight velocity before it has been calculated, for example, would result in a `KeyError`.

Similar results can be obtained from `subsetCirc`. If we wanted all particles with X values within 1 kpc of the Galactic center:

Code Block 18: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subsetCirc(['x'],[1],[0])
>>> tcopy.x
array([-0.59395581, -0.00613003,  0.11393187,  0.38487162,  0.35803147,
        0.3565866 ,  0.52268562])
```

Or if we wanted all particles with X and Y values within 1 kpc of the Galactic center:

Code Block 19: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subsetCirc(['x', 'y'],[1,1],[0,0])
>>> tcopy.x
array([-0.00613003,  0.11393187,  0.38487162,  0.35803147,  0.3565866 ,
        0.52268562])
>>> tcopy.y
array([ 0.43348783,  0.39540971, -0.43899208,  0.14577695,  0.0082146 ,
       -0.34261058])
```

This routine is a bit of a misnomer, as it can be extended to any n-dimensional spheroidal volume of your phase space. For example, say that we wanted particles located within a spheroid defined with an semiaxis of length 1 on the X-axis, a semiaxis of length 2 on the Y-axis, and a semiaxis of length 5 on the Z-axis, centered at (X, Y, Z) = (0, 1, 3). This can be done with

Code Block 20: Cutting data in a Timestep

```
>>> tcopy = t.copy()
>>> tcopy.subsetCirc(['x', 'y', 'z'],[1,2,5],[0,1,3])
>>> tcopy.x
array([-0.59395581, -0.00613003,  0.11393187,  0.35803147,  0.3565866 ])
>>> tcopy.y
array([0.87634781, 0.43348783, 0.39540971, 0.14577695, 0.0082146 ])
>>> tcopy.z
array([ 0.55739028, -0.18463063,  0.32116868,  0.17315727, -0.09600029])
```

Unlike `subsetRect()`, this routine becomes somewhat unclear when you begin mixing values with different units. In that case, it is suggested that you either make multiple circular cuts where the units of each value match, or make rectangular cuts instead.

Other data manipulation routines include sampling and splitting `Timesteps`. There are three types of sampling that *mwahpy* supports: manual, incremental, and random sampling, shown below.

Code Block 21: Sampling a Timestep

```
>>> #manual sampling
>>> tcopy = t.copy()
>>> tcopy.take([0,2,3,6,9]) #take particles at indices 0, 2, 3, 6, and 9
>>> tcopy.x
array([ 4.53813066, -1.41000385, -0.00613003,  0.35803147,  3.57179938])

>>> #incremental sampling
>>> tcopy = t.copy()
>>> tcopy.subsample(2) #take every 2nd particle, starting at 0
>>> tcopy.x
array([ 4.53813066, -1.41000385,  0.11393187,  0.35803147,  0.52268562])

>>> #random sampling
>>> tcopy = t.copy()
>>> tcopy.randSample(5) #randomly take 5 particles from the data
>>> tcopy.x
array([ 4.53813066, -0.59395581,  0.38487162, -0.00613003,  0.35803147])
>>> #run again to demonstrate randomness
>>> tcopy = t.copy()
>>> tcopy.randSample(5)
>>> tcopy.x
array([ 0.52268562, -0.59395581,  0.3565866 , -0.00613003,  0.11393187])
```

It is easy to split a `Timestep` by providing the index at which you want to split:

Code Block 22: Splitting a Timestep

```
>>> tcopy, tcopy2 = t.split(4) #split at index 4
>>> tcopy.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003])
>>> tcopy2.x
array([ 0.11393187,  0.38487162,  0.35803147,  0.3565866,   0.52268562,
3.57179938])
```

Finally, if you would like to add two `Timestep` instances together, use the `appendTimestep` method:

Code Block 23: Adding Timesteps together

```
>>> tcopy.appendTimestep(tcopy2) #append two Timesteps
>>> tcopy.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])

>>> tcopy, tcopy2 = t.split(4)
>>> tcopy.appendPoint(tcopy2, n=5) #add tcopy2 index 5 to tcopy
>>> tcopy.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  3.57179938])
```

Note that actions like just replacing a single `x` value in a `Timestep` does not update the rest of the calculated values in the `Timestep`. That `Timestep` will need to be updated manually using `Timestep.update()`. This method will update all values associated with the `Timestep` in order to keep them in sync with the provided values.

WARNING:

Note that if the `autoUpdate` flag is turned on (it is on by default), *mwahpy* functions will automatically keep the provided and calculated values in sync. However, if you manually update the provided values after the calculated values have been computed, you will need to run `Timestep.update()`.

Code Block 24: Updating a Timestep

```
>>> t = oh.readOutput('../test/test.out')

Reading in data from ../test/test.out...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> t.x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
>>> t.distFromCOM
Calculating basic values...
array([4.53771154, 1.20665236, 2.01267674, 0.48791978, 0.51046082,
       0.99322694, 0.39540509, 0.34974823, 0.85795412, 3.73818636])
>>> t.x[0] = 0
>>> t.distFromCOM
array([4.53771154, 1.20665236, 2.01267674, 0.48791978, 0.51046082,
       0.99322694, 0.39540509, 0.34974823, 0.85795412, 3.73818636])
>>> t.update()
>>> t.distFromCOM
array([0.80324921, 1.29094239, 2.05506687, 0.36725208, 0.51176956,
       0.99371407, 0.34057982, 0.29000981, 0.81102414, 3.37390626])
```

You should now have a fundamental undertsanding of how to use the `Timestep` class. The rest of the methods that were not covered in this section can be found in Section 6.1. Continue reading for information about handling entire simulations with multiple timesteps and plotting timesteps and simulations.

3.2 The *Nbody* Class

The `Nbody` class is one rung above the `Timestep` class in the *mwahpy* hierarchy; one `Nbody` instance is composed of many individual `Timestep` instances. This is useful when you need to compare many timesteps from a single MilkyWay@home simulation. The simplest example of a `Nbody` instance is the default instance:

Code Block 25: Initializing an Nbody

```
>>> from mwahpy.nbody import Nbody
>>> n = Nbody()
```

From here, you can add a `Timestep` to the dictionary in the `Nbody` instance:

Code Block 26: Initializing an Nbody

```
>>> import mwahpy.output_handler as oh
>>> t = oh.readOutput('<path/to/mwahpy>/test/test.out')
Reading in data from ../test/test.out...
[-------------->      ] 73%
10 objects read in
Converting data...done
>>> n[1] = t
>>> n[1].x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
```

Note that the Nbody instance itself does not carry the Timestep data, but just points to it.

Code Block 27: An Nbody does not carry simulation data

```
>>> n.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nbody' object has no attribute 'x'
```

In this case, the Timestep t still exists on its own, but the Nbody instance forms a nice functional container for it. This can be seen by altering t, as the change is also present when accessing the data through the Nbody. Thus, the Nbody structure does not copy the Timestep data, it just references it.

Code Block 28: An Nbody points to Timestep data

```
>>> t.x[0] = 0
>>> t.x
array([ 0.        , -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
>>> n[1].x
array([ 0.        , -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
```

The Timestep does recognize that it belongs to an Nbody class however, and it remembers its designated time and the Nbody object which it belongs to:

Code Block 29: Accessing an Nbody

```
>>> t.time
1
>>> t.nbody
<mwahpy.nbody.Nbody object at 0x[...]>
```

WARNING:

Currently, `Timestep` instances only weakly retain information about parent `Nbody` instances. If a `Timestep` is added to a second `Nbody`, it will replace its `self.time` and `self.nbody` from the first `Nbody` instance with the new information from the second `Nbody`. This is made more confusing by the fact that multiple `Nbody` instances can point to the same `Timestep`, or a single `Nbody` can point to the same `Timestep` more than once. In these cases, the data can be properly accessed through the `Nbody` objects, but is not guaranteed to refer back to the desired information when accessed from the `Timestep` object.

3.2.1 Reading In & Writing Out Data

As with the `Timestep` class, the `Nbody` class is typically not created manually, but is initialized by reading in data. This is also done through the `mwahpy.output_handler` subpackage by providing a folder with the `oh.readFolder()` function.

Code Block 30: Reading data into an Nbody

```
>>> n = oh.readFolder('<...path/to/mwahpy>/mwahpy/test/nbody_test/')
Reading in data from directory ../test/nbody_test/...

Reading in data from ../test/nbody_test/1...
[-------------->     ] 73%
10 objects read in
Converting data...done

Reading in data from ../test/nbody_test/2...
[-------------->     ] 73%
10 objects read in
Converting data...done

Reading in data from ../test/nbody_test/3...
[-------------->     ] 73%
10 objects read in
Converting data...done
>>> n[1].x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
>>> n[2].x
array([21.51019042, 17.97537383, 15.99136357, 12.30622142, 10.80896895,
       10.56226125, 12.94243578, 11.16187291, 11.17781449,  8.04042707])
```

This data has been read in from the `../mwahpy/test/nbody_test/` folder that is included in *mwahpy* .

WARNING:

When reading in a `Nbody` from simulation data, the **only** files in the provided folder must be MilkyWay@home `.out` files, and **must** be named only their time **without the `.out` extension**. This is the default output scheme of MilkyWay@home N-body.

Currently, there is not support for writing out `Nbody` data. This can be done fairly simply by iterating over the `Nbody` (see the next subsection for details) and writing out each `Timestep`. It is also not recommended that a user writes out `Nbody` data using the `pickle` package.

3.2.2 Functionality of *Nbody* Objects

`Nbody` objects have a few nice capabilities that allow for robust treatment of simulation data. To begin, iterating over an `Nbody` object returns its component `Timestep` objects sorted by time.

Code Block 31: Iterating over an Nbody

```
>>> for ts in n:
...     print(ts.time)
1
2
3
```

Being able to read in and iterate over large groups of `Timestep` instances from the same simulation is much easier than going through each one individually, not to mention it is often faster.

The times of each timestep can be scaled in order to reflect the actual physical simulation time of each timestep using the `scaleTimes()` method. For example, if one timestep is equal to 0.5 Myr in physical time, then we can scale the corresponding timesteps. We see that the original information is retained, although renamed:

Code Block 32: Scaling Nbody times

```
>>> n.scaleTimes(0.5)
>>> for ts in n:
...     print(ts.time)
0.5
1.0
1.5
>>> n[0.5].x
array([ 4.53813066, -0.59395581, -1.41000385, -0.00613003,  0.11393187,
        0.38487162,  0.35803147,  0.3565866 ,  0.52268562,  3.57179938])
```

3.3 Plotting

Text

4 Auxiliary Subpackages

The *mwahpy* package comes with a few auxiliary subpackages with additional content that the typical user might find useful. While the `mwahpy.orbit_fitting` and the `mwahpy.orbit_fitting_gc` subpackages have been used in academic contexts, the packages have not yet been refactored for general use. After the packages have been made consistent with the rest of the *mwahpy* code standards, then documentation for those subpackages will be released.

4.1 Coordinate Transformations

The coordinate transformation package `mwahpy.coords` is made up of many first and second-order transformations for position, velocity, and other values. A complete list of the routines and functions provided in this package is given in Section 6.5.

This subpackage contains most of the coordinate transformations that a typical person will need to do Galactic astronomy, excluding some on-sky transformations (such as R.A., Dec. to l, b) that are better accessed through the `astropy` python package.

4.2 Orbit Fitting

The subpackages `mwahpy.orbit_fitting` and `mwahpy.orbit_fitting_gc` are both still under development. Once the code has been refactored and brought to an acceptable level of quality, the documentation for these subpackages will be provided.

5 Flags & Settings

5.1 *flags*

The flags and settings for *mwahpy* can be found in `flags.py`. Currently, there are only a few options, but more may be added in the future.

- `mwahpy.flags.verbose` (default value = 1)

  This flag dictates how much is printed out into the terminal when *mwahpy* code is running. If turned on, functions are much noisier, and will update you on their current activity. If this is turned off, the overall functionality of *mwahpy* will remain unchanged, but the terminal will be less informative.

- `mwahpy.flags.progressBars` (default value = 1)

  If this flag is turned on, *mwahpy* will use progress bars when performing certain actions (such as reading in files) in order to avoid long wait times without updating the terminal. If turned off, *mwahpy* functionality will not be changed; however, be aware that you may occasionally experience long periods of time where your terminal is silent.

- `mwahpy.flags.autoUpdate` (default value = 1)

  This flag determines whether or not to automatically update an entire `Timestep` whenever the provided values for that `Timestep` change **due to a *mwahpy* method or function**. Having this flag turned on will keep things like the center of mass and momentum accurate even after changing the constituent data, and will keep all of the calculated values in sync with the provided values throughout *mwahpy* functions.

  If you are frequently updating certain values in a `Timestep`, this flag can occasionally cause performance issues (if this flag is causing performance issues, that's probably a sign that there's a better way to do what you are trying to do). It is strongly recommended that this flag is only turned off if the user understands what updating means, how to update the data manually, and what data needs to be updated.

5.2 *mwahpy_glob*

Constants and functions that are universally accessed in *mwahpy* are stored in `mwahpy_glob.py`.

- `mwahpy.mwahpy_glob.G`

  Newton's gravitational constant is provided in `astropy` units of m^3/kg s^2.

- `mwahpy.mwahpy_glob.structToSol`

  This is the conversion factor between MilkyWay@home structural masses and solar masses. Equal to 222,288.47 solar masses/structure unit.

- `mwahpy.mwahpy_glob.kmsToKpcgyr`

  The conversion factor from km/s to kpc/Gyr. Roughly equal to unity (1.023).

- `mwahpy.mwahpy_glob.kpcgyrToKms`

  The conversion factor from kpc/Gyr to km/s. Roughly equal to unity (0.978).

- `mwahpy.mwahpy_glob.progressBar(value, endvalue, bar_length=20)`

  `value` (int): the value that is being iterated

  `endvalue` (int): the value at which the loop terminates

  `bar_length` (int): How many characters long the progress bar is in the terminal output

  Adapted from `https://stackoverflow.com/questions/6169217/replace-console-output-in-python`. This function can be placed inside a loop and will output a nice progress bar in the terminal, provided you know the end value that the loop should terminate on.

- `mwahpy.mwahpy_glob.fileLen(f)`

  `f` (str): the filename (path)

  Given a file, this function will determine the number of lines in that file. Useful for implementing `mwahpy.mwahpy_glob.progressBar()` while reading in files.

5.3 *pot*

Constants and functions related to the Milky Way's gravitational potential are found in `mwahpy.pot.py`. The default potential that is used in *mwahpy* is the Orphan Stream Fit #5 Potential from Newberg et al. (2010). This potential is very similar to Law et al. (2005).

- `mwahpy.pot.m_bulge`

  The mass of the bulge in `astropy` solar masses. Equal to 3.4e10 $M_\odot$

- `mwahpy.pot.m_disk`

  The mass of the disk in `astropy` solar masses. Equal to 1.0e11 $M_\odot$

- `mwahpy.pot.v_halo`

  The Milky Way halo dispersion in `astropy` km/s. Equal to 74.61 km/s.

- `mwahpy.pot.pot_bulge`

  The gravitational potential of the Milky Way due to the bulge. Given as a `galpy potential` object.

- `mwahpy.pot.pot_disk`

  The gravitational potential of the Milky Way due to the disk. Given as a `galpy potential` object.

- `mwahpy.pot.pot_halo`

  The gravitational potential of the Milky Way due to the halo. Given as a `galpy potential` object.

- `mwahpy.pot.energy_offset`

  Due to the logarithmic halo potential term in our gravitational potential for the Milky Way, the potential does not disappear as one travels infinitely far from the center of the potential. A side effect of this type of halo potential is that it is not clear what structures are bound or unbound to the potential, as the zero-point of the energy is not representative of the escape velocity. This adjusts the calculated potential energy so that it is consistent with Donlon et al. (2019), and so that clearly bound structures will not have positive energy values. The default value is -60000 km^2/s^2.

- `plotPotential(potential, Rrange=[0.01,10.])`

  `potential` (`galpy potential` object): the potential that you wish to graph

  `Rrange` (list of floats): the Galactocentric cylindrical radius of the points that you wish to evaluate the potential at

  This method will generate a figure with the rotation curve of the potential in the plane of the Milky Way.

6 Functions & Methods

Text

6.1 *timestep*

Text

6.2 *nbody*

Text

6.3 *output_handler*

Text

6.4 *plot*

Text

6.5 *coords*

Text

6.6 *orbit_fitting*

The subpackage `mwahpy.orbit_fitting` is still under development. Once the code has been refactored and brought to an acceptable level of quality, the documentation for this subpackage will be provided.

6.7 *orbit_fitting_gc*

The subpackage `mwahpy.orbit_fitting_gc` is still under development. Once the code has been refactored and brought to an acceptable level of quality, the documentation for this subpackage will be provided.