`ssedit.py`

Last Update 1/26/23

Created by Julian C. Marohnic 1/26/23

Contents

1 Introduction

`ssedit` is a utility thats aids in creating and manipulating `pkdgrav` `ss` files. It does not rely on any existing `pkdgrav` utilities, with the exception of `ssio.py`. Currently, `ssedit` has no interactivity built in, though this may change in the future. Given the way it is designed, `ssedit.py` is most easily used via an interactive Python session (invoked with `python -i ssedit.py`), or by importing `ssedit` into a Python script. This document contains an inventory of the classes, methods, and functions introduced in `ssedit`, as well as miscellaneous information about best practices and usage.

2 Classes

`ssedit` introduces two new classes: `Particle` and `Assembly`, hereafter ”particles” and ”assemblies.” Particles in `ssedit` 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 `ssedit`.

2.1 Units

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. `ssedit` 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.

2.2 Particles

Particles in `ssedit` 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
```

2.3 Assemblies

The `Assembly` class is derived from the Python `list` type, and shares many of its features. An assembly only has one attribute of its own, which is `units`. 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 an optional units argument (the default being `pkdgrav` units). 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 `ssedit` 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 `ssedit` allows for things 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))
```

3 Functions and Methods

Both particles and assemblies have a number of class methods associated with them that provide much, though not all, of the functionality in `ssedit`. Both particles and assemblies have a `print` function implemented, which may be used to display the entire contents of the object. In addition, there are a number of general functions, as well as some methods and functions that are not meant to be called by the user, but are used internally.

3.1 General Functions

3.1.1 `ss_in(filename, units='pkd')`

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

3.1.2 `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.

3.1.3 `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.

3.1.4 `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.

3.1.5 `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.

3.1.6 `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. Intended for a niche use case for JCM, but included here for the sake of completeness. Could be removed or offloaded elsewhere in the future.

3.2 Particle Methods

3.2.1 `pos()`

Return particle position as a `pkdgrav` array.

3.2.2 `vel()`

Return particle velocity as a `numpy` array.

3.2.3 `spin()`

Return particle spin as a `numpy` array.

3.2.4 `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.

3.2.5 `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.

3.2.6 `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.

3.2.7 `copy()`

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

3.3 Assembly Methods

3.3.1 `N()`

Returns the number of particles in the assembly.

3.3.2 `M()`

Returns the total mass of the assembly.

3.3.3 `xbounds()`

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

3.3.4 `ybounds()`

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

3.3.5 `zbounds()`

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

3.3.6 `com()`

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

3.3.7 `comv()`

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

3.3.8 `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.

3.3.9 `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.

3.3.10 `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.

3.3.11 `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.

3.3.12 `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.

3.3.13 `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.

3.3.14 `avg_dens()`

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

3.3.15 `bulk_dens()`

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

3.3.16 `I()`

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

3.3.17 `axes()`

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

3.3.18 `L()`

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

3.3.19 `show_particles()`

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

3.3.20 `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.

3.3.21 `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.

3.3.22 `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.**

3.3.23 `copy()`

Returns an independent copy of the assembly.

3.3.24 `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.

3.3.25 `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.

3.3.26 `condense(direction='a')`

Renumbers iOrder values consecutively, in either ascending `'a'` or descending `'d'` order. 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).

3.3.27 `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. This behavior may change in the future.

3.4 Aggregate Methods

While there is no formal distinction between aggs and assemblies in `ssedit`, there are several assembly methods that are intended for use with aggs.

3.4.1 `agg_max()`

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

3.4.2 `agg_min()`

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

3.4.3 `agg_range()`

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

3.4.4 `agg_list()`

Return a list of all iOrgIdx values in the assembly.

3.4.5 `N_aggs()`

Return the number of aggs in the assembly.

3.4.6 `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.

3.4.7 `del_agg(iOrgIdx)`

Delete any particles with a matching iOrgIdx value from the assembly. Currently, this method can only accept a single iOrgIdx value at once.

3.4.8 `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.

3.4.9 `fix_orphans()`

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

3.5 Functions for Generating Regular Aggregates

`ssedit` 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`.

3.5.1 `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.

3.5.2 `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.

3.5.3 `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.

3.5.4 `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.

3.5.5 `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.

3.6 Functions and Methods for Internal Use

`ssedit` includes a number of functions and methods that are not intended to be called by the end user. These should probably be set aside or organized separately in the code, but they are not currently. They are cataloged here for the benefit of anyone making changes to `ssedit` in the future.

3.6.1 `iOrder_key(particle)`

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

3.6.2 `iOrgIdx_key(particle)`

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

3.6.3 `color_translate(pkd_color)`

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

3.6.4 `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.

3.6.5 `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 `ssedit.py` comments.

3.6.6 `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 `ssedit.py` comments. This function is used by `viz()` for plotting spheres instead of scatter plot markers.

3.6.7 `convert(value='pkd')`

A particle method that changes particle units to `value`. Intended for internal `ssedit` 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.

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 `ssedit.py` (sourced from various locations around the internet) that encode the actual conversion factors. Derek has pointed out that these constants may not match exactly with the constants used in `pkdgrav` or other existing utilies, which may cause small discrepancies to creep in. This should be investigated and reconciled.

3.6.8 `pkd2cgs(particle)`

3.6.9 `cgs2pkd(particle)`

3.6.10 `pkd2mks(particle)`

3.6.11 `mks2pkd(particle)`

3.6.12 `mks2cgs(particle)`

3.6.13 `cgs2mks(particle)`