`pkdtools`

Last Update: 10/11/23

Created by Julian C. Marohnic: 10/2/23

Contents

1 Preface

`pkdtools` is a Python package thats aids in creating and manipulating `pkdgrav` `ss` data. It does not rely on any existing `pkdgrav` utilities, with the exception of `ssio.py`. `pkdtools` is not interactive in the way that tools like `rpx` or `ssinfo` are. Rather, `pkdtools` is designed to be imported by user-created Python scripts as a general toolkit for working with `ss` files in `pkdgrav`. Ideally, interactive Python-based `pkdgrav` utilities developed in the future will use `pkdtools` as a “back end.” This document contains an inventory of the classes, methods, and functions introduced in `pkdtools`, its organization into various subpackages, as well as miscellaneous information about best practices and usage. `pkdtools` is an expansion of the `pkdtools` module and was created by Julian Marohnic (JCM) with important contributions from Joseph DeMartini (JVD).

`pkdtools` is organized into several core modules that make up the base `pkdtools` package, as well as several subpackages that contain a variety of optional, specialized, or experimental features. As of this writing, the subpackages are `pkdtools.aggs`, `pkdtools.rp`, `pkdtools.constants`, and `pkdtools.exp`.

2 Core classes, functions, and methods

The basic functionality of `pkdtools` is split across the following modules: `particle`, `assembly`, `tools`, `util`, `pkdio`, and `ssio`. `ssio` predates `pkdtools` and is documented elsewhere. When importing `pkdtools`, all of the base modules will be imported automatically—the division into various modules is purely for organization and may be ignored by the user. This section describes these core classes, functions, and methods. Additional, more specialized tools are included in various subpackages (e.g., `pkdtools.aggs`) and are described in later sections of this document.

`pkdtools` introduces two new classes: `Particle` and `Assembly`, hereafter "particles" and "assemblies." Particles in `pkdtools` correspond to particles in `pkdgrav`, while assemblies correspond to any collection of zero or more particles. These two classes are the primary data structures used in `pkdtools`. Particles and assemblies both have "units" attributes. Legal unit settings are `'pkd'` (`pkdgrav` customary units), `'cgs'`, and `'mks'`. Units may be changed at will by the user, and the values contained in the relevant particle or assembly should be updated seamlessly. Changing the units of a particle will only update the data values for that particle, while changing the units of an assembly will update the units attribute of the assembly, as well as the units and data values for all of its constituent particles. `pkdtools` makes use of a number of internal functions (described below) to effect unit changes, but users should always change particle units by simply setting the units attribute to the desired value.

2.1 Particles

Particles in `pkdtools` have attributes corresponding to the standard `pkdgrav` particle fields, and an additional units attribute. The names of the attributes are as follows:

```
iOrder
iOrgIdx
m
R
x
y
z
vx
```

```
vy
vz
wx
wy
wz
color
units
```

The `Particle` keyword will create a new particle object. For example:

```
Particle(0,0,10,100,1,0,0,5,0,0,0,0,0,5,‘cgs’)
```

will create a new particle with iOrder 0, iOrgIdx 0, mass 10 g, radius 100 cm, x position 1 cm, y and z positions 0, x speed 5 cm/s, y and z speeds of 0, no spin, color yellow, and a ‘cgs’ units tag. The ‘cgs’ argument is the only way to specify the units of the input arguments, so it is crucial to specify this value. When initializing a new particle, iOrder, iOrgIdx, mass, and radius must be specified. Position, velocity, and spin all default to zero if not specified. Particle color will default to green, and units default to pkdgrav. Particle attributes can be read or written using the `<class instance>.<attribute>` syntax, e.g.:

```
>>> p1 = Particle(0,0,15,50,units=‘cgs’)
>>> p1.m
15.0
>>> p1.m = 20
>>> p1.m
20.0
>>> p1.units = ‘mks’
>>> p1.m
0.02
```

Below is a list of the methods available to the `Particle` class in the base `pkdtools` package. In addition, particles have a `print` function implemented, which may be used to display the entire contents of the object.

2.1.1 `pos()`

Return particle position as a `pkdgrav` array.

2.1.2 `vel()`

Return particle velocity as a `numpy` array.

2.1.3 `spin()`

Return particle spin as a `numpy` array.

2.1.4 `set_pos(pos, units=None)`

Set particle position with a 3-element vector-like input. `pos` may be a `numpy` array, a tuple, or a list. Units default to the particle's current units.

2.1.5 `set_vel(vel, units=None)`

Set particle velocity with a 3-element vector-like input. `vel` may be a `numpy` array, a tuple, or a list. Units default to the particle's current units.

2.1.6 `set_spin(w, units=None)`

Set particle velocity with a 3-element vector-like input. `w` may be a `numpy` array, a tuple, or a list. Units default to the particle's current units.

2.1.7 `copy()`

Returns a copy of the particle. Setting a new variable equal to an existing particle will not create a new, independent particle object.

2.2 Assemblies

The `Assembly` class is derived from the Python `list` type, and shares many of its features. An assembly has two attributes of its own, which are `units` and `time`. Both `time` and `units` may be specified when initializing an assembly, or omitted in favor of the default values (`pkdgrav` units and a timestamp of 0.0). An assembly is a container for zero or more particles, though a list of particles on its own does *not* constitute an assembly. An assembly can be created with the `Assembly` keyword. `Assembly()` expects to be called on an arbitrary number of particles, along with optional units and time arguments. Note that calling `Assembly` on a list of particles will not work, but this can be circumvented by prepending the * operator to your list when creating the assembly. E.g.:

```
Assembly(particle1, particle2, ..., units='cgs')

OR

Assembly(*[particle1, particle2, ...], units='cgs')
```

Assemblies support list-style slicing, though there are some subtleties to be noted here. Consider an assembly `a1` containing 10 particles. `a1[3]` will return the 4th particle in the assembly. The particle will not be removed from `a1`, but any manipulations made to this particle will be reflected in the 4th particle of `a1`. Contrast this with the `get_particle()` assembly method, which will instead return a copy of the particle requested that exists independently of the original. Functions and methods in `pkdtools` that deal with extracting particles from assemblies typically behave this way. Also, `get_particle()` searches by iOrder value, while list slicing does not "know" about iOrder values and will simply return the particle in the position requested. Both approaches may be useful in different circumstances. Assemblies are also iterable, so `pkdtools` allows for constructions like the following:

```
to_be_deleted = [particle.iOrder for particle in a1 if particle.x > 1000]

OR

for particle in a1:
    if particle.iOrgIdx < 0:
        particle.set_vel((0,0,0))
```

Below is a list of the core `Assembly` methods that are included in the base `pkdtools` package. The `print` function can also be called on assemblies.

2.2.1 `N()`

Returns the number of particles in the assembly.

2.2.2 `M()`

Returns the total mass of the assembly.

2.2.3 `xbounds()`

Returns a `numpy` array containing the minimum and maximum x values across all particles in the assembly.

2.2.4 `ybounds()`

Returns a `numpy` array containing the minimum and maximum y values across all particles in the assembly.

2.2.5 `zbounds()`

Returns a `numpy` array containing the minimum and maximum z values across all particles in the assembly.

2.2.6 `com()`

Returns a `numpy` array containing the center of mass position of the assembly.

2.2.7 `comv()`

Returns a `numpy` array containing the center of mass velocity of the assembly.

2.2.8 `center()`

Returns a `numpy` array containing the mid-point of the assembly. In other words, the point located halfway between the extreme x, y, and z values. Note that this is not equivalent to the center of mass of the assembly.

2.2.9 `ang_freq()`

Return the angular frequency vector of an assembly.

2.2.10 `freq()`

Return the rotation frequency of an assembly.

2.2.11 `period()`

Return the spin period of an assembly.

2.2.12 `set_com(com, units=None)`

Translate all particles so that the assembly has the specified center of mass position. `com` must be a 3-element `numpy` array, tuple, or list. Units will default to the current assembly setting.

2.2.13 `set_comv(comv, units=None)`

Edit all particles so that the assembly has the new center of mass velocity. `comv` must be a 3-element `numpy` array, tuple, or list. Units will default to the current assembly setting.

2.2.14 `set_center(center=(0,0,0), units=None)`

Translate all particles to match the desired center location (center is described above under `center()`). `comv` must be a 3-element `numpy` array, tuple, or list. Units will default to the current assembly setting.

2.2.15 `R()`

Return the assembly "radius." This method will return a value regardless of whether the assembly is a single rubble pile or not. The radius is defined here to be the greatest possible distance between the center of mass of the assembly and any single particle plus that particle's radius.

2.2.16 `vol()`

Calculate the volume occupied by particles in the assembly. This method will return a value regardless of whether the assembly is a single rubble pile or not. `vol()` uses a simplistic convex hull method and could probably be improved. Clearly, usefulness and accuracy will depend on the inputs.

2.2.17 `avg_dens()`

Return the average particle density over all particles in the assembly.

2.2.18 `bulk_dens()`

Return the bulk density of the assembly. Relies on `vol()`, and so suffers from the same pitfalls.

2.2.19 `I()`

Return the inertia tensor of the assembly as a `numpy` array.

2.2.20 `axes()`

Return the principal axes of the assembly as a `numpy` array.

2.2.21 `L()`

Return the angular momentum vector of the assembly. (Untested)

2.2.22 `deeve()`

Return the DEEVE semi-axes, equivalent radius, and bulk density of the assembly.

2.2.23 `semi_axes()`

Calculate the lengths of the three semi-axes of an assembly.

2.2.24 `show_particles()`

Prints all particles in the assembly. Largely redundant with `print(<assembly>)`.

2.2.25 `add_particles(*particles)`

Add copies of an arbitrary number of particles to the assembly. Future manipulations of the assembly will not affect the original particles that were added. As in the case of assembly, this method takes each particle to be added as an argument, rather than a list of particles. Use the * operator to add a list of particles.

2.2.26 `get_particle(iOrder)`

Returns a copy of the first particle in the assembly with an iOrder matching the value passed in. To edit the actual particle in the assembly, use list slicing. E.g. `<assembly>[0].m = 100`. Currently, this method can only accept a single iOrder value at a time.

2.2.27 `del_particles(*iOrders)`

Deletes all particles with iOrder values matching any in `iOrders` from the assembly. Any list arguments containing the iOrder values must be unpacked with the * operator. **Use caution when combining `del_particles()` with loops. Deleting particles from an assembly while iterating over its particles is equivalent to removing elements of a list while looping through it and can lead to unexpected behavior.**

2.2.28 `copy()`

Returns an independent copy of the assembly.

2.2.29 `sort_iOrder(direction='a')`

Sorts the assembly by iOrder. Optional `direction` argument allows sorting in ascending `'a'` or descending `'d'` order. The default is ascending.

2.2.30 `sort_iOrgIdx(direction='d')`

Sorts the assembly by iOrgIdx. Optional `direction` argument allows sorting in ascending `'a'` or descending `'d'` order. The default is descending.

2.2.31 `condense_iOrder(direction='a')`

Renumber iOrder values consecutively. Optional `direction` argument allows sorting in ascending `'a'` or descending `'d'` order. The default is ascending. There are no guarantees on which particles will get which iOrder value, just that the particles will be unchanged and the iOrder values will be consecutive beginning with zero (or the largest iOrder value if the descending option is chosen).

2.2.32 `condense(direction='a')`

Reassign iOrder and iOrgIdx values so that particles are sequentially ordered in both iOrgIdx and iOrder. The default order is ascending is the default, and `direction` may be set to `'d'` for descending order.

2.2.33 `rotate(axis, angle)`

Rotate the entire assembly by `angle` about `axis`. Both arguments must be non-zero. **Note: currently, the reference point for the rotation is about the origin.** To rotate the assembly in place, relocate the center or center of mass to zero, rotate, and move the assembly back. An option to specify the center of rotation should be added in the future.

2.3 Reading and writing ss files

`pkdtools` includes dedicated reading and writing functions in the `pkdio` module, which is imported automatically with `pkdtools`. Both functions rely on the existing `pkdgrav` module ssio for interfacing between Python and the binary ss format. An unmodified copy of ssio is currently included in pkdtools. Once pkdtools is formally included in the pkdgrav repository, the pkdgrav version of ssio may be referenced instead.

2.3.1 ss_in(filename, units='pkd')

Read in a pkdgrav ss file. This function makes use of the read_SS() function from ssio, but returns a typical pkdtools assembly structure. Units may be specified, with the default being pkdgrav units. The filename argument must be passed in as a string.

2.3.2 ss_out(assembly, filename)

Write an assembly to an ss file. ss_out() uses the write_SS() function from ssio. When writing an assembly, units will be automatically converted to pkdgrav units by ss_out before writing. ss_out() will warn the user when attempting to write assemblies containing duplicate or non-sequential iOrder values. Since ssio will (reasonably) not respect this input, ss_out() will in all cases make a copy of the assembly to be written and call the sort_iOrder() method on it before passing it to write_SS(). Calling condense() on the original assembly should result in a renumbering equivalent to what will eventually be written to an ss file by ssio, though this has not been exhaustively tested. Apart from the iOrder values and sequence, all particles from the assembly will be reflected faithfully in the written ss file.

2.4 Miscellaneous

This subsection describes several extra functions that didn't fit neatly elsewhere. All of these functions are included in the tools module and are imported automatically with pkdtools.

2.4.1 join(*assemblies, units='pkd')

Combine an arbitrary number of existing assemblies into one new assembly. Particles in the new assembly are copies of those in the input assemblies. Any manipulations of the new assembly will not affect the originals. pkdgrav units are the default.

2.4.2 subasbly(assembly, *iOrders, units=None)

Returns a new assembly with only the input iOrders. List arguments must be unpacked using the * operator. By default, units will be preserved from the original assembly.

2.4.3 viz(assembly, resolution=10)

Visualize an assembly of particles using matplotlib. The resolution argument determines how round or "blocky" the particles appear, with higher values taking more time to render. Unfortunately, for $\gtrsim$ 1000 particles viz() becomes prohibitively slow. This is because viz() is essentially making a 3D scatter plot using matplotlib, which is not designed for so many inputs and cannot easily handle this number of points. Using a standard scatter plot instead of plotting spheres does not fix this issue. Ultimately, the solution would be to use something like `mayavi` or `plotly` to render the particles, but I compromised here in the interest of accessibility since `matplotlib` is so much more widely available. An additional problem is that `matplotlib` cannot currently set an equal or "square" aspect ratio for 3D plots. As a workaround, `viz` determines the most extreme particle locations in x, y, and z and sets an equal range for all 3 axes to accommodate the worst case scenario. This results in a tolerable aspect ratio for the visualization.

2.5 Functions and Methods for Internal Use

`pkdtools` includes a number of functions and methods that are not intended to be called by the end user. In the future, all of these should probably be renamed to match the initial underscore convention used in `pkdgrav`. They are cataloged here for the benefit of anyone making changes to `pkdtools` in the future. Most of these functions are defined in the `util` module.

2.5.1 `iOrder_key(particle)`

A key function used by the `sort_iOrder()` method. Returns the particle's iOrder value.

2.5.2 `iOrgIdx_key(particle)`

A key function used by the `sort_iOrgIdx()` method. Returns the particle's iOrgIdx value.

2.5.3 `color_translate(pkd_color)`

A single-use utility for converting `pkdgrav` numeric color codes to `matplotlib` colors in `viz()`.

2.5.4 `vector_rotate(vector, axis, angle)`

A function for to perform a general rotation on a vector. Returns the rotated vector and leaves the original vector unaltered. This implementation was largely copied from `ssgen2Agg`. This function is used by the `rotate()` assembly method.

2.5.5 `angle_between(v1, v2)`

Return the angle between two vectors. Used when setting the orientation of generated regular aggs. There could be some funny issues here relating to the domain of `np.arccos` which JCM has not looked into very carefully. The current implementation was copied from a Stack Overflow post, which is linked in the `pkdtools.py` comments.

2.5.6 `makesphere(x, y, z, radius, resolution=10)`

Return the coordinates for plotting a "sphere" centered at `(x, y, z)`, though really it produces a sphere-like polyhedron. Increasing `resolution` increases the number of faces, giving a rounder look but increases time the time to render. Copied from a Stack Overflow post, which is linked in the `pkdtools.py` comments. This function is used by `viz()` for plotting spheres instead of scatter plot markers.

2.5.7 `convert(value='pkd')`

A particle method that changes particle units to `value`. Intended for internal `pkdtools` use. This function gets called when the particle `units` attribute is updated. Users should always change particle units by setting the units attribute to the desired units. `convert` is defined in the `particle` module.

The following six functions are called by the `convert()` particle method to handle unit conversions. Each function scales the input particle's attributes appropriately and returns nothing. These functions in turn make use of a series of constants defined in `pkdtools.py` (sourced from various locations around the internet) that encode the actual conversion factors. **NOTE:** DCR has pointed out that these constants may not match exactly with the constants used in `pkdgrav` or other existing utilies, which will cause small discrepancies to creep in. This should be investigated and reconciled.

2.5.8 `pkd2cgs(particle)`

2.5.9 `cgs2pkd(particle)`

2.5.10 `pkd2mks(particle)`

2.5.11 `mks2pkd(particle)`

2.5.12 `mks2cgs(particle)`

2.5.13 `cgs2mks(particle)`

3 `pkdtools.aggs`

`aggs` is a subpackage of `pkdtools`. There is no formal distinction between aggregates and assemblies in `pkdtools`—aggregates are simply assemblies whose particles all have the same iOrgIdx value. However, the `aggs` subpackage includes a number of useful assembly methods that are intended for use with aggregates as well as some functions for generating regular shapes like dumbbells, tetrahedra, etc. Given the increasing interest in aggregates and irregular shapes in `pkdgrav`, this package should grow significantly in the future. In particular, JVD has a wide array of improvements in the works for this package as of Fall '23 which should be fully incorporated soon.

3.1 Aggregate methods

3.1.1 `agg_max()`

Return the largest (negative) iOrgIdx value in the assembly.

3.1.2 `agg_min()`

Return the smallest (negative) iOrgIdx value in the assembly.

3.1.3 `agg_range()`

Returns a tuple with the minimum and maximum iOrgIdx values in the assembly.

3.1.4 `agg_list()`

Return a list of all iOrgIdx values in the assembly.

3.1.5 `N_aggs()`

Return the number of aggs in the assembly.

3.1.6 `get_agg(iOrgIdx)`

Return a new assembly consisting only of particles in the desired aggregate. Any particles in the new assembly are copies of the originals, and any manipulations should not affect original assembly. Currently, this method can only accept a single iOrgIdx value at once.

3.1.7 `del_aggs(*iOrgIdxs)`

Delete any particles with matching iOrgIdx values from the assembly. Use the * operator when passing a list of iOrgIdx values.

3.1.8 `pop_agg(iOrgIdx)`

Delete the agg from the assembly and return a copy of it. Currently, this method can only accept a single iOrgIdx value at once.

3.1.9 `fix_orphans()`

Find any single particles with a negative iOrgIdx value ("orphans") and set their iOrgIdx value equal to their iOrder value.

3.1.10 `all_aggs(assembly, units=None)`

Return a list of assemblies, each containing all of the particles from one agg in the assembly passed in. By default, units will match those of the assembly passed in.

3.2 Functions for Generating Regular Aggregates

`pkdtools` includes a set of functions for generating the 5 standard `pkdgrav` aggregate shapes included in `ssgen2Agg`. These functions generally work, but are a work in progress and may have some bugs or be updated in the future. Each function has a large number of possible arguments. While all arguments are optional individually, at a minimum the user must specify a mass and radius, or a particle mass and particle radius. The `mass` and `radius` arguments set a total mass and overall radius for the entire aggregate, while the `pmass` and `pradius` arguments define the mass and size of each constituent particle. An orientation may be specified, but the angle about the orientation axis cannot currently be set. For example, calling `make_diamond()` with a `orientation` set to (0,1,0) will align the long axis of the diamond with the y-axis, but no guarantees are made as to the orientation of the short axes. In the future, these functions could be split off into their own utility since they aren't really core functions. Some things here may need to be reconciled with `ssgen2Agg`.

3.2.1 `make_db(iOrder=0, iOrgIdx=-1, mass=0, radius=0, center=(0,0,0), orientation=(0,0,1), color=2, pmass=0, pradius=0, sep_coeff=np.sqrt(3), units='pkd')`

Generate an assembly consisting of a single 2-particle, dumbbell-shaped aggregate. The user may specify a mass and radius for the whole agg, or alternately for the particles in the agg by using the `pmass` and `pradius` arguments in lieu of the `mass` and `radius` arguments.

3.2.2 `make_diamond(iOrder=0, iOrgIdx=-1, mass=0, radius=0, center=(0,0,0), orientation=(0,0,1), color=12, pmass=0, pradius=0, sep_coeff=np.sqrt(3), units='pkd')`

Generate an assembly consisting of a single 4-particle, planar diamond-shaped aggregate.

3.2.3 `make_tetra(iOrder=0, iOrgIdx=-1, mass=0, radius=0, center=(0,0,0), orientation=(0,0,1), color=3, pmass=0, pradius=0, sep_coeff=np.sqrt(3), units='pkd')`

Generate an assembly consisting of a single 4-particle, tetrahedron-shaped aggregate.

3.2.4 `make_rod(iOrder=0, iOrgIdx=-1, mass=0, radius=0, center=(0,0,0), orientation=(0,0,1), color=5, pmass=0, pradius=0, sep_coeff=np.sqrt(3), units='pkd')`

Generate an assembly consisting of a single 4-particle, rod-shaped aggregate.

3.2.5 `make_cube(iOrder=0, iOrgIdx=-1, mass=0, radius=0, center=(0,0,0), orientation=(0,0,1), color=7, pmass=0, pradius=0, sep_coeff=np.sqrt(3), units='pkd')`

Generate an assembly consisting of a single 8-particle, cube-shaped aggregate.

4 `pkdtools.rp`

The `rp` package includes set of functions for managing assemblies or datasets that contain multiple distinct rubble piles. The core capability is `find_rp()`, which returns a list of assemblies, each of which represents a single rubble pile. Neighbor searches are conducted using a k-D tree by default, although the user may substitute their own tree function. Large portions of `rp` are modeled on and draw very heavily from the existing C-based `rpa` utility, especially `find_rp()` and its core supporting functions. When time allows, `rp` would benefit greatly from efficiency improvements, in particular for `find_rp()`. `find_rp()` is quite handy, but can be very slow for large N.

4.1 User functions

4.1.1 `find_rp(L=1.1, tree_func=scipy.spatial.KDTree, units=None)`

An assembly method that that returns a list of assemblies, each corresponding to a coherent fragment in the initial assembly. `find_rp()` is modeled on `rpa` and should give similar results. `L` is the linking scale and `tree_func` is used to reduce the time spent finding neighbors.

4.1.2 `rm_single(rp_list, particles=True, aggs=True, mass_frac=False, min_num=False)`

Returns a new list of rubble piles assemblies with any “singlets” (lone particles or lone aggregates) removed. Changing the `particles` or `aggs` inputs to `False` will toggle OFF this behavior. It can also remove all fragments/rubble piles below a given mass fraction (off by default).

4.1.3 `elong()`

An assembly method that returns the elongation of a rubble pile assembly.

4.1.4 `tot_M(rp_list)`

Returns the total mass of a list of rubble piles.

4.1.5 `max_M(rp_list)`

Returns the mass and mass fraction of the most massive fragment/rubble pile in `rp_list`.

4.2 Important utilities

4.2.1 `ok_to_merge(rp1, rp2, L)`

Determine whether two rubble piles meet the merger condition. From `rpa.c`: The following merger strategy is based entirely on geometry and does not take the gravitational potential into account. It is suitable for searching from the bottom up, that is, for starting with individual particles and linking them together into larger groups. The search takes the ellipsoidal shapes of the current groups into account. In order for `rp1` to be merged with `rp2`, spheres drawn with radii equal to the major axes of the bodies (times linking-scale) and centered on the bodies must overlap. If the scaled minor spheres also overlap, the bodies are merged. Failing that, if either body has its center of mass in the other’s scaled ellipsoid, the bodies are merged. Otherwise, no merge occurs. In contrast with `rpa`, linking-scale is passed in here as a parameter `L`, defaulting to 1.1. `rp1` and `rp2` must both be assemblies, containing one or more particles. Currently `ok_to_merge()` does not consider units. This shouldn't be a problem when used as a utility here, but use caution...

4.2.2 `find_closest(rp_com, tree, N=10)`

Identifies closest `N` rubble piles to `rp_com`.

4.2.3 `merge_rp(rp_list, locs, tree, L, units=None)`

Executes one merging pass over the rubble piles in `rp_list`. There is no guarantee that this will work nicely for anything other than a "good faith" rubble pile. This operation is currently very slow, and probably the source of the `fine_rp()` bottleneck.

4.2.4 `in_ellipsoid(r0, R, a)`

Returns True if a ball with radius `R` at `r0` lies entirely within the ellipsoid defined by semi-axes `a` (measured along the Cartesian axes and centered at the origin). To get this right, we need to compute direction from `r0` to nearest point on ellipsoid surface. This is too hard, so settle for more conservative boundary.

4.2.5 `overlap(pos1, pos2, R1, R2)`

Determine whether two spheres are overlapping, given their positions and radii.

4.2.6 `split_particles(units=None)`

Take a standard assembly and return a list of assemblies, each a singlet containing one of the original particles. `split_particles()` may also be called as an assembly method.

5 `pkdtools.constants`

The `constants` subpackage has three modules: `pkd`, `mks`, and `cgs`. Each module defines five useful constants in terms of its units: the gravitational constant G, the masses of the Sun and the Earth, and the radii of the Sun and the Earth. This package could easily grow or expand in scope.

6 `pkdtools.exp`

`exp` is essentially a "scratch" space for ideas that have not been fully implemented. There is currently only one orphaned function here. This package may go away in the future.

6.0.1 `embed_boulder(assembly, center, radius, units='pkd')`

Embed a spherical boulder in a rubble pile. The center argument is defined relative to the agg center of mass. This function is experimental and likely has some bugs to work out. It's intended for a niche use case for JCM, but is included here for the sake of completeness.

7 pkdtools.tidal

The tidal package contains tools that were developed to support the analysis of large suites of tidal disruption or spinup simulations (or any other processes that produce lots of rubble pile fragments). This was intended to support JCM's thesis work, but may end up being useful for other projects, especially the spinup work that JCM never completed. tidal is composed of three modules: dgrid, pgrid, and hist.

7.1 dgrid

The dgrid module handles the collection and storage of data from large suites of tidal runs. It defines the DataGrid class, which is used for the collection and storage of data from tidal runs. It also includes functions for traversing a large suite of runs and calling analysis functions, as well as a number of analysis metrics.

7.1.1 DataGrid

A new class intended to accumulate data as `walk_grid()` progresses through a given tidal suite of runs. DataGrid should be provided with a list of speed at infinity v_∞ values, a list of close approach distance q values, a filename, and an analysis function. In most cases, this function will ingest a list of rubble piles (i.e., output from `find_rp()`) and return a single value for a given run. DataGrid includes the following attributes:

- `q_list`: A list of the q values included in the suite of simulations
- `vinf_list`: A list of the v_∞ values included in the suite of simulations
- `filename`: The filename that the collected data will ultimately be written to.
- `func`: The analysis function that will be called for each simulation. This is where the main body of the grid analysis scheme happens.
- `grid`: A `pandas` dataframe size to match the q and v_∞ values supplied by the user. This is where the data will be stored.

Users must provide `q_list`, `vinf_list`, `filename`, and `func` upon initialization. DataGrid also has four associated methods:

- `read(q, vinf)` Returns the current value of `grid` at the given coordinates.
- `write(q, vinf, val)` Writes `val` to `grid` at the given coordinates.
- `save_csv()` Writes `grid` to `filename` in csv format. Intended for use with single value analysis functions.
- `save_pickle()` Writes `grid` to `filename` in pickle format.

7.1.2 `walk_grid(q_list, vinf_list, filename=None, fraggrids=[], stepgrids=[], othergrids=[], L=1.2, fragunits='pkd', stepunits='pkd', otherunits='pkd')`

Step through each run directory in the provided `DataGrid` objects and call the analysis functions on `filename`. `fragrids` should be a list of `DataGrid` objects for analyses that take a list of all fragments in `filename` as input. `stepgrids` should have analysis functions assigned that take q, v_∞, and a list of all steps in the run directory. `othergrids` should be a list of `DataGrid` objects for all analyses that do not meet this format, and require a more general initialization.

7.1.3 `analyze_frags(q, vinf, *fraggrids, filename=None, L=1.2, del_earth=True, units='pkd')`

Load data from `filename`, remove any Earth particle, and locate fragments using `find_rp()`. Any further fragment cleaning should happen in the analysis function, since this may be treated differently for different metrics. Grouping this way avoids calling `find_rp()` for each individual analysis, which can save quite a bit of time. Any `DataGrid` objects passed in will be updated. Nothing should be returned. If `filename` is not specified, `analyze_frags()` should automatically find the last valid output file written to the current run directory. Analysis functions called in this way must return a single numeric value and associated `DataGrid` objects will be saved in csv format

7.1.4 `analyze_steps(q, vinf, *stepgrids, del_earth=True, units='pkd')`

Similar to `analyze_frags()` in concept, but used for analyses that require stepping through all (or a subset of) output files, rather than considering only the final state. E.g., `tidal_threshold()`. Any function called in this way should accept q, v_∞, and a list of all steps in the run directory as arguments. This list is computed automatically in `analyze_steps()` by `steplist()`.Analysis functions called in this way must return a single numeric value and associated `DataGrid` objects will be saved in csv format

7.1.5 `analyze_other(q, vinf, *othergrids, units='pkd')`

We set aside calls to any functions that *do not* require fragment analysis, since `find_rp()` is expensive and we benefit substantially from only performing this calculation once per run. Also, we may want to use analysis functions that don't return numerical values, but instead return plots, or histograms, etc. Any analysis functions that don't fit the final state fragment or steplist analysis models well can be called here. No assumptions about inputs are made beyond q and v_∞.

7.1.6 `gen_walk(q_list, vinf_list, func)`

A much more general approach to traversing a suite of tidal encounter-style runs, in contrast to `walk_grid()`. Accepts q and v_∞ lists to be treated as a grid and a function to be called with no arguments. No assumptions are made about this function. Allows for maximum flexibility at the expense of efficiency.

7.1.7 `get_stepsize()`

Extract the `iOutInterval` values from the `ss.par` file in the current directory.

7.1.8 `get_zeropad()`

Extract the `nDigits` value from `ss.par` in the current directory.

7.1.9 `final_step(stepsize, zeropad)`

Find the final sequential `ss` output file written to the current directory.

7.1.10 `get_steps(stepsize, zeropad, init="initcond.ss")`

Return a list of all sequential `ss` output files in the current directory, beginning with `init`.

7.1.11 `pop_earth()`

Pop the Earth particle from an assembly and return, if one is present. May be called as a function or an assembly method.

7.1.12 `rm_earth()`

Remove the Earth particle from an assembly, if one is present. May be called as a function or an assembly method.

The remaining entries in this section are all analysis functions.

7.1.13 `largest_fragment_fraction(frags)`

Returns the fraction of total system mass of the most massive fragment in the list.

7.1.14 `largest_fragment_elongation(frags)`

Return the elongation of the most massive fragment in the list.

7.1.15 `largest_fragment_bound(frags)`

Returns the fraction of total system mass *bound* to the largest fragment, determined by a simple two-body calculation. In other words, if a a fragment in isolation is determined to be gravitationally bound to the largest fragment, its mass is added to the total. The total also includes the mass of the largest fragment itself.

7.1.16 `largest_fragment_orbiting(frags)`

Returns the fraction of total system mass in orbit about the largest fragment. This *excludes* the mass of the largest fragment itself.

7.1.17 `largest_fragment_period(frags)`

Returns the spin period of the largest post-encounter fragment in units of hours.

7.1.18 `N_fragments(frags)`

Returns the total number of fragments, after removing anything under 10 particles.

7.1.19 `largest_fragment_volcore(q, vinf, steplist, units='pkd')`

Returns the fraction of material in the largest fragment that came from the inner 50% by *volume* of the initial body.

7.1.20 `largest_fragment_radcore(q, vinf, steplist, units='pkd')`

Returns the fraction of material in the largest post-encounter fragment that came from the inner 50% by *radius* of the initial body. This is almost Identical to `largest_fragment_volcore()`, but with a different criterion for what constitutes the "core" of the initial rubble pile. Here, any particle closer to the center than half of the radius is considered the core. This is a more restrictive condition, so we should expect lower values.

7.1.21 `tidal_threshold(q, vinf, steplist, units='pkd')`

Returns the distance from the primary body at which the rubble pile is disrupted or reshaped. We consider the threshold for a reshaping in this context to be an increase of 0.5% in bulk radius.

7.1.22 `tidal_run_complete(q, vinf, steplist, units='pkd')`

Determine if a tidal run is done evolving. We first check whether the run has proceeded for more than three times the expected time to periapse. If it hasn't, we consider the run incomplete. If it has, we consider the mass fraction of the largest fragment. If it is greater than 95% at this point, the is likely a case of minimal tidal disturbance and we consider the run to be complete. If it isn't, we know that a major disruption did occur. In this case, we compare the mass fraction of the largest fragment in the final recorded output file to the largest mass fraction in the 10th to last output file. If these values are within 10% of each other, we consider the run to be finished evolving. This usually works well, but does give false negatives in the case where a messy fragment train causes the fragment finder to give slightly different results for different time steps.

7.2 `pgrid`

`pgrid` works in tandem with `dgrid` to allow for easy plotting of the `DataGrid` objects created by `dgrid`. The data is saved as a plain `pandas` dataframe and may be loaded and plotted with the functions below.

7.2.1 `load_csv_grid(filename)`

Read in a dataframe written from a `DataGrid` in csv format and process in preparation for plotting.

7.2.2 `load_pickle_grid(filename)`

Read in a dataframe written from a `DataGrid` in pickle format and process in preparation for plotting.

7.2.3 `plot_grid(title, rows, columns, datagrids=[], labels=[], cmaps=[], fig=None, axes=[], center=None, big_size=32, med_size=28, xtick_freq=2, show=True)`

Intended for plotting multiple data grids. User may specify the number and arrangement of plots, font sizes, labels, color maps (from the Seaborn plotting package), and more. The `fig` and `axes` arguments may be used if the user wants to call `plot_grid()` multiple times on different sets of data grids but show all of the resulting plots on the same figure. If an existing figure and axes are specified, the plots will be placed there. Otherwise, a new figure and axes will be created.

7.3 `hist`

An unfinished module designed to create histograms from tidal (or spinup) grid-based data.

7.3.1 `get_mass_hist(q, vinf, units='pkd')`

An analysis function designed to be called with `analyze_other()`. `get_mass_hist()` finds the final output file and calculates histogram data of fragment mass as a function of total system mass.

7.3.2 `get_frag_masses(q, vinf, units='pkd')`

An analysis function designed to be called with `analyze_other()`. Finds the final output file and returns a sorted list of all fragment masses as a function of total system mass.