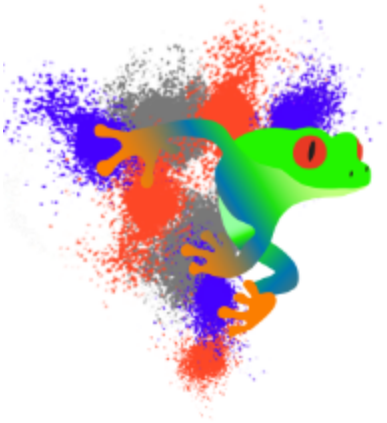
TreeFrog Documentation

Release 1.21

Pascal Jahan Elahi, Rhys Poulton, Rodrigo Canas

Jan 03, 2021

CONTENTS:



TreeFrog is a C++ halo merger tree builder using MPI and OpenMP APIs. The repository also contains several associated analysis tools in python, example configuration files and analysis python scripts (and sample jupyter notebooks).

If you are using **TreeFrog** please cite the following paper, which describe the code in full:

```
@ARTICLE{10.1017/pasa.2019.18,
    author = {{Elahi}, Pascal J. and {Poulton}, Rhys J.~J. and {Tobar}, Rodrigo J. and
    {Ca{\~n}as}, Rodrigo and {Lagos}, Claudia del P. and {Power}, Chris and
    {Robotham}, Aaron S.~G.},
    title = "{Climbing halo merger trees with TreeFrog}",
    journal = {\pasa},
    keywords = {dark matter, methods: numerical, galaxies: evolution, galaxies: halos,
↪ Astrophysics - Instrumentation and Methods for Astrophysics, Astrophysics -␣
↪Astrophysics of Galaxies},
    year = "2019",
    month = "Jan",
    volume = {36},
    eid = {e028},
    pages = {e028},
    doi = {10.1017/pasa.2019.18},
    archivePrefix = {arXiv},
    eprint = {1902.01527},
    primaryClass = {astro-ph.IM},
    adsurl = {https://ui.adsabs.harvard.edu/abs/2019PASA...36...28E},
}
```

An online entry can also be found at NASA's ADS service.

CHAPTER

ONE

GETTING TREEFROG

TreeFrog is currently hosted in GitHub. To get a copy you can clone the repository:

```
git clone https://github.com/pelahi/TreeFrog
```

TreeFrog's compilation system is based on cmake. `cmake` will check that you have a proper compiler (anything supporting C++14 or later should do), and scan the system for all required dependencies.

To compile **TreeFrog** run (assuming you are inside the `TreeFrog` directory already):

```
$> mkdir build
$> cd build
$> cmake ..
$> make
```

With `cmake` you can also specify additional compilation flags. For example, if you want to generate the fastest possible code you can try this:

```
$> cmake .. -DCMAKE_CXX_FLAGS="-O3 -march=native"
```

You can also specify a different installation directory like this:

```
$> cmake .. -DCMAKE_INSTALL_PREFIX=~/my/installation/directory
```

Other `cmake` options that can be given in the command-line include:

A list of compile time options is found below in *Compilation Options*.

1.1 Requirments

TreeFrog depends on:

- GSL - the GNU Scientific Library
- **NBodylib** - a internal scientific library included with **TreeFrog** as a submodule.

1.1.1 Optional requirements

For parallel use may need the following libraries are required for compilation depending on the compilation flags used:

- MPI - the Message Passing Interface (version 1.0 or higher). Many vendor supplied versions exist, in addition to excellent open source implementations, e.g. Open MPI, MPICH or LAM.
- OpenMP - API, generally included with many compilers

TreeFrog also can output in a variety of formats: ASCII, and HDF. HDF can be enabled and disabled, and requires libraries.

- Hiearchical Data Format (HDF) - self describing data format.

1.2 Compilation Options

These can be passed to `cmake`

External library flags

- **Parallel APIs can be enabled by setting**
  - **For MPI**

    `TF_MPI`: boolean to compile with MPI support
    `MPI_LIBRARY`: specify library path to MPI
    `MPI_EXTRA_LIBRARY`: Extra MPI libraries to link against
  - **For OpenMP**

    `TF_OPENMP`: boolean to compile with OpenMP support
    `OpenMP_CXX_FLAGS`: string, compiler flag that enables OpenMP
- **Enable input/output formats**
  - **For HDF**

    `TF_HDF5`: boolean on whether to include HDF support
    `HDF5_ROOT`: specify a local directory containing HDF library.
- **To set directories of required libraries**
  - **Set the directories of the following libraries**

    `GSL_DIR =`

Internal precision and data structure flags

- **Adjust precision in stored variables and calculations**
  - **all integers are 64 bit integers. Enable this if dealing with more than MAXINT total number of particles**
    `TF_LONG_INT`: boolean on whether to use long ints. Default is ON
  - **Use unsigned ids.** `TF_UNSIGNED_IDS`: boolean on whether to use unsigned (long) ints

Operation flags

- **Adjust TreeFrog operation**
  - **Assume halo indices and snapshot do not map to a halo ID and halo ID must be read from input.**
    `TF_HALOIDNOTINDEX`: boolean on whether ids and indices map. Default is OFF

Executable flags

- **Enable debugging** `DEBUG`: boolean on whether to run with debug flags on and no optimisation.

CHAPTER

TWO

USING TREEFROG

2.1 Running the code

TreeFrog is a stand alone executable. It can be run in serial, with OpenMP, or MPI APIs. A typical command to start the code looks like:

```
./treefrog < args >
```

When compiled with OpenMP, setting the environment variable `OMP_NUM_THREADS` will set the number of threads in the openmp sections.

With MPI using 8 MPI threads:

```
mpirun -np 8 ./treefrog < args >
```

where here we assume that the parallel environment uses the `mpirun` command to start MPI applications. Depending on the operating system, other commands may be required for this task, e.g. `srun` on some Cray machines.

The output produced by **TreeFrog** will consist of either a single ASCII file or multiple HDF5 files (on per snapshot). The information contained in it this output is a raw halo merger tree listing the connections halos have at one snapshot in time to later (or earlier) snapshots. More details of the output can be found in *Understanding and Analysing TreeFrog Output*.

2.1.1 Arguments

The code has several command line arguements. To list the arguments, type

```
./treefrog -?
```

The main arguments that can be passed are:

-i < file name of input file that contains list of files to be processed >

-s < number of files (snaphots) >

-I < input format [2 VELOCIraptor, 1 SUSSING, 3 nIFTy, 4 Void] >

-o < output base name >

-C < configuration file name (see *Configuration File*) >

Of these arguements, only an input file, number of files (snapshots) in input and an output name must be provided. In such a case, default values for all other confirguration options are used. We suggest you do NOT run the code in this fashion. Instead we suggest the code be run with at least a configuration file passed.

```
./treefrog -i input -s numsnaps -o output -C configfile.txt
```

This configuration file is an ascii file that lists keywords and values. A list of keywords, along with a description is presented below in *Configuration File*. A more typical command for a large cosmological simulation might be something like

```
export OMP_NUM_THREADS=4
mpirun -np 2 ./treefrog -i listofsnaphots -s 128 -o halotree -C configfile.txt >␣
↪treefrog.log
```

By default the code is works natively with output from the **VELOCIraptor** halo finder (which can be obtained via https://github.com/pelahi/VELOCIraptor-STF, and which has associated online documetation).

The code is able to process other types of halo finder input (see *AHF catalogue* for a description of running **TreeFrog** on output from AHF).

Finally, **TreeFrog** can be used to produce halo merger trees used as input for semi-analytic models of galaxy formation (see *Producing SAM digestible merger trees*).

2.1.2 Configuration File

An example configuration file can be found the examples directory within the repository (see for instance `sample`). This sample file lists all the options. *Only the keywords listed here will be used, all other words/characters are ignored.*

Warning: Note that if misspell a keyword it will not be used.

There are numerous key words that can be passed. Here we list them, grouped into several categories: *Outputs*, *Inputs*, *Particle ID options*, *Tree construction options*, *Merit options*, *Temporal linking options*, *Miscellaneous*, *MPI*.

Output related

`Output format = 2/0`

- Flag indicating whether output is 2 HDF, 0 ASCII

`Output data included = 1/0/2`

- Flag indicating whether to produce standard output, minial output, or extensive output. Standard includes merits. Extensive includes merits and nubmer of particles

Input related

`Input_tree_format = 2/1/3/4`

- Type of input halo catalog. 2 is VELOCIraptor input, 1 SUSSING, 3 nIFTY, 4 Void.

`VELOCIraptor_input_format = 2/0/1`

- Input format of a VELOCIraptor catalog, 2 HDF, 0 ASCII, 1 binary.

VELOCIraptor_input_field_sep_files = 0/1

- Whether VELOCIraptor catalog has separate files for field halos and subhalos.

VELOCIraptor_input_num_files_per_snap = 0/1

- Whether there is more than one file per VELOCIraptor catalog (if it was run in MPI mode)

Particle IDs options

How to handle particle ids which are used to cross correlate catalogs.

Max_ID_Value = 1073741824

- TreeFrog allocates array of size Max_ID_Value to cross correlate particles thus specify maximim ID and code will allocate an array of size max ID of either ints or long ints (depending on compilation options) to cross correlate. If value not set must set an id to index mapping.

Mapping= 0/1/-1

- Can construct a memory efficient ID to index map (computationally expensive but reduces) memory by prodiving a map or by having code produce a map. No map 0, memory efficient treefrog built map -1, user defined (must alter code) map 1

Tree construction options

Tree_direction = 1

- Integer indicating direction in which to process snapshots and build the tree. Descendant [1], Progenitor [0], or Both [-1].

Particle_type_to_use = -1

- Integer describing particle types to use when calculating merits. All [-1], Gas [0], Dark Matter [1], Star [4].

Default_values = 1

- Whether to use default cross matching & merit options when building the tree. 1/0 for True/False.

Merit options

Related to what merit function to use to define matches

Merit_type = 6

- **Integer specifying merit function to use. Several options available using variations of two specific merits:**
  - the shared number of particles $\mathcal{N}_{A_iB_j} = N^2_{A_i \bigcap B_j}/(N_{A_i}N_{B_j})$
  - the rank ordering of particles $\mathcal{S}_{A_iB_j,A_i} = \sum_l^{N_{A_i \cap B_j}} 1/\mathcal{R}_{l,A_i}$
- Optimal descendant tree merit is combination of both rank ordered in both directions [6], common (progenitor tree) merit in is using the shared merit [1].

Core_match_type = 2

- Integer flag indicating the type of core matching used. Off [0], core-to-all [1], core-to-all followed by core-to-core [2], core-to-core only [3].

Particle_core_fraction = 0.4

- Fraction of particles to use when calculating merits. Assumes some meaningful rank ordering to input particle lists and uses the first $f_{\rm TF}$ fraction.

Particle_core_min_numpart = 5

- Minimum number of particles to use when calculating merit if core fraction matching enabled.

Temporal linking options

Related to how code searches for candidate links across multiple snapshots.

Nsteps_search_new_links = 1

- Number of snapshots to search for links.

Multistep_linking_criterion = 3

- **Integer specifying the criteria used when deciding whether more snapshots should be searched for candidate links.**
  - **Descendant Tree**: continue searching if halo is: missing descendant [0]; missing descendant or descendant merit is low [1]; missing descendant or missing primary descendant [2]; missing a descendant, a primary descendant or primary descendant has poor merit [3].
  - **Progenitor tree**: continue searchign if halo is: missing progenitor[0]; missing progenitor or progenitor merit is low [1].

Merit_limit_continuing_search = 0.025

- Float specifying the merit limit a match must meed if using `Multistep_linking_criterion = 1 (progenitor) or 3 (descendant).`

MPI related options

Related to MPI options

Miscellaneous options

Miscellaneous options

Verbose = 0/1/2

- Indicates how verbose the code is while running. 0 is minimial, 1 verbose, 2 chatterbox.

2.2 Running on Other Catalogs

2.2.1 AHF catalogue

To process the ASCII output produced by the AHF halo finder, the mpi flag needs to be switched off and the flag that tells TreeFrog the ID's do not correspond to an index (as with AHF halo ID's) needs to be switched on. These flags are

```
TF_MPI:BOOL=ON
TF_HALOIDNOTINDEX:BOOL=OFF
```

There are several configuration options that must be set. The input format must be set appropriately.

```
Input_tree_format = 3
```

The next one is the maximum particle id value that should be set to the total amount of all particles in the simulation.

```
Max_ID_Value
```

The code can run with

```
./treefrog -C ../treefrog.cfg -i <filelist> -s <numsnapshots> -o <baseoutputfilename>
```

Where <*filelist*> is a file containing the _particles files for each snapshot from the **AHF** output.

CHAPTER

THREE

UNDERSTANDING AND ANALYSING TREEFROG OUTPUT

TreeFrog produces several different types of trees. A descendant tree, a progenitor tree, and a cross catalog. The output is set by the runtime mode of operation, whether the code is building a descendant tree, progenitor tree or simple comparing two input catalogs.

When run with MPI, each thread will write the snapshots that have been processed by that mpi thread when writing HDF output. For ASCII output, either a single continuous ascii file is written or each thread writes a file, adding the rank of the mpi thread writing the file.

Standard files

- `.snapshot_%03d.VELOCIraptor.tree`: a tree file

3.1 TreeFile

The exact format of a tree file depends on whether the code produces a descendant tree or progenitor tree. The information contain also depends on the output format. As the suggetsion is to produce HDF format unless otherwise required we only list the HDF output in detail.

Name	Comments
Header Attributes	
Number_of_snapshots	Number of snapshots in the tree
Total_number_of_halos	Total number of halos across all snapshots
Merit_limit	Merit limit used to determine whether a connection is viable to be a primary connection
Number_of_steps	Number of snapshots searched for primary connections
Search_next_step_criterion	Integer indicating type of criterion used to keep searching for primary connection
Merit_limit_for_next_step	Merit limit below which more snaphots are search for viable primary connection
Core_fraction	Fraction of most bound particles used to calculate merits
Core_min_number_of_particles	Minumum number of most bound particles used to calculate merits
Description	String describing how tree was produced
Tree Data with arrays typically the size of number of halos in given snapshot	
ID	Tree Halo IDs (index of halo + 1 + TEMPORALHALOIDVAL * Snapshot_value)
OrigID	Original ID in halo catalog (IF compiled with HALOIDNOTINDEX. Otherwise not present)
Npart	Number of particles in a halo. Only produced if desired.
NumDescen/NumProgen	Number of descendants/progenitors
DescenOffsets/ProgenOffsets	An offset array indicating where a halo's connections begin associated descen/progen array
Tree Data with arrays the size of the number of viable connections found. This can be larger than the number of halos	
Descendants/Progenitors	Array of Tree ID connections. Halos can have 0,1,>1 connections. Array read using the NumDescen and DescenOffsets arrays
Merit	Merit of the connection
DescenNpart/ProgenNpart	Number of particles in descendant/progenitor. Only produced if desired.

CHAPTER

FOUR

PRODUCING SAM DIGESTIBLE MERGER TREES

One of the key uses of halo merger trees is to produce synthetic galaxies using semi-analytic models such as shark, sage or meraxes.

We provide some background to the production of halo merger trees. Readers simply interested in producing an output can skip the background and simply follow the steps described in *Generating Walkable Tree* and subsequent sections or just follow the steps listed in *Summary of steps*.

We note that the default mode of **TreeFrog** is to process snapshots walking forward in time, identifying descendants but it can operate in a mode walking backward in time identifying progenitors. Halo merger trees for SAMs are better built identifying links by walking forward in time. Thus we focus here on using *Descendant Trees* but it is possible to use *Progenitor Trees*.

Note: We focus on using **VELOCIraptor** halo catalogs as these catalogs can have temporally unique halo ids that contain both snapshot that a halo is found at an the index of the halo in the catalog. Example: **halo ID = snapshot_number * Temporal_halo_id_value + index + 1**, where typically Temporal_halo_id_value = 1000000000000

4.1 Background

The raw output from **TreeFrog** is more akin to a graph than a simple halo merger tree. The file contains for all halos the optimal connection and also secondary connections. This extra information is useful for exploring a variety of topics, from the performance of halo finders to how particles can be exchanged between mergers. However, this extra complexity is not necesary for SAMs.

The raw tree also does not list for a given halo, the first or progenitor or last descendant but instead just the immediate connection. Most SAM models also make use of this information.

Currently, **TreeFrog** makes use of python tools available in the tools directory to produce input that is digestible by such SAMs, converting the raw tree to one which is easily naviable by such codes. There are also scripts availbe in the examples directory that can be used.

Note: This process may change so that **TreeFrog** natively produces a raw tree along with the halo merger trees used by a SAM.

4.2 Generating Walkable Tree

To produce information that stores for each halo, its progenitor (Tail), descendant (Head), first progenitor (Root Tail), and final descendant (Root Head) we make use of the function

```
#for processing a standard descendant based tree frog tree
velociraptor_python_tools.BuildTemporalHeadTailDescendant(numsnaps: int,
    tree: list, numhalos: np.array, halodata: list,
    TEMPORALHALOIDVAL : long = 1000000000000, ireverseorder : bool = False,
    iverbose : int = 1)
#for processing a standard progenitor based tree
velociraptor_python_tools.BuildTemporalHeadTail(numsnaps: int,
    tree: list, numhalos: np.array, halodata: list,
    TEMPORALHALOIDVAL : long = 1000000000000, iverbose : int = 1)
```

This code uses the raw **TreeFrog** data and the information from halo catalogues to produce a simple wakable tree. An example script is located `here`.

The process is as follows:

```
#load the velociraptor python tools
import sys
sys.path.append(TreeFrog/tools/)
import velociraptor_python_tools as vpt

# to determine how many snapshots there are, how many snapshots
# were used to identify links and the temporal halo ID,
# we can use the firt file from TreeFrog
numsnaps = 100
# how do ids map to snapshot, ie. the temporal halo id.
TEMPORALHALOIDVAL = 1000000000000L

# load tree data. Here we assume data is in HDF5 format and follows
# a specific naming convention
basetreename = 'treefrog'
# produce a file listing the treefrog output
snaptreelist=open('snaptreelist.txt','w')
for i in range(numsnaps):
    snaptreelist.write(basetreefname+'.snapshot_%03d.VELOCIraptor\n'%i)
snaptreelist.close()

# read a treefrog descendant tree
reverseorderflag = False
formatflag = 2 #HDF5
iverbose = 0 #not verbose
meritinfoflag = True #merit information present
treefrogdata = vpt.ReadHaloMergerTreeDescendant('snaptreelist.txt',
    reverseorderflag, formatflag, iverbose, meritinfoflag)

# read halo catalog. Here we assume that VELOCIraptor has been used
# we also assume a specific naming convention
# allocate data structures to store the information
numhalos=np.zeros(numsnaps, dtype=np.int64)
halodata = [None for i in range(numsnaps)]
scalefactors = np.zeros(numsnaps)
# as we do not need all the information in the halo catalogs
# only request a subset of the fields
```

(continues on next page)

(continued from previous page)

```
requestedfields = ['ID', 'hostHaloID']

# load halo properties file (this also assumes HDF input)
iverbose = 0
separatefilesforhaloandsubhalos = 0
for i in range(numsnaps):
    fname='snapshot_%03d.VELOCIraptor'%i
    halodata[i],numhalos[i] = vpt.ReadPropertyFile(fname, formatflag,
        separatefilesforhaloandsubhalos, iverbose, requestedfields)
    scalefactors[i] = halodata[i]['SimulationInfo']['ScaleFactor']
```

This loads all the data necessary to make a walkable tree

```
#build the walkable tree
vpt.BuildTemporalHeadTailDescendant(numsnaps,
    treefrogdata, numhalos, halodata, )
```

We now save the data

```
# We store the information related to
# how the tree was built in a dictonary.
# here values are hard coded but can be taken
# from the input.
DescriptionInfo={
        'Title':'Walkable Tree',
        'TreeBuilder':{
            'Name': 'TreeFrog',
            'Version':1.20,
            'Temporal_linking_length':NSNAPSEARCH,
            'Temporal_halo_id_value':TEMPORALHALOIDVAL,
        },
        'HaloFinder': {
            'Name': 'VELOCIraptor',
            'Version': 1.11,
            'Particle_num_threshold':20,
        },
        }
# write file
outputfname = 'walkablehalomergertree.hdf5'
vpt.WriteWalkableHDFTree(outputfname, numsnaps, treefrogdata,
    numhalos, halodata, scalefactors, DescriptionInfo)
```

Using the `script` simply requires altering it to the desired naming convention and running it.

```
#we set the appropriate variables
treefrog_base_filename=treedir/treefrog
#base halo catalog where we assume the names are in directory and follow
#a specific naming convention
halocatalog_dir=halos
output_filename=treedir/walkabletree.hdf5
script=/dir/to/treefrog/examples/example_produce_walkabletree.py
python3 ${script} ${treefrog_base_filename} ${halocatalog_dir} ${output_filename}
```

4.2.1 Generating Input for shark

The semi-analytic code **shark** is designed to load the this walkable tree and the halo catalogues. No further processing of **TreeFrog** is required.

4.3 Generating Forest

Some SAMs require more information to process output. This can range from just extra links to quickly navigate halo catalogs. We focus here on producing output that contains not only a halo's progneitor and descendant ID but also a Forest ID. The idea of a halo forest is a collection of halo merger trees that have interacted with each other at some point in cosmic time. The interaction is typically taken to be that a halo has become a subhalo of another halo at some point. However, such a concept can be generalised to halos that entire some factor of the virial radius of another halo. Here we limit the forest to objects that have become subhalos of another halo as defined by the FOF envelop.

For brevity we simply show code snippets that could be added to the snippets for constructing a walkable tree. Script can be found `here`

```
# load the tree information stored in the file in a dictionary structure
halodata,numsnaps=vpt.ReadWalkableHDFTree(walkabletreefile)
```

For forest files, we suggest that all the desired halo properties be included.

```
requestedfields=[
    'ID', 'hostHaloID',
    'numSubStruct', 'npart',
    'Mass_tot', 'Mass_FOF', 'Mass_200mean', 'Mass_200crit',
    'R_size', 'R_HalfMass', 'R_200mean', 'R_200crit',
    'Xc', 'Yc', 'Zc',
    'Xcminpot', 'Ycminpot', 'Zcminpot',
    'VXc', 'VYc', 'VZc',
    'lambda_B',
    'Lx','Ly','Lz',
    'RVmax_Lx','RVmax_Ly','RVmax_Lz',
    'sigV', 'RVmax_sigV',
    'Rmax', 'Vmax',
    'cNFW',
    'Efrac','Structuretype'
    ]

#load halo properties file
time1 = time.clock()
mp = -1
for i in range(numsnaps):
    fname=halocatalogdir+'snapshot_%03d.VELOCIraptor'%i
    halos, numhalos[i] = vpt.ReadPropertyFile(fname,RAWPROPFORMAT,0,0,requestedfields)
    scalefactors[i]=halos['SimulationInfo']['ScaleFactor']
    halodata[i].update(halos)
```

We suggest you add to the halo catalog entries that allow quick access to substructures and progenitors. These links are used by **sage** and all its variants.

```
#generate subhalo links
vpt.GenerateSubhaloLinks(numsnaps,numhalos,halodata)
#generate progenitor links
vpt.GenerateProgenitorLinks(numsnaps,numhalos,halodata)
```

Now we generate forest IDs.

```
#building forest
#first determine how many snapshots ahead an halo's descendant can be.
maxnsnapsearch=0
for i in range(numsnaps):
    if (numhalos[i] == 0): continue
    headsnap = np.int64(halodata[i]['Head']/TEMPORALHALOIDVAL)
    maxs = np.max(headsnap-i)
    maxnsnapsearch = max(maxnsnapsearch, maxs)

ireverseorder = False
iverbose = 0
iforestcheck = False #this uses extra compute and is generally unnecessary
forestdata = vpt.GenerateForest(numsnaps, numhalos, halodata, scalefactors,
    maxnsnapsearch, ireverseorder, TEMPORALHALOIDVAL, iverbose, iforestcheck)
```

Save the data, seting the information regarding the simulation, tree construction, etc. These can be stored in dictionaries. Here we show the dictionary structure need to write the file.

```
DescriptionInfo={
        'Title':'Forest',
        'TreeBuilder' : copy.deepcopy(treedata['Header']['TreeBuilder']),
        'HaloFinder' : copy.deepcopy(treedata['Header']['HaloFinder']),
        'Flag_subhalo_links':True, 'Flag_progenitor_links':True, 'Flag_forest_ids
↪':True, 'Flag_sorted_forest':False,
        'ParticleInfo':{
            'Flag_dm':True, 'Flag_gas':(igas==1), 'Flag_star':(istar==1), 'Flag_bh
↪':(ibh==1), 'Flag_zoom': False,
            'Particle_mass' : {'dm':mp, 'gas':-1, 'star':-1, 'bh':-1, 'lowres':-1}
            }
        }
vpt.WriteForest(outputfname, numsnaps, numhalos, halodata, forestdata, scalefactors,
    DescriptionInfo, SimulationInfo, UnitInfo, HaloFinderConfigurationInfo
)
```

Using the `script` simply requires altering it to the desired naming convention and running it after having having run the `walkable tree script`.

```
#we set the appropriate variables
#a specific naming convention
halocatalog_dir=halos
walkable_filename=treedir/walkabletree.hdf5
output_filename=treedir/forest
script=/dir/to/treefrog/examples/example_produce_forestID.py
python3 ${script} ${walkable_filename} ${halocatalog_dir} ${output_filename}
```

4.3.1 Generating Input for meraxes

The semi-analytic code **meraxes** is designed to load the forest file, which contains halo properties, tree information and forest information. No further processing of **TreeFrog** is required.

4.3.2 Generating Input for sage

The semi-analytic code **sage** is designed to load the forest file, which contains halo properties, tree information and forest information. The code has in-built converters to convert the information stored in the HDF5 file to its native binary format. HDF5 readers are in development. No further processing of **TreeFrog** is required.

4.4 Summary of steps

To produce halo merger trees for **shark** using scripts:

```
#we set the appropriate variables
treefrog_base_filename=treedir/treefrog
#base halo catalog where we assume the names are in directory and follow
#a specific naming convention
halocatalog_dir=halos
#input snapshot list
snaplist = ${treefrog_base_filename}/snaplist.txt
#num of snaps
nsnaps=200
#walkable tree
walkabletree_filename=treedir/walkabletree.hdf5

#tree frog stuff
tfdir=/dir/to/treefrog/
tf=${tfdir}/build/bin/treefrog
tfconfig=${tfdir}/examples/treefrog_sample.configuration

#post processing scripts
walkablescript=${tfdir}/examples/example_produce_walkabletree.py

#run tree frog
${tf} -i ${snaplist} -s ${nsnaps} -C ${tfconfig} -o ${base_treefrog_filename}

#walkable tree
python3 ${walkablescript} ${treefrog_base_filename} ${halocatalog_dir} ${walkabletree_
↪filename}
```

Now to also produce output for **sage** and **meraxes** also run:

```
#walkable tree
forest_base_filename=treedir/walkabletree.hdf5

#post processing scripts
forestscript=${tfdir}/examples/example_produce_forestID.py

#forest file
python3 ${forestscript} ${walkabletree_filename} ${halocatalog_dir} ${forest_base_
↪filename}
```

CHAPTER

FIVE

DEVELOPING TREEFROG

TreeFrog is an freely available from github.