uchuutools documentation

Manodeep Sinha

Mar 19, 2021

User Documentation

This is the documentation for the *uchuutools* package, a python package for converting ascii simulation data products into more user-friendly datasets. The package was created by Manodeep Sinha @manodeep as part of the Uchuu Project. *uchuutools* is written in pure Python 3 and publicly available on GitHub.

The documentation of *uchuutools* is spread out over several sections:

CHAPTER 1

Getting started

1.1 Installation

uchuutools can be easily installed by either cloning the repository and installing it manually:

```
$ git clone https://github.com/uchuuproject/uchuutools
$ cd uchuutools
$ python -m pip install .
```

or by installing it directly from PyPI with:

```
$ python -m pip install uchuutools
```

uchuutools can now be imported as a package with `import uchuutools`.

CHAPTER 2

HDF5 Output Data Format

Within the following sections we will assume that the generated hdf5 file has been opened with the following code:

```python
import h5py
# if using the defaults, ``h5_filename`` could be
# - ``./forest_0.h5`` (for Consistent-Trees tree catalogue)
# - ``./out_0.list.h5`` (for Rockstar halo catalogue)
# - ``./hlist_<scale_factor>.list.h5`` (for Consistent-Trees halo catalogue)
hf = h5py.File(h5_filename, 'r')
```

2.1 Consistent-Trees HDF5 format for Tree catalogues

The Consistent-Trees tree hdf5 format consists of two types of files:

1. one container hdf5 file,
2. one or more hdf5 files that contain the forest/tree/halo level information (we will refer to this as `hdf5-treecat` file)

2.1.1 Container HDF5 file format

The following attributes of the container hdf5 file may be useful to the user:

1. **Nfiles**: Total number of hdf5 data files that are associated with this container file (`np.int64`)
2. **TotNforests**: Total number of forests stored across all associated `Nfiles` (`np.int64`)
3. **TotNtrees**: Total number of trees stored across all associated `Nfiles` (`np.int64`)
4. **TotNhalos**: Total number of halos stored across all associated `Nfiles` (`np.int64`)

The individual hdf5 files containing the halo-level information are embedded as `ExternalLinks` within the container hdf5 file under `File<ifile>`, where `ifile` ranges from `[0, Nfiles)`. Users can loop over these external links and transparently read all the halos.

```python
import h5py
with h5py.File(container_file, 'r') as hf:
    nfiles = hf.attrs['Nfiles']
    for i in range(nfiles):
        # usage will depend on the value of ``write_halo_props_cont``
        # used during the creation of these files.
        mvir = hf[f"File{i}/Forests/Mvir"]
```

```python
def update_container_h5_file(fname, h5files,
                             standard_consistent_trees=True):
    """
    Writes the container hdf5 file that has external links to
    the hdf5 datafiles with the mergertree information.

    Parameters
    -----------

    fname: string, required
        The name of the output container file (usually ``forest.h5``). A
        new file is always created, however, if the file ``fname`` previously
        existed then the external links are preserved.

    h5files: list of filenames, required
        The list of filenames that were either newly created or updated.

        If the container file ``fname`` exists, then the union of the filenames
        that already existed in ``fname`` and ``h5files`` will be used to
        create the external links

    standard_consistent_tree: boolean, optional, default: True
        Specifies whether the input files were from a parallel
        Consistent-Trees code or the standard Consistent-Trees code. Assumed
        to be standard (i.e., the public version) of the Consistent-Trees
        catalog

    Returns
    -------

    Returns ``True`` on successful completion of the write

    """
    import h5py

    outfiles = h5files
    if not isinstance(h5files, (list, tuple)):
        outfiles = [h5files]

    try:
        with h5py.File(fname, 'r') as hf:
            nfiles = hf['/'].attrs['Nfiles']
            for ifile in range(nfiles):
                outfiles.append(hf[f'File{ifile}'].file)
    except OSError:
        pass

    outfiles = set(outfiles)
    nfiles = len(outfiles)
```

(continues on next page)

(continued from previous page)

```
    with h5py.File(fname, 'w') as hf:
        hf['/'].attrs['Nfiles'] = nfiles
        hf['/'].attrs['TotNforests'] = 0
        hf['/'].attrs['TotNtrees'] = 0
        hf['/'].attrs['TotNhalos'] = 0
        attr_props = [('TotNforests', 'Nforests'),
                      ('TotNtrees', 'Ntrees'),
                      ('TotNhalos', 'Nhalos')]
        for ifile, outfile in enumerate(outfiles):
            with h5py.File(outfile, 'a') as hf_task:
                if standard_consistent_trees:
                    hf_task.attrs['consistent-trees-type'] = 'standard'
                else:
                    hf_task.attrs['consistent-trees-type'] = 'parallel'
                for (out, inp) in attr_props:
                    hf['/'].attrs[out] += hf_task['/'].attrs[inp]

            hf[f'File{ifile}'] = h5py.ExternalLink(outfile, '/')
    return
```

2.1.2 HDF5-treecat file format

There may be one or more hdf5 data-files written as part of the conversion process. These files contain the actual halo information, as well as tree-level and forest-level information contained in the original ascii Consistent-Trees tree catalogues. In this section, we will describe this `hdf5-treecat` file format.

Note: The total number of hdf5 data-files associated with the container file is simply the number of parallel tasks used during the ascii->hdf5 conversion. For serial conversions, there will be *exactly* one hdf5 data-file (by default, named `./forest_0.h5`)

File-level Attributes (`list(hf.attrs)`)

The `hdf5-treecat` file has attributes at the root-level to store metadata about the input ascii Consistent-trees catalogues. The following attributes of the container hdf5 file facilitate reading the hdf5 file:

1. **Nforests**: Total number of forests stored in this file(`np.int64`)
2. **Ntrees**: Total number of trees stored in this file (`np.int64`)
3. **Nhalos**: Total number of halos stored in this file (`np.int64`)
4. **simulation_params**: An hdf5 group that contains cosmological parameters (`Omega_M`, `Omega_L`, `hubble`) and the simulation boxsize (`Boxsize`)

```
# give the HDF5 root some attributes
hf.attrs['input_files'] = np.string_(alltreedatafiles)
mtimes = [os.path.getmtime(f) for f in alltreedatafiles]
hf.attrs['input_filedatestamp'] = np.array(mtimes)
hf.attrs["input_catalog_type"] = np.string_(input_catalog_type)
hf.attrs[f"{input_catalog_type}_version"] = np.string_(version_info)
hf.attrs[f"{input_catalog_type}_columns"] = np.string_(hdrline)
hf.attrs[f"{input_catalog_type}_metadata"] = np.string_(metadata)
hf.attrs['contiguous-halo-props'] = write_halo_props_cont
```

(continues on next page)

(continued from previous page)

```python
sim_grp = hf.create_group('simulation_params')
simulation_params = metadata_dict['simulation_params']
for k, v in simulation_params.items():
    sim_grp.attrs[f"{k}"] = v

hf.attrs['HDF5_version'] = np.string_(h5py.version.hdf5_version)
hf.attrs['h5py_version'] = np.string_(h5py.version.version)

hf.attrs['Nforests'] = 0
hf.attrs['Ntrees'] = 0
hf.attrs['Nhalos'] = 0

### These two lines are executed at the end, while creating
### the container file :func:`update_container_h5_file`.
### ``hf_task`` here refers to ``hf`` in the preceeding
### chunk of code
if standard_consistent_trees:
    hf_task.attrs['consistent-trees-type'] = 'standard'
else:
    hf_task.attrs['consistent-trees-type'] = 'parallel'
```

Halo-level info (`hf['Forests']`)

Halos are written under a `Forests` group within the hdf5 file. If each selected halo property is written separately (i.e., with the default option of `write_halo_props_cont=True`), then individual halo properties are written as a separate dataset as `Forests/<property_name>` (e.g., `Forests/M200c`). If all selected properties of a halo are written contiguously (i.e., with the user-specified option of `write_halo_props_cont=False`), then the halos are written as a single dataset `Forests/halos`.

For each forest, all halos are written contiguously. Further, within each forest, all halos from the same tree are written contiguously. Hence the starting index and number of halos stored in the `TreeInfo` and `ForestInfo` datasets can be directly used to read all halos from the same tree/forest.

```python
forests_grp = hf.create_group('Forests')
if write_halo_props_cont:
    # Create a dataset for every halo property
    # For any given halo property, the value
    # for halos will be written contiguously
    # (structure of arrays)
    for name, dtype in output_dtype.descr:
        forests_grp.create_dataset(name, (0,), dtype=dtype,
                                   chunks=chunks,
                                   compression=compression,
                                   maxshape=(None,))
else:
    # Create a single dataset that contains all properties
    # of a given halo, then all properties of the next halo,
    # and so on (array of structures)
    forests_grp.create_dataset('halos', (0,),
                               dtype=output_dtype,
                               chunks=chunks,
                               compression=compression,
                               maxshape=(None,))
```

By design, the halo properties are written as chunked and compressed. If you plan to read these hdf5 files repeatedly, then you will get faster read-times if you re-write the hdf5 files as unchunked. If you intend to keep the compression, then you will likely get a better compression ratio as well (compression in hdf5 only works on the chunks). You can accomplish that by running the following on the command-line:

```
h5repack -i forest_0.h5 -o forest_0_conti.h5 -l CONTI
h5repack -i forest_0_conti.h5 -o forest_0_conti_gz4.h5 -f GZIP=4
## if the previous two are successfull
mv forest_0_conti_gz4.h5 forest_0.h5 && rm forest_0_conti.h5
```

Note: Any special characters in the Consistent-Trees halo property name are replaced with a single underscore _. For example, `A[x](500c)` in the input ascii file is written as `A_x_500c` in the hdf5 file. This name conversion is done by the function `uchuutools.utils.sanitize_ctrees_header()`.

```
def sanitize_ctrees_header(headerline):
    import re

    header = [re.sub('\(\d+\)$', '', s) for s in headerline]
    # print("After normal sub: header = {}\n".format(header))
    header = [re.sub('[^a-zA-Z0-9 \n\.]', '_', s) for s in header]
    # print(f"After replacing special characters with _: header = {header}\n")
    header = [re.sub('_$', '', s) for s in header]
    # print(f"After replacing trailing underscore: header = {header}\n")
    header = [re.sub('(_)+', '_', s) for s in header]
    # print(f"After replacing multiple underscores: header = {header}")
    return header
```

Forest-level info (`hf['Forestinfo]`)

Since all halos from the same forest are written contiguously, the forest level info is there to allow easy access to entire forests. This info is stored in the dataset `ForestInfo` and contains the following fields:

1. **ForestID**: Contains the `ForestID` as assigned by Consistent-Trees (`np.int64`)
2. **ForestHalosOffset**: Contains the index of the first halo contained within each forest
3. **ForestNhalos**: Contains the total number of halos within each forest (`np.int64`)
4. **ForestNtrees**: Contains the total number of trees within each forest (`np.int64`)

The number of entries in this `ForestInfo` dataset (i.e., the shape) equals the number of forests stored in the hdf5 file.

```
forest_dtype = np.dtype([('ForestID', np.int64),
                         ('ForestHalosOffset', np.int64),
                         ('ForestNhalos', np.int64),
                         ('ForestNtrees', np.int64), ])
hf.create_dataset('ForestInfo', (0,), dtype=forest_dtype,
                  chunks=True, compression=compression,
                  maxshape=(None,))
```

Tree-level info (`hf['TreeInfo']`)

Since the halos are stored on a **per tree** basis in the input ascii Consistent-Trees catalogue, data provenance requires that we store that original information at a tree level as well. In addition, this allows us to quickly read a single tree for visualisation/testing (rather than the entire forest). This info is stored in the dataset `TreeInfo` and contains the following fields:

1. **ForestID**: Contains the `ForestID` as assigned by Consistent-Trees (`np.int64`)
2. **TreeRootID**: Contains the `TreeRootID` as assigned by Consistent-Trees (`np.int64`)
3. **TreeHalosOffset**: Contains the index of the first halo contained within each tree (`np.int64`)
4. **TreeNhalos**: Contains the total number of halos within each tree (`np.int64`)
5. **Input_Filename**: Contains the input ascii Consistent-Trees filename(string, `'S1024'`)
6. **Input_FileDateStamp**: Contains the modification time of the input ascii Consistent-Trees file (`np.float`)
7. **Input_TreeByteOffset**: Contains the byte offset of the first halo within the input ascii Consistent-Trees file (`np.int64`)
8. **Input_TreeNbytes**: Contains the total number of bytes for this tree within the input ascii Consistent-Trees file (`np.int64`)

Fields prefixed with `Input_` are there solely for tracking back to the original files or ease of access (`Input_TreeNbytes`). The number of entries in this `TreeInfo` dataset (i.e., the shape) equals the number of trees stored in the hdf5 file.

```
tree_dtype = np.dtype([('ForestID', np.int64),
                       ('TreeRootID', np.int64),
                       ('TreeHalosOffset', np.int64),
                       ('TreeNhalos', np.int64),
                       ('Input_Filename', string_dtype),
                       ('Input_FileDateStamp', np.float),
                       ('Input_TreeByteOffset', np.int64),
                       ('Input_TreeNbytes', np.int64), ])
hf.create_dataset('TreeInfo', (0,), dtype=tree_dtype,
                  chunks=True, compression=compression,
                  maxshape=(None,))
```

2.2 Rockstar/Consistent-Trees HDF5 format for halo catalogues

Each Rockstar `out_*.list`, or Consistent-Trees `hlist_*.list` files is converted into a single hdf5 file (`hdf5-halocat` file). The halos in the hdf5 files are written in the exact same order as the input ascii files.

2.2.1 HDF5-halocat file format

File-level Attributes

The `hdf5-halocat` file has attributes at the root-level to store metadata about the input ascii Consistent-trees catalogues. The following attributes of the container hdf5 file facilitate reading the hdf5 file:

1. **TotNhalos**: Total number of halos stored in this file (`np.int64`)
2. **scale_factor**: Total number of forests stored in this file(`np.float`)
3. **redshift**: The redshift for the halo catalogue (`np.float`)
4. **redshift_params**: An hdf5 group that contains cosmological parameters (`Omega_M`, `Omega_L`, `hubble`) and the simulation boxsize (`Boxsize`)

```
line_with_scale_factor = ([line for line in metadata
                           if line.startswith("#a")])[0]
scale_factor = float((line_with_scale_factor.split('='))[1])
redshift = 1.0/scale_factor - 1.0

# give the HDF5 root some attributes
hf.attrs[u"input_filename"] = np.string_(input_file)
hf.attrs[u"input_filedatestamp"] = np.array(os.path.getmtime(input_file))
hf.attrs[u"input_catalog_type"] = np.string_(input_catalog_type)
hf.attrs[f"{input_catalog_type}_version"] = np.string_(version_info)
hf.attrs[f"{input_catalog_type}_columns"] = np.string_(hdrline)
hf.attrs[f"{input_catalog_type}_metadata"] = np.string_(metadata)
sim_grp = hf.create_group('simulation_params')
simulation_params = metadata_dict['simulation_params']
for k, v in simulation_params.items():
    sim_grp.attrs[f"{k}"] = v

hf.attrs[u"HDF5_version"] = np.string_(h5py.version.hdf5_version)
hf.attrs[u"h5py_version"] = np.string_(h5py.version.version)
hf.attrs[u"TotNhalos"] = -1
hf.attrs[u"scale_factor"] = scale_factor
hf.attrs[u"redshift"] = redshift
```

Halo-level info

```
halos_grp = hf.create_group('HaloCatalogue')
halos_grp.attrs['scale_factor'] = scale_factor
halos_grp.attrs['redshift'] = redshift

dset_size = approx_totnumhalos
if write_halo_props_cont:
    halos_dset = dict()
    # Create a dataset for every halo property
    # For any given halo property, the value
    # for halos will be written contiguously
    # (structure of arrays)
    for name, dtype in parser.dtype.descr:
        halos_dset[name] = halos_grp.create_dataset(name,
                                                    (dset_size, ),
                                                    dtype=dtype,
                                                    chunks=True,
                                                    compression=compression,
                                                    maxshape=(None,))
else:
    # Create a single dataset that contains all properties
    # of a given halo, then all properties of the next halo,
    # and so on (array of structures)
    halos_dset = halos_grp.create_dataset('halos', (dset_size,),
                                          dtype=parser.dtype,
                                          chunks=True,
                                          compression=compression,
                                          maxshape=(None,))
```

CHAPTER 3

Community guidelines

uchuutools is an open-source and free-to-use software package provided under the MIT license (see below for the full license).

Users are highly encouraged to make contributions to the package or request new features by opening a GitHub issue. As with contributions, if you find a problem or issue with *uchuutools*, please do not hesitate to open a GitHub issue about it.

3.1 License

CHAPTER 4

Converters

Provides a collection of routines to convert from Rockstar and Consistent-Tree ascii catalogues into hdf5 files

Recommended usage:

```
import uchuutools.converters as utconv
```

4.1 Available submodules

`convert_ctrees_to_h5()` Converts ascii Consistent-Trees catalogues to hdf5

`convert_halocat_to_h5()` Converts ascii Rockstar and Consistent-Trees halo catalogues to hdf5

`uchuutools.converters.`**`convert_ctrees_to_h5`**(*filenames*, *standard_consistent_trees=None*, *outputdir='./'*, *output_filebase='forest'*, *write_halo_props_cont=True*, *fields=None*, *drop_fields=None*, *truncate=True*, *compression='gzip'*, *buffersize=None*, *use_pread=True*, *max_nforests=None*, *comm=None*, *show_progressbar=False*)

Convert a set of forests from Consistent Trees ascii file(s) into an (optionally compressed) hdf5 file. Can be invoked with MPI.

Parameters

- **filenames** (*list of strings for Consistent-Trees catalogues, required*) – The input ascii files will be decompressed, if required.
- **standard_consistent_tree** (*boolean, optional, default: None*) – Whether the input filres were generated by the Uchuu collaboration's parallel Consistent-Trees code. If only two files are specified in `filenames`, and these two filenames end with 'forests.list', and 'locations.dat', then a standard Consistent-Trees output will be inferred. If all files specified in `filenames` end with '.tree', then parallel Consistent-Trees is inferred.

- **outputdir** (*string, optional, default: current working directory ('./')*) – The directory where the converted hdf5 file will be written in. The output filename is obtained by appending '.h5' to the `input_file`.
- **output_filebase** (*string, optional, default: "forest"*) – The output filename is constructed using '{outputdir}/{output_filebase}_{rank}.h5'
- **write_halo_props_cont** (*boolean, optional, default: True*) – Controls if the individual halo properties are written as distinct datasets such that any given property for *all* halos is written contiguously (structure of arraysA).

  When set to False, only one dataset ('halos') is created under the group 'Forests', and *all* properties of a halo is written out contiguously (array of structures).
- **fields** (*list of strings, optional, default: None*) – Describes which specific columns in the input file to carry across to the hdf5 file. Default action is to convert ALL columns.
- **drop_fields** (*list of strings, optional, default: None*) – Contains a list of column names that will *not* be carried through to the hdf5 file. If `drop_fields` is not set for a parallel Consistent-Trees run, then [`Tidal_Force`, `Tidal_ID`] will be used.

  `drop_fields` is processed after `fields`, i.e., you can specify `fields=None` to create an initial list of *all* columns in the ascii file, and then specify `drop_fields = [colname2, colname7, ...]`, and *only* those columns will not be present in the hdf5 output.
- **truncate** (*boolean, default: True*) – Controls whether a new file is created on this 'rank'. When set to `True`, the header info file is written out. Otherwise, the file is appended to. The code checks to make sure that the existing metadata in the hdf5 file is identical to the new metadata in the ascii files being currently converted (i.e., tries to avoid different simulation + mergertree results being present in the same file)
- **compression** (*string, optional, default: 'gzip'*) – Controls the kind of compression applied. Valid options are anything that `h5py` accepts.
- **buffersize** (*integer, optional, default: 1 MB*) – Controls the size of the buffer how many halos are written out per write call to the hdf5 file. The number of halos written out is this buffersize divided the size of the datatype for individual halos.
- **use_pread** (*boolean, optional, default: True*) – Controls whether low-level i/o operations (through `os.pread`) is used. Otherwise, higher-level i/o operations (via `io.open`) is used. This option is only meaningful on linux systems (and python3+). Since `pread` does not change the file offset, additional parallelisation can be implemented reasonably easily.
- **max_nforests** (*integer >= 1, optional, default: None*) – The maximum number of forests to convert across all tasks. If a positive value is passed then the total number of forests converted will be `min(totnforests, max_nforests)`. ValueError is raised if the passed parameter value is less than 1.
- **comm** (*MPI communicator, optional, default: None*) – Controls whether the conversion is run in MPI parallel. Should be compatible with *mpi4py.MPI.COMM_WORLD*.
- **show_progressbar** (*boolean, optional, default: False*) – Controls whether a progressbar is printed. Only enables progressbar on rank==0, the remaining ranks ignore this keyword.

Returns Returns `True` on successful completion.

`uchuutools.converters.`**`convert_halocat_to_h5`**(*filenames*, *outputdir='./'*, *write_halo_props_cont=True*, *fields=None*, *drop_fields=None*, *chunksize=100000*, *compression='gzip'*, *comm=None*, *show_progressbar=False*)

Converts a list of Rockstar/Consistent-Trees halo catalogues from ascii to hdf5.

Can be used with MPI but requires that the number of files to be larger than the number of MPI tasks spawned.

Parameters

- **filenames** (*list of strings, required*) – A list of filename(s) for the Rockstar/Consistent Trees file. Can be compressed (.gz, .bz2, .xz, .zip) files.
- **outputdir** (*string, optional, default: current working directory ('./')*) – The directory where the converted hdf5 file will be written in. The output filename is obtained by appending '.h5' to the `input_file`. If the output file already exists, then it will be truncated.
- **write_halo_props_cont** (*boolean, optional, default: True*) – Controls if the individual halo properties are written as distinct datasets such that any given property for ALL halos is written contiguously (structure of arrays, SOA).

  When set to False, only one dataset ('halos') is created, and ALL properties of a halo is written out contiguously (array of structures).
- **fields** (*list of strings, optional, default: None*) – Describes which specific columns in the input file to carry across to the hdf5 file. Default action is to convert ALL columns.
- **drop_fields** (*list of strings, optional, default: None*) – Describes which columns are not carried through to the hdf5 file. Processed after `fields`, i.e., you can specify `fields=None` to create an initial list of *all* columns in the ascii file, and then specify `drop_fields = [colname2, colname7, ...]`, and those columns will not be present in the hdf5 output.
- **chunksize** (*integer, optional, default: 100000*) – Controls how many lines are read in from the input file before being written out to the hdf5 file.
- **compression** (*string, optional, default: 'gzip'*) – Controls the kind of compression applied. Valid options are anything that `h5py` accepts.
- **comm** (*MPI communicator, optional, default: None*) – Controls whether the conversion is run in MPI parallel. Should be compatible with *mpi4py.MPI.COMM_WORLD*.
- **show_progressbar** (*boolean, optional, default: False*) – Controls whether a progressbar is printed. Only enables progressbar on rank==0, the remaining ranks ignore this keyword.

Returns Returns `True` on successful completion.

CHAPTER 5

Utilities for Consistent-Trees Catalogues

Provides several useful utility functions that work with tree catalogues generated by Consistent-Trees.

`uchuutools.ctrees_utils.`**`read_locations_and_forests`**(*forests_fname*, *locations_fname*, *rank=0*)

Returns a numpy structured array that contains *both* the forest and tree level info.

Parameters

- **forests_fname** (*string, required*) – The name of the file containing forest-level info like the Consistent-Trees 'forests.list' file
- **locations_fname** (*string, required*) – The name of the file containing tree-level info like the Consistent-Trees 'locations.dat' file
- **rank** (*integer, optional, default:0*) – An integer identifying which task is calling this function. Only used in status messages

Returns

trees_and_locations (*A numpy structured array*) – A numpy structured array containing the fields <TreeRootID ForestID Filename FileID Offset TreeNbytes> The array is sorted by `(ForestID, Filename, Offset)` in that order. This sorting means that *all* trees belonging to the same forest *will* appear consecutively regardless of the file that the corresponding tree data might appear in. The number of elements in the array is equal to the number of trees.

Note: Sorting by `Filename` is implemented by an equivalent, but faster sorting on `FileID`.

`uchuutools.ctrees_utils.`**`get_aggregate_forest_info`**(*trees_and_locations*, *rank=0*)

Returns forest-level information from the tree-level information supplied.

Parameters

- **trees_and_locations** (*numpy structured array, required*) – A numpy structured array that contains the tree-level information. Should be the output from the function `read_locations_and_forests()`
- **rank** (*integer, optional, default:0*) – An integer identifying which task is calling this function. Only used in status messages

Returns **forest_info** (*A numpy structured array*) – The structured array contains the fields `['ForestID', 'ForestNhalos', 'Input_ForestNbytes', 'Ntrees']` The 'ForestNhalos' field is set to 0, and is populated as the trees are processed. The number of elements in the array is equal to the number of forests.

`uchuutools.ctrees_utils.`**`get_all_parallel_ctrees_filenames`**(*fname*)

Returns three filenames corresponding to the `'forests.list'`, `'locations.dat'`, and `'tree_*.dat'` files *assuming* the naming convention the Uchuu collaboration's parallel Consistent-Trees code

Parameters **fname** (*string, required*) – A filename specifying the tree data file for a parallel Consistent-Trees soutput

Returns

forests_file, locations_file, treedata_file (*strings, filenames*) – The filenames corresponding to the `'forests.list'`, `'locations.dat'` and the `'tree_*_*_*.dat'` file *generated using* the convention of the Uchuu collaboration. The convention is:

Standard CTrees	Parallel CTrees
'forests.list'	'<prefix>.forest'
'locations.dat'	'<prefix>.loc'
'tree_*_*_*.dat'	'<prefix>.tree'

`uchuutools.ctrees_utils.`**`check_forests_locations_filenames`**(*filenames*)

Accepts two filenames (in any order) and checks whether the files contain the correct data as expected in 'forests.list', 'locations.dat' and returns the files as (equivalent to) 'forests.list' and 'locations.dat'

Parameters **filenames** (*list of two filenames, string, required*) – List containing two filenames corresponding to the standard 'forests.list' and 'locations.dat' (in any order, i.e., both ['forests.list', 'locations.dat'] *and* ['locations.dat', 'forests.list'] are valid)

Returns **forests_file, locations_file** (*strings, filenames*) – The filenames equivalent to the 'forests.list', 'locations.dat' (in that order)

`uchuutools.ctrees_utils.`**`validate_inputs_are_ctrees_files`**(*ctrees_filenames, base_metadata=None, base_version=None, base_input_catalog_type=None*)

Checks the files contain Consistent-Trees catalogues derived from the same simulation and Consistent-Trees configuration.

Parameters **ctrees_filenames** (*list of filenames, string, required*) – The input filenames (potentially) containing Consistent-Trees catalogues.

Note: Only unique filenames within this list are checked

Returns *True on successful validation, ValueError otherwise*

`uchuutools.ctrees_utils.`**`get_treewalk_dtype_descr`**()

Returns the description for the additional fields to add to the forest for walking the mergertree

Parameters **None**

Returns **mergertree_descr** (*list of tuples*) – A list of tuples containing the names and datatypes for the additional columns needed for walking the mergertree. This list can be used to create a numpy datatype suitable to contain the additional mergertree indices

`uchuutools.ctrees_utils.`**`add_tree_walk_indices`**(*forest, rank=0*)

Adds the various mergertree walking indices and IDs. These include indices to access the host FOF halo<->subhalo hierarchy, and the progenitor-descendant hierarchy.

The mergertree indices are filled in-place and no additional return occurs. The specific indices are listed in the function `get_treewalk_dtype_descr()`

Parameters

- **forest** (*numpy structured array, required*) – An array containing all halos from the same forest
- **rank** (*integer, optional, default=0*) – The unique identifier for the current task. Only used within an error statement

Returns *True on successful completion*

CHAPTER 6

General Utilities

Provides several utility functions.

`uchuutools.utils.`**`get_parser`**(*filename*, *fields=None*, *drop_fields=None*)

Returns a parser that parses a single line from the input ascii file

Parameters

- **filename** (*string, required*) – A filename containg Rockstar/Consistent-Trees data. Can be a compressed file if the compression is one of the types supported by the `generic_reader` function.
- **fields** (*list of strings, optional, default: None*) – Describes which specific columns in the input file to carry across to the hdf5 file. Default action is to convert ALL columns.
- **drop_fields** (*list of strings, optional, default: None*) – Describes which columns are not carried through to the hdf5 file. Processed after `fields`, i.e., you can specify `fields=None` to create an initial list of *all* columns in the ascii file, and then specify `drop_fields = [colname2, colname7, ...]`, and those columns will not be present in the hdf5 output.

Returns **parser** (*an instance of BaseParseFields*) – A parser that will parse a single line (read from a Rockstar/Consistent-Trees file) and create a tuple containing *only* the relevant columns.

`uchuutools.utils.`**`get_approx_totnumhalos`**(*input_file*, *ndatabytes=None*)

Returns an (approximate) number of lines containing data in the `input_file`.

Assumes that the only comment lines in the file occur at the beginning. Comment lines are assumed to begin with '#'.

Parameters

- **input_file** (*string, required*) – The input filename for the Rockstar/Consistent Trees file
- **ndatabytes** (*integer, optional*) – The total number of bytes being processed. If not passed, the entire disk size of the `input_file` minus the initial header lines will be used (i.e. assumes that the entire file is being processed)

Returns **approx_totnumhalos** (*integer*) – The approximate number of halos in the input file. The actual number of halos should be close but can be smaller/greater than the approximate value.

`uchuutools.utils.`**`generic_reader`**(*filename*, *mode='rt'*)

Returns a file-reader with capability to read line-by-line for both compressed and normal text files.

Parameters

- **filename** (*string, required*) – The filename for the associated input/output. Can be a compressed (.bz2, .gz, .xz, .zip) file as well as a regular ascii file
- **mode** (*string, optional, default: 'rt' (readonly-text mode)*) – Controls the kind of i/o operation that will be performed

Returns **f** (*file handle, generator*) – Yields a generator that has the `readline` feature (i.e., supports the paradigm `for line in f:`). This file-reader generator is suitable for use in `with` statements, e.g., `with generic_reader(<fname>) as f:`

`uchuutools.utils.`**`get_metadata`**(*input_file*)

Returns metadata information for `input_file`. Includes all comment lines in the header, Rockstar/Consistent-Trees version, and the input catalog type (either Rockstar or Consistent-Trees).

Assumes that the only comment lines in the file occur at the beginning. Comment lines are assumed to begin with '#'.

Parameters **input_file** (*string, required*) – The input filename for the Rockstar/Consistent Trees file Compressed files ('.bz2', '.gz', '.xz', '.zip') are also allowed as valid kinds of `input_file`

Returns

- **metadata_dict** (*dictionary*) – The dictionary contains four key-value pairs corresponding to the keys: ['metadata', 'version', 'catalog_type', 'headerline'].
- **metadata** (*string*) – All lines in the beginning of the file that start with the character '#'.
- **version** (*string*) – Rockstar or Consistent-Trees version that was used to generate `input_file`
- **catalog_type** (*string*) – Is one of [`Rockstar`, `Consistent Trees`, `Consistent Trees (hlist)`] and indicates what kind of catalog is contained in `input_file`
- **headerline** (*string*) – The first line in the input file with any leading/trailing white-space, and any leading '#' removed

`uchuutools.utils.`**`resize_halo_datasets`**(*halos_dset*, *new_size*, *write_halo_props_cont*, *dtype*)

Resizes the halo datasets

Parameters

- **halos_dset** (*dictionary, required*)
- **new_size** (*scalar integer, required*)
- **write_halo_props_cont** (*boolean, required*) – Controls if the individual halo properties are written as distinct datasets such that any given property for ALL halos is written contiguously (structure of arrays, SOA).
- **dtype** (*numpy datatype*)

Returns Returns `True` on successful completion

`uchuutools.utils.`**`check_and_decompress`**(*fname*)

Decompresses the input file (if necessary) and returns the decompressed filename

Parameters **fname** (*string, required*) – Input filename, can be compressed

Returns **decomp_fname** (*string*) – The decompressed filename

`uchuutools.utils.`**`distribute_array_over_ntasks`**(*cost_array*, *rank*, *ntasks*)

Calculates the subscript range for the `rank`'th task such that the work-load is evenly distributed across `ntasks`.

Parameters

- **cost_array** (*numpy array, required*) – Contains the cost associated with processing *each* element of the array
- **rank** (*integer, required*) – The integer rank for the task that we need to compute the work-load division for
- **ntasks** (*integer, required*) – Total number of tasks that the array should be (evenly) distributed across

Returns

(start, stop) (*A tuple of (np.int64, np.int64)*) – Contains the initial and final subscripts that the `rank` task should process.

Note: start, stop are both inclusive, i.e., all elements from `start` to `stop` should be included. For python array indexing with slices, this translates to arr[start:stop + 1].

`uchuutools.utils.`**`check_for_contiguous_halos`**(*h5_task_file*, *write_halo_props_cont*)

Checks that the hdf5 file can be appended to with the requested writing of halo properties

Parameters

- **h5_task_file** (*string, required*) – An existing hdf5 file. The file may or may not contain halos, but the dataset (or datasets, depending on the value of `write_halo_props_cont`) for the halo properties should already be created
- **write_halo_props_cont** (*boolean, required*) – Controls if the individual halo properties are written as distinct datasets such that any given property for ALL halos is written contiguously (structure of arrays, SOA).

Returns Returns `True` on successful completion

`uchuutools.utils.`**`write_halos`**(*halos_dset*, *halos_dset_offset*, *halos*, *nhalos_to_write*, *write_halo_props_cont*)

Writes halos into the relevant dataset(s) within a hdf5 file

Parameters

- **halos_dset** (*dictionary, required*) – Contains the halos dataset(s) within a hdf5 file where either the entire halos array or the individual halo properties should be written to. See parameter `write_halo_props_cont` for further details
- **halos_dset_offset** (*scalar integer, required*) – Contains the index within the halos dataset(s) where the write should start
- **halos** (*numpy structured array, required*) – An array containing the halo properties that should be written out into the hdf5 file. The entire array may not be written out, see the parameter `nhalos_to_write`
- **nhalos_to_write** (*scalar integer, required*) – Number of halos from the `halos` array that should be written out. Can be smaller than the shape of the `halos` array
- **write_halo_props_cont** (*boolean, required*) – Controls if the individual halo properties are written as distinct datasets such that any given property for ALL halos is written contiguously (structure of arrays, SOA).

Returns Returns `True` on successful completion of the write `uchuutools.utils.`**`update_container_h5_file`**(*fname*, *h5files*, *standard_consistent_trees=True*)

Writes the container hdf5 file that has external links to the hdf5 datafiles with the mergertree information.

Parameters

- **fname** (*string, required*) – The name of the output container file (usually `forest.h5`). A new file is always created, however, if the file `fname` previously existed then the external links are preserved.
- **h5files** (*list of filenames, required*) – The list of filenames that were either newly created or updated.

  If the container file `fname` exists, then the union of the filenames that already existed in `fname` and `h5files` will be used to create the external links
- **standard_consistent_tree** (*boolean, optional, default: True*) – Specifies whether the input files were from a parallel Consistent-Trees code or the standard Consistent-Trees code. Assumed to be standard (i.e., the public version) of the Consistent-Trees catalog

Returns Returns `True` on successful completion of the write

Python Module Index

Index