galpy Documentation

Release v1.7.2

Jo Bovy

Jun 28, 2022

Contents

galpy is a Python package for galactic dynamics. It supports orbit integration in a variety of potentials, evaluating and sampling various distribution functions, and the calculation of action-angle coordinates for all static potentials. galpy is an astropy affiliated package and provides full support for astropy's Quantity framework for variables with units.

galpy is developed on GitHub. If you are looking to report an issue, join the galpy slack community, or for information on how to contribute to the code, please head over to galpy's GitHub page for more information.

As a preview of the kinds of things you can do with galpy, here's a video introducing some of the new features in galpy v1.5:

CHAPTER 1

Quick-start guide

1.1 Installation

1.1.1 Dependencies

galpy requires the `numpy`, `scipy`, and `matplotlib` packages; these must be installed or the code will not be able to be imported. The installation methods described below will all automatically install these required dependencies.

Optional dependencies are:

- `astropy` for Quantity support (used throughout galpy when installed),
- `astroquery` for the `Orbit.from_name` initialization method (to initialize using a celestial object's name),
- `tqdm` for displaying a progress bar for certain operations (e.g., orbit integration of multiple objects at once)
- `numexpr` for plotting arbitrary expressions of `Orbit` quantities,
- `numba` for speeding up the evaluation of certain functions when using C orbit integration,
- `JAX` for use of constant-anisotropy DFs in `galpy.df.constantbetadf`, and
- pynbody for use of `SnapshotRZPotential` and `InterpSnapshotRZPotential`.

To be able to use the fast C extensions for orbit integration and action-angle calculations, the GNU Scientific Library (GSL) needs to be installed (*see below*).

1.1.2 With conda

The easiest way to install the latest released version of galpy is using conda or pip (see below):

```
conda install galpy -c conda-forge
```

or:

```
conda config --add channels conda-forge
conda install galpy
```

Installing with conda will automatically install the required dependencies (`numpy`, `scipy`, and `matplotlib`) and the GSL, but not the optional dependencies.

1.1.3 With pip

galpy can also be installed using pip. Since v1.6.0, the pip installation will install binary *wheels* for most major operating systems (Mac, Windows, and Linux) and commonly-used Python 3 versions. When this is the case, you do not need to separately install the GSL.

When you are on a platform or Python version for which no binary wheel is available, pip will compile the source code on your machine. Some advanced features require the GNU Scientific Library (GSL; *see below*). If you want to use these with a pip-from-source install, install the GSL first (or install it later and re-install using the upgrade command. Then do:

```
pip install galpy
```

or to upgrade without upgrading the dependencies:

```
pip install -U --no-deps galpy
```

Installing with pip will automatically install the required dependencies (`numpy`, `scipy`, and `matplotlib`), but not the optional dependencies. On a Mac/UNIX system, you can make sure to include the necessary GSL environment variables by doing (see *below*):

```
export CFLAGS="$CFLAGS -I`gsl-config --prefix`/include" && export LDFLAGS="$LDFLAGS -
↪L`gsl-config --prefix`/lib" && pip install galpy
```

1.1.4 Latest version

The latest updates in galpy can be installed using:

```
pip install -U --no-deps git+https://github.com/jobovy/galpy.git#egg=galpy
```

or:

```
pip install -U --no-deps --install-option="--prefix=~/local" git+https://github.com/
↪jobovy/galpy.git#egg=galpy
```

for a local installation. The latest updates can also be installed from the source code downloaded from github using the standard python `setup.py` installation:

```
python setup.py install
```

or:

```
python setup.py install --prefix=~/local
```

for a local installation.

Note that these latest-version commands all install directly from the source code and thus require you to have the GSL and a C compiler installed to build the C extension(s). If you are having issues with this, you can also download a

binary wheel for the latest `main` version, which are available here. To install these wheels, download the relevant version for your operating system and Python version and do:

```
pip install WHEEL_FILE.whl
```

Note that there is also a Pure Python wheel available there, but use of this is not recommended. These wheels have stable . . . *latest*. . . names, so you can embed them in workflows that should always be using the latest version of *galpy* (e.g., to test your code against the latest development version).

1.1.5 Installing from a branch

If you want to use a feature that is currently only available in a branch, do:

```
pip install -U --no-deps git+https://github.com/jobovy/galpy.git@dev#egg=galpy
```

to, for example, install the `dev` branch.

Note that we currently do not build binary wheels for branches other than `main`. If you *really* wanted this, you could fork galpy, edit the GitHub Actions workflow file that generates the wheel to include the branch that you want to build (in the `on:` section), and push to GitHub; then the binary wheel will be built as part of your fork. Alternatively, you could do a pull request, which would also trigger the building of the wheels.

1.1.6 Installing from source on Windows

Tip: You can install a pre-compiled Windows "wheel" of the latest `main` version that is automatically built using `GitHub Actions` for all recent Python versions here. Download the wheel for your version of Python, and install with `pip install WHEEL_FILE.whl` (see above).

Versions >1.3 should be able to be compiled on Windows systems using the Microsoft Visual Studio C compiler (>= 2015). For this you need to first install the GNU Scientific Library (GSL), for example using Anaconda (*see below*). Similar to on a UNIX system, you need to set paths to the header and library files where the GSL is located. On Windows, using the CDM commandline, this is done as:

```
set INCLUDE=%CONDA_PREFIX%\Library\include;%INCLUDE%
set LIB=%CONDA_PREFIX%\Library\lib;%LIB%
set LIBPATH=%CONDA_PREFIX%\Library\lib;%LIBPATH%
```

If you are using the Windows PowerShell (which newer versions of the Anaconda prompt might set as the default), do:

```
$env:INCLUDE="$env:CONDA_PREFIX\Library\include"
$env:LIB="$env:CONDA_PREFIX\Library\lib"
$env:LIBPATH="$env:CONDA_PREFIX\Library\lib"
```

where in this example `CONDA_PREFIX` is the path of your current conda environment (the path that ends in `\ENV_NAME`). If you have installed the GSL somewhere else, adjust these paths (but do not use `YOUR_PATH\include\gsl` or `YOUR_PATH\lib\gsl` as the paths, simply use `YOUR_PATH\include` and `YOUR_PATH\lib`).

To compile with OpenMP on Windows, you have to install Intel OpenMP via:

```
conda install -c anaconda intel-openmp
```

and then to compile the code:

```
python setup.py install
```

If you encounter any issue related to OpenMP during compilation, you can do:

```
python setup.py install --no-openmp
```

1.1.7 Installing from source with Intel Compiler

Compiling galpy with an Intel Compiler can give significant performance improvements on 64-bit Intel CPUs. Moreover students can obtain a free copy of an Intel Compiler at this link.

To compile the galpy C extensions with the Intel Compiler on 64bit MacOS/Linux do:

```
python setup.py build_ext --inplace --compiler=intelem
```

and to compile the galpy C extensions with the Intel Compiler on 64bit Windows do:

```
python setup.py build_ext --inplace --compiler=intel64w
```

Then you can simply install with:

```
python setup.py install
```

or other similar installation commands, or you can build your own wheels with:

```
python setup.py sdist bdist_wheel
```

1.1.8 Installing the TorusMapper code

Warning: The TorusMapper code is *not* part of any of galpy's binary distributions (installed using conda or pip); if you want to gain access to the TorusMapper, you need to install from source as explained in this section and above.

Since v1.2, `galpy` contains a basic interface to the TorusMapper code of Binney & McMillan (2016). This interface uses a stripped-down version of the TorusMapper code, that is not bundled with the galpy code, but kept in a fork of the original TorusMapper code. Installation of the TorusMapper interface is therefore only possible when installing from source after downloading or cloning the galpy code and using the `python setup.py install` method above.

To install the TorusMapper code, *before* running the installation of galpy, navigate to the top-level galpy directory (which contains the `setup.py` file) and do:

```
git clone https://github.com/jobovy/Torus.git galpy/actionAngle/actionAngleTorus_c_
↪ext/torus
cd galpy/actionAngle/actionAngleTorus_c_ext/torus
git checkout galpy
cd -
```

Then proceed to install galpy using the `python setup.py install` technique or its variants as usual.

1.1.9 Installation FAQ

What is the required `numpy` version?

`galpy` should mostly work for any relatively recent version of `numpy`, but some advanced features, including calculating the normalization of certain distribution functions using Gauss-Legendre integration require `numpy` version 1.7.0 or higher.

I get warnings like "galpyWarning: libgalpy C extension module not loaded, because libgalpy.so image was not found"

This typically means that the GNU Scientific Library (GSL) was unavailable during galpy's installation, causing the C extensions not to be compiled. Most of the galpy code will still run, but slower because it will run in pure Python. The code requires GSL versions >= 1.14. If you believe that the correct GSL version is installed for galpy, check that the library can be found during installation (see *below*).

I get the warning "galpyWarning: libgalpy_actionAngleTorus C extension module not loaded, because libgalpy_actionAngleTorus.so image was not found"

This is typically because the TorusMapper code was not compiled, because it was unavailable during installation. This code is only necessary if you want to use `galpy.actionAngle.actionAngleTorus`. See *above* for instructions on how to install the TorusMapper code. Note that in recent versions of galpy, you should *not* be getting this warning, unless you set `verbose=True` in the *configuration file*.

How do I install the GSL?

Certain advanced features require the GNU Scientific Library (GSL), with action calculations requiring version 1.14 or higher. The easiest way to install this is using its Anaconda build:

```
conda install -c conda-forge gsl
```

If you do not want to go that route, on a Mac, the next easiest way to install the GSL is using Homebrew as:

```
brew install gsl --universal
```

You should be able to check your version using (on Mac/Linux):

```
gsl-config --version
```

On Linux distributions with `apt-get`, the GSL can be installed using:

```
apt-get install libgsl0-dev
```

or on distros with `yum`, do:

```
yum install gsl-devel
```

The `galpy` installation fails because of C compilation errors

`galpy`'s installation can fail due to compilation errors, which look like:

```
error: command 'gcc' failed with exit status 1
```

or:

```
error: command 'clang' failed with exit status 1
```

or:

```
error: command 'cc' failed with exit status 1
```

This is typically because the compiler cannot locate the GSL header files or the GSL library. You can tell the installation about where you've installed the GSL library by defining (for example, when the GSL was installed under `/usr`; the `LD_LIBRARY_PATH` part of this may or may not be necessary depending on your system):

```
export CFLAGS=-I/usr/include
export LDFLAGS=-L/usr/lib
export LD_LIBRARY_PATH=-L/usr/lib
```

or:

```
setenv CFLAGS -I/usr/include
setenv LDFLAGS -L/usr/lib
setenv LD_LIBRARY_PATH -L/usr/lib
```

depending on your shell type (change the actual path to the include and lib directories that have the gsl directory). If you already have `CFLAGS`, `LDFLAGS`, and `LD_LIBRARY_PATH` defined you just have to add the `'-I/usr/include'` and `'-L/usr/lib'` to them.

If you are on a Mac or UNIX system (e.g., Linux), you can find the correct `CFLAGS` and `LDFLAGS`/`LD_LIBRARY_path` entries by doing:

```
gsl-config --cflags
gsl-config --prefix
```

where you should add `/lib` to the output of the latter. In a bash shell, you could also simply do:

```
export CFLAGS="$CFLAGS -I`gsl-config --prefix`/include" && export LDFLAGS="$LDFLAGS -
↪L`gsl-config --prefix`/lib" && pip install galpy
```

or:

```
export CFLAGS="$CFLAGS -I`gsl-config --prefix`/include" && export LDFLAGS="$LDFLAGS -
↪L`gsl-config --prefix`/lib" && python setup.py install
```

depending on whether you are installing using `pip` or from source.

I have defined `CFLAGS`, `LDFLAGS`, and `LD_LIBRARY_PATH`, but the compiler does not seem to include these and still returns with errors

This typically happens if you install using `sudo`, but have defined the `CFLAGS` etc. environment variables without using sudo. Try using `sudo -E` instead, which propagates your own environment variables to the `sudo` user.

I'm having issues with OpenMP

galpy uses OpenMP to parallelize various of the computations done in C. galpy can be installed without OpenMP by specifying the option `--no-openmp` when running the `python setup.py` commands above:

```
python setup.py install --no-openmp
```

or when using pip as follows:

```
pip install -U --no-deps --install-option="--no-openmp" git+https://github.com/jobovy/
↪galpy.git#egg=galpy
```

or:

```
pip install -U --no-deps --install-option="--prefix=~/local" --install-option="--no-
↪openmp" git+https://github.com/jobovy/galpy.git#egg=galpy
```

for a local installation. This might be useful if one is using the `clang` compiler, which is the new default on macs with OS X (>= 10.8), but does not support OpenMP. `clang` might lead to errors in the installation of galpy such as:

```
ld: library not found for -lgomp

clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

If you get these errors, you can use the commands given above to install without OpenMP, or specify to use `gcc` by specifying the `CC` and `LDSHARED` environment variables to use `gcc`. Note that `clang` does not seem to have this issue anymore in more recent versions, but it still does not support `OpenMP`.

1.1.10 Configuration file

Since v1.2, `galpy` uses a configuration file to set a small number of configuration variables. This configuration file is parsed using ConfigParser/configparser. It is currently used:

- to set a default set of distance and velocity scales (`ro` and `vo` throughout galpy) for conversion between physical and internal galpy unit
- to decide whether to use seaborn plotting with galpy's defaults (which affects *all* plotting after importing `galpy.util.plot`),
- to specify whether output from functions or methods should be given as an astropy Quantity with units as much as possible or not, and whether or not to use astropy's coordinate transformations (these are typically somewhat slower than galpy's own coordinate transformations, but they are more accurate and more general)
- to set the level of verbosity of galpy's warning system (the default `verbose=False` turns off non-crucial warnings).

The current configuration file therefore looks like this:

```
[normalization]
ro = 8.
vo = 220.

[plot]
seaborn-bovy-defaults = False

[astropy]
astropy-units = False
astropy-coords = True

[warnings]
verbose = False
```

where `ro` is the distance scale specified in kpc, `vo` the velocity scale in km/s, and the setting is to *not* return output as a Quantity. These are the current default settings.

A user-wide configuration file should be located at `$HOME/.galpyrc`. This user-wide file can be overridden by a `$PWD/.galpyrc` file in the current directory. If no configuration file is found, the code will automatically write the default configuration to `$HOME/.galpyrc`. Thus, after installing galpy, you can simply use some of its simplest functionality (e.g., integrate an orbit), and after this the default configuration file will be present at `$HOME/.galpyrc`. If you want to change any of the settings (for example, you want Quantity output), you can edit this file. The default configuration file can also be found `here`.

1.2 What's new?

This page gives some of the key improvements in each galpy version. See the `HISTORY.txt` file in the galpy source for full details on what is new and different in each version.

1.2.1 v1.7

Version 1.7 adds many new features, mainly in the `galpy.potential` and `galpy.df` modules. The biggest new additions are:

- A general framework for spherical distribution functions defined using $f(E,L)$ models. Specifically, general solutions for (a) isotropic $f(E)$ models, (b) $f(E,L)$ models with constant anisotropy β, and (c) $f(E,L)$ models with Osipkov-Merritt-type anisotropy are implemented for any potential/density pair (not necessarily self-consistent). These distribution functions can be evaluated, sampled exactly, and any moment of the distribution function can be calculated. Documentation of this is currently available at *NEW in v1.7 Spherical distribution functions*. Distribution functions with constant anisotropy require the JAX.
- In addition to the general solution, the distribution function of a few well-known distribution functions was added, including (a) Hernquist distribution functions that are isotropic, have constant anisotropy, or have Osipkov-Merritt type anisotropy; (b) an isotropic Plummer profile; (c) the isotropic NFW profile (either using the approx. from Widrow 2000 or using an improved approximation) and the Osipkov-Merritt NFW profile (new approximate form); (d) the King model (also added as a potential as KingPotential).

Other new additions include:

- New or improved potentials and *potential wrappers*:
  - *interpSphericalPotential*: general class to build interpolated versions of spherical potentials.
  - *AdiabaticContractionWrapperPotential*: wrapper potential to adiabatically contract a spherical dark-matter halo in response to the adiabatic growth of a baryonic component.
  - *TriaxialGaussianPotential*: potential of a Gaussian stratified on triaxial ellipsoids (Emsellem et al. 1994).
  - *PowerTriaxialPotential*: potential of a triaxial power-law density (like `PowerSphericalPotential`, but triaxial).
  - *AnyAxisymmetricRazorThinDiskPotential*: potential of an arbitrary razor-thin axisymmetric disk (not in C, mainly useful for rotation-curve modeling).
  - *AnySphericalPotential*: potential of an arbitrary spherical density distribution (not in C, mainly useful for rotation-curve modeling).
  - *RotateAndTiltWrapperPotential*: wrapper potential to re-orient a potential arbitrarily in three dimensions.
- Other changes to Potential classes, methods, and functions:

  - Functions to compute the SCF density/potential expansion coefficients based on an N-body representation of the density (*scf_compute_coeffs_spherical_nbody*, *scf_compute_coeffs_axi_nbody*, and *scf_compute_coeffs_nbody*).
  - An *NFWPotential* can now be initialized using `rmax/vmax`, the radius and value of the maximum circular velocity.
  - Potential functions and methods to compute the zero-velocity curve: `zvc` and `zvc_range`. The latter computes the range in R over which the zero-velocity curve is defined, the former gives the positive z position on the zero-velocity curve for a given radius in this range.
  - `rhalf` Potential function/method for computing the half-mass radius.
  - `tdyn` Potential function/method for computing the dynamical time using the average density.
  - `Potential.mass` now always returns the mass within a spherical shell if only one argument is given. Implemented faster versions of many mass implementations using Gauss' theorem (including *SCFPotential* and *DiskSCFPotential*).
  - Mixed azimuthal,vertical 2nd derivatives for all non-axisymmetric potentials in function `evaluatephizderivs` and method `phizderiv`. Now all second derivatives in cylindrical coordinates are implemented.
  - Function/method `plotSurfaceDensities/plotSurfaceDensity` for plotting, you'll never guess, the surface density of a potential.
  - Re-implementation of `DoubleExponentialDiskPotential` using the double-exponential formula for integrating Bessel functions, resulting in a simpler, more accurate, and more stable implementation. This potential is now accurate to ~machine precision.
  - Potentials are now as much as possible numerically stable at `r=0` and `r=inf`, meaning that they can be evaluated there.

Other additions and changes include:

- Added the inverse action-angle transformations for the isochrone potential (in *actionAngleIsochroneInverse*) and for the one-dimensional harmonic oscillator (in *actionAngleHarmonicInverse*). Also added the action-angle calculation for the harmonic oscilator in *actionAngleHarmonic*. Why yes, I have been playing around with the TorusMapper a bit!
- Renamed `galpy.util.bovy_coords` to `galpy.util.coords`, `galpy.util.bovy_conversion` to `galpy.util.conversion`, and `galpy.util.bovy_plot` to `galpy.util.plot` (but old `from galpy.util import bovy_X` will keep working for now). Also renamed some other internal utility modules in the same way (`bovy_symplecticode`, `bovy_quadpack`, and `bovy_ars`; these are not kept backwards-compatible). Trying to make the code a bit less egotistical!
- Support for Python 3.9.

1.2.2 v1.6

This version mainly consists of changes to the internal functioning of `galpy`; some of the new outward-facing features are:

- ChandrasekharDynamicalFrictionForce is now implemented in C, leading to 100x to 1000x speed-ups for orbit integrations using dynamical friction compared to the prior pure-Python version.
- New potentials:
  - HomogeneousSpherePotential: the potential of a constant density sphere out to some radius R.
  - DehnenSphericalPotential: the Dehnen Spherical Potential from Dehnen (1993).

  - DehnenCoreSphericalPotential: the Dehnen Spherical Potential from (Dehnen 1993) with alpha=0 (corresponding to an inner core).

- Some notable internal changes:
  - Fixed a bug in how `DiskSCFPotential` instances are passed to C for orbit integration that in particular affected the `McMillan17` Milky-Way potential (any hole in the surface density was effectively ignored in the C code in v1.5).
  - The performance of orbit animations is significantly improved.
  - All main galpy C extensions are now compiled into a single shared-object library `libgalpy`.
  - Binary wheels are now automatically built for Windows, Mac, and most major Linux distributions upon every push to the `master` (now `main`) branch and these are automatically uploaded to PyPI upon release. See the *Installation Instructions* for more info. Binary wheels on Windows are also built for every push on AppVeyor, see the *Windows installation instructions*.

1.2.3 v1.5

This version will be the last to support Python 2.7 as this version of Python is reaching end-of-life on January 1 2020.

- This version's highlight is a fully re-written implementation of `galpy.orbit.Orbit` such that it can now contain and manipulate multiple objects at once. `galpy.orbit.Orbit` can be initialized with an arbitrary shape of input objects in a *variety of ways*, manipulated in a manner similar to Numpy arrays, and all `Orbit` methods work efficiently on `Orbit` instances containing multiple objects. Some methods, such as *orbit integration* and those for *fast orbital characterization* are parallelized on multi-core machines. `Orbit` instances can contain and manipulate millions of objects simultaneously now.
- Added the `galpy.potentials.mwpotentials` module with various Milky-Way-like potentials. Currently included are MWPotential2014, McMillan17 for the potential from McMillan (2017), models 1 through 4 from Dehnen & Binney (1998), and the three models from Irrgang et al. (2013). See *this section of the API documentation* for details.
- Added a (JSON) list with the phase-space coordinates of known objects (mainly Milky Way globular clusters and dwarf galaxies) for easy *Orbit.from_name initialization*. For ease of use, Orbit.from_name also supports tab completion for known objects in this list in IPython/Jupyter.
- Added `galpy.potential.to_amuse` to create an AMUSE representation of any galpy potential, *allowing galpy potentials to be used as external gravitational fields in AMUSE N-body simulations*.
- New or improved potentials and *potential wrappers*:
  - MovingObjectPotential: Re-wrote `potential.MovingObjectPotential` to allow general mass distributions for the moving object, implemented now as standard galpy potentials. Also added a C implementation of this potential for fast orbit integration.
  - IsothermalDiskPotential: The one-dimensional potential of an isothermal self-gravitating disk (sech^2 profile).
  - NumericalPotentialDerivativesMixin: a Mixin class to add numerically-computed forces and second derivatives to any Potential class, allowing new potentials to be implmented quickly by only implementing the potential itself and obtaining all forces and second derivatives numerically.
  - DehnenSmoothWrapperPotential: Can now decay rather than grow a potential by setting `decay=True`.
  - Added support to combine Potential instances or lists thereof through the addition operator. E.g., `pot=pot1+pot2+pot3` to create the combined potential of the three component potentials (pot1,pot2,pot3). Each of these components can be a combined potential itself. As before, combined potentials are simply lists of potentials, so this is simply an alternative (and perhaps more intuitive) way to create these lists.

  - Added support to adjust the amplitude of a Potential instance through multiplication of the instance by a number or through division by a numer. E.g., `pot= 2.*pot1` returns a Potential instance that is the same as pot1, except that the amplitude is twice larger. Similarly, `pot= pot1/2.` decreases the amplitude by a factor of two. This is useful, for example, to quickly change the mass of a potential. Only works for Potential instances, not for lists of Potential instances.

- New or improved `galpy.orbit.Orbit` functionality and methods:
  - Added support for 1D orbit integration in C.
  - Added support to plot arbitrary combinations of the basic Orbit attributes by giving them as an expresion (e.g., `orb.plot(d2='vR*R/r+vz*z/r')`; requires the numexpr package.
  - Switched default Sun's vertical height zo parameter for Orbit initialization to be the value of 20.8 pc from Bennett & Bovy (2019).
  - Add Python and C implementation of Dormand-Prince 8(5,3) integrator.

1.2.4 v1.4

- Added dynamical friction as the ChandrasekharDynamicalFrictionForce class, an implementation of dynamical friction based on the classical Chandrasekhar formula (with recent tweaks from the literature to better represent the results from N-body simulations).
- A general `EllipsoidalPotential` superclass for implementing potentials with densities that are constant on ellipsoids (functions of $m^2 = x^2 + y^2/b^2 + z^2/c^2$). Also implemented in C. Implementing new types of ellipsoidal potentials now only requires three simple functions to be defined: the density as a function of m, its derivative with respect to m, and its integral with respect to m^2. Makes implementing any ellipsoidal potential a breeze. See examples in the new-potentials section below.
- New or improved potentials and *potential wrappers*:
  - CorotatingRotationWrapperPotential: wrapper to make a pattern (e.g., a SpiralArmsPotential) wind up over time such that it is always corotating (see Hunt et al. (2018) for an example of this).
  - GaussianAmplitudeWrapperPotential: wrapper to modulate the amplitude of a (list of) `Potential` (s) with a Gaussian.
  - PerfectEllipsoidPotential: Potential of a perfect triaxial ellipsoid (de Zeeuw 1985).
  - SphericalShellPotential: Potential of a thin, spherical shell.
  - RingPotential: Potential of a circular ring.
  - Re-implemented `TwoPowerTriaxialPotential`, `TriaxialHernquistPotential`, `TriaxialJaffePotential`, and `TriaxialNFWPotential` using the general `EllipsoidalPotential` class.
- New `Potential` methods and functions:
  - Use nested lists of `Potential` instances wherever lists of `Potential` instances can be used. Allows easy adding of components (e.g., a bar) to previously defined potentials (which may be lists themselves): new_pot= [pot,bar_pot].
  - rtide and ttensor: compute the tidal radius of an object and the full tidal tensor.
  - surfdens method and evaluateSurfaceDensities function to evaluate the surface density up to a given z.
  - r2deriv and evaluater2derivs: 2nd derivative wrt spherical radius.
  - evaluatephi2derivs: second derivative wrt phi.
  - evaluateRphiderivs: mixed (R,phi) derivative.

- New or improved `galpy.orbit.Orbit` functionality and methods:
  - Orbit.from_name to initialize an `Orbit` instance from an object's name. E.g., `orb= Orbit.from_name('LMC')`.
  - Orbit initialization without arguments is now the orbit of the Sun.
  - Orbits can be initialized with a SkyCoord.
  - Default `solarmotion=` parameter is now 'schoenrich' for the Solar motion of Schoenrich et al. (2010).
  - rguiding: Guiding-center radius.
  - Lz: vertical component of the angular momentum.
  - If astropy version > 3, Orbit.SkyCoord method returns a SkyCoord object that includes the velocity information and the Galactocentric frame used by the Orbit instance.
- `galpy.df.jeans` module with tools for Jeans modeling. Currently only contains the functions sigmar and sigmalos to calculate the velocity dispersion in the radial or line-of-sight direction using the spherical Jeans equation in a given potential, density profile, and anisotropy profile (anisotropy can be radially varying).
- Support for compilation on Windows with MSVC.

1.2.5 v1.3

- A fast and precise method for approximating an orbit's eccentricity, peri- and apocenter radii, and maximum height above the midplane using the Staeckel approximation (see Mackereth & Bovy 2018). Can determine these parameters to better than a few percent accuracy in as little as 10 μs per object, more than 1,000 times faster than through direct orbit integration. See *this section* of the documentation for more info.
- A general method for modifying `Potential` classes through potential wrappers—simple classes that wrap existing potentials to modify their behavior. See *this section* of the documentation for examples and *this section* for information on how to easily define new wrappers. Example wrappers include SolidBodyRotationWrapperPotential to allow *any* potential to rotate as a solid body and DehnenSmoothWrapperPotential to smoothly grow *any* potential. See *this section of the galpy.potential API page* for an up-to-date list of wrappers.
- New or improved potentials:
  - DiskSCFPotential: a general Poisson solver well suited for galactic disks
  - Bar potentials SoftenedNeedleBarPotential and FerrersPotential (latter only in Python for now)
  - 3D spiral arms model SpiralArmsPotential
  - Henon & Heiles (1964) potential HenonHeilesPotential
  - Triaxial version of LogarithmicHaloPotential
  - 3D version of DehnenBarPotential
  - Generalized version of CosmphiDiskPotential
- New or improved `galpy.orbit.Orbit` methods:
  - Method to display an animation of an integrated orbit in jupyter notebooks: Orbit.animate. See *this section* of the documentation.
  - Improved default method for fast calculation of eccentricity, zmax, rperi, rap, actions, frequencies, and angles by switching to the Staeckel approximation with automatically-estimated approximation parameters.
  - Improved plotting functions: plotting of spherical radius and of arbitrary user-supplied functions of time in Orbit.plot, Orbit.plot3d, and Orbit.animate.
- `actionAngleStaeckel` upgrades:

  - `actionAngleStaeckel` methods now allow for different focal lengths delta for different phase-space points and for the order of the Gauss-Legendre integration to be specified (default: 10, which is good enough when using actionAngleStaeckel to compute approximate actions etc. for an axisymmetric potential).
  - Added an option to the estimateDeltaStaeckel function to facilitate the return of an estimated delta parameter at every phase space point passed, rather than returning a median of the estimate at each point.

- galpy.df.schwarzschilddf:the simple Schwarzschild distribution function for a razor-thin disk (useful for teaching).

1.2.6 v1.2

- Full support for providing inputs to all initializations, methods, and functions as astropy Quantity with units and for providing outputs as astropy Quantities.
- `galpy.potential.TwoPowerTriaxialPotential`, a set of triaxial potentials with iso-density contours that are arbitrary, similar, coaxial ellipsoids whose 'radial' density is a (different) power-law at small and large radii: 1/m^alpha/(1+m)^beta-alpha (the triaxial generalization of TwoPowerSphericalPotential, with flattening in the density rather than in the potential; includes triaxial Hernquist and NFW potentials.
- `galpy.potential.SCFPotential`, a class that implements general density/potential pairs through the basis expansion approach to solving the Poisson equation of Hernquist & Ostriker (1992). Also implemented functions to compute the coefficients for a given density function. See more explanation *here*.
- `galpy.actionAngle.actionAngleTorus`: an experimental interface to Binney & McMillan's TorusMapper code for computing positions and velocities for given actions and angles. See the installation instructions for how to properly install this. See *this section* and the `galpy.actionAngle` API page for documentation.
- `galpy.actionAngle.actionAngleIsochroneApprox` (Bovy 2014) now implemented for the general case of a time-independent potential.
- `galpy.df.streamgapdf`, a module for modeling the effect of a dark-matter subhalo on a tidal stream. See Sanders et al. (2016). Also includes the fast methods for computing the density along the stream and the stream track for a perturbed stream from Bovy et al. (2016).
- `Orbit.flip` can now flip the velocities of an orbit in-place by specifying `inplace=True`. This allows correct velocities to be easily obtained for backwards-integrated orbits.
- `galpy.potential.PseudoIsothermalPotential`, a standard pseudo-isothermal-sphere potential. `galpy.potential.KuzminDiskPotential`, a razor-thin disk potential.
- Internal transformations between equatorial and Galactic coordinates are now performed by default using astropy's coordinates module. Transformation of (ra,dec) to Galactic coordinates for general epochs.

1.2.7 v1.1

- Full support for Python 3.
- `galpy.potential.SnapshotRZPotential`, a potential class that can be used to get a frozen snapshot of the potential of an N-body simulation.
- Various other potentials: `PlummerPotential`, a standard Plummer potential; `MN3ExponentialDiskPotential`, an approximation to an exponential disk using three Miyamoto-Nagai potentials (Smith et al. 2015); `KuzminKutuzovStaeckelPotential`, a Staeckel potential that can be used to approximate the potential of a disk galaxy (Batsleer & Dejonghe 1994).
- Support for converting potential parameters to NEMO format and units.

- Orbit fitting in custom sky coordinates.

1.3 Introduction

The most basic features of galpy are its ability to display rotation curves and perform orbit integration for arbitrary combinations of potentials. This section introduce the most basic features of `galpy.potential` and `galpy.orbit`.

1.3.1 Rotation curves

The following code example shows how to initialize a Miyamoto-Nagai disk potential and plot its rotation curve

```
>>> from galpy.potential import MiyamotoNagaiPotential
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,normalize=1.)
>>> mp.plotRotcurve(Rrange=[0.01,10.],grid=1001)
```

The `normalize=1.` option normalizes the potential such that the radial force is a fraction `normalize=1.` of the radial force necessary to make the circular velocity 1 at R=1. Starting in v1.2 you can also initialize potentials with amplitudes and other parameters in physical units; see below and other parts of this documentation.

Tip: You can copy all of the code examples in this documentation to your clipboard by clicking the button in the top, right corner of each example. This can be directly pasted into a Python interpreter (including the >>>).

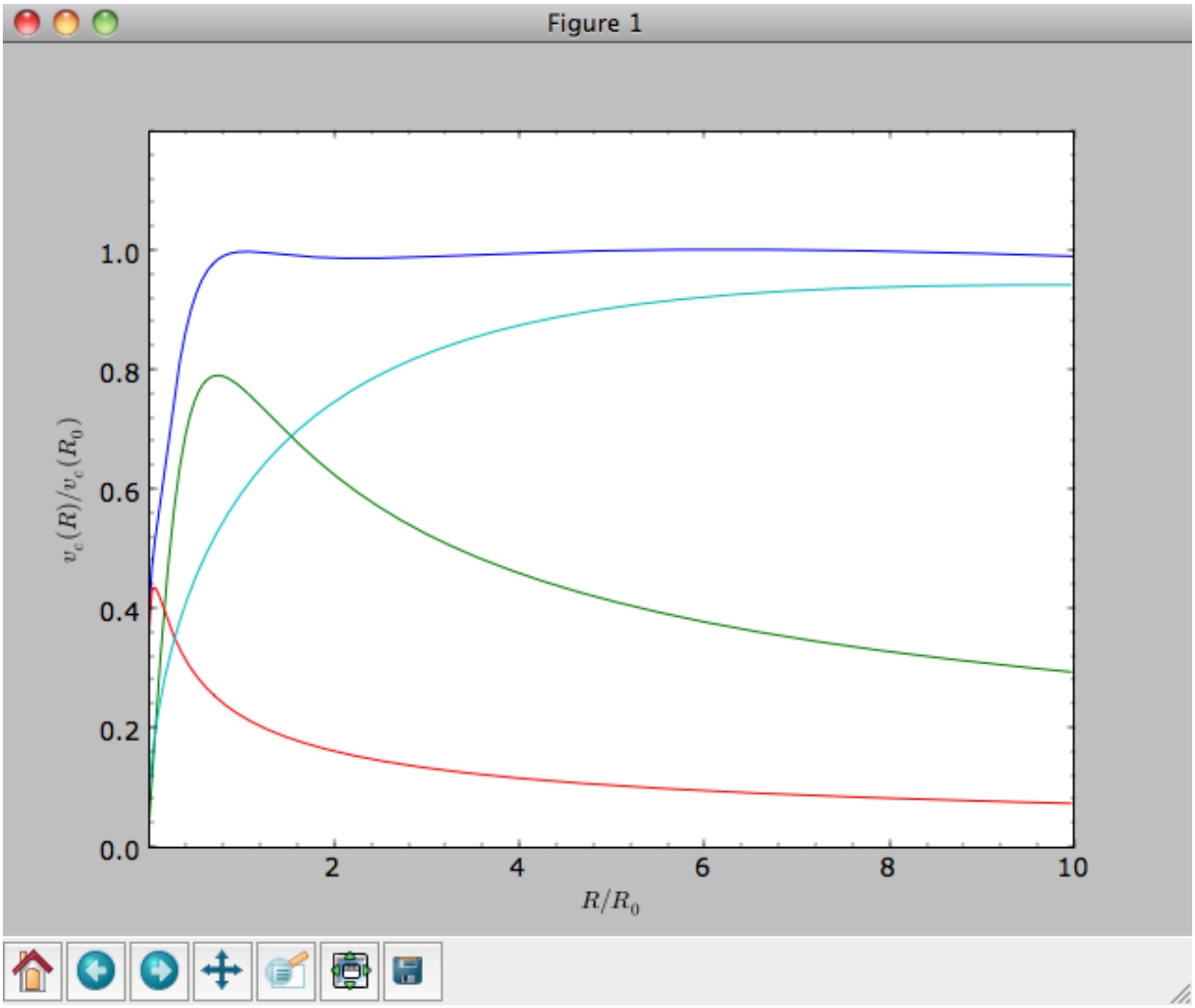
Similarly we can initialize other potentials and plot the combined rotation curve

```
>>> from galpy.potential import NFWPotential, HernquistPotential
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,normalize=.6)
>>> np= NFWPotential(a=4.5,normalize=.35)
>>> hp= HernquistPotential(a=0.6/8,normalize=0.05)
>>> from galpy.potential import plotRotcurve
>>> plotRotcurve(hp+mp+np,Rrange=[0.01,10.],grid=1001,yrange=[0.,1.2])
```

Note that the `normalize` values add up to 1. such that the circular velocity will be 1 at R=1. Potentials can be combined into a composite potential either by combining them in a list as `[hp,mp,np]` or by adding them up `hp+mp+np` (the latter simply returns the list `[hp,mp,np]`). The resulting rotation curve is approximately flat. To show the rotation curves of the three components do

```
>>> mp.plotRotcurve(Rrange=[0.01,10.],grid=1001,overplot=True)
>>> hp.plotRotcurve(Rrange=[0.01,10.],grid=1001,overplot=True)
>>> np.plotRotcurve(Rrange=[0.01,10.],grid=1001,overplot=True)
```

You'll see the following



As a shortcut the `[hp,mp,np]` Milky-Way-like potential is defined as

```
>>> from galpy.potential import MWPotential
```

This is *not* the recommended Milky-Way-like potential in `galpy`. The (currently) recommended Milky-Way-like potential is `MWPotential2014`:

```
>>> from galpy.potential import MWPotential2014
```

`MWPotential2014` has a more realistic bulge model and is actually fit to various dynamical constraints on the Milky Way (see *here* and the `galpy` paper).

1.3.2 Units in galpy

Internal (natural) units

Above we normalized the potentials such that they give a circular velocity of 1 at R=1. These are the standard galpy units (sometimes referred to as *natural units* in the documentation). galpy will work most robustly when using these natural units. When using galpy to model a real galaxy with, say, a circular velocity of 220 km/s at R=8 kpc, all of the

velocities should be scaled as v= V/[220 km/s] and all of the positions should be scaled as x = X/[8 kpc] when using galpy's natural units.

For convenience, a utility module `conversion` is included in galpy that helps in converting between physical units and natural units for various quantities. Alternatively, you can use the `astropy` units module to specify inputs in physical units and get outputs with units (see the *next subsection* below). For example, in natural units the orbital time of a circular orbit at R=1 is 2π; in physical units this corresponds to

```
>>> from galpy.util import conversion
>>> print(2.*numpy.pi*conversion.time_in_Gyr(220.,8.))
# 0.223405444283
```

or about 223 Myr. We can also express forces in various physical units. For example, for the Milky-Way-like potential defined in galpy, we have that the vertical force at 1.1 kpc is

```
>>> from galpy.potential import MWPotential2014, evaluatezforces
>>> -evaluatezforces(MWPotential2014, 1.,1.1/8.)*conversion.force_in_pcMyr2(220.,8.)
# 2.0259181908629933
```

which we can also express as an equivalent surface-density by dividing by $2\pi G$

```
>>> -evaluatezforces(MWPotential2014, 1.,1.1/8.)*conversion.force_in_2piGmsolpc2(220.,
↪8.)
# 71.658016957792356
```

Because the vertical force at the solar circle in the Milky Way at 1.1 kpc above the plane is approximately 70 $(2\pi G\,M_\odot\,\mathrm{pc}^{-2})$ (e.g., 2013arXiv1309.0809B), this shows that our Milky-Way-like potential has a realistic disk (at least in this respect).

`conversion` further has functions to convert densities, masses, surface densities, and frequencies to physical units (actions are considered to be too obvious to be included); see *here* for a full list. As a final example, the local dark matter density in the Milky-Way-like potential is given by

```
>>> MWPotential2014[2].dens(1.,0.)*conversion.dens_in_msolpc3(220.,8.)
# 0.0075419566970079373
```

or

```
>>> MWPotential2014[2].dens(1.,0.)*conversion.dens_in_gevcc(220.,8.)
# 0.28643101789044584
```

or about $0.0075\,M_\odot\,\mathrm{pc}^{-3} \approx 0.3\,\mathrm{GeV\,cm}^{-3}$, in line with current measurements (e.g., 2012ApJ...756...89B).

When `galpy` Potentials, Orbits, actionAngles, or DFs are initialized using a distance scale `ro=` and a velocity scale `vo=` output quantities returned and plotted in physical coordinates. Specifically, positions are returned in the units in the table below. If `astropy-units = True` in the *configuration file*, then an astropy Quantity which includes the units is returned instead (see below).

Quantity	Default unit
position	kpc
velocity	km/s
angular velocity	km/s/kpc
energy	(km/s)^2
Jacobi integral	(km/s)^2
angular momentum	km/s x kpc
actions	km/s x kpc
frequencies	1/Gyr
time	Gyr
period	Gyr
potential	(km/s)^2
force	km/s/Myr
force derivative	1/Gyr^2
density	Msun/pc^3
number density	1/pc^3
surface density	Msun/pc^2
mass	Msun
angle	rad
proper motion	mas/yr
phase-space density	1/(kpc x km/s)^3

Physical units

Tip: With `apy-units = True` in the configuration file and specifying all inputs using astropy Quantity with units, `galpy` will return outputs in convenient, unambiguous units.

Full support for unitful quantities using astropy Quantity was added in v1.2. Thus, *any* input to a galpy Potential, Orbit, actionAngle, or DF instantiation, method, or function can now be specified in physical units as a Quantity. For example, we can set up a Miyamoto-Nagai disk potential with a mass of $5 \times 10^{10}\,M_\odot$, a scale length of 3 kpc, and a scale height of 300 pc as follows

```
>>> from galpy.potential import MiyamotoNagaiPotential
>>> from astropy import units
>>> mp= MiyamotoNagaiPotential(amp=5*10**10*units.Msun,a=3.*units.kpc,b=300.*units.pc)
```

Internally, galpy uses a set of normalized units, where positions are divided by a scale `ro` and velocities are divided by a scale `vo`. If these are not specified, the default set from the *configuration file* is used. However, they can also be specified on an instance-by-instance manner for all Potential, Orbit, actionAngle, and DF instances. For example

```
>>> mp= MiyamotoNagaiPotential(amp=5*10**10*units.Msun,a=3.*units.kpc,b=300.*units.pc,
↪ro=9*units.kpc,vo=230.*units.km/units.s)
```

uses differently normalized internal units. When you specify the parameters of a Potential, Orbit, etc. in physical units (e.g., the Miyamoto-Nagai setup above), the internal set of units is unimportant as long as you receive output in physical units (see below) and it is unnecessary to change the values of `ro` and `vo`, unless you are modeling a system with very different distance and velocity scales from the default set (for example, if you are looking at internal globular cluster dynamics rather than galaxy dynamics). If you find an input to any galpy function that does not take a Quantity as an input (or that does it wrong), please report an Issue.

Warning: If you combine potentials by adding them (`comb_pot= pot1+pot2`), galpy uses the `ro` and `vo` scales from the first potential in the list for physical <-> internal unit conversion. If you add potentials using the '+' operator, galpy will check that the units are compatible. galpy does **not** always check whether the unit systems of various objects are consistent when they are combined (but does check this for many common cases, e.g., integrating an Orbit in a Potential, setting up an actionAngle object for a given potential, setting up a DF object for a given potential, etc.).

galpy can also return values with units as an astropy Quantity. Whether or not this is done is specified by the `apy-units` option in the *configuration file*. If you want to get return values as a Quantity, set `apy-units = True` in the configuration file. Then you can do for the Miyamoto-Nagai potential above

```
>>> mp.vcirc(10.*units.kpc)
# <Quantity 135.72399857308042 km / s>
```

Note that if you do not specify the argument as a Quantity with units, galpy will assume that it is given in natural units, viz.

```
>>> mp.vcirc(10.)
# <Quantity 51.78776595740726 km / s>
```

because this input is considered equal to 10 times the distance scale (this is for the case using the default `ro` and `vo`, the first Miyamoto-Nagai instantiation of this subsection)

```
>>> mp.vcirc(10.*8.*units.kpc)
# <Quantity 51.78776595740726 km / s>
```

Warning: If you do not specify arguments of methods and functions using a Quantity with units, galpy assumes that the argument has internal (natural) units.

If you do not use astropy Quantities (`apy-units = False` in the configuration file), you can still get output in physical units when you have specified `ro=` and `vo=` during instantiation of the Potential, Orbit, etc. For example, for the Miyamoto-Nagai potential above in a session with `apy-units = False`

```
>>> mp= MiyamotoNagaiPotential(amp=5*10**10*units.Msun,a=3.*units.kpc,b=300.*units.pc)
>>> mp.vcirc(10.*units.kpc)
# 135.72399857308042
```

This return value is in km/s (see the *table* at the end of the previous section for default units for different quantities). Note that as long as astropy is installed, we can still provide arguments as a Quantity, but the return value will not be a Quantity when `apy-units = False`. If you setup a Potential, Orbit, actionAngle, or DF object with parameters specified as a Quantity, the default is to return any output in physical units. This is why `mp.vcirc` returns the velocity in km/s above. Potential and Orbit instances (or lists of Potentials) also support the functions `turn_physical_off` and `turn_physical_on` to turn physical output off or on. For example, if we do

```
>>> mp.turn_physical_off()
```

outputs will be in internal units

```
>>> mp.vcirc(10.*units.kpc)
# 0.61692726624127459
```

If you setup a Potential, Orbit, etc. object without specifying the parameters as a Quantity, the default is to return output in natural units, except when `ro=` and `vo=` scales are specified (exception: when you wrap a potential that has

physical outputs on, the wrapped potential will also have them on). `ro=` and `vo=` can always be given as a Quantity themselves. `ro=` and `vo=` can always also be specified on a method-by-method basis, overwriting an object's default. For example

```
>>> mp.vcirc(10.*units.kpc,ro=12.*units.kpc)
# 0.69273212489609337
```

Physical output can also be turned off on a method-by-method or function-by-function basis, for example

```
>>> mp.turn_physical_on() # turn overall physical output on
>>> mp.vcirc(10.*units.kpc)
135.72399857308042 # km/s
>>> mp.vcirc(10.*units.kpc,use_physical=False)
# 0.61692726624127459 # in natural units
```

Further examples of specifying inputs with units will be given throughout the documentation.

1.3.3 Orbit integration

Warning: `galpy` uses a left-handed Galactocentric coordinate frame, as is common in studies of the kinematics of the Milky Way. This means that in particular cross-products, like the angular momentum $\vec{L} = \vec{r} \times \vec{p}$, behave differently than in a right-handed coordinate frame.

We can also integrate orbits in all galpy potentials. Going back to a simple Miyamoto-Nagai potential, we initialize an orbit as follows

```
>>> from galpy.orbit import Orbit
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,amp=1.,normalize=1.)
>>> o= Orbit([1.,0.1,1.1,0.,0.1])
```

Since we gave `Orbit()` a five-dimensional initial condition `[R,vR,vT,z,vz]`, we assume we are dealing with a three-dimensional axisymmetric potential in which we do not wish to track the azimuth. We then integrate the orbit for a set of times `ts`

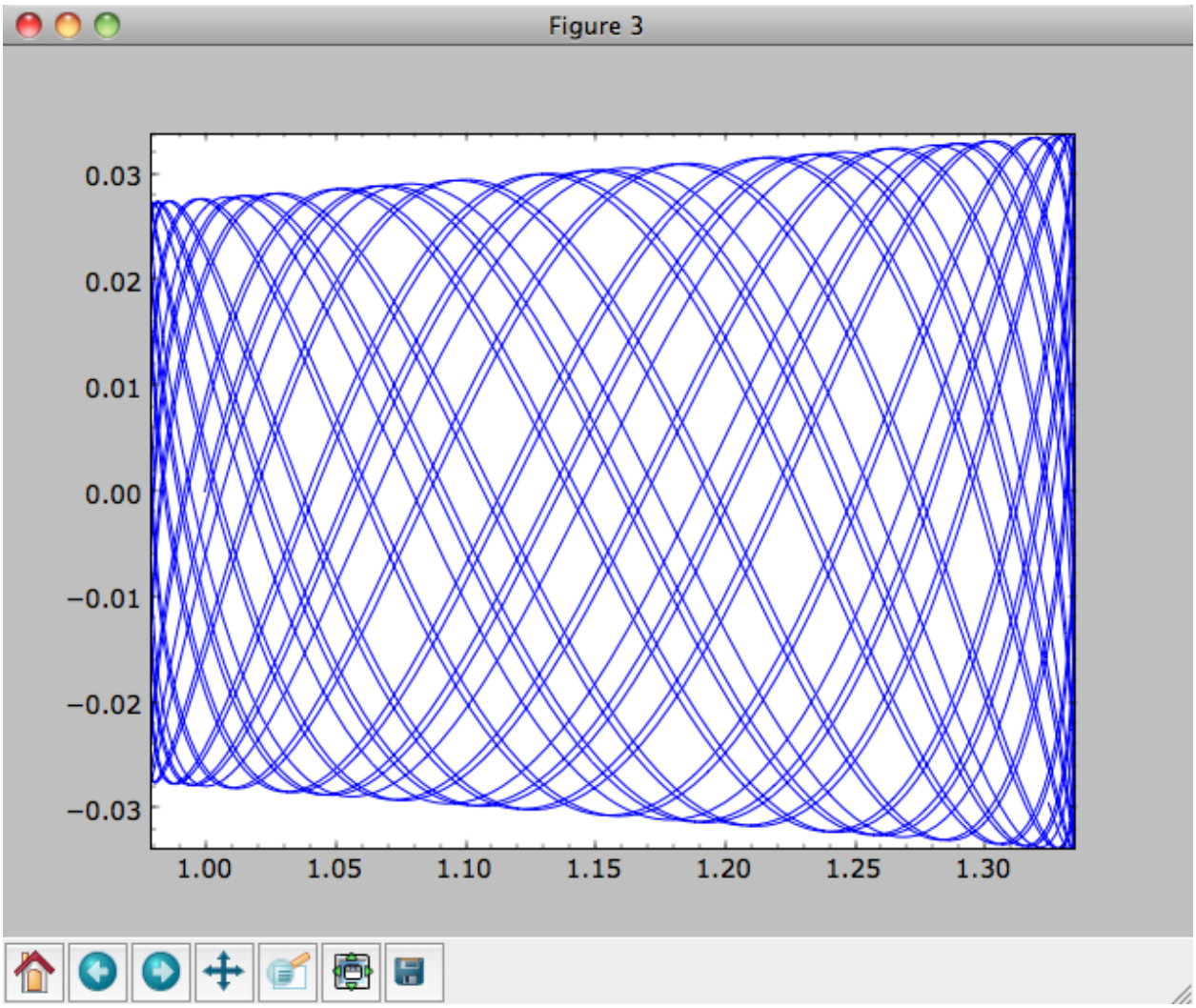
```
>>> import numpy
>>> ts= numpy.linspace(0,100,10000)
>>> o.integrate(ts,mp,method='odeint')
```

Tip: Like for the Miyamoto-Nagai example in the section above, the Orbit and integration times can also be specified in physical units, e.g., `o= Orbit([8.*units.kpc,22.*units.km/units.s,242.*units.km/units.s,0.*units.pc,20.*units.km/s])` and `ts= numpy.linspace(0.,10.,10000)*units.Gyr`

Now we plot the resulting orbit as

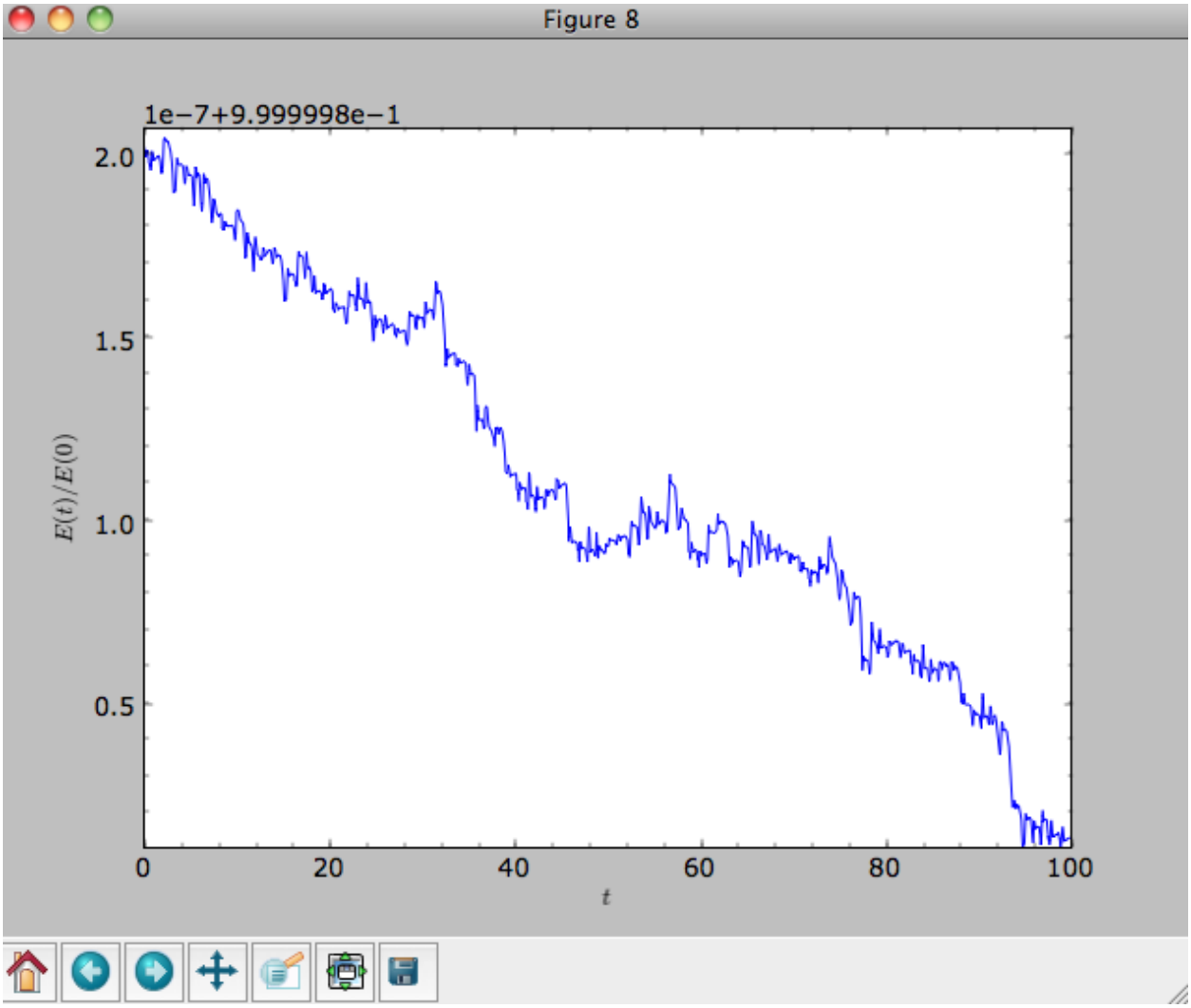
```
>>> o.plot()
```

Which gives



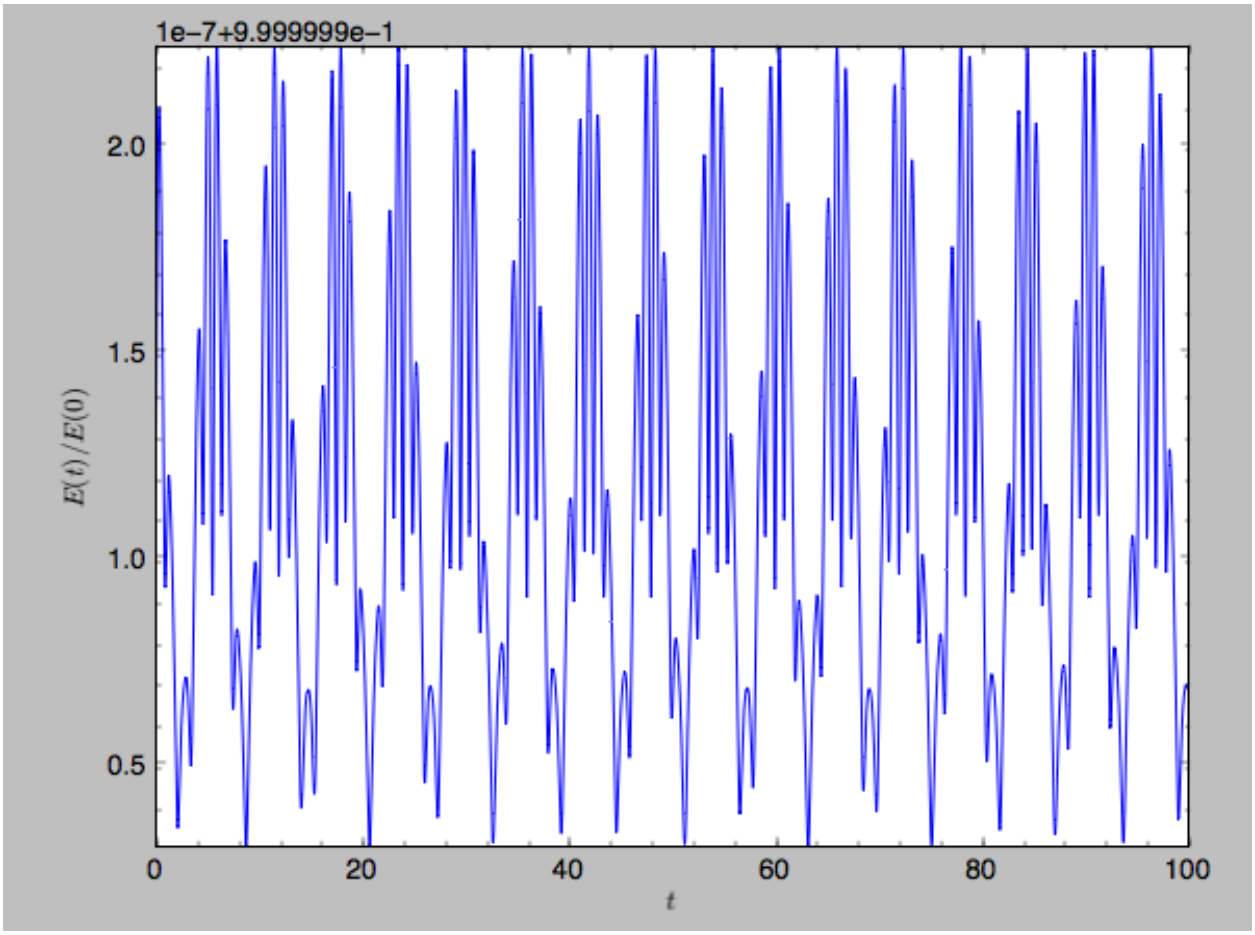
The integrator used is not symplectic, so the energy error grows with time, but is small nonetheless

```
>>> o.plotE(normed=True)
```



When we use a symplectic leapfrog integrator, we see that the energy error remains constant

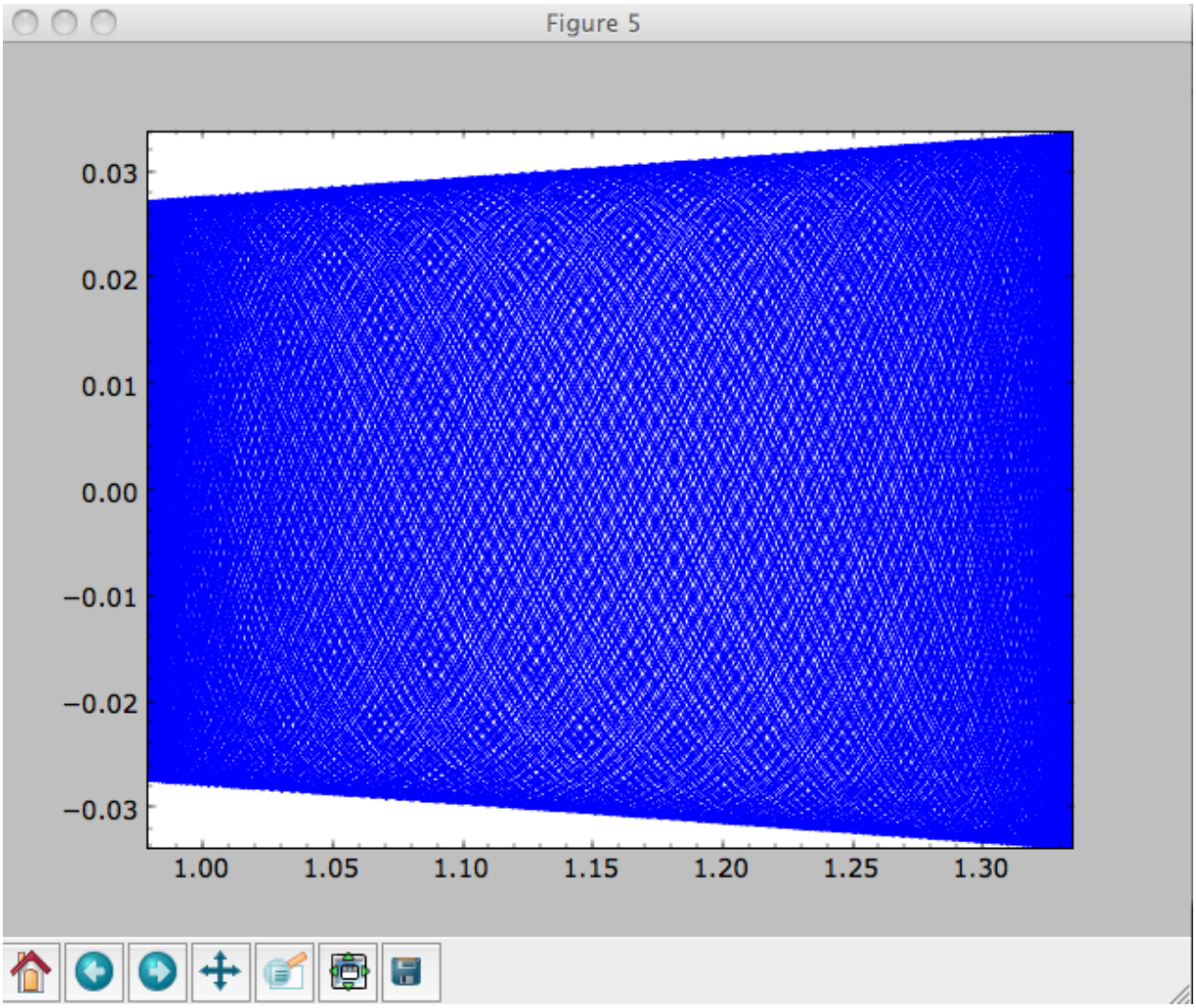
```
>>> o.integrate(ts,mp,method='leapfrog')
>>> o.plotE(xlabel=r'$t$',ylabel=r'$E(t)/E(0)$')
```



Because stars have typically only orbited the center of their galaxy tens of times, using symplectic integrators is mostly unnecessary (compared to planetary systems which orbits millions or billions of times). galpy contains *fast integrators* written in C, which can be accessed through the `method=` keyword (e.g., `integrate(..., method='dopr54_c')` is a fast high-order Dormand-Prince method).

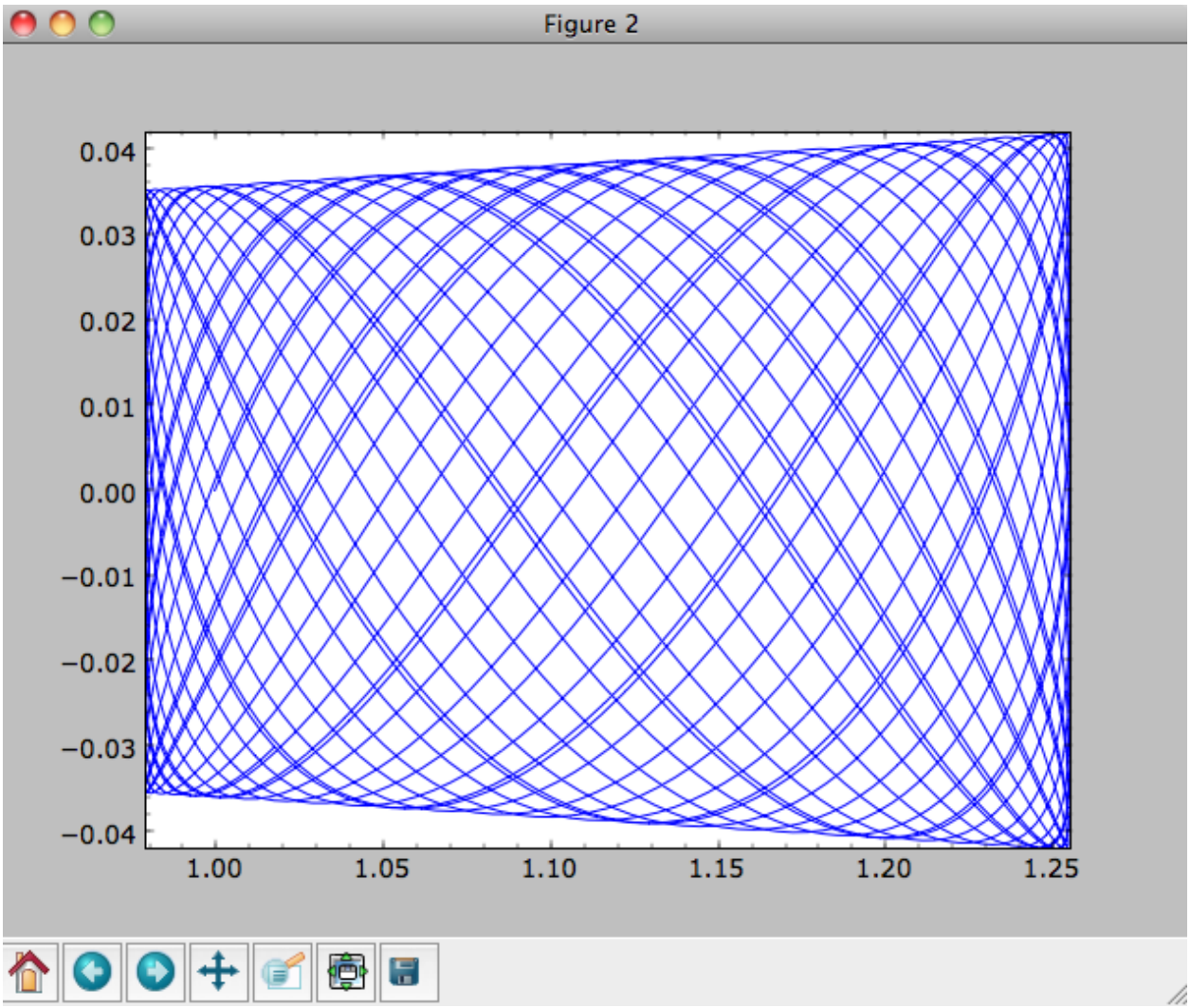
When we integrate for much longer we see how the orbit fills up a torus (this could take a minute)

```
>>> ts= numpy.linspace(0,1000,10000)
>>> o.integrate(ts,mp,method='odeint')
>>> o.plot()
```



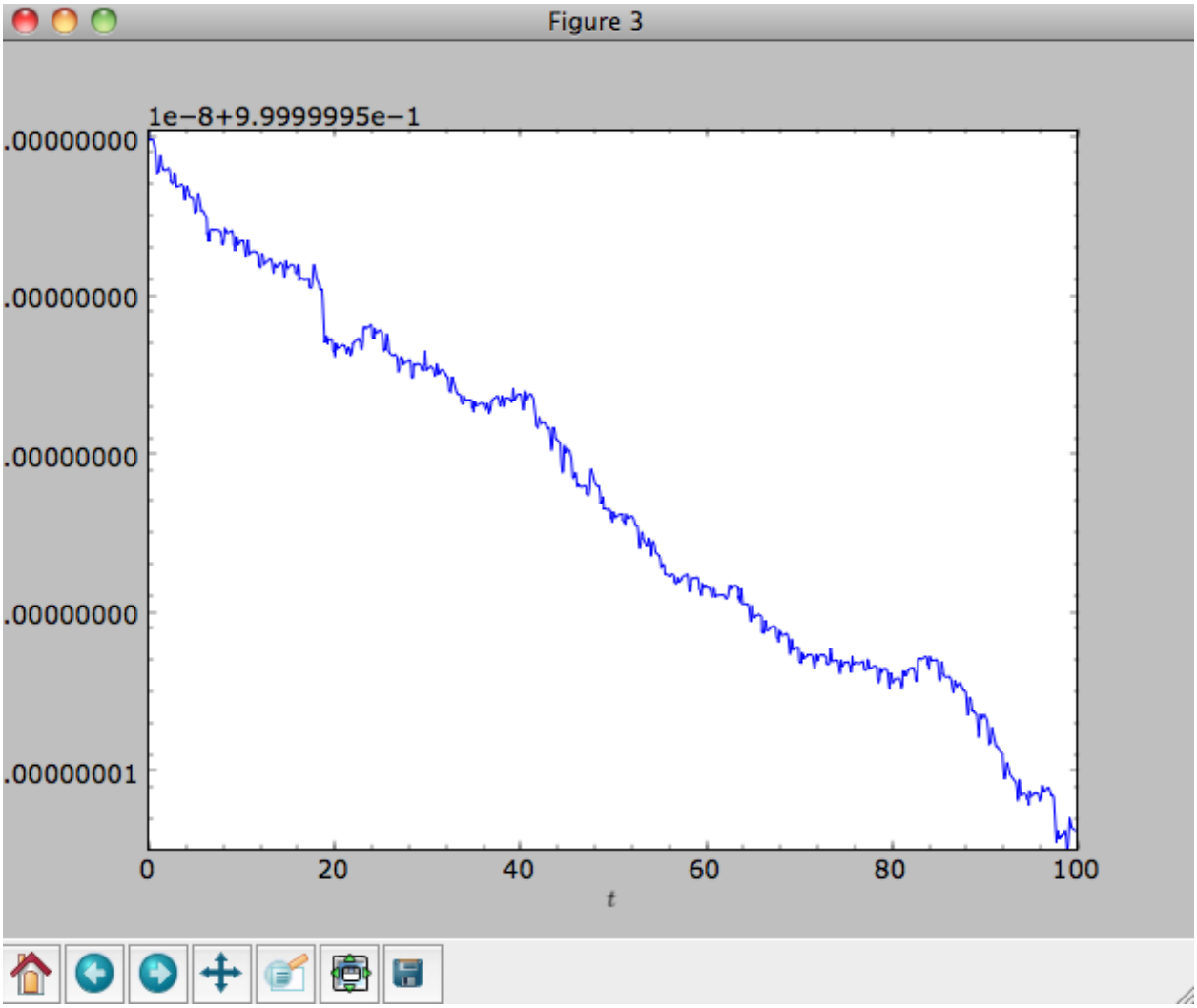
As before, we can also integrate orbits in combinations of potentials. Assuming `mp`, `np`, and `hp` were defined as above, we can

```
>>> ts= numpy.linspace(0,100,10000)
>>> o.integrate(ts,mp+hp+np)
>>> o.plot()
```



Energy is again approximately conserved

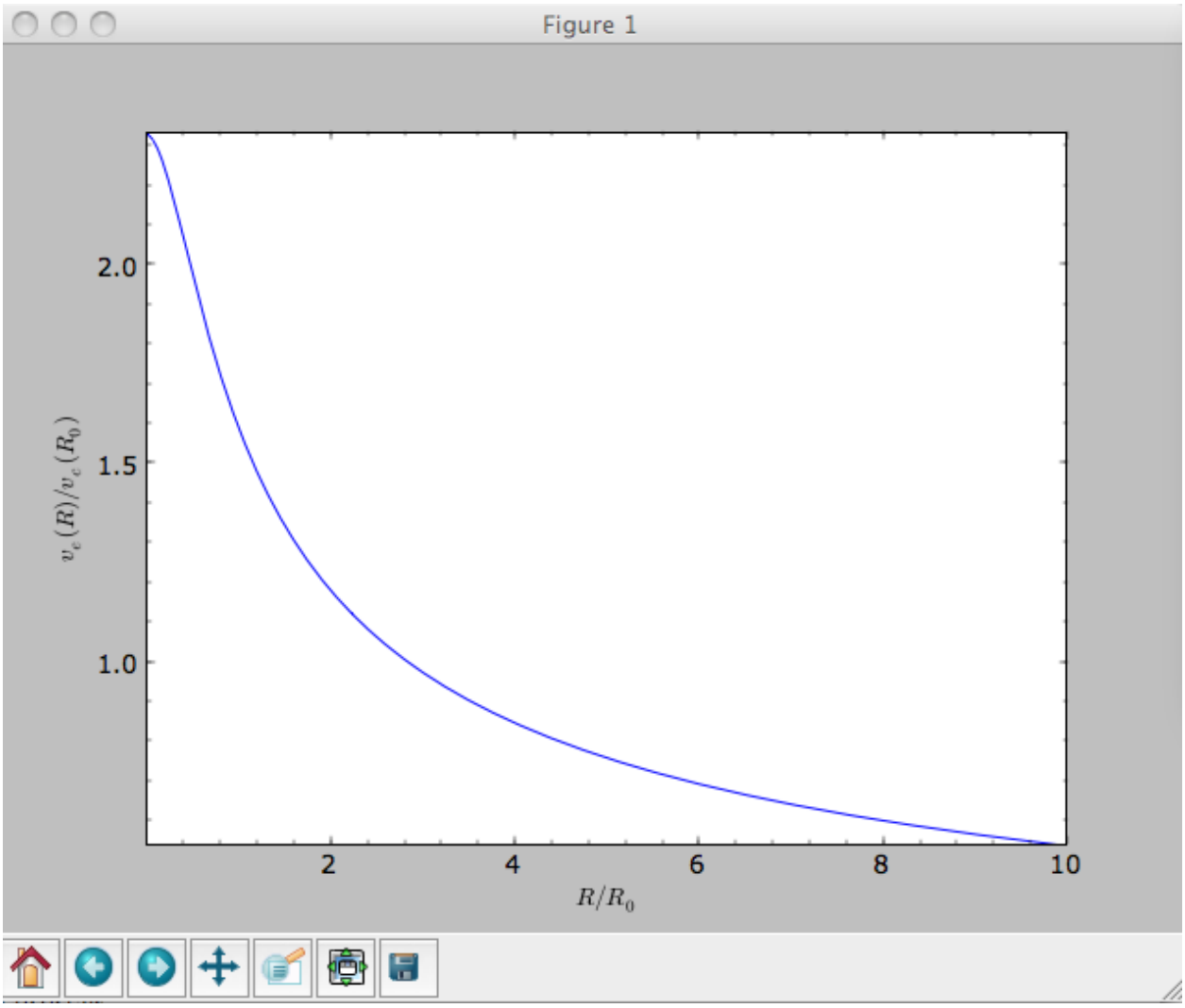
```
>>> o.plotE(xlabel=r'$t$',ylabel=r'$E(t)/E(0)$')
```



1.3.4 Escape velocity curves

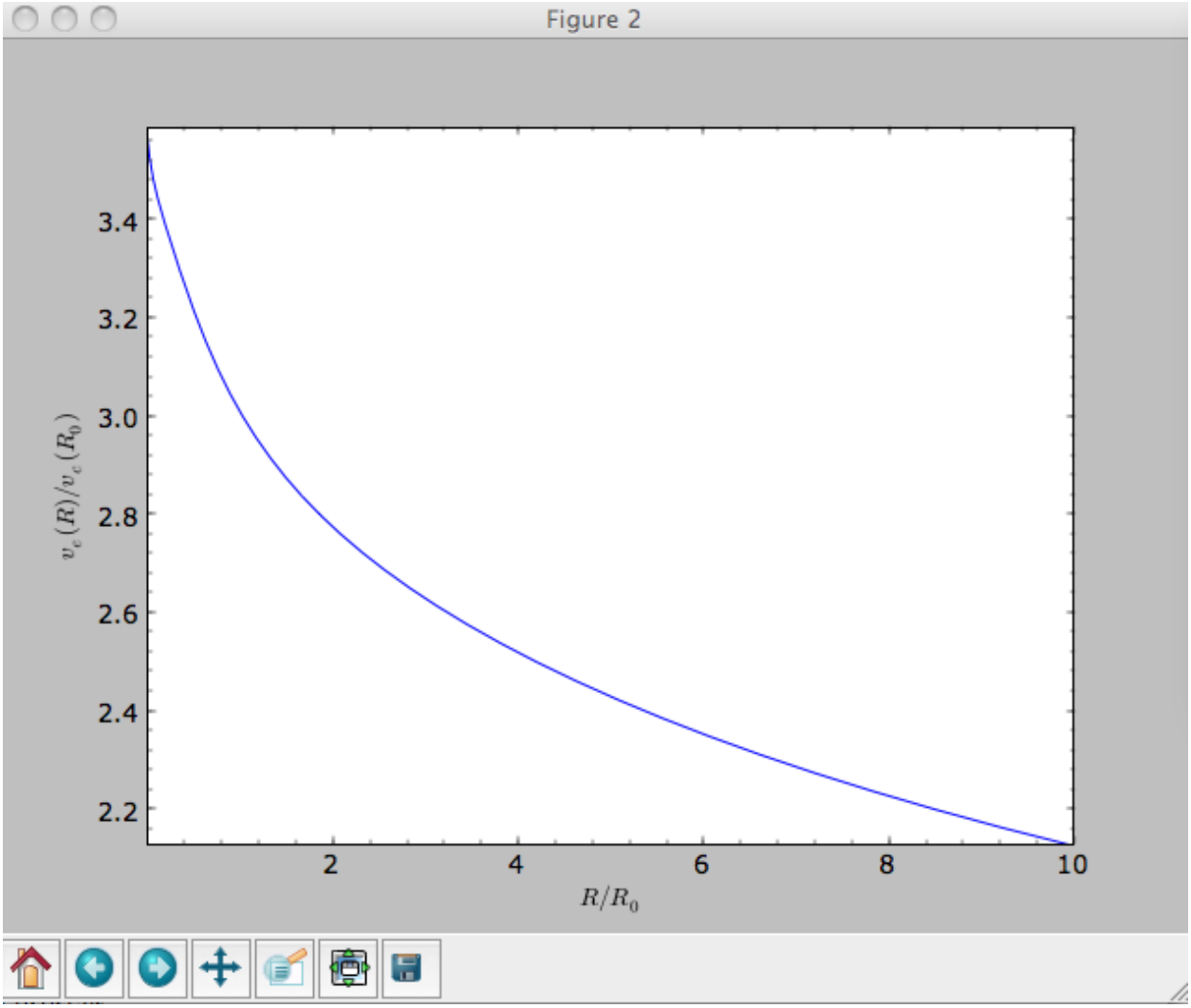
Just like we can plot the rotation curve for a potential or a combination of potentials, we can plot the escape velocity curve. For example, the escape velocity curve for the Miyamoto-Nagai disk defined above

```
>>> mp.plotEscapecurve(Rrange=[0.01,10.],grid=1001)
```



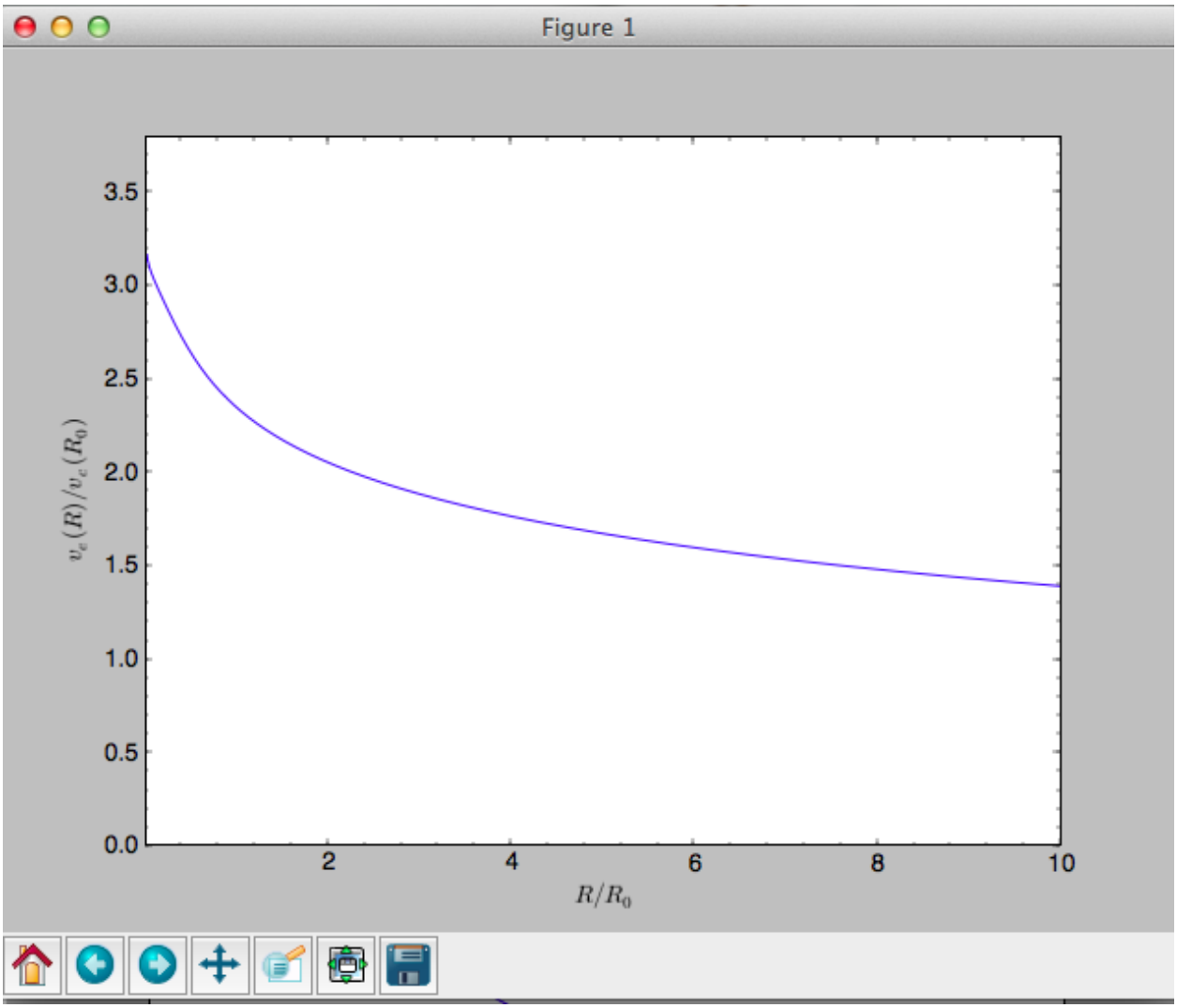
or of the combination of potentials defined above

```
>>> from galpy.potential import plotEscapecurve
>>> plotEscapecurve(mp+hp+np,Rrange=[0.01,10.],grid=1001)
```



For the Milky-Way-like potential `MWPotential2014`, the escape-velocity curve is

```
>>> plotEscapecurve(MWPotential2014,Rrange=[0.01,10.],grid=1001)
```



At the solar radius, the escape velocity is

```
>>> from galpy.potential import vesc
>>> vesc(MWPotential2014,1.)
2.3316389848832784
```

Or, for a local circular velocity of 220 km/s

```
>>> vesc(MWPotential2014,1.)*220.
# 512.96057667432126
```

similar to direct measurements of this (e.g., 2007MNRAS.379..755S and 2014A%26A...562A..91P).

1.4 Potentials in galpy

galpy contains a large variety of potentials in `galpy.potential` that can be used for orbit integration, the calculation of action-angle coordinates, as part of steady-state distribution functions, and to study the properties of gravitational potentials. This section introduces some of these features.

1.4.1 Potentials and forces

Various 3D and 2D potentials are contained in galpy, list in the *API page*. Another way to list the latest overview of potentials included with galpy is to run

```
>>> import galpy.potential
>>> print([p for p in dir(galpy.potential) if 'Potential' in p])
# ['CosmphiDiskPotential',
#  'DehnenBarPotential',
#  'DoubleExponentialDiskPotential',
#  'EllipticalDiskPotential',
#  'FlattenedPowerPotential',
#  'HernquistPotential',
# ....]
```

(list cut here for brevity). Section *Rotation curves* explains how to initialize potentials and how to display the rotation curve of single Potential instances or of combinations of such instances. Similarly, we can evaluate a Potential instance

```
>>> from galpy.potential import MiyamotoNagaiPotential
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,normalize=1.)
>>> mp(1.,0.)
# -1.2889062500000001
```

Most member functions of Potential instances have corresponding functions in the galpy.potential module that allow them to be evaluated for lists of multiple Potential instances (and in versions >=1.4 even for nested lists of Potential instances). `galpy.potential.MWPotential2014` is such a list of three Potential instances

```
>>> from galpy.potential import MWPotential2014
>>> print(MWPotential2014)
# [<galpy.potential.PowerSphericalPotentialwCutoff.PowerSphericalPotentialwCutoff␣
↪instance at 0x1089b23b0>, <galpy.potential.MiyamotoNagaiPotential.
↪MiyamotoNagaiPotential instance at 0x1089b2320>, <galpy.potential.
↪TwoPowerSphericalPotential.NFWPotential instance at 0x1089b2248>]
```

and we can evaluate the potential by using the `evaluatePotentials` function

```
>>> from galpy.potential import evaluatePotentials
>>> evaluatePotentials(MWPotential2014,1.,0.)
# -1.3733506513947895
```

Tip: Lists of Potential instances can be nested, allowing you to easily add components to existing gravitational-potential models. For example, to add a `DehnenBarPotential` to `MWPotential2014`, you can do: `pot= [MWPotential2014,DehnenBarPotential()]` and then use this `pot` everywhere where you can use a list of Potential instances. You can also add potential simply as `pot= MWPotential2014+DehnenBarPotential()`.

Warning: `galpy` potentials do *not* necessarily approach zero at infinity. To compute, for example, the escape velocity or whether or not an orbit is unbound, you need to take into account the value of the potential at infinity. E.g., $v_{\rm esc}(r) = \sqrt{2[\Phi(\infty) - \Phi(r)]}$. If you want to create a potential that does go to zero at infinity, you can add a *NullPotential* with value equal to minus the original potential evaluated at infinity.

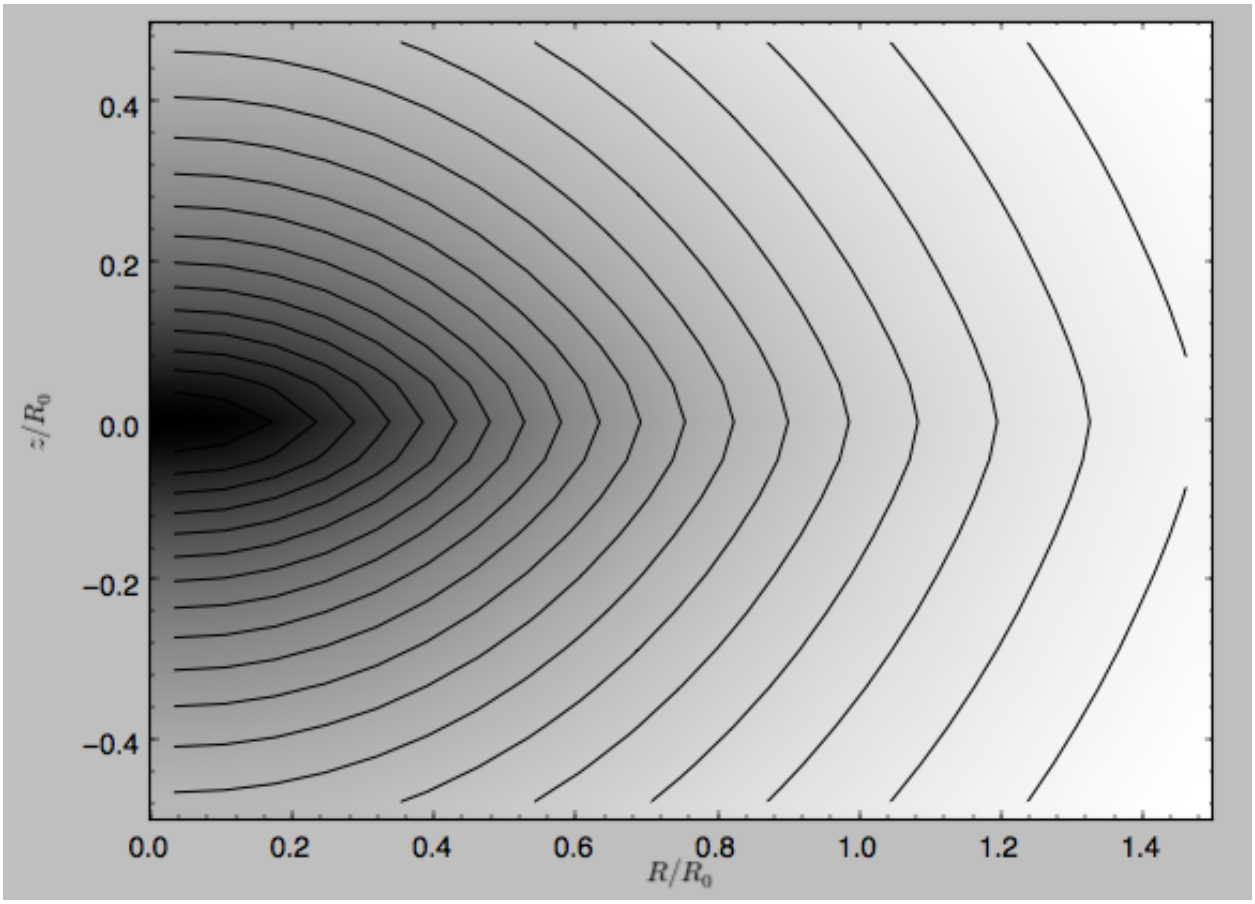
Tip: As discussed in the section on *physical units*, potentials can be initialized and evaluated with arguments specified

as a astropy Quantity with units. Use the configuration parameter `apy-units = True` to get output values as a Quantity. See also the subsection on *Initializing potentials with parameters with units* below.

We can plot the potential of axisymmetric potentials (or of non-axisymmetric potentials at phi=0) using the `plot` member function

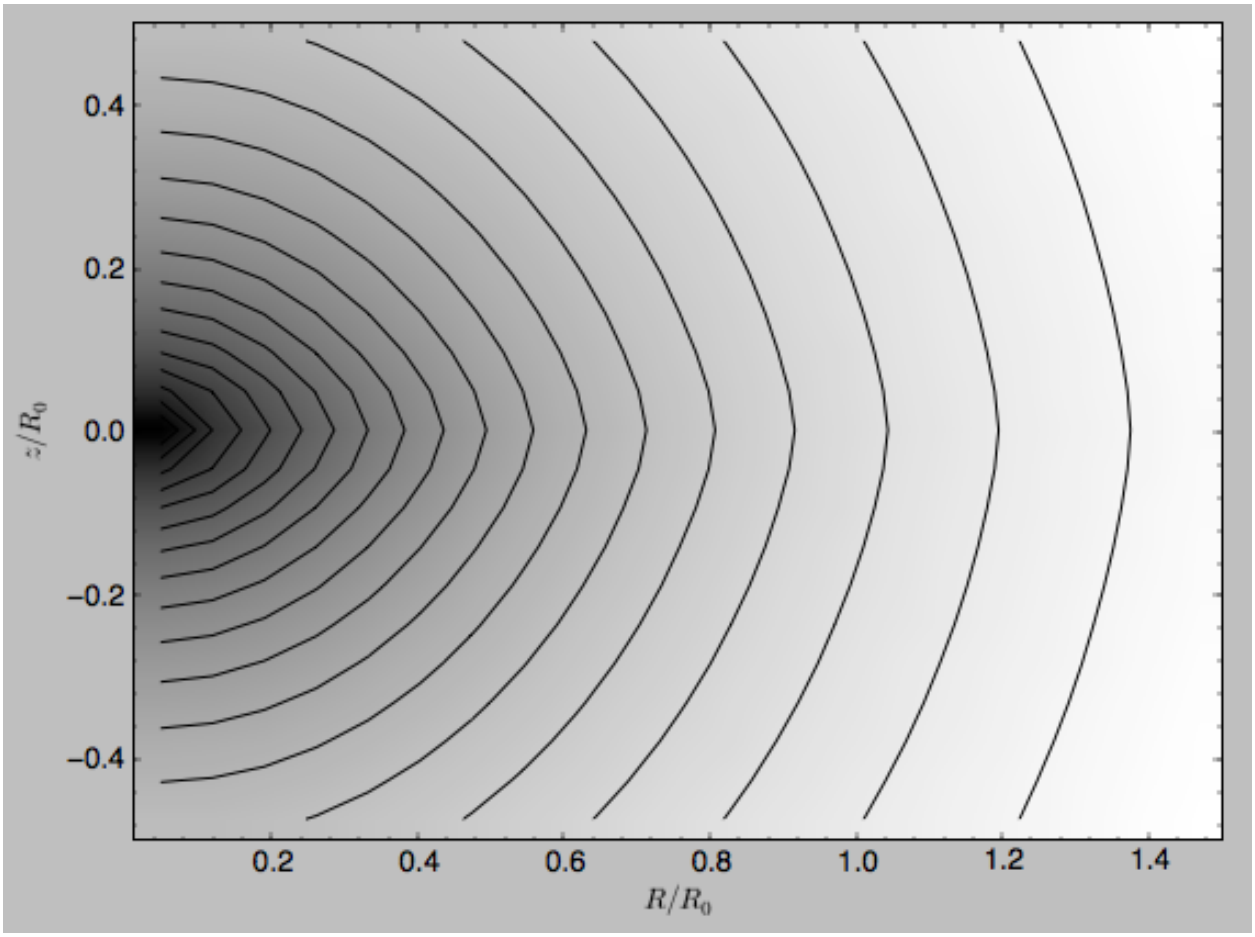
```
>>> mp.plot()
```

which produces the following plot



Similarly, we can plot combinations of Potentials using `plotPotentials`, e.g.,

```
>>> from galpy.potential import plotPotentials
>>> plotPotentials(MWPotential2014,rmin=0.01)
```



These functions have arguments that can provide custom `R` and `z` ranges for the plot, the number of grid points, the number of contours, and many other parameters determining the appearance of these figures.

galpy also allows the forces corresponding to a gravitational potential to be calculated. Again for the Miyamoto-Nagai Potential instance from above

```
>>> mp.Rforce(1.,0.)
# -1.0
```

This value of -1.0 is due to the normalization of the potential such that the circular velocity is 1. at R=1. Similarly, the vertical force is zero in the mid-plane

```
>>> mp.zforce(1.,0.)
# -0.0
```

but not further from the mid-plane

```
>>> mp.zforce(1.,0.125)
# -0.53488743705310848
```

As explained in *Units in galpy*, these forces are in standard galpy units, and we can convert them to physical units using methods in the `galpy.util.conversion` module. For example, assuming a physical circular velocity of 220 km/s at R=8 kpc

```
>>> from galpy.util import conversion
>>> mp.zforce(1.,0.125)*conversion.force_in_kmsMyr(220.,8.)
# -3.3095671288657584 #km/s/Myr
>>> mp.zforce(1.,0.125)*conversion.force_in_2piGmsolpc2(220.,8.)
# -119.72021771473301 #2 \pi G Msol / pc^2
```

Again, there are functions in `galpy.potential` that allow for the evaluation of the forces for lists of Potential instances, such that

```
>>> from galpy.potential import evaluateRforces
>>> evaluateRforces(MWPotential2014,1.,0.)
# -1.0
>>> from galpy.potential import evaluatezforces
>>> evaluatezforces(MWPotential2014,1.,0.125)*conversion.force_in_2piGmsolpc2(220.,8.)
>>> -69.680720137571114 #2 \pi G Msol / pc^2
```

We can evaluate the flattening of the potential as $\sqrt{|z\,F_R/R\,F_Z|}$ for a Potential instance as well as for a list of such instances

```
>>> mp.flattening(1.,0.125)
# 0.4549542914935209
>>> from galpy.potential import flattening
>>> flattening(MWPotential2014,1.,0.125)
# 0.61231675305658628
```

Warning: While we call them 'forces' in `galpy`, the forces are really gravitational fields (forces per unit mass) or accelerations (through Newton's second law).

1.4.2 Densities

galpy can also calculate the densities corresponding to gravitational potentials. For many potentials, the densities are explicitly implemented, but if they are not, the density is calculated using the Poisson equation (second derivatives of the potential have to be implemented for this). For example, for the Miyamoto-Nagai potential, the density is explicitly implemented

```
>>> mp.dens(1.,0.)
# 1.1145444383277576
```

and we can also calculate this using the Poisson equation

```
>>> mp.dens(1.,0.,forcepoisson=True)
# 1.1145444383277574
```

which are the same to machine precision

```
>>> mp.dens(1.,0.,forcepoisson=True)-mp.dens(1.,0.)
# -2.2204460492503131e-16
```

Similarly, all of the potentials in `galpy.potential.MWPotential2014` have explicitly-implemented densities, so we can do

```
>>> from galpy.potential import evaluateDensities
>>> evaluateDensities(MWPotential2014,1.,0.)
# 0.57508603122264867
```

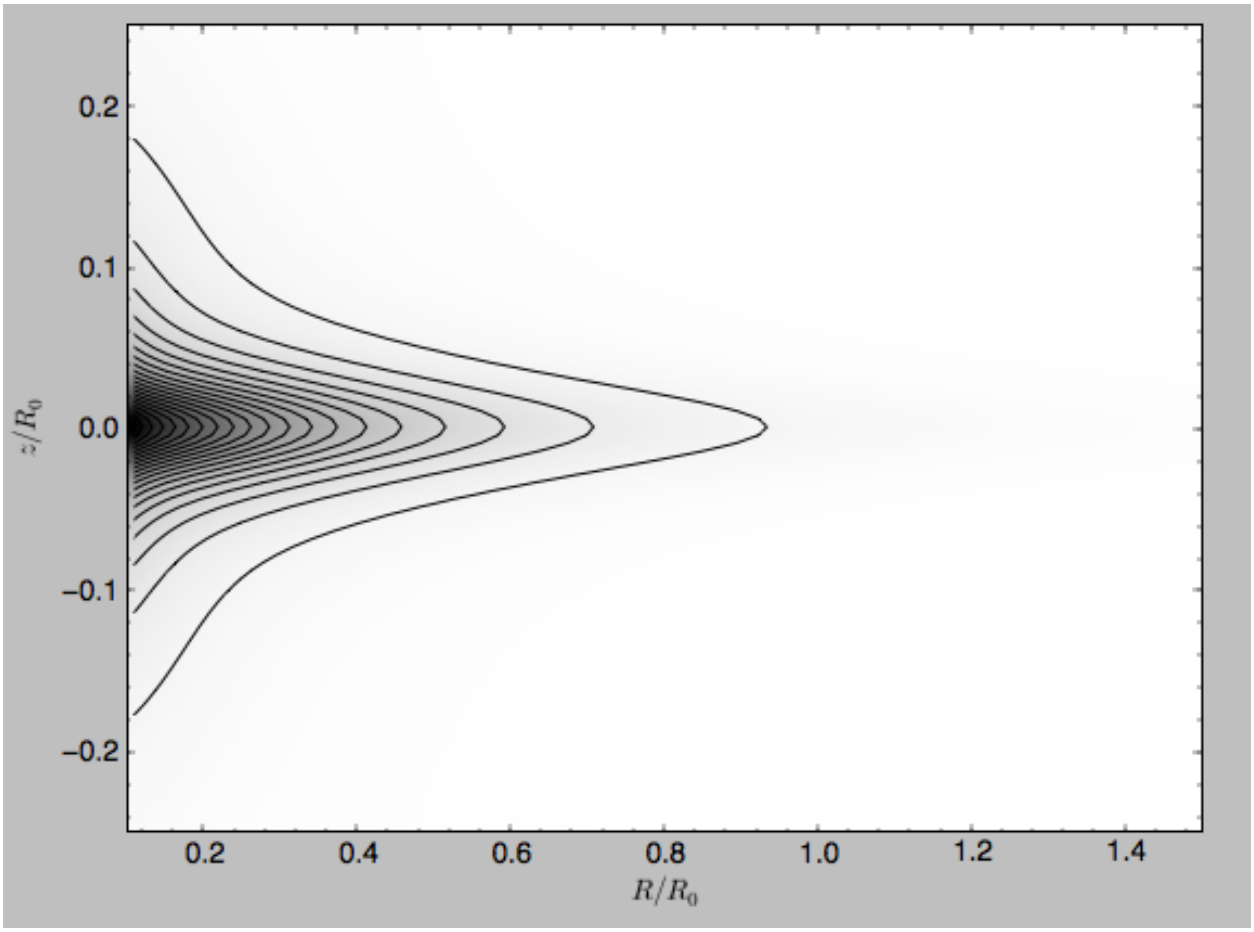
In physical coordinates, this becomes

```
>>> evaluateDensities(MWPotential2014,1.,0.)*conversion.dens_in_msolpc3(220.,8.)
# 0.1010945632524705 #Msol / pc^3
```

We can also plot densities

```
>>> from galpy.potential import plotDensities
>>> plotDensities(MWPotential2014,rmin=0.1,zmax=0.25,zmin=-0.25,nrs=101,nzs=101)
```

which gives



Another example of this is for an exponential disk potential

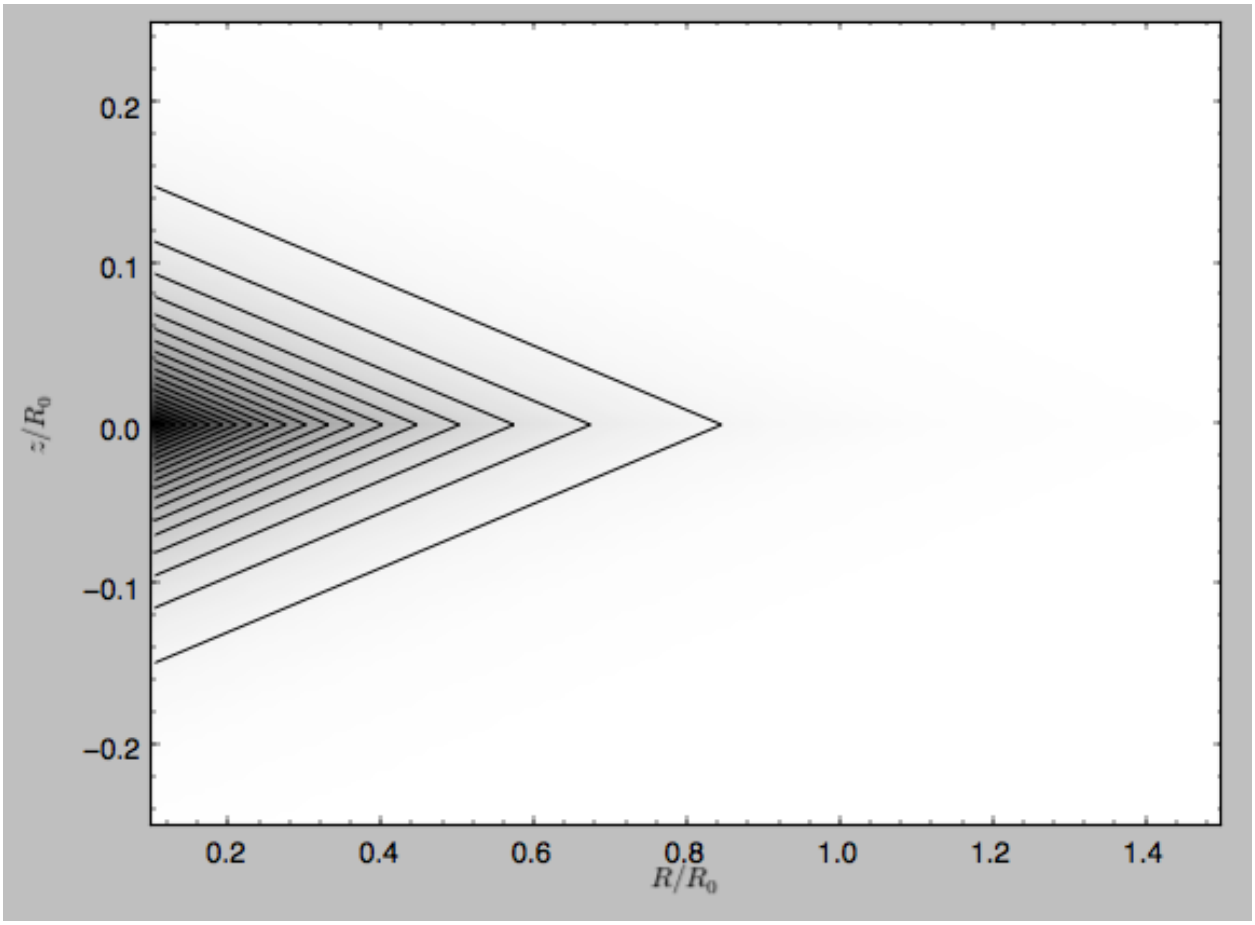
```
>>> from galpy.potential import DoubleExponentialDiskPotential
>>> dp= DoubleExponentialDiskPotential(hr=1./4.,hz=1./20.,normalize=1.)
```

The density computed using the Poisson equation now requires multiple numerical integrations, so the agreement between the analytical density and that computed using the Poisson equation is slightly less good, but still better than a percent

```
>>> (dp.dens(1.,0.,forcepoisson=True)-dp.dens(1.,0.))/dp.dens(1.,0.)
# 0.0032522956769123019
```

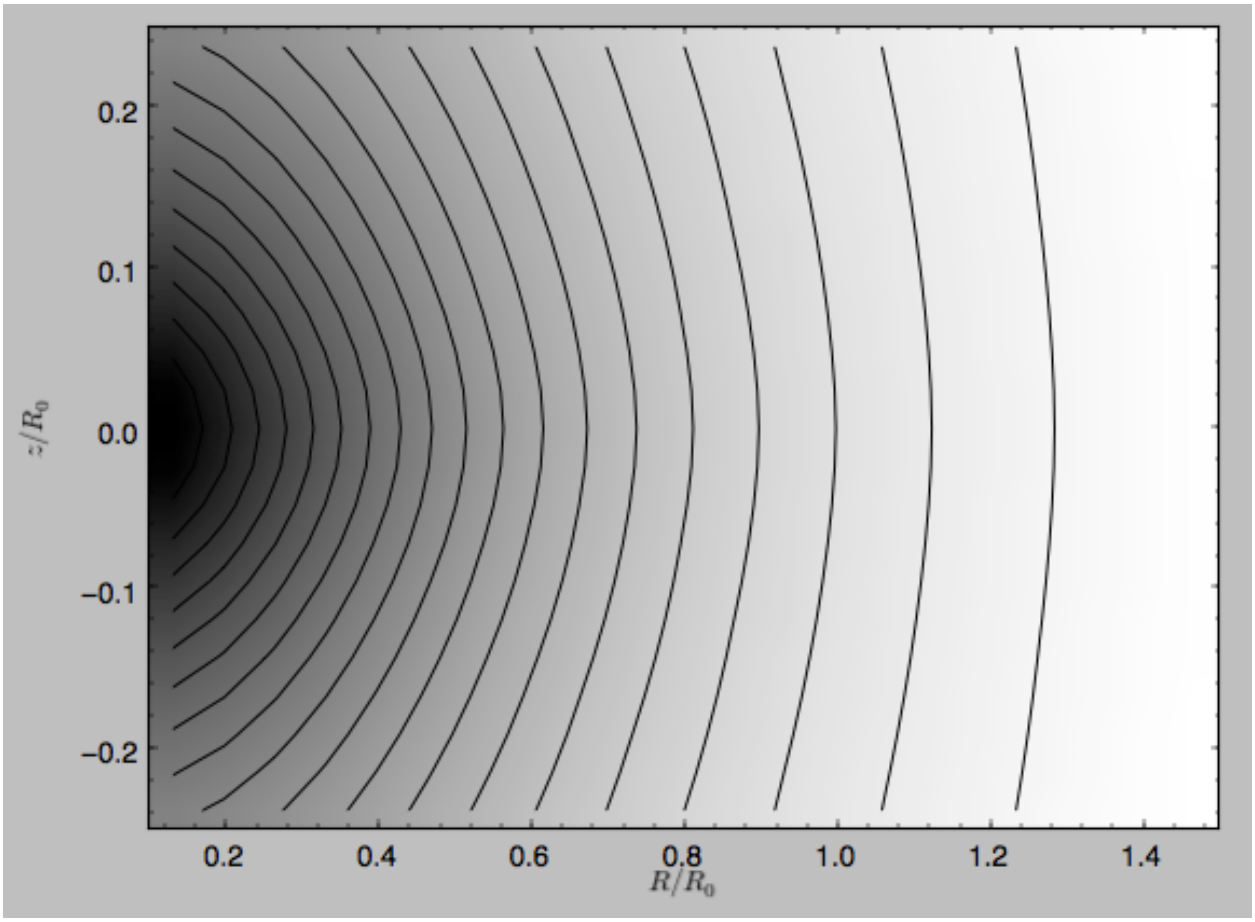
The density is

```
>>> dp.plotDensity(rmin=0.1,zmax=0.25,zmin=-0.25,nrs=101,nzs=101)
```



and the potential is

```
>>> dp.plot(rmin=0.1,zmin=-0.25,zmax=0.25)
```



Clearly, the potential is much less flattened than the density.

1.4.3 Modifying potential instances using wrappers

Potentials implemented in galpy can be modified using different kinds of wrappers. These wrappers modify potentials to, for example, change their amplitude as a function of time (e.g., to grow or decay the bar contribution to a potential) or to make a potential rotate. Specific kinds of wrappers are listed on the *Potential wrapper API page*. These wrappers can be applied to instances of *any* potential implemented in galpy (including other wrappers). An example is to grow a bar using the polynomial smoothing of Dehnen (2000). We first setup an instance of a `DehnenBarPotential` that is essentially fully grown already

```
>>> from galpy.potential import DehnenBarPotential
>>> dpn= DehnenBarPotential(tform=-100.,tsteady=0.) # DehnenBarPotential has a custom␣
↪implementation of growth that we ignore by setting tform to -100
```

and then wrap it

```
>>> from galpy.potential import DehnenSmoothWrapperPotential
>>> dswp= DehnenSmoothWrapperPotential(pot=dpn,tform=-4.*2.*numpy.pi/dpn.OmegaP(),
↪tsteady=2.*2.*numpy.pi/dpn.OmegaP())
```

This grows the `DehnenBarPotential` starting at 4 bar periods before `t=0` over a period of 2 bar periods. `DehnenBarPotential` has an older, custom implementation of the same smoothing and the `(tform,tsteady)` pair used here corresponds to the default setting for `DehnenBarPotential`. Thus we can compare the two

```
>>> dp= DehnenBarPotential()
>>> print(dp(0.9,0.3,phi=3.,t=-2.)-dswp(0.9,0.3,phi=3.,t=-2.))
# 0.0
>>> print(dp.Rforce(0.9,0.3,phi=3.,t=-2.)-dswp.Rforce(0.9,0.3,phi=3.,t=-2.))
# 0.0
```

Other wrappers to modify the amplitude of a potential include `GaussianAmplitudeWrapperPotential`, for modulating the amplitude using a Gaussian, and the fully general `TimeDependentAmplitudeWrapperPotential`, which can modulate the amplitude of any potential with an arbitrary function of time.

Tip: To simply adjust the amplitude of a Potential instance, you can multiply the instance with a number or divide it by a number. For example, `pot= 2.*LogarithmicHaloPotential(amp=1.)` is equivalent to `pot= LogarithmicHaloPotential(amp=2.)`. This is useful if you want to, for instance, quickly adjust the mass of a potential.

The wrapper `SolidBodyRotationWrapperPotential` allows one to make any potential rotate around the z axis. This can be used, for example, to make general bar-shaped potentials, which one could construct from a basis-function expansion with `SCFPotential`, rotate without having to implement the rotation directly. As an example consider this `SoftenedNeedleBarPotential` (which has a potential-specific implementation of rotation)

```
>>> sp= SoftenedNeedleBarPotential(normalize=1.,omegab=1.8,pa=0.)
```

The same potential can be obtained from a non-rotating `SoftenedNeedleBarPotential` run through the `SolidBodyRotationWrapperPotential` to add rotation

```
>>> sp_still= SoftenedNeedleBarPotential(omegab=0.,pa=0.,normalize=1.)
>>> swp= SolidBodyRotationWrapperPotential(pot=sp_still,omega=1.8,pa=0.)
```

Compare for example

```
>>> print(sp(0.8,0.2,phi=0.2,t=3.)-swp(0.8,0.2,phi=0.2,t=3.))
# 0.0
>>> print(sp.Rforce(0.8,0.2,phi=0.2,t=3.)-swp.Rforce(0.8,0.2,phi=0.2,t=3.))
# 8.881784197e-16
```

`RotateAndTiltWrapperPotential` is a wrapper that allows you to rotate, tilt, or offset a potential. This can be useful if you are trying to see a potential they way an external galaxy is tilted, or, in combination with `SolidBodyRotationWrapperPotential`, to make a potential rotate around an arbitrary axis (you can tilt, solid-body rotate, and tilt back to do this).

Wrapper potentials can be used anywhere in galpy where general potentials can be used. They can be part of lists of Potential instances. Wrappers can be wrapped again. They can also be used in C for orbit integration provided that both the wrapper and the potentials that it wraps are implemented in C. For example, a static `LogarithmicHaloPotential` with a bar potential grown as above would be

```
>>> from galpy.potential import LogarithmicHaloPotential, evaluateRforces
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> pot= lp+dswp
>>> print(evaluateRforces(pot,0.9,0.3,phi=3.,t=-2.))
# -1.00965326579
```

Warning: When wrapping a potential that has *physical outputs turned on*, the wrapper object inherits the units of the wrapped potential and automatically turns them on, even when you do not explictly set `ro=` and `vo=`.

1.4.4 Close-to-circular orbits and orbital frequencies

We can also compute the properties of close-to-circular orbits. First of all, we can calculate the circular velocity and its derivative

```
>>> mp.vcirc(1.)
# 1.0
>>> mp.dvcircdR(1.)
# -0.163777427566978
```

or, for lists of Potential instances

```
>>> from galpy.potential import vcirc
>>> vcirc(MWPotential2014,1.)
# 1.0
>>> from galpy.potential import dvcircdR
>>> dvcircdR(MWPotential2014,1.)
# -0.10091361254334696
```

We can also calculate the various frequencies for close-to-circular orbits. For example, the rotational frequency

```
>>> mp.omegac(0.8)
# 1.2784598203204887
>>> from galpy.potential import omegac
>>> omegac(MWPotential2014,0.8)
# 1.2733514576122869
```

and the epicycle frequency

```
>>> mp.epifreq(0.8)
# 1.7774973530267848
>>> from galpy.potential import epifreq
>>> epifreq(MWPotential2014,0.8)
# 1.7452189766287691
```

as well as the vertical frequency

```
>>> mp.verticalfreq(1.0)
# 3.7859388972001828
>>> from galpy.potential import verticalfreq
>>> verticalfreq(MWPotential2014,1.)
# 2.7255405754769875
```

We can also for example easily make the diagram of $\Omega - n\kappa/m$ that is important for understanding kinematic spiral density waves. For example, for `MWPotential2014`

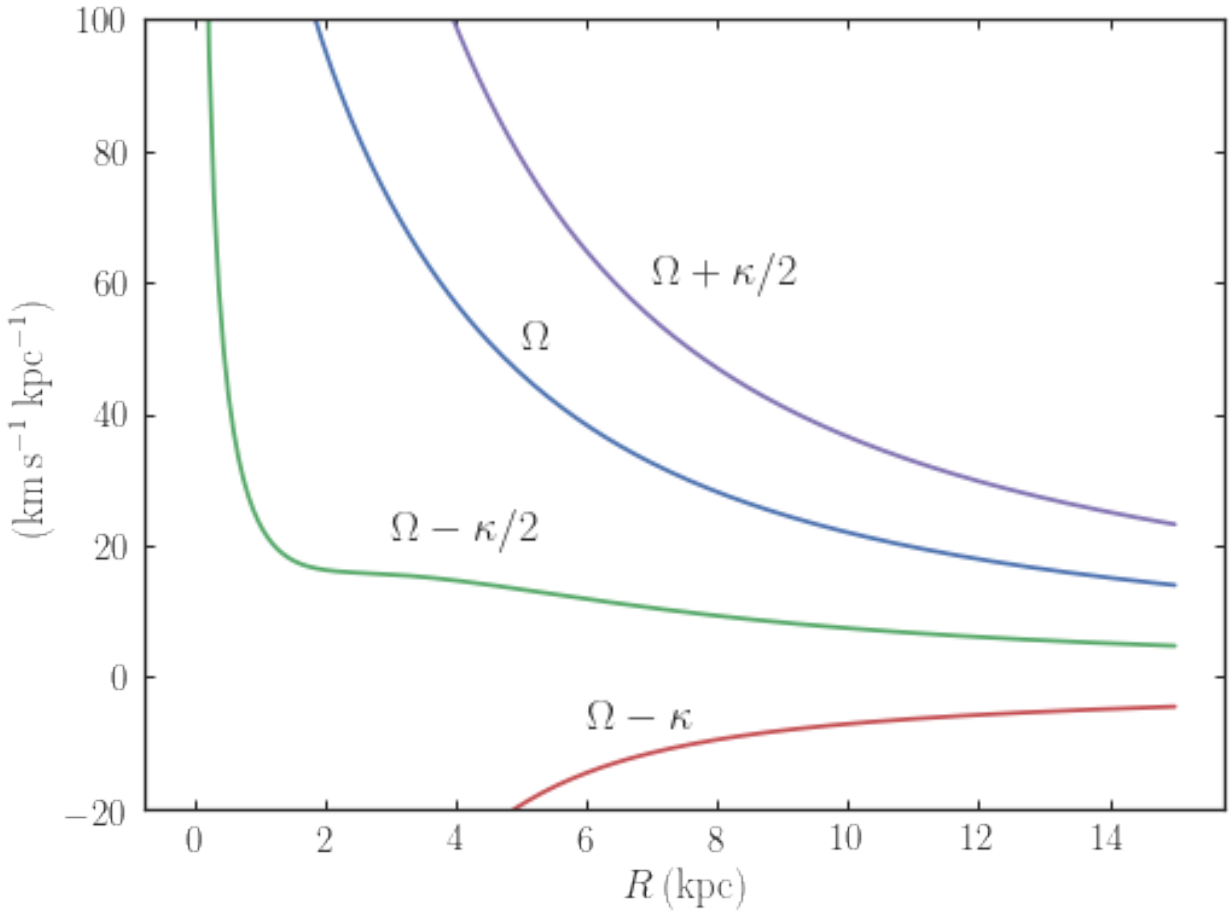
```
>>> from galpy.potential import MWPotential2014, omegac, epifreq
>>> def OmegaMinusKappa(pot,Rs,n,m,ro=8.,vo=220.):
        # ro,vo for physical units, Rs in units of ro
        return omegac(pot,Rs/ro,ro=ro,vo=vo)-n/m*epifreq(pot,Rs/ro,ro=ro,vo=vo)
>>> Rs= numpy.linspace(0.,16.,101) # kpc
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,0,1))
```

(continues on next page)

(continued from previous page)

```
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,1,2))
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,1,1))
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,1,-2))
>>> ylim(-20.,100.)
>>> xlabel(r'$R\,(\mathrm{kpc})$')
>>> ylabel(r'$(\mathrm{km\,s}^{-1}\,\mathrm{kpc}^{-1})$')
>>> text(3.,21.,r'$\Omega-\kappa/2$',size=18.)
>>> text(5.,50.,r'$\Omega$',size=18.)
>>> text(7.,60.,r'$\Omega+\kappa/2$',size=18.)
>>> text(6.,-7.,r'$\Omega-\kappa$',size=18.)
```

which gives



For close-to-circular orbits, we can also compute the radii of the Lindblad resonances. For example, for a frequency similar to that of the Milky Way's bar

```
>>> mp.lindbladR(5./3.,m='corotation') #args are pattern speed and m of pattern
# 0.6027911166042229 #~ 5kpc
>>> print(mp.lindbladR(5./3.,m=2))
# None
>>> mp.lindbladR(5./3.,m=-2)
# 0.9906190683480501
```

The `None` here means that there is no inner Lindblad resonance, the `m=-2` resonance is in the Solar neighborhood (see the section on the *Hercules stream* in this documentation).

1.4.5 UPDATED IN v1.7 Using interpolations of potentials

`galpy` contains various ways to set up interpolated versions of potentials that can be used to generate interpolations of potentials that can be used in their stead to speed up calculations when the calculation of the original potential is computationally expensive (for example, for the `DoubleExponentialDiskPotential`).

To interpolated spherical potentials, use the `interpSphericalPotential` class, described in detail *here*. To set up an instance, simply provide a function that gives the radial force as a function of (spherical) radius and a grid to interpolate it over (to set up a potential for a given enclosed mass, give the enclosed mass divided by radius squared). Alternatively, provide a spherical `galpy` potential instance or a list of such instances to build an interpolated version of them.

To interpolate axisymmetric potentials, use the `interpRZPotential` class. Full details on how to set this up are given *here*.

Interpolated potentials can be used anywhere that general three-dimensional galpy potentials can be used. Some care must be taken with outside-the-interpolation-grid evaluations for functions that use `C` to speed up computations.

1.4.6 Initializing potentials with parameters with units

As already discussed in the section on *physical units*, potentials in galpy can be specified with parameters with units since v1.2. For most inputs to the initialization it is straightforward to know what type of units the input Quantity needs to have. For example, the scale length parameter `a=` of a Miyamoto-Nagai disk needs to have units of distance.

The amplitude of a potential is specified through the `amp=` initialization parameter. The units of this parameter vary from potential to potential. For example, for a logarithmic potential the units are velocity squared, while for a Miyamoto-Nagai potential they are units of mass. Check the documentation of each potential on the *API page* for the units of the `amp=` parameter of the potential that you are trying to initialize and please report an Issue if you find any problems with this.

1.4.7 UPDATED IN v1.7 General density/potential pairs with basis-function expansions

`galpy` allows for the potential and forces of general, time-independent density functions to be computed by expanding the potential and density in terms of basis functions. This is supported for ellipsoidal-ish as well as for disk-y density distributions, in both cases using the basis-function expansion of the self-consistent-field (SCF) method of Hernquist & Ostriker (1992). On its own, the SCF technique works well for ellipsoidal-ish density distributions, but using a trick due to Kuijken & Dubinski (1995) it can also be made to work well for disky potentials. We first describe the basic SCF implementation and then discuss how to use it for disky potentials.

The basis-function approach in the SCF method is implemented in the *SCFPotential* class, which is also implemented in C for fast orbit integration. The coefficients of the basis-function expansion can be computed using the *scf_compute_coeffs_spherical* (for spherically-symmetric density distribution), *scf_compute_coeffs_axi* (for axisymmetric densities), and *scf_compute_coeffs* (for the general case). The coefficients obtained from these functions can be directly fed into the *SCFPotential* initialization. The basis-function expansion has a free scale parameter `a`, which can be specified for the `scf_compute_coeffs_XX` functions and for the `SCFPotential` itself. Make sure that you use the same `a`! Note that the general functions are quite slow. Equivalent functions for computing the coefficients based on an N-body snapshot are also available: *scf_compute_coeffs_spherical_nbody*, *scf_compute_coeffs_axi_nbody*, and *scf_compute_coeffs_nbody*.

The simplest example is that of the Hernquist potential, which is the lowest-order basis function. When we compute the first ten radial coefficients for this density we obtain that only the lowest-order coefficient is non-zero

```
>>> from galpy.potential import HernquistPotential
>>> from galpy.potential import scf_compute_coeffs_spherical
>>> hp= HernquistPotential(amp=1.,a=2.)
>>> Acos, Asin= scf_compute_coeffs_spherical(hp.dens,10,a=2.)
>>> print(Acos)
# array([[[  1.00000000e+00]],
#          [[ -2.83370393e-17]],
#          [[  3.31150709e-19]],
#          [[ -6.66748299e-18]],
#          [[  8.19285777e-18]],
#          [[ -4.26730651e-19]],
#          [[ -7.16849567e-19]],
#          [[  1.52355608e-18]],
#          [[ -2.24030288e-18]],
#          [[ -5.24936820e-19]]])
```

As a more complicated example, consider a prolate NFW potential

```
>>> from galpy.potential import TriaxialNFWPotential
>>> np= TriaxialNFWPotential(normalize=1.,c=1.4,a=1.)
```

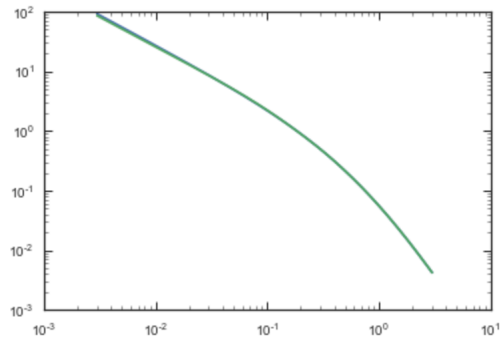
and we compute the coefficients using the axisymmetric `scf_compute_coeffs_axi`

```
>>> a_SCF= 50. # much larger a than true scale radius works well for NFW
>>> Acos, Asin= scf_compute_coeffs_axi(np.dens,80,40,a=a_SCF)
>>> sp= SCFPotential(Acos=Acos,Asin=Asin,a=a_SCF)
```

If we compare the densities along the `R=Z` line as

```
>>> xs= numpy.linspace(0.,3.,1001)
>>> loglog(xs,np.dens(xs,xs))
>>> loglog(xs,sp.dens(xs,xs))
```

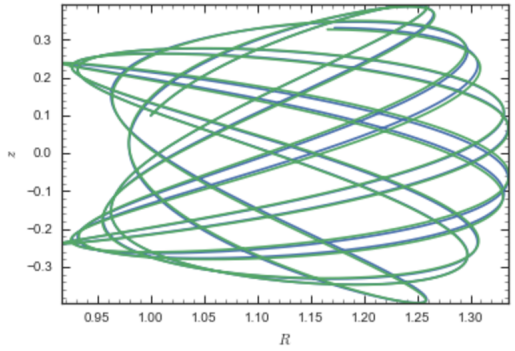
we get



If we then integrate an orbit, we also get good agreement

```
>>> from galpy.orbit import Orbit
>>> o= Orbit([1.,0.1,1.1,0.1,0.3,0.])
>>> ts= numpy.linspace(0.,100.,10001)
>>> o.integrate(ts,hp)
>>> o.plot()
>>> o.integrate(ts,sp)
>>> o.plot(overplot=True)
```

which gives



Near the end of the orbit integration, the slight differences between the original potential and the basis-expansion version cause the two orbits to deviate from each other.

To use the SCF method for disky potentials, we use the trick from Kuijken & Dubinski (1995). This trick works by approximating the disk density as $\rho_{\rm disk}(R,\phi,z) \approx \sum_i \Sigma_i(R)\,h_i(z)$, with $h_i(z) = \mathrm{d}^2H(z)/\mathrm{d}z^2$ and searching for solutions of the form

$$\Phi(R,\phi,z = \Phi_{\rm ME}(R,\phi,z) + 4\pi G \sum_i \Sigma_i(r)\,H_i(z)\,,$$

where r is the spherical radius $r^2 = R^2 + z^2$. The density which gives rise to $\Phi_{\rm ME}(R,\phi,z)$ is not strongly confined to a plane when $\rho_{\rm disk}(R,\phi,z) \approx \sum_i \Sigma_i(R)\,h_i(z)$ and can be obtained using the SCF basis-function-expansion technique discussed above. See the documentation of the *DiskSCFPotential* class for more details on this procedure.

As an example, consider a double-exponential disk, which we can compare to the `DoubleExponentialDiskPotential` implementation

```
>>> from galpy import potential
>>> dp= potential.DoubleExponentialDiskPotential(amp=13.5,hr=1./3.,hz=1./27.)
```

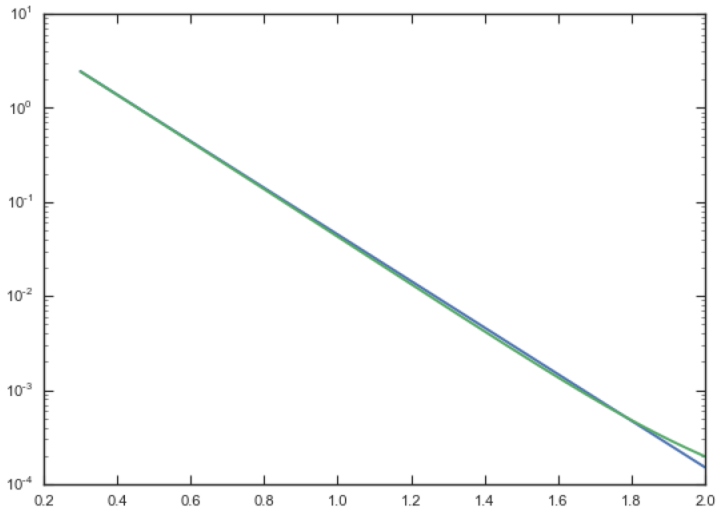
and then setup the `DiskSCFPotential` approximation to this as

```
>>> dscfp= potential.DiskSCFPotential(dens=lambda R,z: dp.dens(R,z),
                                      Sigma={'type':'exp','h':1./3.,'amp':1.},
                                      hz={'type':'exp','h':1./27.},
                                      a=1.,N=10,L=10)
```

The `dens=` keyword specifies the target density, while the `Sigma=` and `hz=` inputs specify the approximation functions $\Sigma_i(R)$ and $h_i(z)$. These are specified as dictionaries here for a few pre-defined approximation functions, but general functions are supported as well. Care should be taken that the `dens=` input density and the approximation functions have the same normalization. We can compare the density along the `R=10 z` line as

```
>>> xs= numpy.linspace(0.3,2.,1001)
>>> semilogy(xs,dp.dens(xs,xs/10.))
>>> semilogy(xs,dscfp.dens(xs,xs/10.))
```

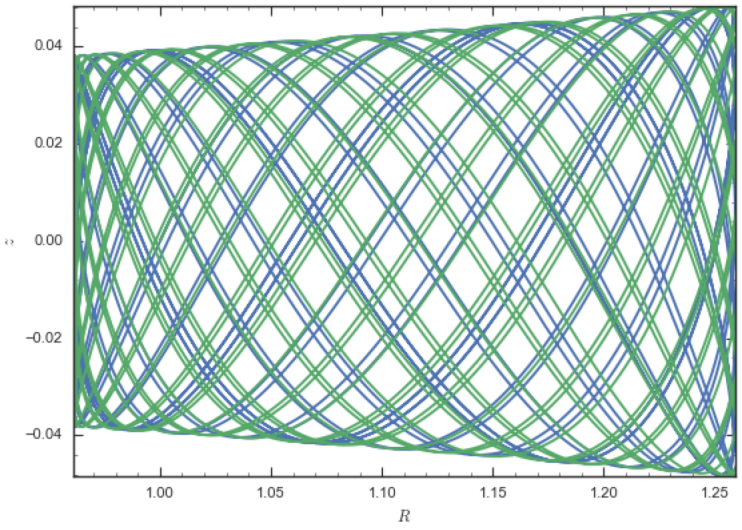
which gives



The agreement is good out to 5 scale lengths and scale heights and then starts to degrade. We can also integrate orbits and compare them

```
>>> from galpy.orbit import Orbit
>>> o= Orbit([1.,0.1,0.9,0.,0.1,0.])
>>> ts= numpy.linspace(0.,100.,10001)
>>> o.integrate(ts,dp)
>>> o.plot()
>>> o.integrate(ts,dscfp)
>>> o.plot(overplot=True)
```

which gives



The orbits diverge slightly because the potentials are not quite the same, but have very similar properties otherwise (peri- and apogalacticons, eccentricity, ...). By increasing the order of the SCF approximation, the potential can be gotten closer to the target density. Note that orbit integration in the `DiskSCFPotential` is much faster than that of the `DoubleExponentialDisk` potential

```
>>> timeit(o.integrate(ts,dp))
# 1 loops, best of 3: 5.83 s per loop
```

(continues on next page)

(continued from previous page)

```
>>> timeit(o.integrate(ts,dscfp))
# 1 loops, best of 3: 286 ms per loop
```

The *SCFPotential* and *DiskSCFPotential* can be used wherever general potentials can be used in galpy.

1.4.8 The potential of N-body simulations

`galpy` can setup and work with the frozen potential of an N-body simulation. This allows us to study the properties of such potentials in the same way as other potentials in `galpy`. We can also investigate the properties of orbits in these potentials and calculate action-angle coordinates, using the `galpy` framework. Currently, this functionality is limited to axisymmetrized versions of the N-body snapshots, although this capability could be somewhat straightforwardly expanded to full triaxial potentials. The use of this functionality requires pynbody to be installed; the potential of any snapshot that can be loaded with `pynbody` can be used within `galpy`.

As a first, simple example of this we look at the potential of a single simulation particle, which should correspond to galpy's `KeplerPotential`. We can create such a single-particle snapshot using `pynbody` by doing

```
>>> import pynbody
>>> s= pynbody.new(star=1)
>>> s['mass']= 1.
>>> s['eps']= 0.
```

and we get the potential of this snapshot in `galpy` by doing

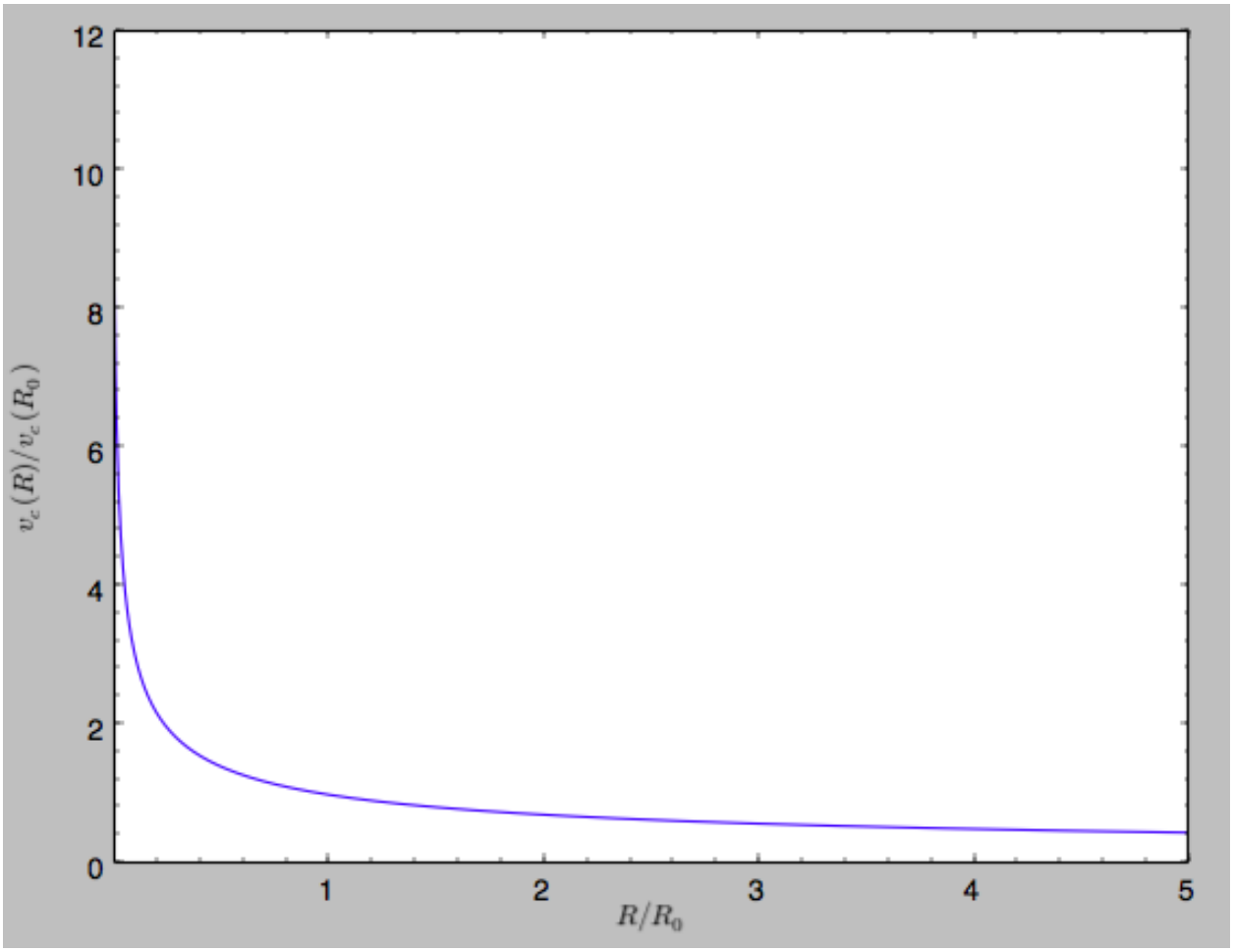
```
>>> from galpy.potential import SnapshotRZPotential
>>> sp= SnapshotRZPotential(s,num_threads=1)
```

With these definitions, this snapshot potential should be the same as `KeplerPotential` with an amplitude of one, which we can test as follows

```
>>> from galpy.potential import KeplerPotential
>>> kp= KeplerPotential(amp=1.)
>>> print(sp(1.1,0.),kp(1.1,0.),sp(1.1,0.)-kp(1.1,0.))
# (-0.90909090909090906, -0.9090909090909091, 0.0)
>>> print(sp.Rforce(1.1,0.),kp.Rforce(1.1,0.),sp.Rforce(1.1,0.)-kp.Rforce(1.1,0.))
# (-0.82644628099173545, -0.8264462809917353, -1.1102230246251565e-16)
```

`SnapshotRZPotential` instances can be used wherever other `galpy` potentials can be used (note that the second derivatives have not been implemented, such that functions depending on those will not work). For example, we can plot the rotation curve

```
>>> sp.plotRotcurve()
```



Because evaluating the potential and forces of a snapshot is computationally expensive, most useful applications of frozen N-body potentials employ interpolated versions of the snapshot potential. These can be setup in `galpy` using an `InterpSnapshotRZPotential` class that is a subclass of the `interpRZPotential` described above and that can be used in the same manner. To illustrate its use we will make use of one of `pynbody`'s example snapshots, `g15784`. This snapshot is used here to illustrate `pynbody`'s use. Please follow the instructions there on how to download this snapshot.

Once you have downloaded the `pynbody` testdata, we can load this snapshot using

```
>>> s = pynbody.load('testdata/g15784.lr.01024.gz')
```

(please adjust the path according to where you downloaded the `pynbody` testdata). We get the main galaxy in this snapshot, center the simulation on it, and align the galaxy face-on using

```
>>> h = s.halos()
>>> h1 = h[1]
>>> pynbody.analysis.halo.center(h1,mode='hyb')
>>> pynbody.analysis.angmom.faceon(h1, cen=(0,0,0),mode='ssc')
```

we also convert the simulation to physical units, but set *G=1* by doing the following

```
>>> s.physical_units()
>>> from galpy.util.conversion import _G
>>> g= pynbody.array.SimArray(_G/1000.)
```

(continues on next page)

(continued from previous page)

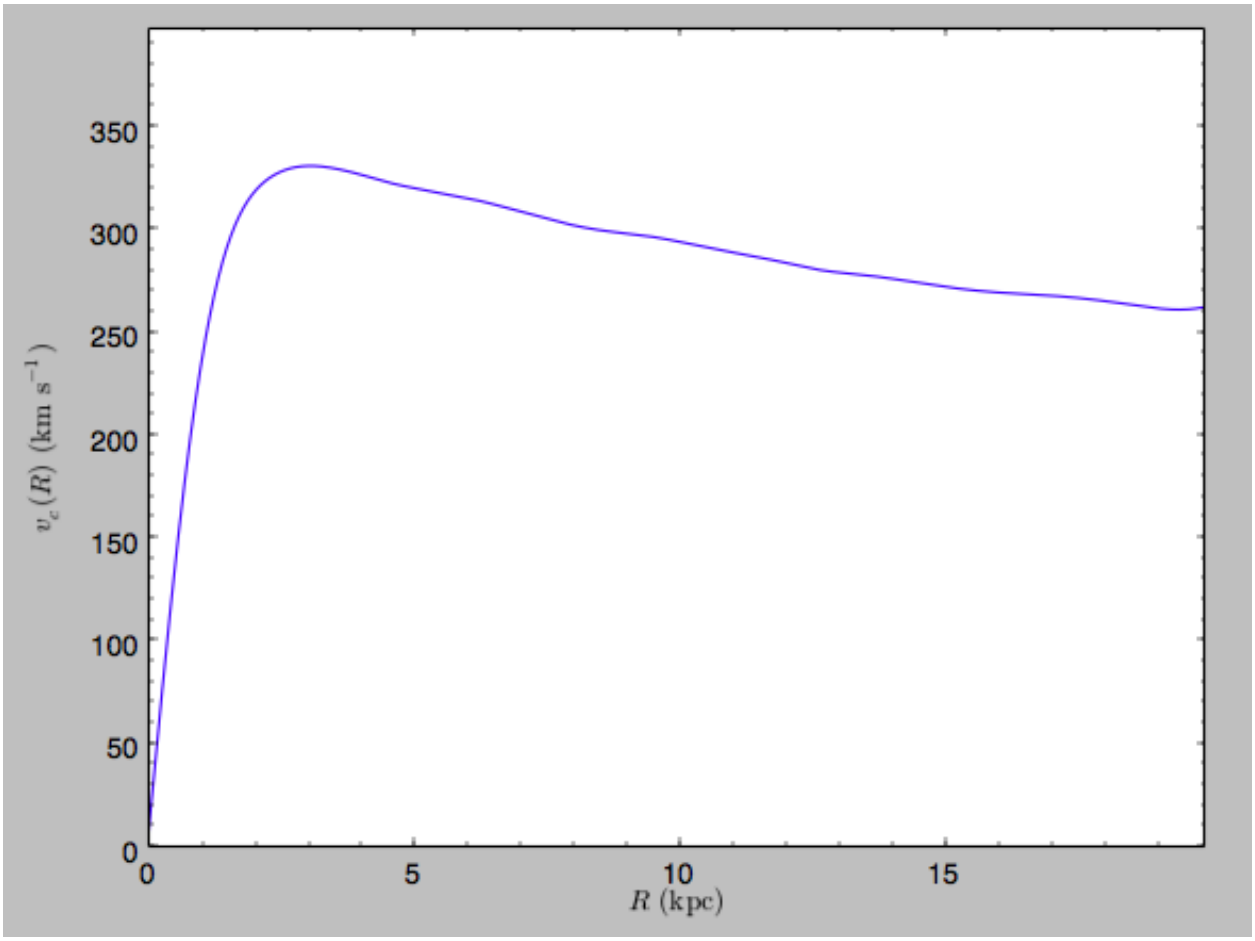
```
>>> g.units= 'kpc Msol**-1 km**2 s**-2 G**-1'
>>> s._arrays['mass']= s._arrays['mass']*g
```

We can now load an interpolated version of this snapshot's potential into `galpy` using

```
>>> from galpy.potential import InterpSnapshotRZPotential
>>> spi= InterpSnapshotRZPotential(h1,rgrid=(numpy.log(0.01),numpy.log(20.),101),
↪logR=True,zgrid=(0.,10.,101),interpPot=True,zsym=True)
```

where we further assume that the potential is symmetric around the mid-plane (*z=0*). This instantiation will take about ten to fiteen minutes. This potential instance has *physical* units (and thus the `rgrid=` and `zgrid=` inputs are given in kpc if the simulation's distance unit is kpc). For example, if we ask for the rotation curve, we get the following:

```
>>> spi.plotRotcurve(Rrange=[0.01,19.9],xlabel=r'$R\,(\mathrm{kpc})$',ylabel=r'$v_
↪c(R)\,(\mathrm{km\,s}^{-1})$')
```



This can be compared to the rotation curve calculated by `pynbody`, see here.

Because `galpy` works best in a system of *natural units* as explained in *Units in galpy*, we will convert this instance to natural units using the circular velocity at *R=10* kpc, which is

```
>>> spi.vcirc(10.)
# 294.62723076942245
```

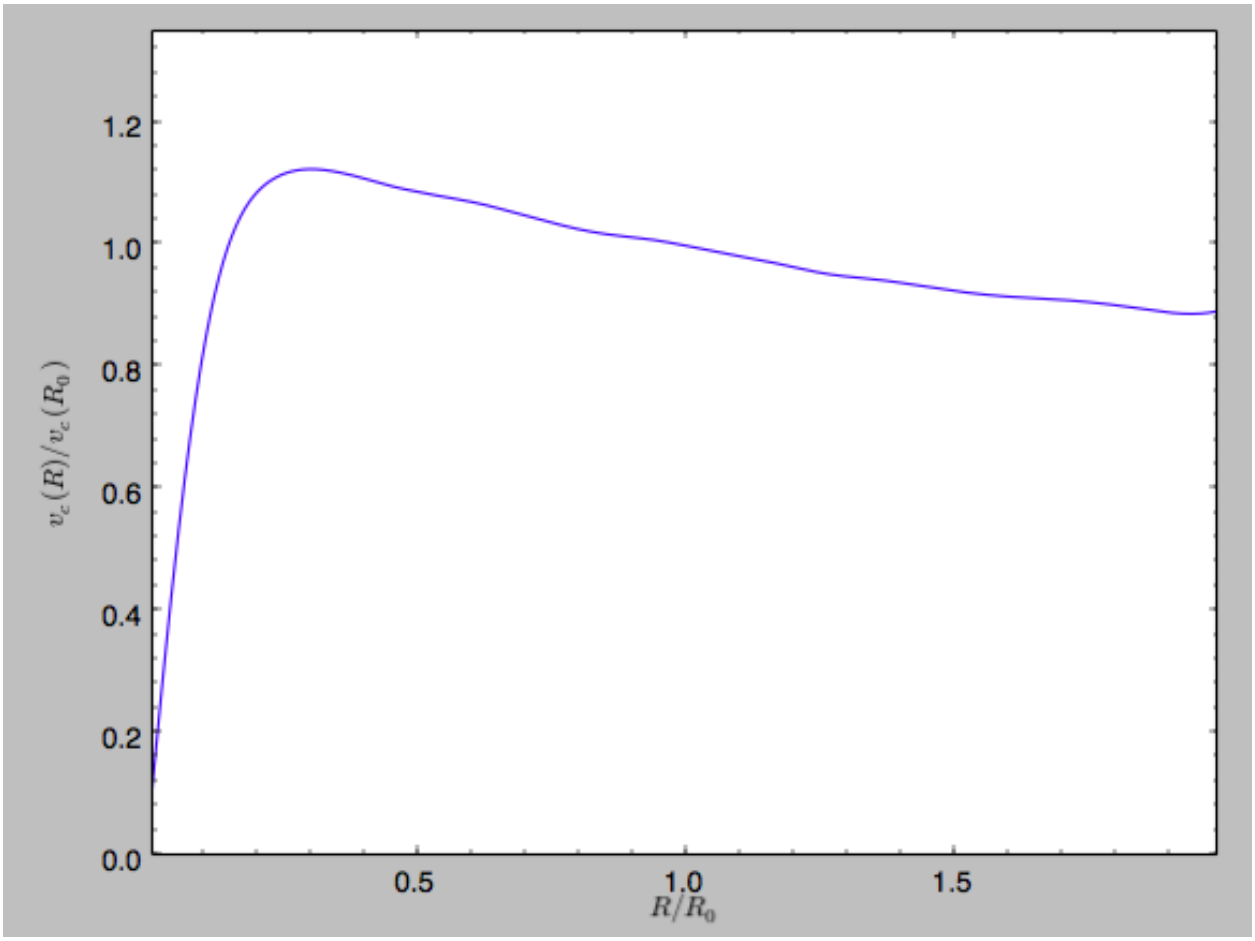
To convert to *natural units* we do

```
>>> spi.normalize(R0=10.)
```

We can then again plot the rotation curve, keeping in mind that the distance unit is now R_0

```
>>> spi.plotRotcurve(Rrange=[0.01,1.99])
```

which gives

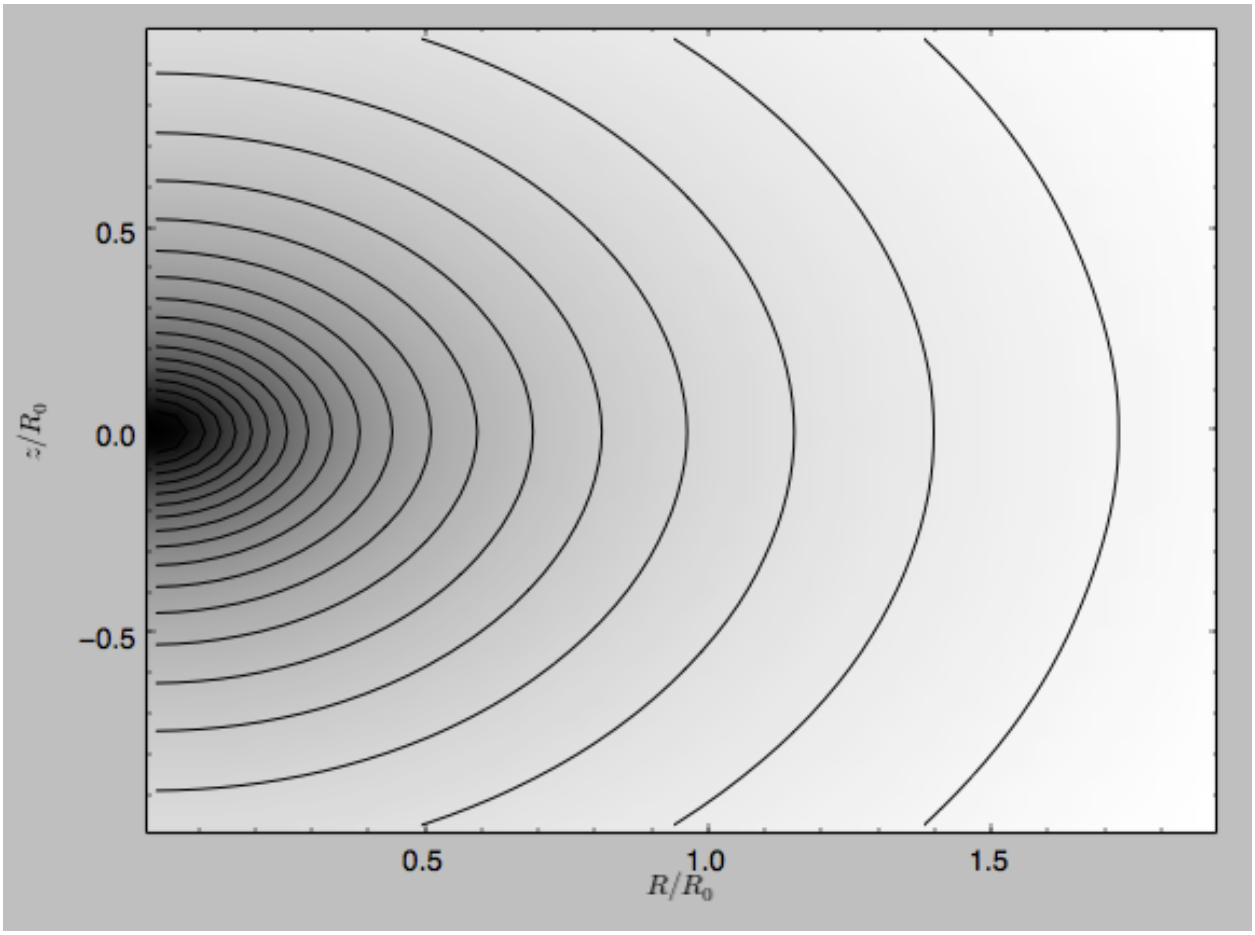


in particular

```
>>> spi.vcirc(1.)
# 1.0000000000000002
```

We can also plot the potential

```
>>> spi.plot(rmin=0.01,rmax=1.9,nrs=51,zmin=-0.99,zmax=0.99,nzs=51)
```



Clearly, this simulation's potential is quite spherical, which is confirmed by looking at the flattening

```
>>> spi.flattening(1.,0.1)
# 0.86675711023391921
>>> spi.flattening(1.5,0.1)
# 0.94442750306256895
```

The epicyle and vertical frequencies can also be interpolated by setting the `interpepifreq=True` or `interpverticalfreq=True` keywords when instantiating the `InterpSnapshotRZPotential` object.

1.4.9 Conversion to NEMO potentials

NEMO is a set of tools for studying stellar dynamics. Some of its functionality overlaps with that of `galpy`, but many of its programs are very complementary to `galpy`. In particular, it has the ability to perform N-body simulations with a variety of poisson solvers, which is currently not supported by `galpy` (and likely will never be directly supported). To encourage interaction between `galpy` and NEMO it is quite useful to be able to convert potentials between these two frameworks, which is not completely trivial. In particular, NEMO contains Walter Dehnen's fast collisionless `gyrfalcON` code (see 2000ApJ...536L..39D and 2002JCoPh.179...27D) and the discussion here focuses on how to run N-body simulations using external potentials defined in `galpy`.

Some `galpy` potential instances support the functions `nemo_accname` and `nemo_accpars` that return the name of the NEMO potential corresponding to this `galpy` Potential and its parameters in NEMO units. These functions assume that you use NEMO with WD_units, that is, positions are specified in kpc, velocities in kpc/Gyr, times in Gyr, and G=1. For the Miyamoto-Nagai potential above, you can get its name in the NEMO framework as

```
>>> mp.nemo_accname()
# 'MiyamotoNagai'
```

and its parameters as

```
>>> mp.nemo_accpars(220.,8.)
# '0,592617.11132,4.0,0.3'
```

assuming that we scale velocities by `vo=220` km/s and positions by `ro=8` kpc in galpy. These two strings can then be given to the `gyrfalcON` `accname=` and `accpars=` keywords.

We can do the same for lists of potentials. For example, for `MWPotential2014` we do

```
>>> from galpy.potential import nemo_accname, nemo_accpars
>>> nemo_accname(MWPotential2014)
# 'PowSphwCut+MiyamotoNagai+NFW'
>>> nemo_accpars(MWPotential2014,220.,8.)
# '0,1001.79126907,1.8,1.9#0,306770.418682,3.0,0.28#0,16.0,162.958241887'
```

Therefore, these are the `accname=` and `accpars=` that one needs to provide to `gyrfalcON` to run a simulation in `MWPotential2014`.

Note that the NEMO potential `PowSphwCut` is *not* a standard NEMO potential. This potential can be found in the nemo/ directory of the `galpy` source code; this directory also contains a Makefile that can be used to compile the extra NEMO potential and install it in the correct NEMO directory (this requires one to have NEMO running, i.e., having sourced nemo_start).

You can use the `PowSphwCut.cc` file in the nemo/ directory as a template for adding additional potentials in `galpy` to the NEMO framework. To figure out how to convert the normalized `galpy` potential to an amplitude when scaling to physical coordinates (like kpc and kpc/Gyr), one needs to look at the scaling of the radial force with R. For example, from the definition of MiyamotoNagaiPotential, we see that the radial force scales as R^{-2}. For a general scaling $R^{-\alpha}$, the amplitude will scale as $V_0^2\,R_0^{\alpha-1}$ with the velocity V_0 and position R_0 of the `v=1` at `R=1` normalization. Therefore, for the MiyamotoNagaiPotential, the physical amplitude scales as $V_0^2\,R_0$. For the LogarithmicHaloPotential, the radial force scales as R^{-1}, so the amplitude scales as V_0^2.

Currently, only the `MiyamotoNagaiPotential`, `NFWPotential`, `PowerSphericalPotentialwCutoff`, `HernquistPotential`, `PlummerPotential`, `MN3ExponentialDiskPotential`, and the `LogarithmicHaloPotential` have this NEMO support. Combinations of all but the `LogarithmicHaloPotential` are allowed in general (e.g., `MWPotential2014`); they can also be combined with spherical `LogarithmicHaloPotentials`. Because of the definition of the logarithmic potential in NEMO, it cannot be flattened in `z`, so to use a flattened logarithmic potential, one has to flip `y` and `z` between `galpy` and NEMO (one can flatten in `y`).

1.4.10 Conversion to AMUSE potentials

AMUSE is a Python software framework for astrophysical simulations, in which existing codes from different domains, such as stellar dynamics, stellar evolution, hydrodynamics and radiative transfer can be easily coupled. AMUSE allows you to run N-body simulations that include a wide range of physics (gravity, stellar evolution, hydrodynamics, radiative transfer) with a large variety of numerical codes (collisionless, collisional, etc.).

The `galpy.potential.to_amuse` function allows you to create an AMUSE representation of any `galpy` potential. This is useful, for instance, if you want to run a simulation of a stellar cluster in an external gravitational field, because `galpy` has wide support for representing external gravitational fields. Creating the AMUSE representation is as simple as (for `MWPotential2014`):

```
>>> from galpy.potential import to_amuse, MWPotential2014
>>> mwp_amuse= to_amuse(MWPotential2014)
>>> print(mwp_amuse)
# <galpy.potential.amuse.galpy_profile object at 0x7f6b366d13c8>
```

Schematically, this potential can then be used in AMUSE as

```
>>> gravity = bridge.Bridge(use_threading=False)
>>> gravity.add_system(cluster_code, (mwp_amuse,))
>>> gravity.add_system(mwp_amuse,)
```

where `cluster_code` is a code to perform the N-body integration of a system (e.g., a `BHTree` in AMUSE). A fuller example is given below.

AMUSE uses physical units when interacting with the galpy potential and it is therefore necessary to make sure that the correct physical units are used. The `to_amuse` function takes the `galpy` unit conversion parameters `ro=` and `vo=` as keyword parameters to perform the conversion between internal galpy units and physical units; if these are not explicitly set, `to_amuse` attempts to set them automatically using the potential that you input using the `galpy.util.conversion.get_physical` function.

Another difference between `galpy` and AMUSE is that in AMUSE integration times can only be positive and they have to increase in time. `to_amuse` takes as input the `t=` and `tgalpy=` keywords that specify (a) the initial time in AMUSE and (b) the initial time in `galpy` that this time corresponds to. Typically these will be the same (and equal to zero), but if you want to run a simulation where the initial time in `galpy` is negative it is useful to give them different values. The time inputs can be either given in `galpy` internal units or using AMUSE's units. Similarly, to integrate backwards in time in AMUSE, `to_amuse` has a keyword `reverse=` (default: `False`) that reverses the time direction given to the `galpy` potential; `reverse=True` does this (note that you also have to flip the velocities to actually go backwards).

A full example of setting up a Plummer-sphere cluster and evolving its N-body dynamics using an AMUSE `BHTree` in the external `MWPotential2014` potential is:

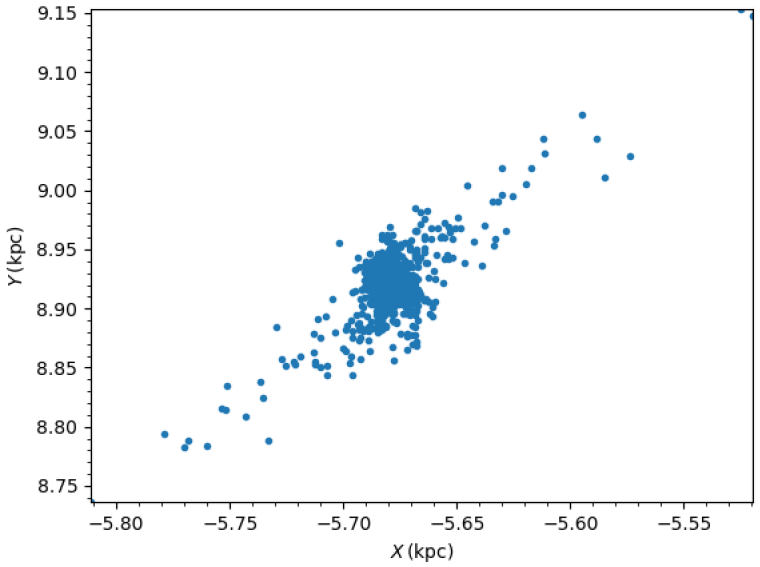
```
>>> from amuse.lab import *
>>> from amuse.couple import bridge
>>> from amuse.datamodel import Particles
>>> from galpy.potential import to_amuse, MWPotential2014
>>> from galpy.util import plot as galpy_plot
>>>
>>> # Convert galpy MWPotential2014 to AMUSE representation
>>> mwp_amuse= to_amuse(MWPotential2014)
>>>
>>> # Set initial cluster parameters
>>> N= 1000
>>> Mcluster= 1000. | units.MSun
>>> Rcluster= 10. | units.parsec
>>> Rinit= [10.,0.,0.] | units.kpc
>>> Vinit= [0.,220.,0.] | units.km/units.s
>>> # Setup star cluster simulation
>>> tend= 100.0 | units.Myr
>>> dtout= 5.0 | units.Myr
>>> dt= 1.0 | units.Myr
>>>
>>> def setup_cluster(N,Mcluster,Rcluster,Rinit,Vinit):
>>>     converter= nbody_system.nbody_to_si(Mcluster,Rcluster)
>>>     stars= new_plummer_sphere(N,converter)
>>>     stars.x+= Rinit[0]
>>>     stars.y+= Rinit[1]
```

(continues on next page)

(continued from previous page)

```
>>>     stars.z+= Rinit[2]
>>>     stars.vx+= Vinit[0]
>>>     stars.vy+= Vinit[1]
>>>     stars.vz+= Vinit[2]
>>>     return stars,converter
>>>
>>> # Setup cluster
>>> stars,converter= setup_cluster(N,Mcluster,Rcluster,Rinit,Vinit)
>>> cluster_code= BHTree(converter,number_of_workers=1) #Change number of workers␣
↪depending no. of CPUs
>>> cluster_code.parameters.epsilon_squared= (3. | units.parsec)**2
>>> cluster_code.parameters.opening_angle= 0.6
>>> cluster_code.parameters.timestep= dt
>>> cluster_code.particles.add_particles(stars)
>>>
>>> # Setup channels between stars particle dataset and the cluster code
>>> channel_from_stars_to_cluster_code= stars.new_channel_to(cluster_code.particles,
>>>                                         attributes=["mass", "x", "y", "z", "vx",
↪"vy", "vz"])
>>> channel_from_cluster_code_to_stars= cluster_code.particles.new_channel_to(stars,
>>>                                         attributes=["mass", "x", "y", "z", "vx",
↪"vy", "vz"])
>>>
>>> # Setup gravity bridge
>>> gravity= bridge.Bridge(use_threading=False)
>>> # Stars in cluster_code depend on gravity from external potential mwp_amuse (i.e.,
↪ MWPotential2014)
>>> gravity.add_system(cluster_code, (mwp_amuse,))
>>> # External potential mwp_amuse still needs to be added to system so it evolves␣
↪with time
>>> gravity.add_system(mwp_amuse,)
>>> # Set how often to update external potential
>>> gravity.timestep= cluster_code.parameters.timestep/2.
>>> # Evolve
>>> time= 0.0 | tend.unit
>>> while time<tend:
>>>     gravity.evolve_model(time+dt)
>>>     # If you want to output or analyze the simulation, you need to copy
>>>     # stars from cluster_code
>>>     #channel_from_cluster_code_to_stars.copy()
>>>
>>>     # If you edited the stars particle set, for example to remove stars from the
>>>     # array because they have been kicked far from the cluster, you need to
>>>     # copy the array back to cluster_code:
>>>     #channel_from_stars_to_cluster_code.copy()
>>>
>>>     # Update time
>>>     time= gravity.model_time
>>>
>>> channel_from_cluster_code_to_stars.copy()
>>> gravity.stop()
>>>
>>> galpy_plot.plot(stars.x.value_in(units.kpc),stars.y.value_in(units.kpc),'.',
>>>                     xlabel=r'$X\,(\mathrm{kpc})$',ylabel=r'$Y\,(\mathrm{kpc})$')
```

After about 30 seconds, you should get a plot like the following, which shows a cluster in the first stages of disruption:



1.4.11 Dissipative forces

While almost all of the forces that you can use in `galpy` derive from a potential (that is, the force is the gradient of a scalar function, the potential, meaning that the forces are *conservative*), `galpy` also supports dissipative forces. Dissipative forces all inherit from the `DissipativeForce` class and they are required to take the velocity `v=[vR, vT, vZ]` in cylindrical coordinates as an argument to the force in addition to the standard `(R,z,phi=0,t=0)`. The set of functions `evaluateXforces` (with `X=R, z, r, phi, etc.`) will evaluate the force due to `Potential` instances, `DissipativeForce` instances, or lists of combinations of these two.

Currently, the dissipative forces implemented in `galpy` include *ChandrasekharDynamicalFrictionForce*, an implementation of the classic Chandrasekhar dynamical-friction formula, with recent tweaks to better represent the results from N-body simulations, and *NonInertialFrameForce*, the fictitious forces of a non-inertial reference frame.

Warning: Dissipative forces can currently only be used for 3D orbits in `galpy`. The code should throw an error when they are used for 2D orbits.

Warning: While we call them 'dissipative', what is really meant is that the force depends on the velocity, whether the force is really dissipative or not.

1.4.12 Adding potentials to the galpy framework

Potentials in galpy can be used in many places such as orbit integration, distribution functions, or the calculation of action-angle variables, and in most cases any instance of a potential class that inherits from the general `Potential` class (or a list of such instances) can be given. For example, all orbit integration routines work with any list of instances of the general `Potential` class. Adding new potentials to galpy therefore allows them to be used everywhere in galpy where general `Potential` instances can be used. Adding a new class of potentials to galpy consists of the following series of steps (for steps to add a new wrapper potential, also see *the next section*):

1. Implement the new potential in a class that inherits from `galpy.potential.Potential` (velocity-dependent forces should inherit from `galpy.potential.DissipativeForce` instead; see below for a brief discussion on differences in implementing such forces). The new class should have an `__init__`

method that sets up the necessary parameters for the class. An amplitude parameter `amp=` and two units parameters `ro=` and `vo=` should be taken as an argument for this class and before performing any other setup, the `galpy.potential.Potential.__init__(self,amp=amp,ro=ro,vo=vo, amp_units=)` method should be called to setup the amplitude and the system of units; the `amp_units=` keyword specifies the physical units of the amplitude parameter (e.g., `amp_units='velocity2'` when the units of the amplitude are velocity-squared) To add support for normalizing the potential to standard galpy units, one can call the `galpy.potential.Potential.normalize` function at the end of the __init__ function.

The new potential class should implement some of the following functions:

- `_evaluate(self,R,z,phi=0,t=0)` which evaluates the potential itself (*without* the amp factor, which is added in the `__call__` method of the general Potential class).
- `_Rforce(self,R,z,phi=0.,t=0.)` which evaluates the radial force in cylindrical coordinates (-d potential / d R).
- `_zforce(self,R,z,phi=0.,t=0.)` which evaluates the vertical force in cylindrical coordinates (-d potential / d z).
- `_R2deriv(self,R,z,phi=0.,t=0.)` which evaluates the second (cylindrical) radial derivative of the potential (d^2 potential / d R^2).
- `_z2deriv(self,R,z,phi=0.,t=0.)` which evaluates the second (cylindrical) vertical derivative of the potential (d^2 potential / d z^2).
- `_Rzderiv(self,R,z,phi=0.,t=0.)` which evaluates the mixed (cylindrical) radial and vertical derivative of the potential (d^2 potential / d R d z).
- `_dens(self,R,z,phi=0.,t=0.)` which evaluates the density. If not given, the density is computed using the Poisson equation from the first and second derivatives of the potential (if all are implemented).
- `_mass(self,R,z=0.,t=0.)` which evaluates the mass. For spherical potentials this should give the mass enclosed within the spherical radius; for axisymmetric potentials this should return the mass up to `R` and between `-Z` and `Z`. If not given, the mass is computed by integrating the density (if it is implemented or can be calculated from the Poisson equation).
- `_phiforce(self,R,z,phi=0.,t=0.)`: the azimuthal force in cylindrical coordinates (assumed zero if not implemented).
- `_phi2deriv(self,R,z,phi=0.,t=0.)`: the second azimuthal derivative of the potential in cylindrical coordinates (d^2 potential / d phi^2; assumed zero if not given).
- `_Rphideriv(self,R,z,phi=0.,t=0.)`: the mixed radial and azimuthal derivative of the potential in cylindrical coordinates (d^2 potential / d R d phi; assumed zero if not given).
- `OmegaP(self)`: returns the pattern speed for potentials with a pattern speed (used to compute the Jacobi integral for orbits).

If you want to be able to calculate the concentration for a potential, you also have to set `self._scale` to a scale parameter for your potential.

The code for `galpy.potential.MiyamotoNagaiPotential` gives a good template to follow for 3D axisymmetric potentials. Similarly, the code for `galpy.potential.CosmphiDiskPotential` provides a good template for 2D, non-axisymmetric potentials.

During development or if some of the forces or second derivatives are too tedious to implement, it is possible to numerically compute any non-implemented forces and second derivatives by inheriting from the *NumericalPotentialDerivativesMixin* class. Thus, a functioning potential can be implemented by simply implementing the `_evaluate` function and adding all forces and second derivatives using the `NumericalPotentialDerivativesMixin`.

After this step, the new potential will work in any part of galpy that uses pure python potentials. To get the potential to work with the C implementations of orbit integration or action-angle calculations, the potential also has to be implemented in C and the potential has to be passed from python to C (see below).

The `__init__` method should be written in such a way that a relevant object can be initialized using `Classname()` (i.e., there have to be reasonable defaults given for all parameters, including the amplitude); doing this allows the nose tests for potentials to automatically check that your Potential's potential function, force functions, second derivatives, and density (through the Poisson equation) are correctly implemented (if they are implemented). The continuous-integration platform that builds the galpy codebase upon code pushes will then automatically test all of this, streamlining push requests of new potentials.

A few atrributes need to be set depending on the potential: `hasC=True` for potentials for which the forces and potential are implemented in C (see below); `self.hasC_dxdv=True` for potentials for which the (planar) second derivatives are implemented in C; `self.hasC_dens=True` for potentials for which the density is implemented in C as well (necessary for them to work with dynamical friction in C); `self.isNonAxi=True` for non-axisymmetric potentials.

2. To add a C implementation of the potential, implement it in a .c file under `potential/potential_c_ext`. Look at `potential/potential_c_ext/LogarithmicHaloPotential.c` for the right format for 3D, axisymmetric potentials, or at `potential/potential_c_ext/LopsidedDiskPotential.c` for 2D, non-axisymmetric potentials.

   For orbit integration, the functions such as:

   - double LogarithmicHaloPotentialRforce(double R,double Z, double phi,double t,struct potentialArg * potentialArgs)
   - double LogarithmicHaloPotentialzforce(double R,double Z, double phi,double t,struct potentialArg * potentialArgs)

   are most important. For some of the action-angle calculations

   - double LogarithmicHaloPotentialEval(double R,double Z, double phi,double t,struct potentialArg * potentialArgs)

   is most important (i.e., for those algorithms that evaluate the potential). If you want your potential to be able to be used as the density for the *ChandrasekharDynamicalFrictionForce* implementation in C, you need to implement the density in C as well

   - double LogarithmicHaloPotentialDens(double R,double Z, double phi,double t,struct potentialArg * potentialArgs)

   The arguments of the potential are passed in a `potentialArgs` structure that contains `args`, which are the arguments that should be unpacked. Again, looking at some example code will make this clear. The `potentialArgs` structure is defined in `potential/potential_c_ext/galpy_potentials.h`.

3. Add the potential's function declarations to `potential/potential_c_ext/galpy_potentials.h`

4. (4. and 5. for planar orbit integration) Edit the code under `orbit/orbit_c_ext/integratePlanarOrbit.c` to set up your new potential (in the **parse_leapFuncArgs** function).

5. Edit the code in `orbit/integratePlanarOrbit.py` to set up your new potential (in the **_parse_pot** function).

6. Edit the code under `orbit/orbit_c_ext/integrateFullOrbit.c` to set up your new potential (in the **parse_leapFuncArgs_Full** function).

7. Edit the code in `orbit/integrateFullOrbit.py` to set up your new potential (in the **_parse_pot** function).

8. Finally, add `self.hasC= True` to the initialization of the potential in question (after the initialization of the super class, or otherwise it will be undone). If you have implemented the necessary second derivatives for integrating

phase-space volumes, also add `self.hasC_dxdv=True`. If you have implemented the density in C, set `self.hasC_dens=True`.

After following the relevant steps, the new potential class can be used in any galpy context in which C is used to speed up computations.

Velocity-dependent forces (e.g., *ChandrasekharDynamicalFrictionForce*) should inherit from `galpy.potential.DissipativeForce` instead of from `galpy.potential.Potential`. Because such forces are not conservative, you only need to implement the forces themselves, in the same way as for a regular `Potential`. For dissipative forces, the force-evaluation functions (`Rforce`, etc.) need to take the velocity in cylindrical coordinates as a keyword argument: `v=[vR,vT,vZ]`. Implementing dissipative forces in C is similar: you only need to implement the forces themselves and the forces should take the velocity in cylindrical coordinates as an additional input, e.g.,

- double ChandrasekharDynamicalFrictionForceRforce(double R,double z, double phi,double t,struct potentialArg * potentialArgs,double vR,double vT,double vz)

1.4.13 Adding wrapper potentials to the galpy framework

Wrappers all inherit from the general `WrapperPotential` or `planarWrapperPotential` classes (which themselves inherit from the `Potential` and `planarPotential` classes and therefore all wrappers are `Potentials` or `planarPotentials`). Depending on the complexity of the wrapper, wrappers can be implemented much more economically in Python than new `Potential` instances as described *above*.

To add a Python implementation of a new wrapper, classes need to inherit from `parentWrapperPotential`, take the potentials to be wrapped as a `pot=` (a `Potential`, `planarPotential`, or a list thereof; automatically assigned to `self._pot`) input to `__init__`, and implement the `_wrap(self,attribute,*args,**kwargs)` function. This function modifies the Potential functions `_evaluate`, `_Rforce`, etc. (all of those listed *above*), with `attribute` the function that is being modified. Inheriting from `parentWrapperPotential` gives the class access to the `self._wrap_pot_func(attribute)` function which returns the relevant function for each attribute. For example, `self._wrap_pot_func('_evaluate')` returns the `evaluatePotentials` function that can then be called as `self._wrap_pot_func('_evaluate')(self._pot,R,Z,phi=phi,t=t)` to evaluate the potentials being wrapped. By making use of `self._wrap_pot_func`, wrapper potentials can be implemented in just a few lines. Your `__init__` function should *only* initialize things in your wrapper; there is no need to manually assign `self._pot` or to call the superclass' `__init__` (all automatically done for you!).

To correctly work with both 3D and 2D potentials, inputs to `_wrap` need to be specified as `*args,**kwargs`: grab the values you need for `R,z,phi,t` from these as `R=args[0], z=0 if len(args) == 1 else args[1], phi=kwargs.get('phi',0.), t=kwargs.get('t',0.)`, where the complicated expression for z is to correctly deal with both 3D and 2D potentials (of course, if your wrapper depends on z, it probably doesn't make much sense to apply it to a 2D planarPotential; you could check the dimensionality of `self._pot` in your wrapper's `__init__` function with `from galpy.potential.Potential._dim` and raise an error if it is not 3 in this case). Wrapping a 2D potential automatically results in a wrapper that is a subclass of `planarPotential` rather than `Potential`; this is done by the setup in `parentWrapperPotential` and hidden from the user. For wrappers of planar Potentials, `self._wrap_pot_func(attribute)` will return the `evaluateplanarPotentials` etc. functions instead, but this is again hidden from the user if you implement the `_wrap` function as explained above.

As an example, for the `DehnenSmoothWrapperPotential`, the `_wrap` function is

```
def _wrap(self,attribute,*args,**kwargs):
    return self._smooth(kwargs.get('t',0.))\
            *self._wrap_pot_func(attribute)(self._pot,*args,**kwargs)
```

where `smooth(t)` returns the smoothing function of the amplitude. When any of the basic `Potential` functions are called (`_evaluate`, `_Rforce`, etc.), `_wrap` gets called by the superclass `WrapperPotential`, and the `_wrap` function returns the corresponding function for the wrapped potentials with the amplitude modified by `smooth(t)`. Therefore, one does not need to implement each of the `_evaluate`, `_Rforce`, etc. functions like

for regular potential. The rest of the DehnenSmoothWrapperPotential is essentially (slightly simplified in non-crucial aspects)

```
def __init__(self,amp=1.,pot=None,tform=-4.,tsteady=None,ro=None,vo=None):
    # Note: (i) don't assign self._pot and (ii) don't run super.__init__
    self._tform= tform
    if tsteady is None:
        self._tsteady= self._tform/2.
    else:
        self._tsteady= self._tform+tsteady
    self.hasC= True
    self.hasC_dxdv= True

def _smooth(self,t):
    #Calculate relevant time
    if t < self._tform:
        smooth= 0.
    elif t < self._tsteady:
        deltat= t-self._tform
        xi= 2.*deltat/(self._tsteady-self._tform)-1.
        smooth= (3./16.*xi**5.-5./8*xi**3.+15./16.*xi+.5)
    else: #bar is fully on
        smooth= 1.
    return smooth
```

The source code for DehnenSmoothWrapperPotential potential may act as a guide to implementing new wrappers.

C implementations of potential wrappers can also be added in a similar way as C implementations of regular potentials (all of the steps listed in the *previous section* for adding a potential to C need to be followed). All of the necessary functions (...Rforce, ...zforce, ..phiforce, etc.) need to be implemented separately, but by including galpy_potentials.h calling the relevant functions of the wrapped potentials is easy. Look at DehnenSmoothWrapperPotential.c for an example that can be straightforwardly edited for other wrappers.

The glue between Python and C for wrapper potentials needs to glue both the wrapper and the wrapped potentials. This can be easily achieved by recursively calling the _parse_pot glue functions in Python (see the previous section; this needs to be done separately for each potential currently) and the parse_leapFuncArgs and parse_leapFuncArgs_Full functions in C (done automatically for all wrappers). Again, following the example of DehnenSmoothWrapperPotential.py should allow for a straightforward implementation of the glue for any new wrappers. Wrapper potentials should be given negative potential types in the glue to distinguish them from regular potentials.

1.4.14 Adding dissipative forces to the galpy framework

Dissipative forces are implemented in much the same way as forces that derive from potentials. Rather than inheriting from galpy.potential.Potential, dissipative forces inherit from galpy.potential.DissipativeForce. The procedure for implementing a new class of dissipative force is therefore very similar to that for *implementing a new potential*. The main differences are that (a) you only need to implement the forces and (b) the forces are required to take an extra keyword argument v= that gives the velocity in cylindrical coordinates (because dissipative forces will in general depend on the current velocity). Thus, the steps are:

1. Implement the new dissipative force in a class that inherits from galpy.potential.DissipativeForce. The new class should have an __init__ method that sets up the necessary parameters for the class. An amplitude parameter amp= and two units parameters ro= and vo= should be taken as an argument for this class and before performing any other setup, the galpy.potential.DissipativeForce.__init__(self,amp=amp,ro=ro,vo=vo,amp_units=) method should

be called to setup the amplitude and the system of units; the `amp_units=` keyword specifies the physical units of the amplitude parameter (e.g., `amp_units='mass'` when the units of the amplitude are mass)

The new dissipative-force class should implement the following functions:

- `_Rforce(self,R,z,phi=0.,t=0.,v=None)` which evaluates the radial force in cylindrical coordinates
- `_phiforce(self,R,z,phi=0.,t=0.,v=None)` which evaluates the azimuthal force in cylindrical coordinates
- `_zforce(self,R,z,phi=0.,t=0.,v=None)` which evaluates the vertical force in cylindrical coordinates

The code for `galpy.potential.ChandrasekharDynamicalFrictionForce` gives a good template to follow.

2. That's it, as for now there is no support for implementing a C version of dissipative forces.

1.5 A closer look at orbit integration

1.5.1 Orbit initialization

Standard initialization

Orbits can be initialized in various coordinate frames. The simplest initialization gives the initial conditions directly in the Galactocentric cylindrical coordinate frame (or in the rectangular coordinate frame in one dimension). `Orbit()` automatically figures out the dimensionality of the space from the initial conditions in this case. In three dimensions initial conditions are given either as `[R,vR,vT,z,vz,phi]` or one can choose not to specify the azimuth of the orbit and initialize with `[R,vR,vT,z,vz]`. Since potentials in galpy are easily initialized to have a circular velocity of one at a radius equal to one, initial coordinates are best given as a fraction of the radius at which one specifies the circular velocity, and initial velocities are best expressed as fractions of this circular velocity. For example,

```
>>> from galpy.orbit import Orbit
>>> o= Orbit([1.,0.1,1.1,0.,0.1,0.])
```

initializes a fully three-dimensional orbit, while

```
>>> o= Orbit([1.,0.1,1.1,0.,0.1])
```

initializes an orbit in which the azimuth is not tracked, as might be useful for axisymmetric potentials.

In two dimensions, we can similarly specify fully two-dimensional orbits `o=Orbit([R,vR,vT,phi])` or choose not to track the azimuth and initialize with `o= Orbit([R,vR,vT])`.

In one dimension we simply initialize with `o= Orbit([x,vx])`.

Initialization with physical units

Orbits are normally used in galpy's *natural coordinates*. When Orbits are initialized using a distance scale `ro=` and a velocity scale `vo=`, then many Orbit methods return quantities in physical coordinates. Specifically, physical distance and velocity scales are specified as

```
>>> op= Orbit([1.,0.1,1.1,0.,0.1,0.],ro=8.,vo=220.)
```

All output quantities will then be automatically be specified in physical units: kpc for positions, km/s for velocities, (km/s)^2 for energies and the Jacobi integral, km/s kpc for the angular momentum o.L() and actions, 1/Gyr for frequencies, and Gyr for times and periods. See below for examples of this.

The actual initial condition can also be specified in physical units. For example, the Orbit above can be initialized as

```
>>> from astropy import units
>>> op= Orbit([8.*units.kpc,22.*units.km/units.s,242*units.km/units.s,0.*units.kpc,22.
↪*units.km/units.s,0.*units.deg])
```

In this case, it is unnecessary to specify the `ro=` and `vo=` scales; when they are not specified, `ro` and `vo` are set to the default values from the *configuration file*. However, if they are specified, then those values rather than the ones from the configuration file are used.

Tip: If you do input and output in physical units, the internal unit conversion specified by `ro=` and `vo=` does not matter!

Inputs to any Orbit method can also be specified with units as an astropy Quantity. galpy's natural units are still used under the hood, as explained in the section on *physical units in galpy*. For example, integration times can be specified in Gyr if you want to integrate for a specific time period.

If for any output you do *not* want the output in physical units, you can specify this by supplying the keyword argument `use_physical=False`.

Initialization from observed coordinates or astropy `SkyCoord`

For orbit integration and characterization of observed stars or clusters, initial conditions can also be specified directly as observed quantities when `radec=True` is set (see further down in this section on how to use an `astropy` SkyCoord instead). In this case a full three-dimensional orbit is initialized as `o= Orbit([RA,Dec,distance, pmRA,pmDec,Vlos],radec=True)` where RA and Dec are expressed in degrees, the distance is expressed in kpc, proper motions are expressed in mas/yr (pmra = pmra' * cos[Dec]), and `Vlos` is the heliocentric line-of-sight velocity given in km/s. The observed epoch is currently assumed to be J2000.00. These observed coordinates are translated to the Galactocentric cylindrical coordinate frame by assuming a Solar motion that can be specified as either `solarmotion='hogg'` (2005ApJ...629..268H), `solarmotion='dehnen'` (1998MNRAS.298..387D) or `solarmotion='schoenrich'` (default; 2010MNRAS.403.1829S). A circular velocity can be specified as `vo=220` in km/s and a value for the distance between the Galactic center and the Sun can be given as `ro=8.0` in kpc (e.g., 2012ApJ...759..131B). While the inputs are given in physical units, the orbit is initialized assuming a circular velocity of one at the distance of the Sun (that is, the orbit's position and velocity is scaled to galpy's *natural* units after converting to the Galactocentric coordinate frame, using the specified `ro=` and `vo=`). The parameters of the coordinate transformations are stored internally, such that they are automatically used for relevant outputs (for example, when the RA of an orbit is requested). An example of all of this is:

```
>>> o= Orbit([20.,30.,2.,-10.,20.,50.],radec=True,ro=8.,vo=220.)
```

However, the internally stored position/velocity vector is

```
>>> print(o.vxvv)
# [1.1480792664061401, 0.1994859759019009, 1.8306295160508093, -0.13064400474040533,␣
↪0.58167185623715167, 0.14066246212987227]
```

and is therefore in *natural* units.

Tip: Initialization using observed coordinates can also use units. So, for example, proper motions can be specified as

2*units.mas/units.yr.

Similarly, one can also initialize orbits from Galactic coordinates using o= Orbit([glon,glat,distance, pmll,pmbb,Vlos],lb=True), where glon and glat are Galactic longitude and latitude expressed in degrees, and the proper motions are again given in mas/yr ((pmll = pmll' * cos[glat]):

```
>>> o= Orbit([20.,30.,2.,-10.,20.,50.],lb=True,ro=8.,vo=220.)
>>> print(o.vxvv)
# [0.79959714332811838, 0.073287283885367677, 0.5286278286083651, 0.12748861331872263,
↪ 0.89074407199364924, 0.0927414387396788]
```

When radec=True or lb=True is set, velocities can also be specified in Galactic coordinates if UVW=True is set. The input is then [RA,Dec,distance,U,V,W], where the velocities are expressed in km/s. U is, as usual, defined as -vR (minus vR).

Finally, orbits can also be initialized using an astropy.coordinates.SkyCoord object. For example, the (ra,dec) example from above can also be initialized as:

```
>>> from astropy.coordinates import SkyCoord
>>> import astropy.units as u
>>> c= SkyCoord(ra=20.*u.deg,dec=30.*u.deg,distance=2.*u.kpc,
                pm_ra_cosdec=-10.*u.mas/u.yr,pm_dec=20.*u.mas/u.yr,
                radial_velocity=50.*u.km/u.s)
>>> o= Orbit(c)
```

In this case, you can still specify the properties of the transformation to Galactocentric coordinates using the standard ro, vo, zo, and solarmotion keywords, or you can use the SkyCoord Galactocentric frame specification and these are propagated to the Orbit instance. For example,

```
>>> from astropy.coordinates import CartesianDifferential
>>> c= SkyCoord(ra=20.*u.deg,dec=30.*u.deg,distance=2.*u.kpc,
                pm_ra_cosdec=-10.*u.mas/u.yr,pm_dec=20.*u.mas/u.yr,
                radial_velocity=50.*u.km/u.s,
                galcen_distance=8.*u.kpc,z_sun=15.*u.pc,
                galcen_v_sun=CartesianDifferential([10.0,235.,7.]*u.km/u.s))
>>> o= Orbit(c)
```

A subtlety here is that the galcen_distance and ro keywords are not interchangeable, because the former is the distance between the Sun and the Galactic center and ro is the projection of this distance onto the Galactic midplane. Another subtlety is that the astropy Galactocentric frame is a right-handed frame, while galpy normally uses a left-handed frame, so the sign of the x component of galcen_v_sun is the opposite of what it would be in solarmotion. Because the Galactocentric frame in astropy does not specify the circular velocity, but only the Sun's velocity, you still need to specify vo to use a non-default circular velocity.

When orbits are initialized using radec=True, lb=True, or using a SkyCoord physical scales ro= and vo= are automatically specified (because they have defaults of ro=8 and vo=220). Therefore, all output quantities will be specified in physical units (see above). If you do want to get outputs in galpy's natural coordinates, you can turn this behavior off by doing

```
>>> o.turn_physical_off()
```

All outputs will then be specified in galpy's natural coordinates.

Initializing multiple objects at once

In all of the examples above, the `Orbit` instance corresponds to a single object, but `Orbit` instances can also contain and analyze multiple objects at once. This makes handling `Orbit` instances highly convenient and also allows for efficient handling of multiple objects. Many of the most computationally-intense methods have been parallelized (orbit integration; analytic eccentricity, zmax, etc. calculation; action-angle calculations) and some other methods switch to more efficient algorithms for larger numbers of objects (e.g., `rguiding`, `rE`, `LcE`).

All of the methods for initializing `Orbit` instances above work for multiple objects. Specifically, the initial conditions can be:

- **Array of arbitrary shape (shape,phasedim); needs to be in internal units (for Quantity input; see 'list' option below or use**
  - in Galactocentric cylindrical coordinates with phase-space coordinates arranged as [R,vR,vT(,z,vz,phi)];
  - [ra,dec,d,mu_ra, mu_dec,vlos] or [l,b,d,mu_l, mu_b, vlos] in [deg,deg,kpc,mas/yr,mas/yr,km/s], or [ra,dec,d,U,V,W] or [l,b,d,U,V,W] in [deg,deg,kpc,km/s,km/s,kms] (ICRS where relevant; mu_ra = mu_ra * cos dec and mu_l = mu_l * cos); use the `radec=`, `lb=`, and `UVW=` keywords as before
- astropy (>v3.0) SkyCoord with arbitrary shape, including velocities;
- **lists of initial conditions, entries can be**
  - individual Orbit instances (of single objects)
  - Quantity arrays arranged as in the first bullet above (so things like [R,vR,vT,z,vz,phi], where R, vR, ... can be arbitrary shape Quantity arrays)
  - list of Quantities (so things like [R1,vR1,..,], where R1, vR1, ... are scalar Quantities
  - None: assumed to be the Sun; if None occurs in a list it is assumed to be the Sun *and all other items in the list are assumed to be [ra,dec,...]*; cannot be combined with Quantity lists
  - lists of scalar phase-space coordinates arranged as in the first bullet above (so things like [R,vR,...] where R,vR are scalars in internal units

Tip: For multiple object initialization using an array or SkyCoord, arbitrary input shapes are supported.

An example initialization with an array is:

```
>>> vxvvs= numpy.array([[1.,0.1,1.,0.1,-0.2,1.5],[0.1,0.3,1.1,-0.3,0.4,2.]])
>>> orbits= Orbit(vxvvs)
>>> print(orbits.R())
# [ 1.   0.1]
```

and with a SkyCoord:

```
>>> numpy.random.seed(1)
>>> nrand= 30
>>> ras= numpy.random.uniform(size=nrand)*360.*u.deg
>>> decs= 90.*(2.*numpy.random.uniform(size=nrand)-1.)*u.deg
>>> dists= numpy.random.uniform(size=nrand)*10.*u.kpc
>>> pmras= 20.*(2.*numpy.random.uniform(size=nrand)-1.)*20.*u.mas/u.yr
>>> pmdecs= 20.*(2.*numpy.random.uniform(size=nrand)-1.)*20.*u.mas/u.yr
>>> vloss= 200.*(2.*numpy.random.uniform(size=nrand)-1.)*u.km/u.s
# Without any custom coordinate-transformation parameters
>>> co= SkyCoord(ra=ras,dec=decs,distance=dists,
```

(continues on next page)

(continued from previous page)

```
                pm_ra_cosdec=pmras,pm_dec=pmdecs,
                radial_velocity=vloss,
                frame='icrs')
>>> orbits= Orbit(co)
>>> print(orbits.ra()[:3],ras[:3])
# [  1.50127922e+02   2.59316818e+02   4.11749371e-02] deg [  1.50127922e+02   2.
↪59316818e+02   4.11749342e-02] deg
```

As before, you can use the `SkyCoord` Galactocentric frame specification here.

`Orbit` instances containing multiple objects act like numpy arrays in many ways, but have some subtly different behaviors for some functions. For example, one can do:

```
>>> print(len(orbits))
# 30
>>> print(orbits.shape)
# (30,)
>>> print(orbits.size)
# 30
>>> orbits.reshape((6,5)) # reshape is done inplace
>>> print(len(orbits))
# 6
>>> print(orbits.shape)
# (6,5)
>>> print(orbits.size)
# 30
>>> sliced_orbits= orbits[:3,1:5] # Extract a subset using numpy's slicing rules
>>> print(sliced_orbits.shape)
# (3,4)
>>> single_orbit= orbits[1,3] # Extract a single object
>>> print(single_orbit.shape)
# ()
```

Slicing creates a new `Orbit` instance. When slicing an `Orbit` instance that has been integrated, the integrated orbit will be transferred to the new instance.

The shape of the `Orbit` instances is retained for all relevant outputs. Continuing on from the previous example (where `orbits` has shape `(6,5)` after we reshaped it), we have:

```
>>> print(orbits.R().shape)
# (6,5)
>>> print(orbits.L().shape)
# (6,5,3)
```

After orbit integration, evaluating `orbits.R(times)` would return an array with shape `(6,5,ntimes)` here.

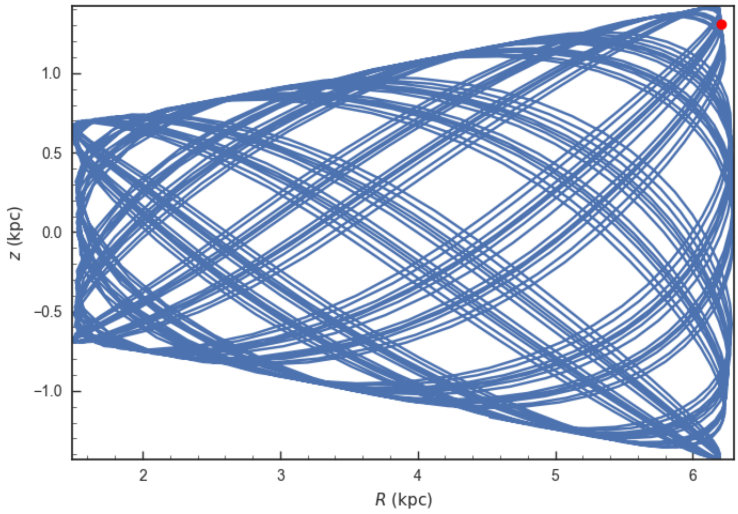
Initialization from an object's name

A convenience method, `Orbit.from_name`, is also available to initialize orbits from the name of an object. For example, for the star Lacaille 8760:

```
>>> o= Orbit.from_name('Lacaille 8760', ro=8., vo=220.)
>>> [o.ra(), o.dec(), o.dist(), o.pmra(), o.pmdec(), o.vlos()]
# [319.31362023999276, -38.86736390000036, 0.003970940656277758, -3258.5529999996584,␣
↪-1145.3959999996205, 20.560000000006063]
```

but this also works for globular clusters, e.g., to obtain Omega Cen's orbit and current location in the Milky Way do:

```
>>> o= Orbit.from_name('Omega Cen')
>>> from galpy.potential import MWPotential2014
>>> ts= numpy.linspace(0.,100.,2001)
>>> o.integrate(ts,MWPotential2014)
>>> o.plot()
>>> plot([o.R()],[o.z()],'ro')
```



We see that Omega Cen is currently close to its maximum distance from both the Galactic center and from the Galactic midplane (note that Omega Cen's phase-space coordinates were updated internally in `galpy` after this plot was made and the orbit is now slightly different).

Similarly, you can do:

```
>>> o= Orbit.from_name('LMC')
>>> [o.ra(), o.dec(), o.dist(), o.pmra(), o.pmdec(), o.vlos()]
# [80.894200000000055, -69.756099999999847, 49.999999999999993, 1.909999999999999, 0.
↪2290000000000037, 262.19999999999993]
```

It is also possible to initialize using multiple names, for example:

```
>>> o= Orbit.from_name(['LMC','SMC'])
>>> print(o.ra(),o.dec(),o.dist())
# [ 80.8942  13.1583] deg [-69.7561 -72.8003] deg [ 50.  60.] kpc
```

The names are stored in the `name` attribute:

```
>>> print(o.name)
# ['LMC', 'SMC']
```

The `Orbit.from_name` method attempts to resolve the name of the object in SIMBAD, and then use the observed coordinates found there to generate an `Orbit` instance. In order to query SIMBAD, `Orbit.from_name` requires the astroquery package to be installed. A small number of objects, mainly Milky Way globular clusters and dwarf satellite galaxies, have their phase-space coordinates stored in a file that is part of galpy and for these objects the values from this file are used rather than querying SIMBAD. `Orbit.from_name` supports tab completion in IPython/Jupyter for this list of objects

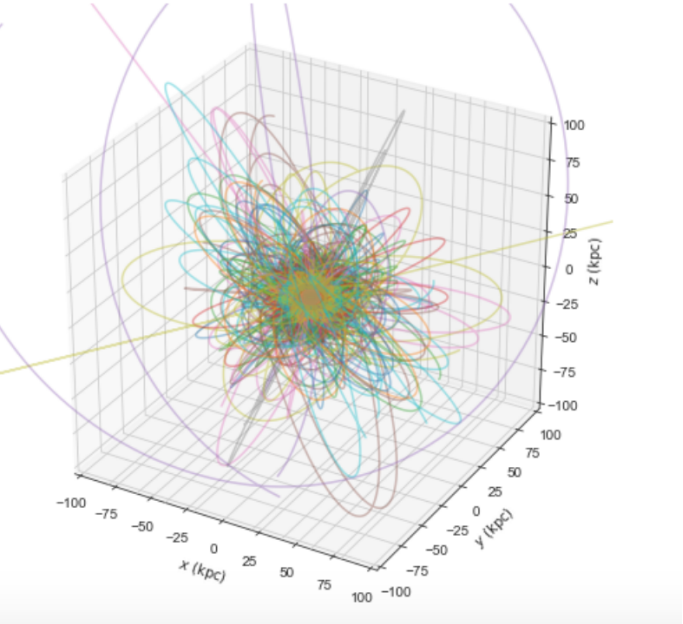
```
In [1]: from galpy.orbit import Orbit

In [2]: Orbit.from_name('
 ro=                47Tuc              BH229              Djorg2
 solarmotion=       Arp2               BH261              E1
 vo=                BH176              Crater             E3
 zo=                BH184              Djorg1             ESO280006
```

The `Orbit.from_name` method also allows you to load some collections of objects in a simple manner. Currently, three collections are supported: 'MW globular clusters', 'MW satellite galaxies', and 'solar system'. Specifying 'MW globular clusters' loads all of the Milky-Way globular clusters with data from Gaia DR2 (using the Vasiliev 2019 catalog):

```
>>> o= Orbit.from_name('MW globular clusters')
>>> print(len(o))
# 150
>>> print(o.name)
# ['NGC5286', 'Terzan12', 'Arp2', 'NGC5024', ... ]
>>> print(o.r())
# [  8.86999028   3.33270877  21.42173795  18.41411889, ...]
```

It is then easy to, for example, integrate the orbits of all Milky-Way globular clusters in `MWPotential2014` and plot them in 3D:

```
>>> ts= numpy.linspace(0.,300.,1001)
>>> o.integrate(ts,MWPotential2014)
>>> o.plot3d(alpha=0.4)
>>> xlim(-100.,100.)
>>> ylim(-100.,100)
>>> gca().set_zlim3d(-100.,100);
```



Similarly, 'MW satellite galaxies' loads all of the Milky-Way satellite galaxies from Fritz et al. (2018):

```
>>> o= Orbit.from_name('MW satellite galaxies')
>>> print(len(o))
# 40
>>> print(o.name)
# ['AquariusII', 'BootesI', 'BootesII', 'CanesVenaticiI', 'CanesVenaticiII', 'Carina',
```

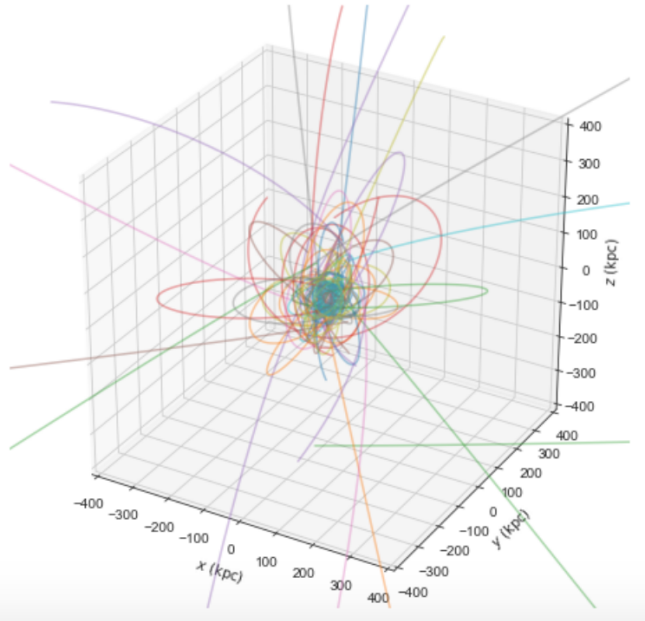
(continues on next page)

(continued from previous page)

```
   'CarinaII', 'CarinaIII', 'ComaBerenices', 'CraterII', 'Draco', 'DracoII',
↪'EridanusII',
   'Fornax', 'GrusI', 'Hercules', 'HorologiumI', 'HydraII', 'HydrusI', 'LeoI', 'LeoII
↪',
   'LeoIV', 'LeoV', 'LMC', 'PhoenixI', 'PiscesII', 'ReticulumII', 'Sgr', 'Sculptor',
   'Segue1', 'Segue2', 'SMC', 'Sextans', 'TriangulumII', 'TucanaII', 'TucanaIII',
   'UrsaMajorI', 'UrsaMajorII', 'UrsaMinor', 'Willman1']
>>> print(o.r())
# [105.11517882  64.02897066  39.72074174 210.66938806 160.60529059
 105.36807259  37.01725917  28.92738515  43.43084545 111.15279646
  79.06498854  23.70062857 364.87901007 141.29780146 116.28700298
 128.81345239  83.47228672 147.90512269  25.68800918 272.93146472
 227.39435987 154.7574384  173.77516411  49.60813235 418.76813979
 181.9540996   32.92030664  19.06561359  84.81046251  27.67609888
  42.13122436  60.28760354  88.86197382  34.58139798  53.79147743
  21.05413655 101.85224099  40.70060809  77.72601419  42.65853777] kpc
```

and we can integrate and plot them in 3D as above:

```
>>> o.plot3d(alpha=0.4)
>>> xlim(-400.,400.)
>>> ylim(-400.,400)
>>> gca().set_zlim3d(-400.,400)
```



Because `MWPotential2014` has a relatively low-mass dark-matter halo, a bunch of the satellites are unbound (to make them bound, you can increase the mass of the halo by, for example, multiplying it by 1.5, as in `MWPotential2014[2]*= 1.5`).

Finally, for illustrative purposes, the solar system is included as a collection as well. The solar system is set up such that the center of what is normally the Galactocentric coordinate frame in `galpy` is now the solar system barycenter and the coordinate frame is a heliocentric one. The solar system data are taken from Bovy et al. (2010) and they represent the positions and planets on April 1, 2009. To load the solar system do:

```
>>> o= Orbit.from_name('solar system')
```

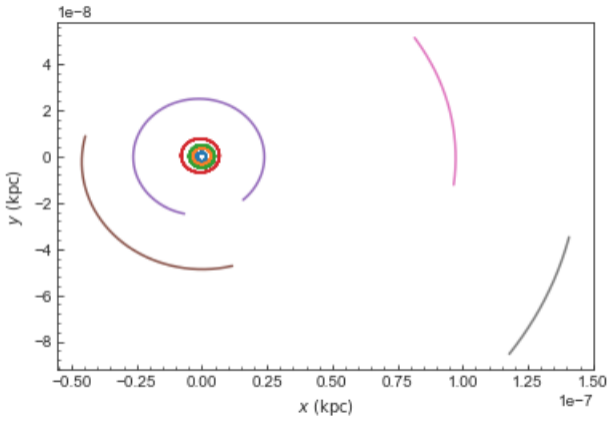
Giving for example:

```
>>> print(o.name)
# ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

You can then, for example, integrate the solar system for 10 years as follows

```
>>> import astropy.units as u
>>> from galpy.potential import KeplerPotential
>>> from galpy.util.conversion import get_physical
>>> kp= KeplerPotential(amp=1.*u.Msun,**get_physical(o)) # Need to use **get_physical␣
↪to get the ro= and vo= parameters, which differ from the default for the solar␣
↪system
>>> ts= numpy.linspace(0.,10.,1001)*u.yr
>>> o.integrate(ts,kp)
>>> o.plot(d1='x',d2='y')
```

which gives



Note that, as usual, physical outputs are in kpc, leading to very small numbers!

Tip: Setting up an `Orbit` instance *without* arguments will return an Orbit instance representing the Sun: `o= Orbit()`. This instance has physical units *turned on by default*, so methods will return outputs in physical units unless you `o.turn_physical_off()`.

Warning: Orbits initialized using `Orbit.from_name` have physical output *turned on by default*, so methods will return outputs in physical units unless you `o.turn_physical_off()`.

1.5.2 Orbit integration

After an orbit is initialized, we can integrate it for a set of times `ts`, given as a numpy array. For example, in a simple logarithmic potential we can do the following

```
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
```

(continues on next page)

(continued from previous page)

```
>>> o= Orbit([1.,0.1,1.1,0.,0.1,0.])
>>> import numpy
>>> ts= numpy.linspace(0,100,10000)
>>> o.integrate(ts,lp)
```

to integrate the orbit from `t=0` to `t=100`, saving the orbit at 10000 instances. In physical units, we can integrate for 10 Gyr as follows

```
>>> from astropy import units
>>> ts= numpy.linspace(0,10.,10000)*units.Gyr
>>> o.integrate(ts,lp)
```

Warning: When the integration times are not specified using a Quantity, they are assumed to be in natural units.

If we initialize the Orbit using a distance scale `ro=` and a velocity scale `vo=`, then Orbit plots and outputs will use physical coordinates (currently, times, positions, and velocities)

```
>>> op= Orbit([1.,0.1,1.1,0.,0.1,0.],ro=8.,vo=220.) #Use Vc=220 km/s at R= 8 kpc as␣
↪the normalization
>>> op.integrate(ts,lp)
```

An `Orbit` instance containing multiple objects can be integrated in the same way and the orbit integration will be performed in parallel on machines with multiple cores. For the fast C integrators (*see below*), this parallelization is done using OpenMP in C and requires one to set the `OMP_NUM_THREADS` environment variable to control the number of cores used. The Python integrators are parallelized in Python and by default also use the `OMP_NUM_THREADS` variable to set the number of cores (but for the Python integrators this can be overwritten). A simple example is

```
>>> vxvvs= numpy.array([[1.,0.1,1.,0.1,-0.2,1.5],[0.1,0.3,1.1,-0.3,0.4,2.]])
>>> orbits= Orbit(vxvvs)
>>> orbits.integrate(ts,lp)
>>> print(orbits.R(ts).shape)
# (2,10000)
>>> print(orbits.R(ts))
# [[ 1.          1.00281576  1.00563403 ...,  1.05694767  1.05608923
#    1.0551804 ]
# [ 0.1         0.18647825  0.27361065 ...,  3.39447863  3.34992543
#    3.30527001]]
```

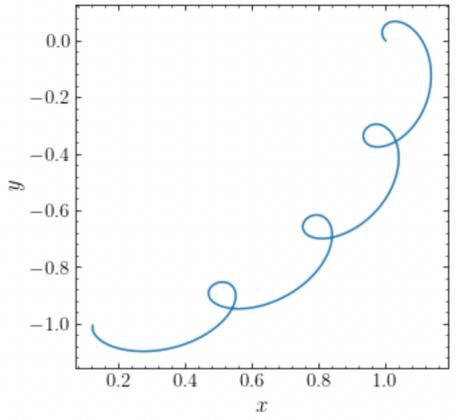
1.5.3 Orbit integration in non-inertial frames

The default assumption in `galpy` is that the frame that an orbit is integrated in is an inertial one. However, `galpy` also supports orbit integration in non-inertial frames that are rotating or whose center is accelerating (or a combination of the two). When a frame is not an inertial frame, fictitious forces such as the centrifugal and Coriolis forces need to be taken into account. `galpy` implements all of the necessary forces as part of the *NonInertialFrameForce* class. objects of this class are instantiated with arbitrary three-dimensional rotation frequencies (and their time derivative) and/or arbitrary three-dimensional acceleration of the origin. The class documentation linked to above provides full mathematical details on the rotation and acceleration of the non-inertial frame.

We can then, for example, integrate the orbit of the Sun in the LSR frame, that is, the frame that is corotating with that of the circular orbit at the location of the Sun. To do this for `MWPotential2014`, do

```
>>> from galpy.potential import MWPotential2014, NonInertialFrameForce
>>> nip= NonInertialFrameForce(Omega=1.) # LSR has Omega=1 in natural units
>>> o= Orbit() # Orbit() is the orbit of the Sun in the inertial frame
>>> o.turn_physical_off() # To use internal units
>>> o= Orbit([o.R(),o.vR(),o.vT()-1.,o.z(),o.vz(),o.phi()]) # Convert to the LSR frame
>>> ts= numpy.linspace(0.,20.,1001)
>>> o.integrate(ts,MWPotential2014+nip)
>>> o.plot(d1='x',d2='y')
```

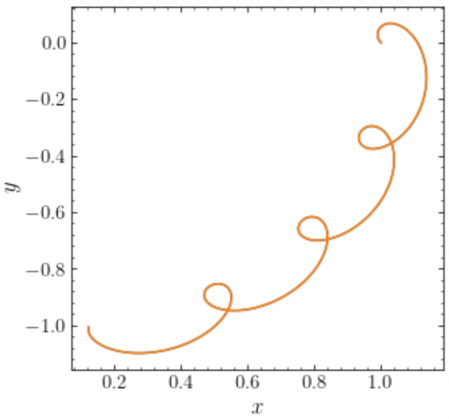
which gives



we can compare this to integrating the orbit in the inertial frame and displaying it in the non-inertial LSR frame as follows:

```
>>> o.plot(d1='x',d2='y') # Repeat plot from above
>>> o= Orbit() # Orbit() is the orbit of the Sun in the inertial frame
>>> o.turn_physical_off() # To use internal units
>>> o.integrate(ts,MWPotential2014)
>>> o.plot(d1='R*cos(phi-t)',d2='R*sin(phi-t)',overplot=True) # Omega = 1, so Omega t␣
↪= t
```

which gives



We can also do all of the above in physical units, in which case the first example above becomes

```
>>> from galpy.potential import MWPotential2014, NonInertialFrameForce
>>> from astropy import units
```

(continues on next page)

(continued from previous page)

```
>>> nip= NonInertialFrameForce(Omega=220./8.*units.km/units.s/units.kpc)
>>> o= Orbit() # Orbit() is the orbit of the Sun in the inertial frame
>>> o= Orbit([o.R(quantity=True),o.vR(quantity=True),
              o.vT(quantity=True)-220.*units.km/units.s,
              o.z(quantity=True),o.vz(quantity=True),
              o.phi(quantity=True)]) # Convert to the LSR frame
>>> ts= numpy.linspace(0.,20.,1001)
>>> o.integrate(ts,MWPotential2014+nip)
>>> o.plot(d1='x',d2='y')
```

We can also provide the `Omega=` frequency as an arbitrary function of time. In this case, the frequency must be returned in internal units and the input time of this function must be in internal units as well (use the routines in *galpy.util.conversion* for converting from physical to internal units; you need to *divide* by these to go from physical to internal). For the example above, this would amount to setting

```
>>> nip= NonInertialFrameForce(Omega=lambda t: 1.,Omegadot=lambda t: 0.)
```

Note that when we supply `Omega` as a function, it is necessary to specify its time derivative as well as `Omegadot` (all again in internal units).

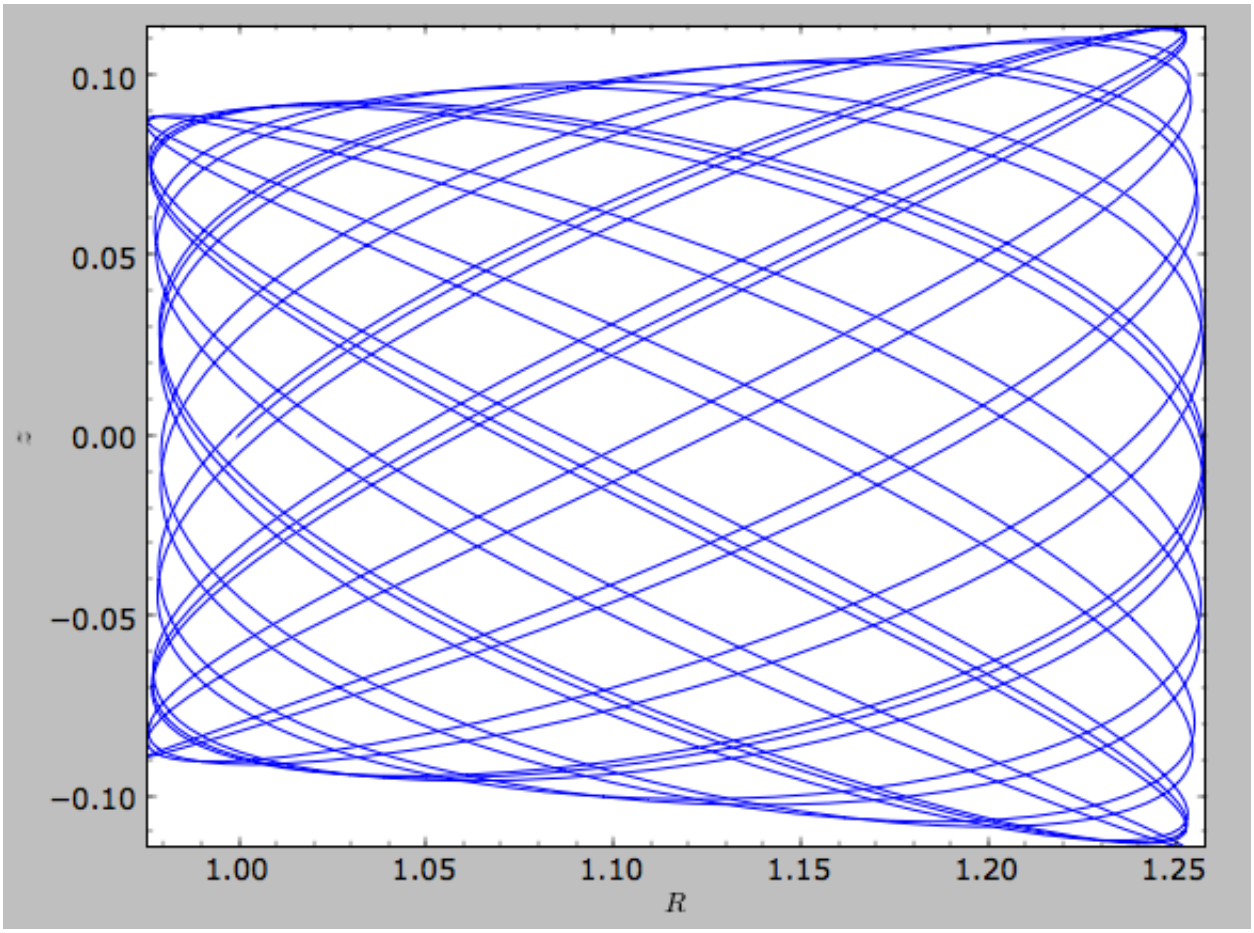
We give an example of having the origin of the non-inertial frame accelerate in the *Example: Including the Milky Way center's barycentric acceleration due to the Large Magellanic Cloud in orbit integrations* section below.

1.5.4 Displaying the orbit

After integrating the orbit, it can be displayed by using the `plot()` function. The quantities that are plotted when `plot()` is called depend on the dimensionality of the orbit: in 3D the (R,z) projection of the orbit is shown; in 2D either (X,Y) is plotted if the azimuth is tracked and (R,vR) is shown otherwise; in 1D (x,vx) is shown. E.g., for the example given above at the start of the *Orbit integration* section above,

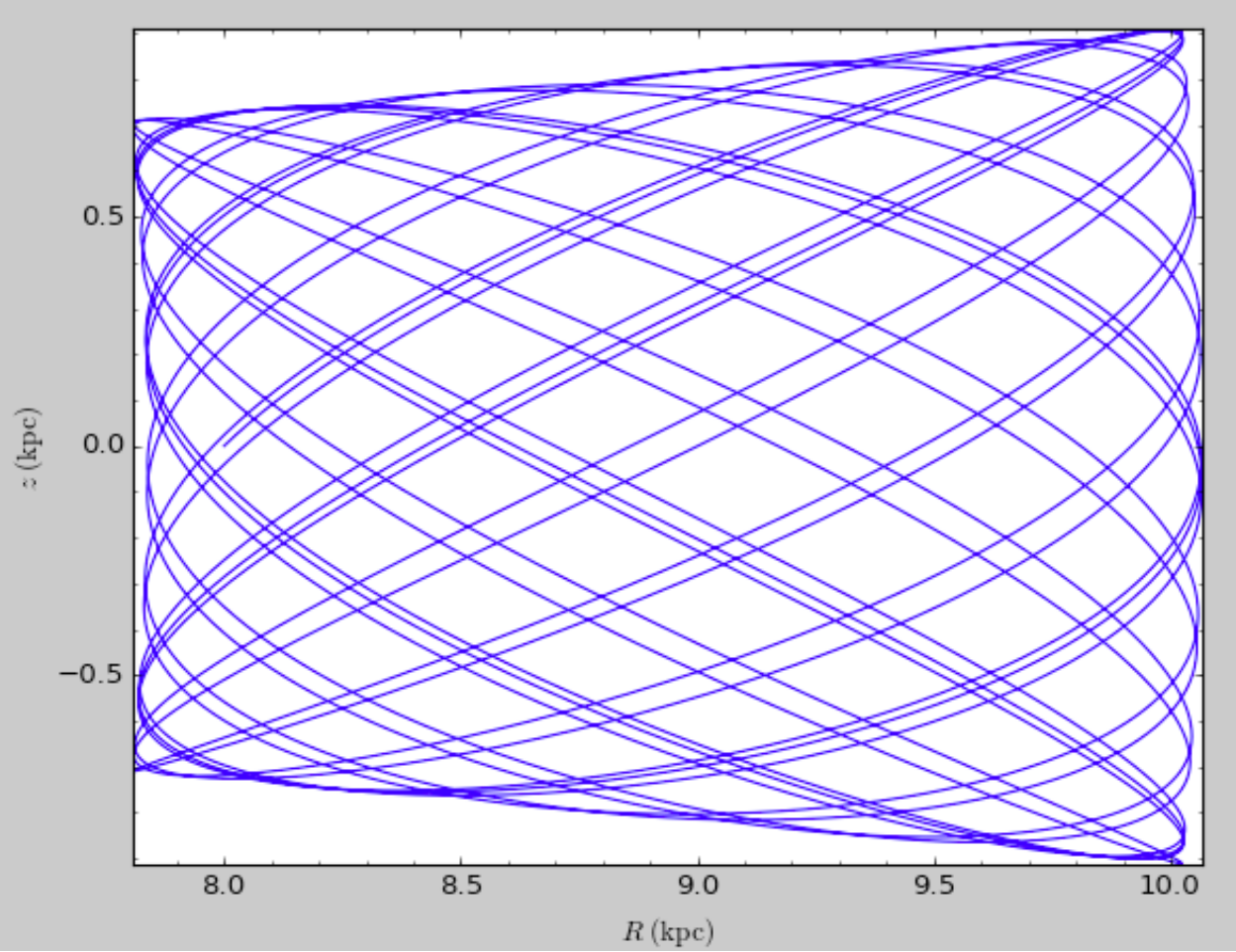
```
>>> o.plot()
```

gives



If we do the same for the Orbit that has physical distance and velocity scales associated with it, we get the following

```
>>> op.plot()
```

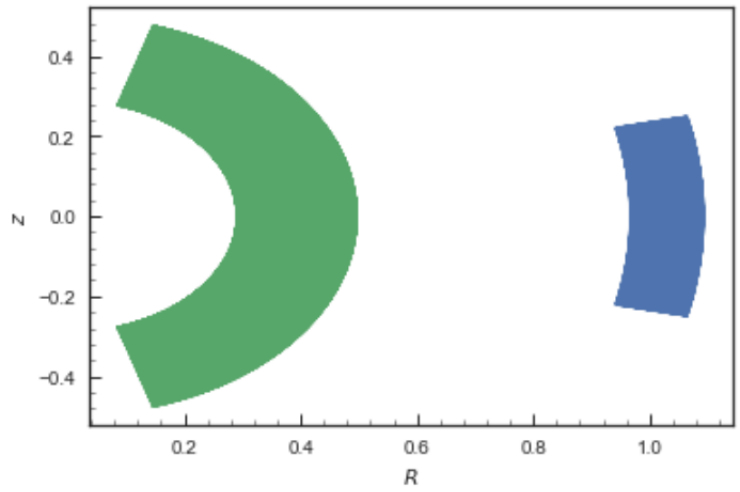


If we call `op.plot(use_physical=False)`, the quantities will be displayed in natural galpy coordinates.

Plotting an `Orbit` instance that consists of multiple objects plots all objects at once, e.g.,

```
>>> orbits.plot()
```

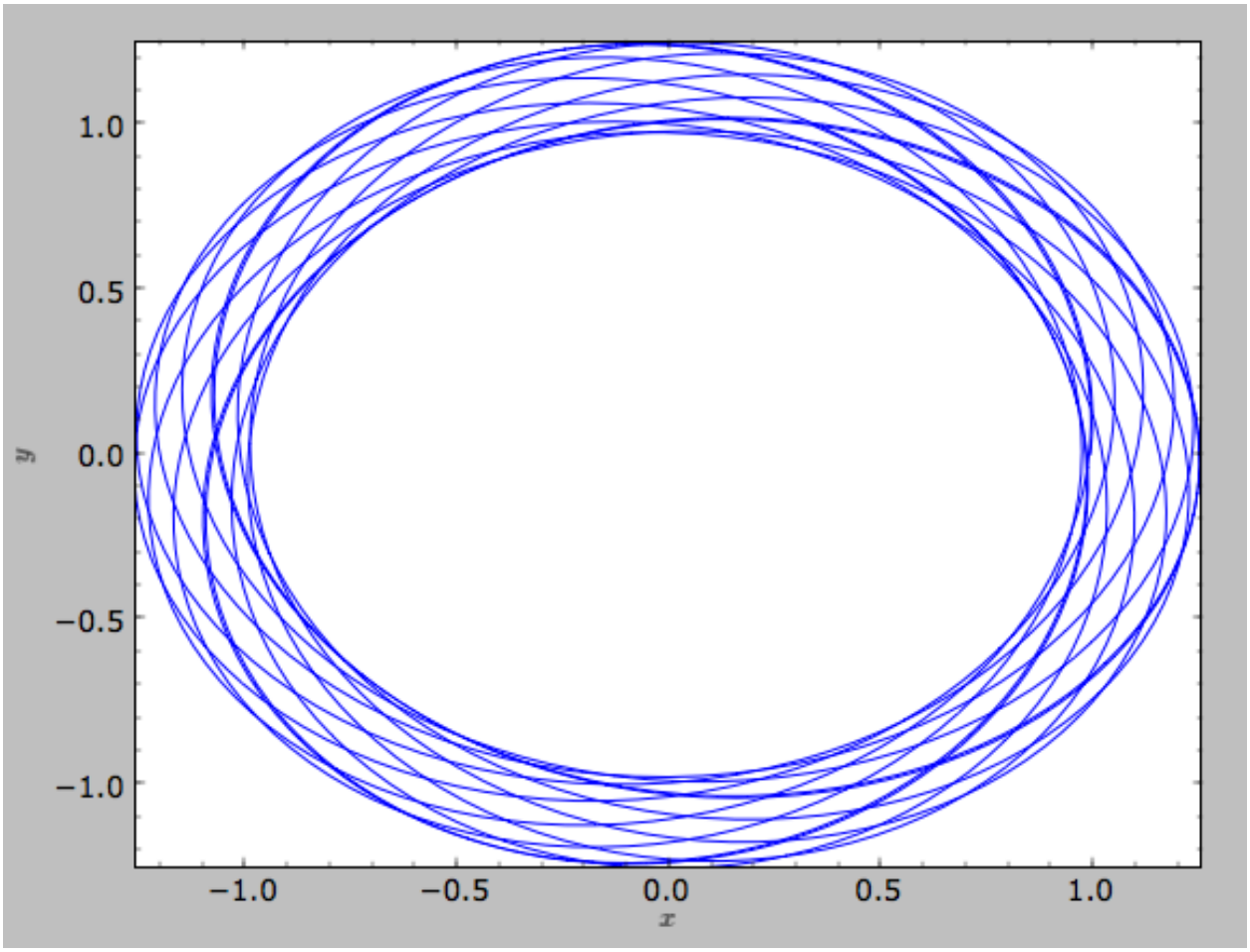
gives



Other projections of the orbit can be displayed by specifying the quantities to plot. E.g.,

```
>>> o.plot(d1='x',d2='y')
```

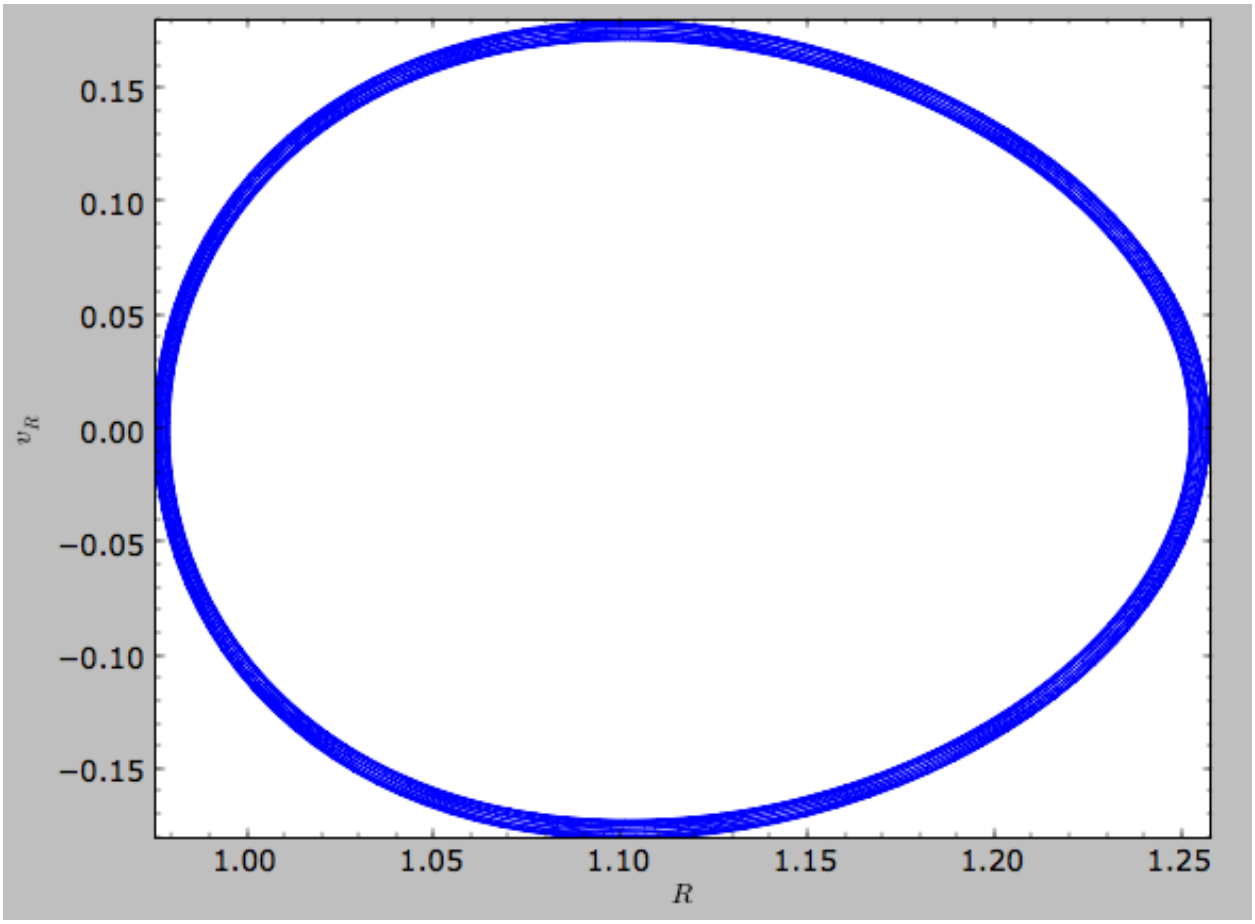
gives the projection onto the plane of the orbit:



while

```
>>> o.plot(d1='R',d2='vR')
```

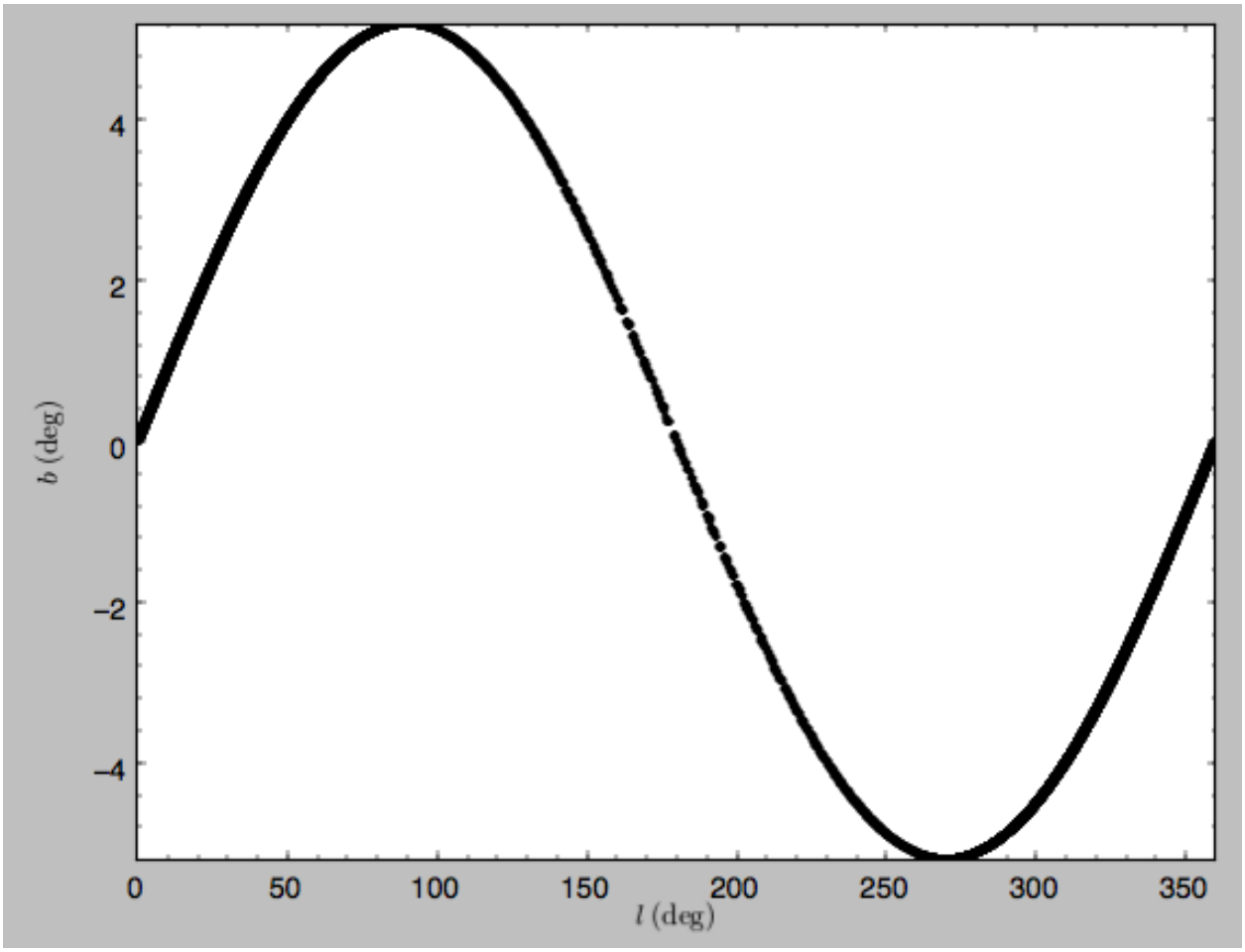
gives the projection onto (R,vR):



We can also plot the orbit in other coordinate systems such as Galactic longitude and latitude

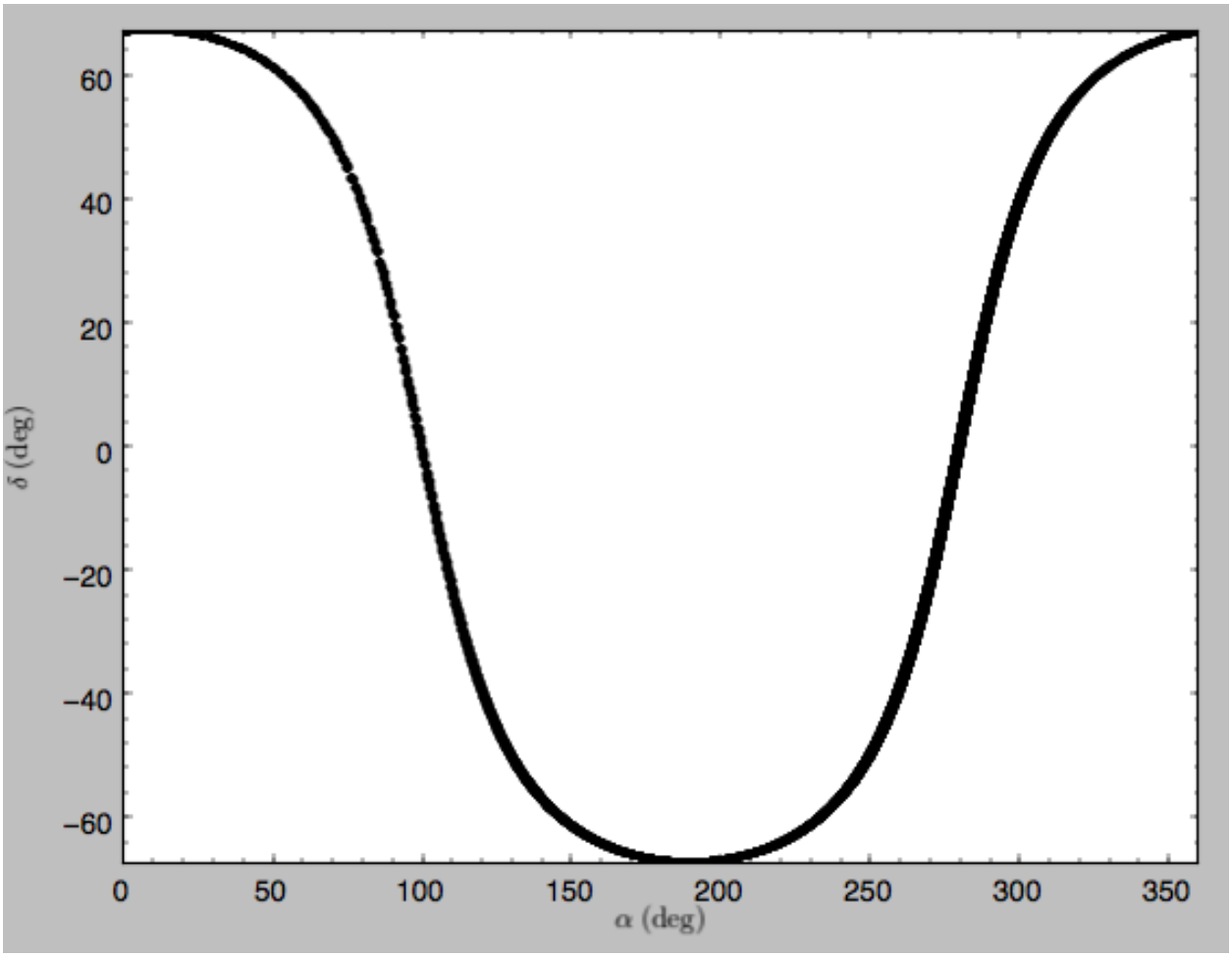
```
>>> o.plot('k.',d1='ll',d2='bb')
```

which shows



or RA and Dec

```
>>> o.plot('k.',d1='ra',d2='dec')
```

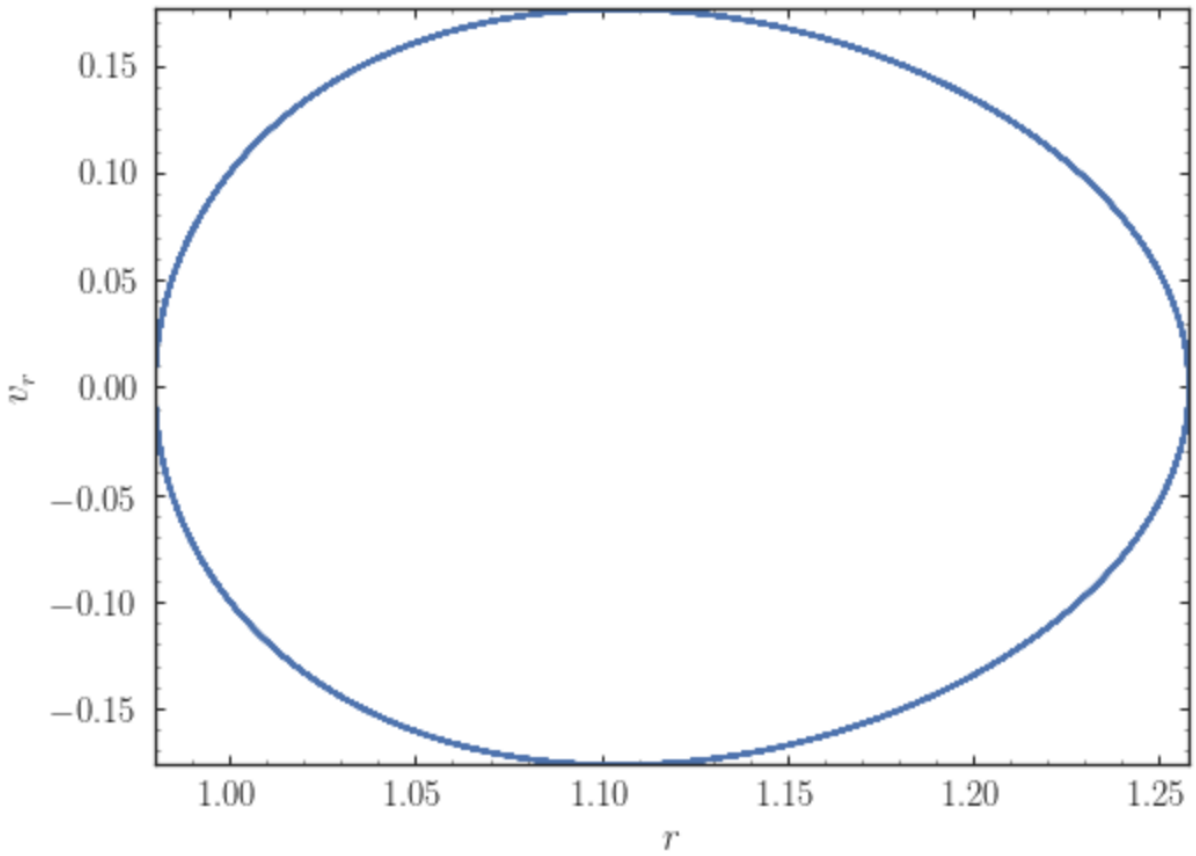


See the documentation of the o.plot function and the o.ra(), o.ll(), etc. functions on how to provide the necessary parameters for the coordinate transformations.

It is also possible to plot quantities computed from the basic Orbit outputs like `o.x()`, `o.r()`, etc. For this to work, the numexpr module needs to be installed; this can be done using `pip` or `conda`. Then you can ask for plots like

```
>>> o.plot(d1='r',d2='vR*R/r+vz*z/r')
```

where `d2=` converts the velocity to spherical coordinates (this is currently not a pre-defined option). This gives the following orbit (which is closed in this projection, because we are using a spherical potential):



You can also do more complex things like

```
>>> o.plot(d1='x',d2='y')
>>> o.plot(d1='R*cos(phi-{:f}*t)'.format(o.Op(quantity=False)),
           d2='R*sin(phi-{:f}*t)'.format(o.Op(quantity=False)),
           overplot=True)
```

which shows the orbit in the regular `(x,y)` frame as well as in a `(x,y)` frame that is rotating at the angular frequency of the orbit. When doing more complex calculations like this, you need to make sure that you are getting the units right: parameters `param` in the expression you provide are directly evaluated as `o.param()`, which depending on how you setup the object may or may not return output in physical units. The expression above is safe, because `o.Op` evaluated like this will be in a consistent unit system with the rest of the expression. Expressions cannot contain astropy Quantities (these cannot be parsed by the parser), which is why `quantity=False` is specified; this is also used internally.

Finally, it is also possible to plot arbitrary functions of time with `Orbit.plot`, by specifying `d1=` or `d2=` as a function. For example, to display the orbital velocity in the spherical radial direction, which we also did with the expression above, you can do the following

```
>>> o.plot(d1='r',
           d2=lambda t: o.vR(t)*o.R(t)/o.r(t)+o.vz(t)*o.z(t)/o.r(t),
           ylabel='v_r')
```

For a function like this, just specifying it as the expression `d2='vR*R/r+vz*z/r'` is much more convenient, but expressions that cannot be parsed automatically could be directly given as a function.

1.5.5 Animating the orbit

Warning: Animating orbits is a new, experimental feature at this time that may be changed in later versions. It has only been tested in a limited fashion. If you are having problems with it, please open an Issue and list all relevant details about your setup (python version, jupyter version, browser, any error message in full). It may also be helpful to check the javascript console for any errors.

In a jupyter notebook or in jupyterlab (jupyterlab versions >= 0.33) you can also create an animation of an orbit *after* you have integrated it. For example, to do this for the `op` orbit from above (but only integrated for 2 Gyr to create a shorter animation as an example here), do

```
>>> op.animate()
```

This will create the following animation

Tip: There is currently no option to save the animation within `galpy`, but you could use screen capture software (for example, QuickTime's Screen Recording feature) to record your screen while the animation is running and save it as a video.

`animate` has options to specify the width and height of the resulting animation, and it can also animate up to three projections of an orbit at the same time. For example, we can look at the orbit in both (x,y) and (R,z) at the same time with

```
>>> op.animate(d1=['x','R'],d2=['y','z'],width=800)
```

which gives

If you want to embed the animation in a webpage, you can obtain the necessary HTML using the `_repr_html_()` function of the IPython.core.display.HTML object returned by `animate`. By default, the HTML includes the entire orbit's data, but `animate` also has an option to store the orbit in a separate `JSON` file that will then be loaded by the output HTML code.

`animate` also works in principle for `Orbit` instances containing multiple objects, but in practice the resulting animation is very slow once more than a few orbits/projections are used.

1.5.6 Orbit characterization

The properties of the orbit can also be found using galpy. For example, we can calculate the peri- and apocenter radii of an orbit, its eccentricity, and the maximal height above the plane of the orbit

```
>>> o.rap(), o.rperi(), o.e(), o.zmax()
# (1.2581455175173673,0.97981663263371377,0.12436710999105324,0.11388132751079502)
```

or for multiple objects at once

```
>>> orbits.rap(), orbits.rperi(), orbits.e(), orbits.zmax()
# (array([ 1.0918143 ,  0.49557137]),
# array([ 0.96779816,  0.29150873]),
# array([ 0.06021334,  0.2592654 ]),
# array([ 0.24734084,  0.47327396]))
```

These four quantities can also be computed using analytical means (exact or approximations depending on the potential) by specifying `analytic=True`

```
>>> o.rap(analytic=True), o.rperi(analytic=True), o.e(analytic=True), o.
↪zmax(analytic=True)
# (1.2581448917376636,0.9798164095995842,0.12436697719989584,0.11390708640305315)
```

or for multiple objects at once (this calculation is done in parallel on systems that support it)

```
>>> orbits.rap(analytic=True), orbits.rperi(analytic=True), orbits.e(analytic=True),␣
↪orbits.zmax(analytic=True)
# (array([ 1.09181433,  0.49557137]),
# array([ 0.96779816,  0.29150873]),
# array([ 0.06021335,  0.2592654 ]),
# array([ 0.24734693,  0.4733304 ]))
```

We can also calculate the energy of the orbit, either in the potential that the orbit was integrated in, or in another potential:

```
>>> o.E(), o.E(pot=mp)
# (0.6150000000000001, -0.67390625000000015)
```

where `mp` is the Miyamoto-Nagai potential of *Introduction: Rotation curves*.

Many other quantities characterizing the orbit can be calculated as well, for example, orbital actions, frequencies, and angles (see *this section*), the guiding-center radius `rguiding`, and the radius `rE` and angular momentum `LcE` of the circular orbit with the same energy as the Orbit instance. See the *Orbit API page* for a full list of quantities that can accessed for any `Orbit` instance.

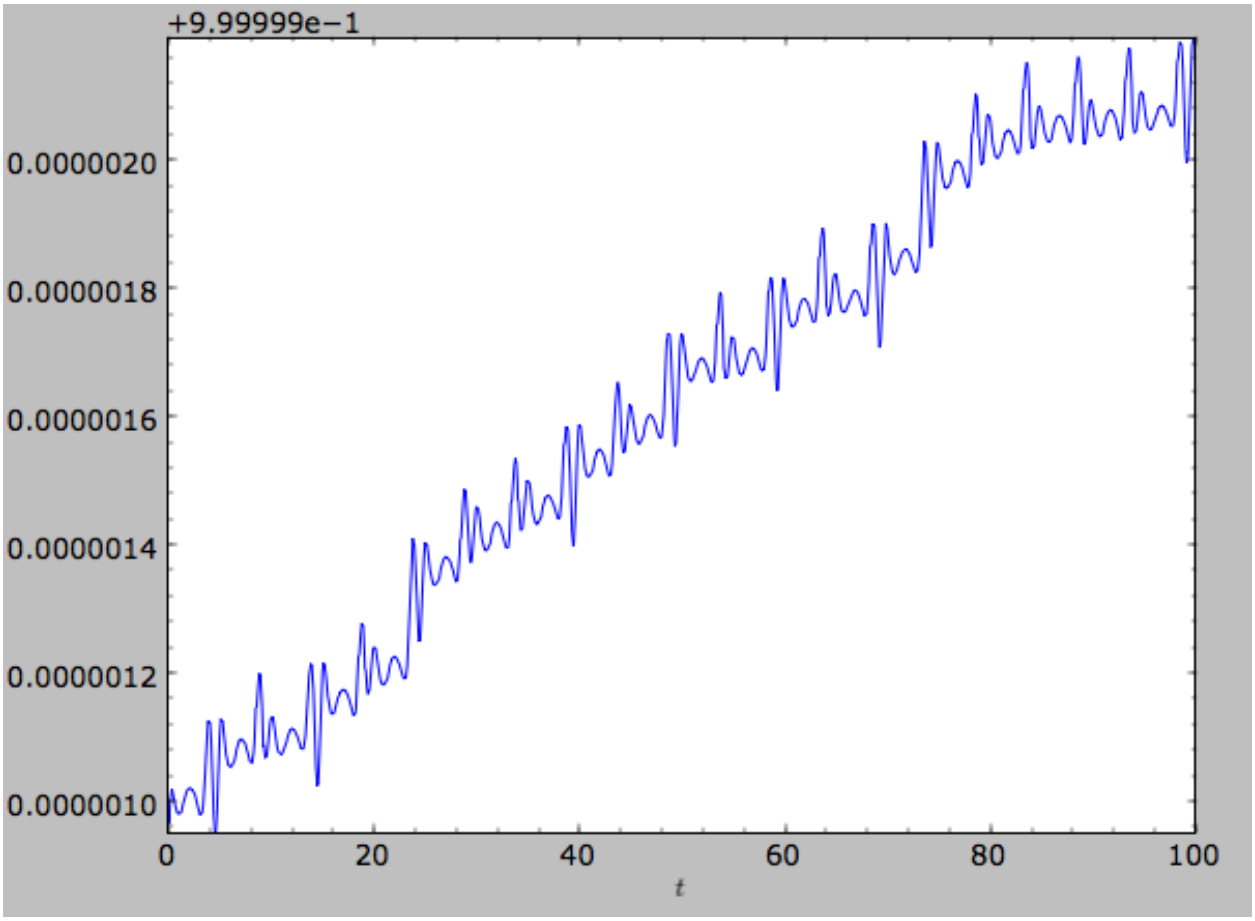
For the Orbit `op` that was initialized above with a distance scale `ro=` and a velocity scale `vo=`, these outputs are all in physical units

```
>>> op.rap(), op.rperi(), op.e(), op.zmax()
# (10.065158988860341,7.8385312810643057,0.12436696983841462,0.91105035688072711) #kpc
>>> op.E(), op.E(pot=mp)
# (29766.000000000004, -32617.062500000007) #(km/s)^2
```

We can also show the energy as a function of time (to check energy conservation)

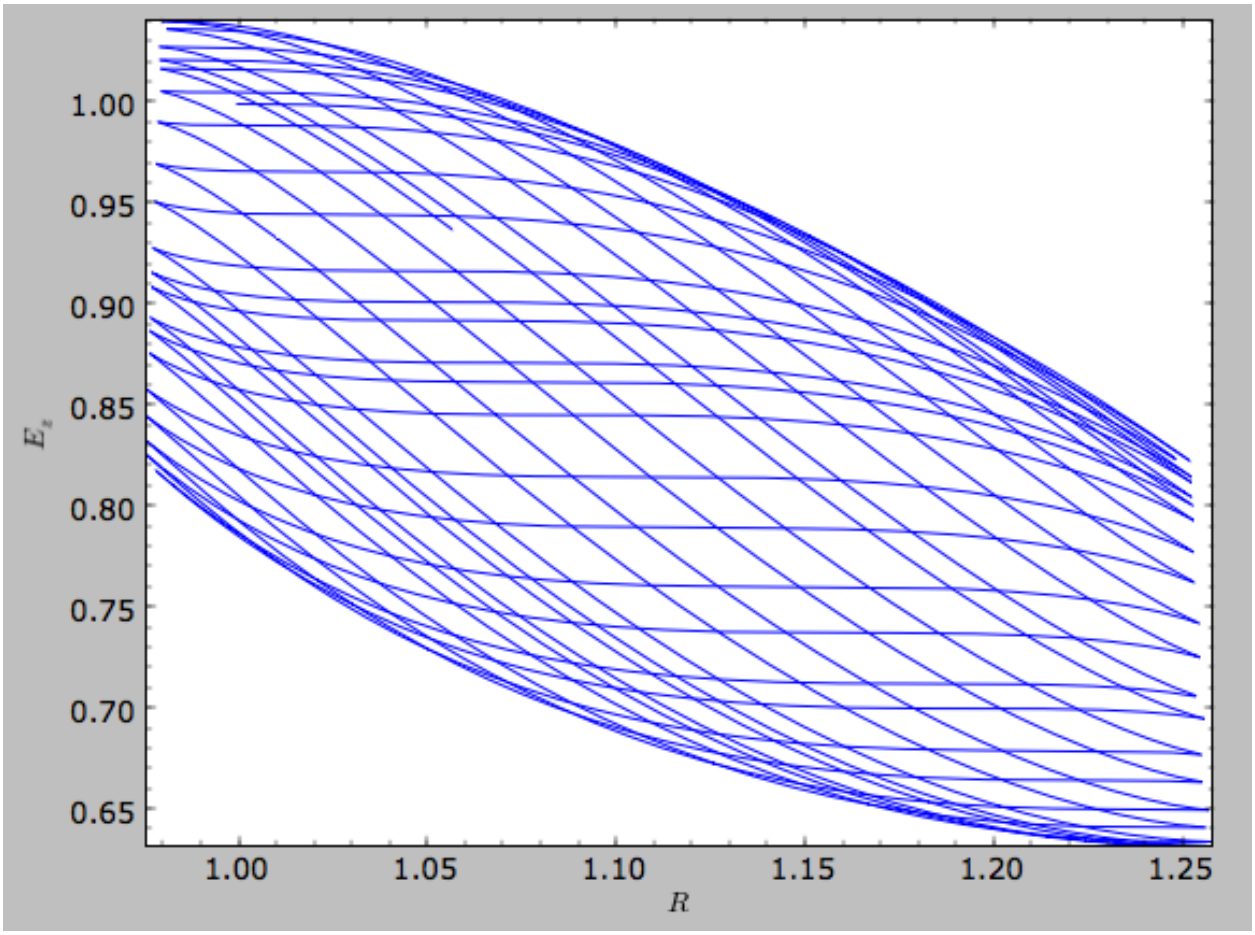
```
>>> o.plotE(normed=True)
```

gives



We can specify another quantity to plot the energy against by specifying `d1=`. We can also show the vertical energy, for example, as a function of R

```
>>> o.plotEz(d1='R',normed=True)
```



1.5.7 Fast orbit characterization

It is also possible to use galpy for the fast estimation of orbit parameters as demonstrated in Mackereth & Bovy (2018) via the Staeckel approximation (originally used by Binney (2012) for the appoximation of actions in axisymmetric potentials), without performing any orbit integration. The method uses the geometry of the orbit tori to estimate the orbit parameters. After initialising an `Orbit` instance, the method is applied by specifying `analytic=True` and selecting `type='staeckel'`.

```
>>> o.e(analytic=True, type='staeckel')
```

if running the above without integrating the orbit, the potential should also be specified in the usual way

```
>>> o.e(analytic=True, type='staeckel', pot=mp)
```

This interface automatically estimates the necessary delta parameter based on the initial condition of the `Orbit` object. (delta is the focal-length parameter of the prolate spheroidal coordinate system used in the approximation, see *the documentation of the actionAngleStaeckel class*).

While this is useful and fast for individual `Orbit` objects, it is likely that users will want to rapidly evaluate the orbit parameters of large numbers of objects. The easiest way to do this is by setting up an `Orbit` instance that contains all objects and call the same functions as above (in this case, the necessary delta parameter will be automatically determined for each object in the instance based on its initial condition)

```
>>> os= Orbit([R, vR, vT, z, vz, phi])
>>> os.e(analytic=True,type='staeckel',pot=mp)
```

In this case, the returned array has the same shape as the input `R,vR,...` arrays.

Rather than automatically estimating delta, you can specify an array for `delta` when calling `os.e` (or `zmax`, `rperi`, and `rap`), for example by first estimating good `delta` parameters as follows:

```
>>> from galpy.actionAngle import estimateDeltaStaeckel
>>> delta= estimateDeltaStaeckel(mp, R, z, no_median=True)
```

where `no_median=True` specifies that the function return the delta parameter at each given point rather than the median of the calculated deltas (which is the default option). Then one can compute the eccentricity etc. using individual delta values as:

```
>>> os.e(analytic=True,type='staeckel',pot=mp,delta=delta)
```

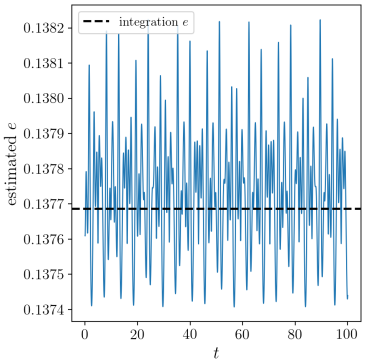
We can test the speed of this method in iPython by finding the parameters at 100000 steps along an orbit in MWPotential2014, like this

```
>>> o= Orbit([1.,0.1,1.1,0.,0.1,0.])
>>> ts = numpy.linspace(0,100,10000)
>>> o.integrate(ts,MWPotential2014)
>>> os= o(ts) # returns an Orbit instance with nt objects, each initialized at the␣
↪position at one of the ts
>>> delta= estimateDeltaStaeckel(MWPotential2014,o.R(ts),o.z(ts),no_median=True)
>>> %timeit -n 10 os.e(analytic=True,pot=MWPotential2014,delta=delta)
# 584 ms ± 8.63 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

you can see that in this potential, each phase space point is calculated in roughly 60µs. further speed-ups can be gained by using the `galpy.actionAngle.actionAngleStaeckelGrid` module, which first calculates the parameters using a grid-based interpolation

```
>>> from galpy.actionAngle import actionAngleStaeckelGrid
>>> R, vR, vT, z, vz, phi = o.getOrbit().T
>>> aASG= actionAngleStaeckelGrid(pot=MWPotential2014,delta=0.4,nE=51,npsi=51,nLz=61,
↪c=True,interpecc=True)
>>> %timeit -n 10 es, zms, rps, ras = aASG.EccZmaxRperiRap(R,vR,vT,z,vz,phi)
# 47.4 ms ± 5.11 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

where `interpecc=True` is required to perform the interpolation of the orbit parameter grid. Looking at how the eccentricity estimation varies along the orbit, and comparing to the calculation using the orbit integration, we see that the estimation good job



1.5.8 Accessing the raw orbit

The value of R, vR, vT, z, vz, x, vx, y, vy, phi, and vphi at any time can be obtained by calling the corresponding function with as argument the time (the same holds for other coordinates ra, dec, pmra, pmdec, vra, vdec, ll, bb, pmll, pmbb, vll, vbb, vlos, dist, helioX, helioY, helioZ, U, V, and W). If no time is given the initial condition is returned, and if a time is requested at which the orbit was not saved spline interpolation is used to return the value. Examples include

```
>>> o.R(1.)
# 1.1545076874679474
>>> o.phi(99.)
# 88.105603035901169
>>> o.ra(2.,obs=[8.,0.,0.],ro=8.)
# array([ 285.76403985])
>>> o.helioX(5.)
# array([ 1.24888927])
>>> o.pmll(10.,obs=[8.,0.,0.,0.,245.,0.],ro=8.,vo=230.)
# array([-6.45263888])
```

For the Orbit op that was initialized above with a distance scale ro= and a velocity scale vo=, the first of these would be

```
>>> op.R(1.)
# 9.2360614837829225 #kpc
```

which we can also access in natural coordinates as

```
>>> op.R(1.,use_physical=False)
# 1.1545076854728653
```

We can also specify a different distance or velocity scale on the fly, e.g.,

```
>>> op.R(1.,ro=4.) #different velocity scale would be vo=
# 4.6180307418914612
```

For Orbit instances that contain multiple objects, the functions above return arrays with the shape of the Orbit.

We can also initialize an Orbit instance using the phase-space position of another Orbit instance evaulated at time t. For example,

```
>>> newOrbit= o(10.)
```

will initialize a new Orbit instance with as initial condition the phase-space position of orbit o at time=10.. If multiple times are given, an Orbit instance with one object for each time will be instantiated (this works even if the original Orbit instance contained multiple objects already).

The whole orbit can also be obtained using the function getOrbit

```
>>> o.getOrbit()
```

which returns a matrix of phase-space points with dimensions [ntimes,nphasedim] or [shape,ntimes,nphasedim] for Orbit instances with multiple objects.

1.5.9 Fast orbit integration and available integrators

The standard orbit integration is done purely in python using standard scipy integrators. When fast orbit integration is needed for batch integration of a large number of orbits, a set of orbit integration routines are written in C that can

be accessed for most potentials, as long as they have C implementations, which can be checked by using the attribute `hasC`

```
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,amp=1.,normalize=1.)
>>> mp.hasC
# True
```

Fast C integrators can be accessed through the `method=` keyword of the `orbit.integrate` method. Currently available integrators are

- rk4_c
- rk6_c
- dopr54_c
- dop853_c

which are Runge-Kutta and Dormand-Prince methods. There are also a number of symplectic integrators available

- leapfrog_c
- symplec4_c
- symplec6_c

The higher order symplectic integrators are described in Yoshida (1993). In pure Python, the available integrators are

- leapfrog
- odeint
- dop853

For most applications I recommend `symplec4_c` or `dop853_c`, which are speedy and reliable. For example, compare

```
>>> o= Orbit([1.,0.1,1.1,0.,0.1])
>>> timeit(o.integrate(ts,mp,method='leapfrog'))
# 1.34 s ± 41.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
>>> timeit(o.integrate(ts,mp,method='leapfrog_c'))
# galpyWarning: Using C implementation to integrate orbits
# 91 ms ± 2.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> timeit(o.integrate(ts,mp,method='symplec4_c'))
# galpyWarning: Using C implementation to integrate orbits
# 9.67 ms ± 48.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> timeit(o.integrate(ts,mp,method='dop853_c'))
# 4.65 ms ± 86.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

If the C extensions are unavailable for some reason, I recommend using the `odeint` pure-Python integrator, as it is the fastest. Using the same example as above

```
>>> o= Orbit([1.,0.1,1.1,0.,0.1])
>>> timeit(o.integrate(ts,mp,method='leapfrog'))
# 2.62 s ± 128 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
>>> timeit(o.integrate(ts,mp,method='odeint'))
# 153 ms ± 2.59 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> timeit(o.integrate(ts,mp,method='dop853'))
# 1.61 s ± 218 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

1.5.10 Integration of the phase-space volume

galpy further supports the integration of the phase-space volume through the method integrate_dxdv, although this is currently only implemented for two-dimensional orbits (planarOrbit). As an example, we can check Liouville's theorem explicitly. We initialize the orbit

```
>>> o= Orbit([1.,0.1,1.1,0.])
```

and then integrate small deviations in each of the four phase-space directions

```
>>> ts= numpy.linspace(0.,28.,1001) #~1 Gyr at the Solar circle
>>> o.integrate_dxdv([1.,0.,0.,0.],ts,mp,method='dopr54_c',rectIn=True,rectOut=True)
>>> dx= o.getOrbit_dxdv()[-1,:] # evolution of dxdv[0] along the orbit
>>> o.integrate_dxdv([0.,1.,0.,0.],ts,mp,method='dopr54_c',rectIn=True,rectOut=True)
>>> dy= o.getOrbit_dxdv()[-1,:]
>>> o.integrate_dxdv([0.,0.,1.,0.],ts,mp,method='dopr54_c',rectIn=True,rectOut=True)
>>> dvx= o.getOrbit_dxdv()[-1,:]
>>> o.integrate_dxdv([0.,0.,0.,1.],ts,mp,method='dopr54_c',rectIn=True,rectOut=True)
>>> dvy= o.getOrbit_dxdv()[-1,:]
```

We can then compute the determinant of the Jacobian of the mapping defined by the orbit integration from time zero to the final time

```
>>> tjac= numpy.linalg.det(numpy.array([dx,dy,dvx,dvy]))
```

This determinant should be equal to one

```
>>> print(tjac)
# 0.999999991189
>>> numpy.fabs(tjac-1.) < 10.**-8.
# True
```

The calls to integrate_dxdv above set the keywords rectIn= and rectOut= to True, as the default input and output uses phase-space volumes defined as (dR,dvR,dvT,dphi) in cylindrical coordinates. When rectIn or rectOut is set, the in- or output is in rectangular coordinates ([x,y,vx,vy] in two dimensions).

Implementing the phase-space integration for three-dimensional FullOrbit instances is straightforward and is part of the longer term development plan for galpy. Let the main developer know if you would like this functionality, or better yet, implement it yourself in a fork of the code and send a pull request!

1.5.11 Example: The eccentricity distribution of the Milky Way's thick disk

A straightforward application of galpy's orbit initialization and integration capabilities is to derive the eccentricity distribution of a set of thick disk stars. We start by downloading the sample of SDSS SEGUE (2009AJ....137.4377Y) thick disk stars compiled by Dierickx et al. (2010arXiv1009.1616D) from CDS at this link. Downloading the table and the ReadMe will allow you to read in the data using astropy.io.ascii like so

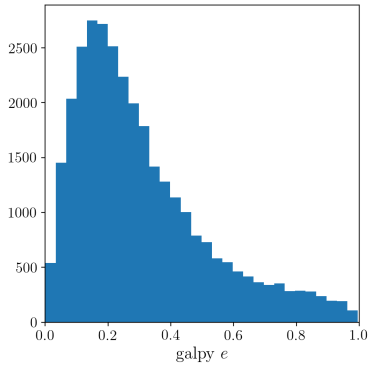
```
>>> from astropy.io import ascii
>>> dierickx = ascii.read('table2.dat', readme='ReadMe')
>>> vxvv = numpy.dstack([dierickx['RAdeg'], dierickx['DEdeg'], dierickx['Dist']/1e3,␣
↪dierickx['pmRA'], dierickx['pmDE'], dierickx['HRV']])[0]
```

After reading in the data (RA,Dec,distance,pmRA,pmDec,vlos; see above) as a vector vxvv with dimensions [6,ndata] we (a) define the potential in which we want to integrate the orbits, and (b) integrate all orbits and compute their eccentricity numerically from the orbit integration and analytically following the *Staeckel approximation method* (the

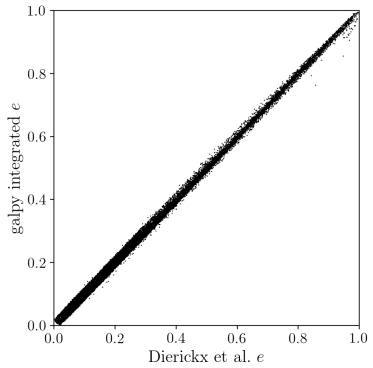
following takes lots of memory; you might want to slice the `orbits` object to a smaller number to run this code faster)

```
>>> ts= np.linspace(0.,20.,10000)
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> orbits= Orbit(vxvv,radec=True,ro=8.,vo=220.,solarmotion='hogg')
>>> e_ana= orbits.e(analytic=True,pot=lp,delta=1e-6)
>>> orbits.integrate(ts,lp)
>>> e_int= orbits.e()
```

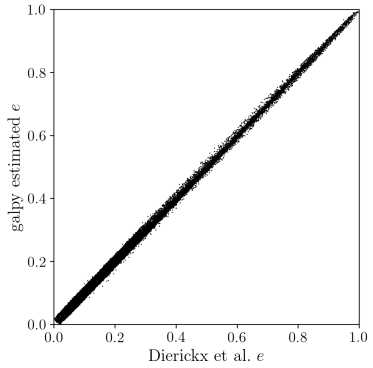
We then find the following eccentricity distribution (from the numerical eccentricities)



The eccentricity calculated by integration in galpy compare well with those calculated by Dierickx et al., except for a few objects

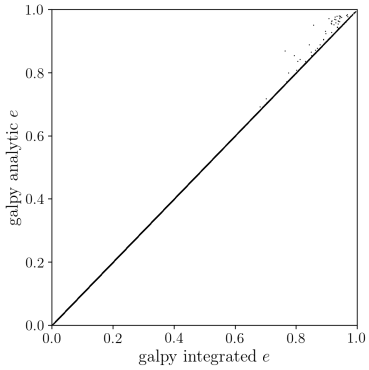


and the analytical estimates are equally as good:



In comparing the analytic and integrated eccentricity estimates - one can see that in this case the estimation is almost

exact, due to the spherical symmetry of the chosen potential:



A script that calculates and plots everything can be downloaded here. To generate the plots just run:

```
python dierickx_eccentricities.py ../path/to/folder
```

specifiying the location you want to put the plots and data.

Alternatively - one can transform the observed coordinates into spherical coordinates and perform the estimations in one batch using the actionAngle interface, which takes considerably less time:

```
>>> from galpy import actionAngle
>>> deltas = actionAngle.estimateDeltaStaeckel(lp, Rphiz[:,0], Rphiz[:,2], no_
↪median=True)
>>> aAS = actionAngleStaeckel(pot=lp, delta=0.)
>>> par = aAS.EccZmaxRperiRap(Rphiz[:,0], vRvTvz[:,0], vRvTvz[:,1], Rphiz[:,2],␣
↪vRvTvz[:,2], Rphiz[:,1], delta=deltas)
```

The above code calculates the parameters in roughly 100ms on a single core.

1.5.12 Example: The orbit of the Large Magellanic Cloud in the presence of dynamical friction

As a further example of what you can do with galpy, we investigate the Large Magellanic Cloud's (LMC) past and future orbit. Because the LMC is a massive satellite of the Milky Way, its orbit is affected by dynamical friction, a frictional force of gravitational origin that occurs when a massive object travels through a sea of low-mass objects (halo stars and dark matter in this case). First we import all the necessary packages:

```
>>> from astropy import units
>>> from galpy.potential import MWPotential2014, ChandrasekharDynamicalFrictionForce
>>> from galpy.orbit import Orbit
```

(also do %pylab inline if running this in a jupyter notebook or turn on the pylab option in ipython for plotting). We can load the current phase-space coordinates for the LMC using the Orbit.from_name function described *above*:

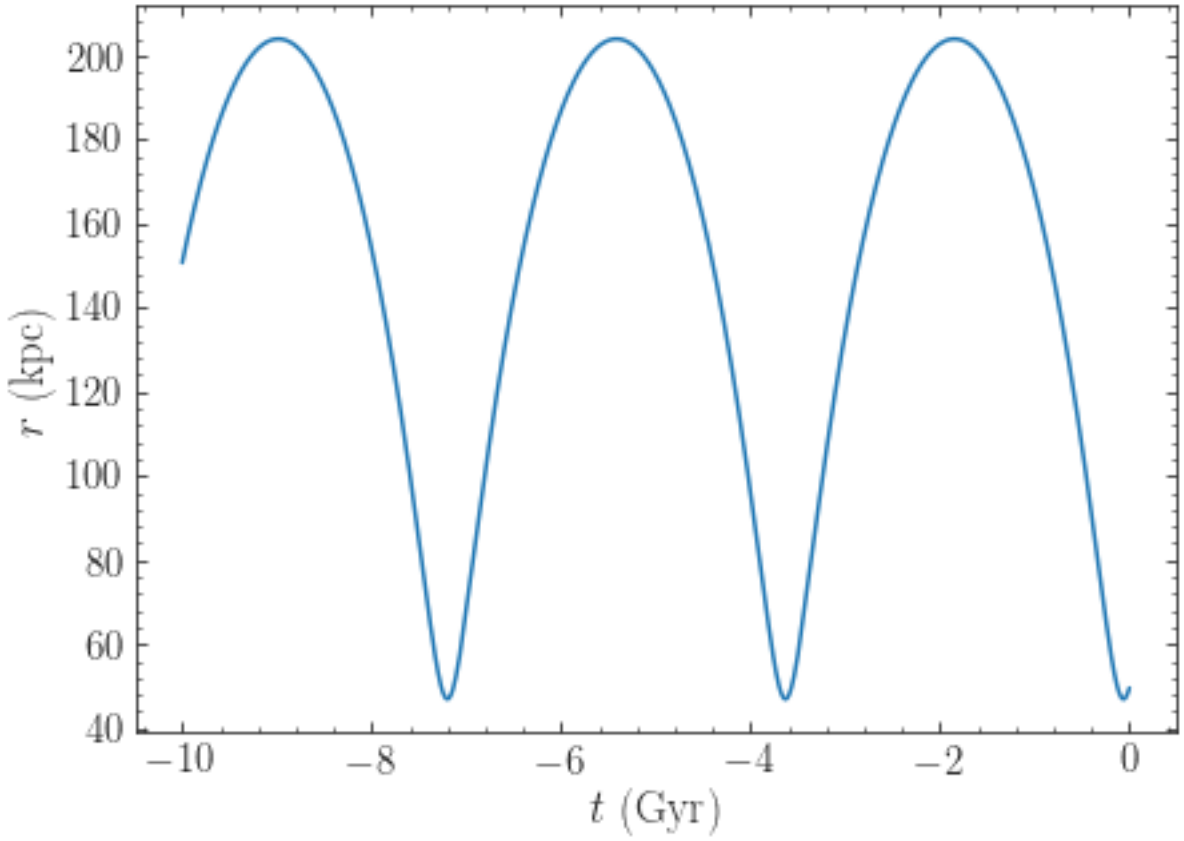
```
>>> o= Orbit.from_name('LMC')
```

We will use MWPotential2014 as our Milky-Way potential model. Because the LMC is in fact unbound in MWPotential2014, we increase the halo mass by 50% to make it bound (this corresponds to a Milky-Way halo mass of $\approx 1.2 \times 10^{12}\,M_\odot$, a not unreasonable value). We can adjust a galpy Potential's amplitude simply by multiplying the potential by a number, so to increase the mass by 50% we do

```
>>> MWPotential2014[2]*= 1.5
```

Let us now integrate the orbit backwards in time for 10 Gyr and plot it:

```
>>> ts= numpy.linspace(0.,-10.,1001)*units.Gyr
>>> o.integrate(ts,MWPotential2014)
>>> o.plot(d1='t',d2='r')
```



We see that the LMC is indeed bound, with an apocenter just over 200 kpc. Now let's add dynamical friction for the LMC, assuming that its mass is $5 \times 10^{10}\,M_\odot$. We setup the dynamical-friction object:

```
>>> cdf= ChandrasekharDynamicalFrictionForce(GMs=5.*10.**10.*units.Msun,rhm=5.*units.
↪kpc,
                                             dens=MWPotential2014)
```

Dynamical friction depends on the velocity distribution of the halo, which is assumed to be an isotropic Gaussian distribution with a radially-dependent velocity dispersion. If the velocity dispersion is not given (like in the example above), it is computed from the spherical Jeans equation. We have set the half-mass radius to 5 kpc for definiteness. We now make a copy of the orbit instance above and integrate it in the potential that includes dynamical friction:

```
>>> odf= o()
>>> odf.integrate(ts,MWPotential2014+cdf)
```

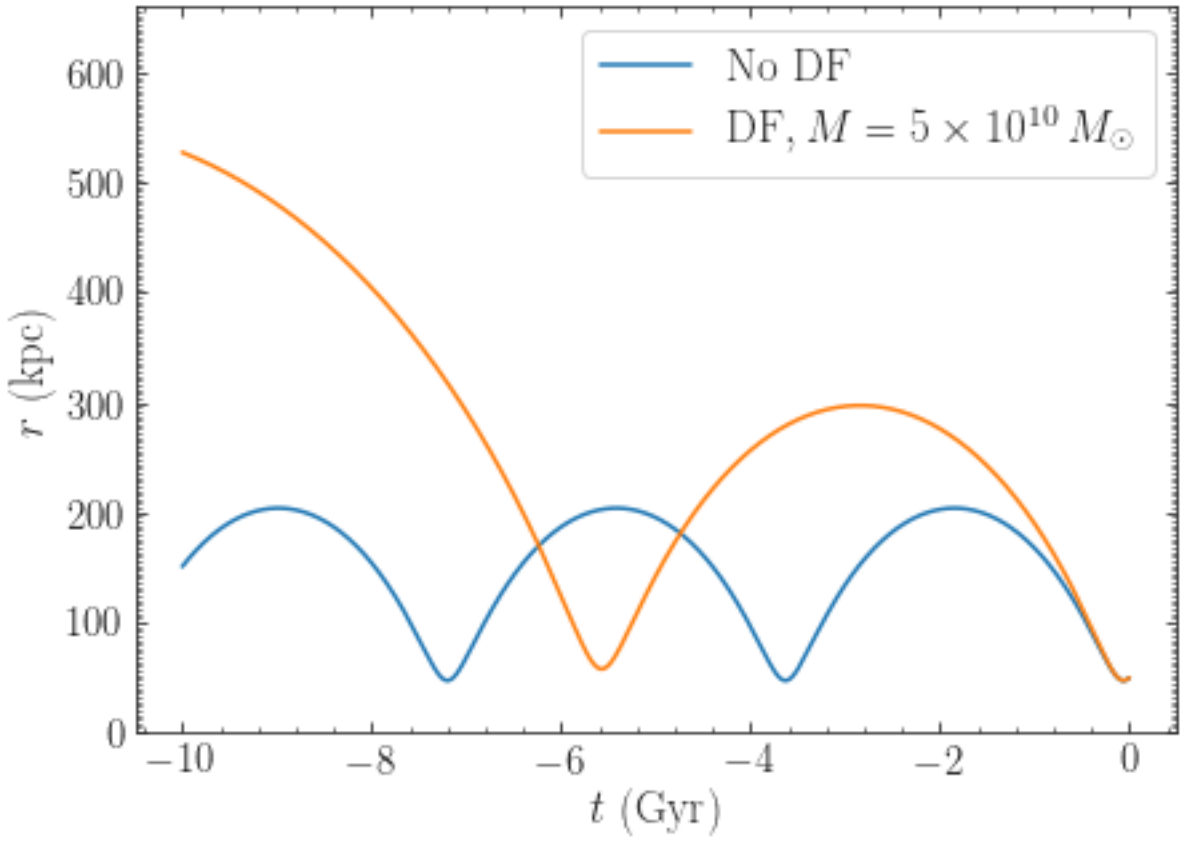
Overlaying the orbits, we can see the difference in the evolution:

```
>>> o.plot(d1='t',d2='r',label=r'$\mathrm{No\ DF}$')
>>> odf.plot(d1='t',d2='r',overplot=True,label=r'$\mathrm{DF}, M=5\times10^{10}\,M_
↪\odot$')
```

(continues on next page)

(continued from previous page)

```
>>> ylim(0.,660.)
>>> legend(fontsize=17.)
```



We see that dynamical friction removes energy from the LMC's orbit, such that its past apocenter is now around 500 kpc rather than 200 kpc! The period of the orbit is therefore also much longer. Clearly, dynamical friction has a big impact on the orbit of the LMC.

Recent observations have suggested that the LMC may be even more massive than what we have assumed so far, with masses over $10^{11}\,M_\odot$ seeming in good agreement with various observations. Let's see how a mass of $10^{11}\,M_\odot$ changes the past orbit of the LMC. We can change the mass of the LMC used in the dynamical-friction calculation as

```
>>> cdf.GMs= 10.**11.*units.Msun
```

This way of changing the mass is preferred over re-initializing the `ChandrasekharDynamicalFrictionForce` object, because it avoids having to solve the Jeans equation again to obtain the velocity dispersion. Then we integrate the orbit and overplot it on the previous results:

```
>>> odf2= o()
>>> odf2.integrate(ts,MWPotential2014+cdf)
```

and

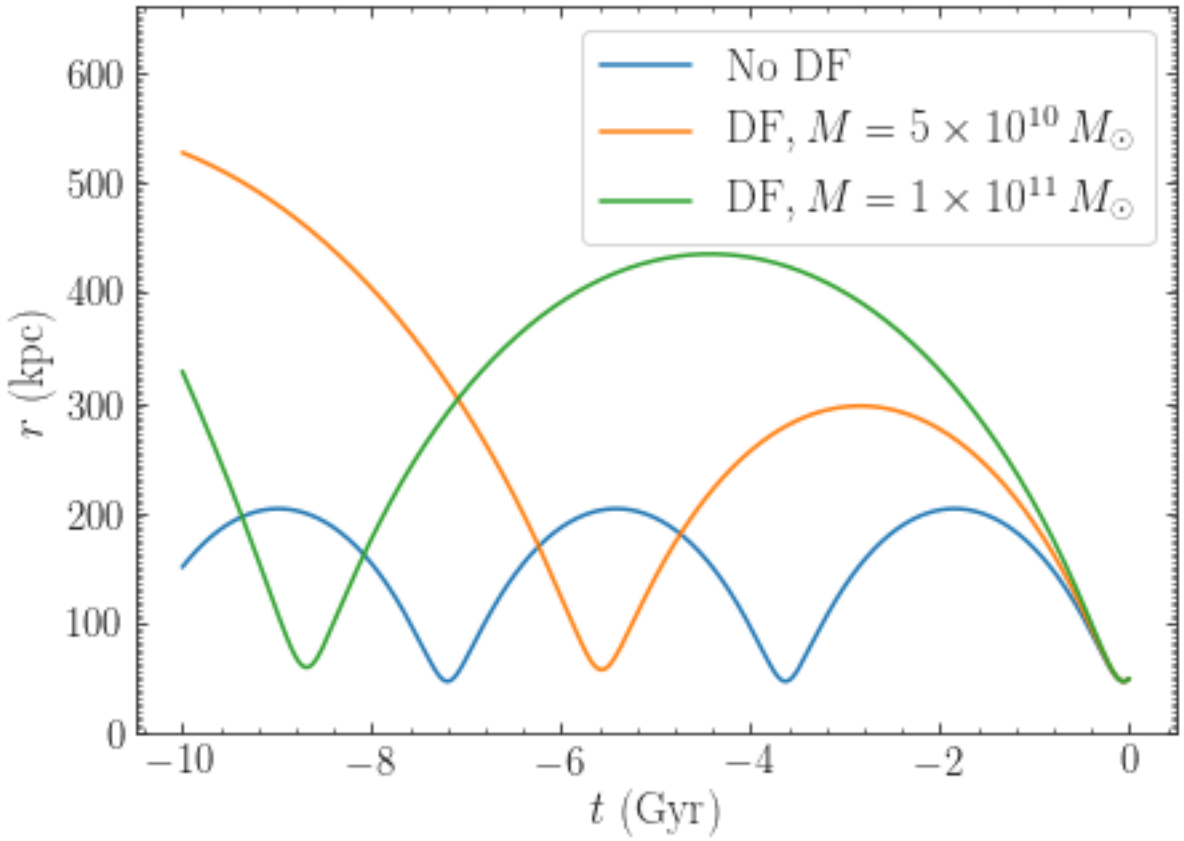
```
>>> o.plot(d1='t',d2='r',label=r'$\mathrm{No\ DF}$')
>>> odf.plot(d1='t',d2='r',overplot=True,label=r'$\mathrm{DF}, M=5\times10^{10}\,M_
↪\odot$')
>>> odf2.plot(d1='t',d2='r',overplot=True,label=r'$\mathrm{DF}, M=1\times10^{11}\,M_
↪\odot$')
```

(continues on next page)

(continued from previous page)

```
>>> ylim(0.,660.)
>>> legend(fontsize=17.)
```

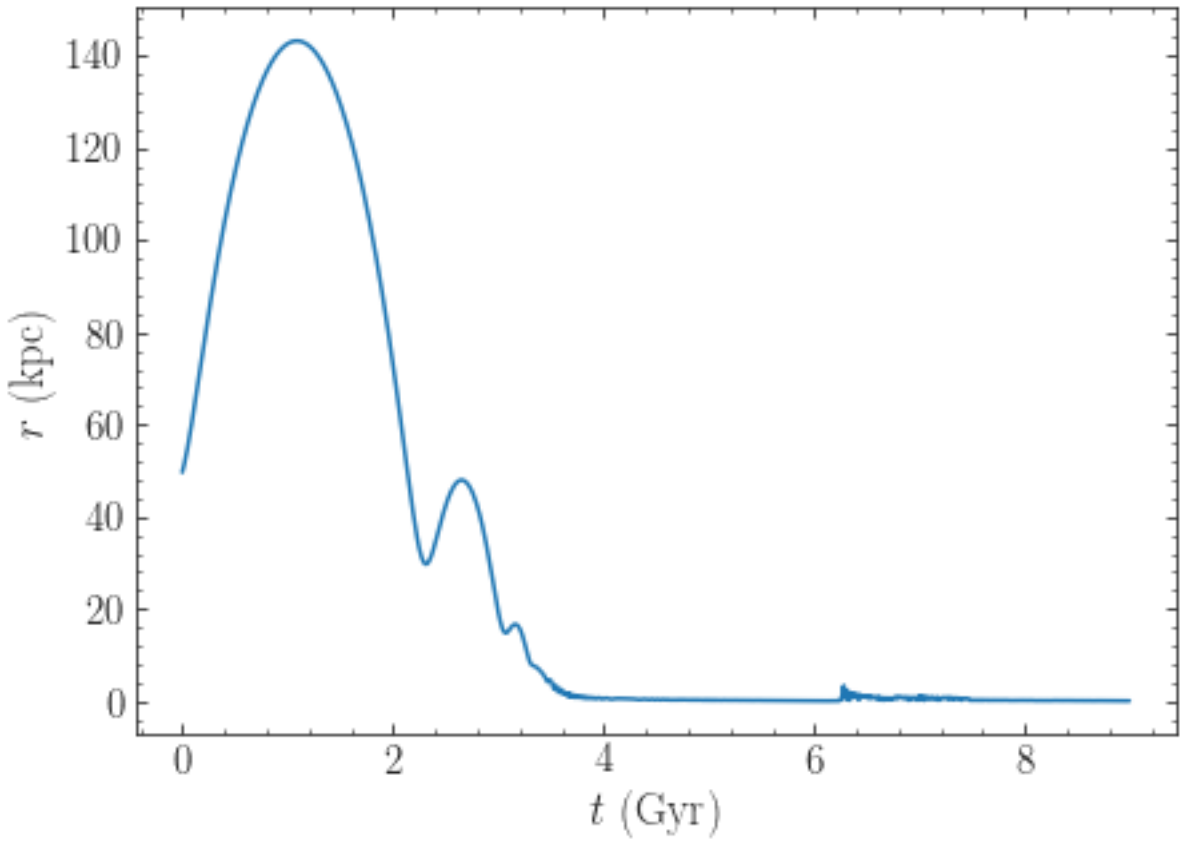
which gives



Now the LMC barely performs a full orbit over the last 10 Gyr.

Finally, let's see what will happen in the future if the LMC is as massive as $10^{11}\,M_\odot$. We simply flip the sign of the integration times to get the future trajectory:

```
>>> odf2.integrate(-ts[-ts < 9*units.Gyr],MWPotential2014+cdf)
>>> odf2.plot(d1='t',d2='r')
```



Because of the large effect of dynamical friction, the LMC will merge with the Milky-Way in about 4 Gyr after a few more pericenter passages. Note that we have not taken any mass-loss into account. Because mass-loss would lead to a smaller dynamical-friction force, this would somewhat increase the merging timescale, but dynamical friction will inevitably lead to the merger of the LMC with the Milky Way.

Warning: When using dynamical friction, if the radius gets very small, the integration sometimes becomes very erroneous, which can lead to a big, unphysical kick (even though we turn off friction at very small radii); this is the reason why we have limited the future integration to 9 Gyr in the example above. When using dynamical friction, inspect the full orbit to make sure to catch whether a merger has happened.

1.5.13 Example: Including the Milky Way center's barycentric acceleration due to the Large Magellanic Cloud in orbit integrations

Observations over the last few decades have revealed that the Large Magellanic Cloud (LMC) is so massive that it pulls the center of the Milky Way towards it to a non-vanishing degree. This implies that the Galactocentric reference frame is not an inertial reference frame and that orbit integrations should take the fictitious forces due to the frame's acceleration into account. In this example, we demonstrate how this can be done in `galpy` in a simplified manner.

To take the Galactocentric frame's acceleration into account, we use the *NonInertialFrameForce*. This `Force` class requires one to input the acceleration of the origin of the non-inertial reference frame, so we first need to determine that. To do this properly, one would have to run some sort of N-body simulation of the LMC's infall into the Milky Way. To avoid doing that, for the purpose of this simple illustration, we will make the following approximation. We will first compute the orbit of the LMC in the Milky Way, assuming that the Milky Way remains at rest (and is thus an inertial frame), and then we will compute the acceleration of the origin induced by the pull from the LMC along this

orbit. Because the effect of the LMC is rather small, this is a decent approximation.

We therefore start by computing the orbit of the LMC in the past like in the previous example in section *Example: The orbit of the Large Magellanic Cloud in the presence of dynamical friction*. We repeat the code here for convenience (we choose again to increase the halo mass in `MWPotential2014` by 50% and we choose the heaviest LMC for this example)

```
>>> from astropy import units
>>> from galpy.potential import MWPotential2014, ChandrasekharDynamicalFrictionForce
>>> from galpy.orbit import Orbit
>>> o= Orbit.from_name('LMC')
>>> MWPotential2014[2]*= 1.5 # Don't run this if you've already run it before in the␣
↪session
>>> cdf= ChandrasekharDynamicalFrictionForce(GMs=10.**11.*units.Msun,rhm=5.*units.kpc,
                                             dens=MWPotential2014)
>>> ts= numpy.linspace(0.,-10.,1001)*units.Gyr
>>> o.integrate(ts,MWPotential2014+cdf)
```

We then define functions giving the acceleration of the origin due to the gravitational pull from the LMC. To do this, we define a `MovingObjectPotential` for the orbiting LMC and then evaluate its forces in rectangular coordinates. We'll model the LMC as a `HernquistPotential`, with mass and half-mass radius consistent with what we used in the dynamical friction above):

```
>>> from galpy.potential import HernquistPotential, MovingObjectPotential
>>> lmcpot= HernquistPotential(amp=2*10.**11.*units.Msun,
                               a=5.*units.kpc/(1.+numpy.sqrt(2.))) #rhm = (1+sqrt(2))␣
↪a
>>> moving_lmcpot= MovingObjectPotential(o,pot=lmcpot)
```

and then we define the functions giving the acceleration of the origin. This is slightly tricky, because there are currently no pre-defined functions giving the force in rectangular coordinates and because evaluating forces at the origin is numerically unstable due to `galpy`'s use of cylindrical coordinates internally. So we will put the origin at a small offset to avoid the numerical issues at the origin and define the rectangular forces ourselves. By placing the origin at $\phi = 0$, the rectangular forces are simple:

```
>>> from galpy.potential import (evaluateRforces, evaluatephiforces,
                                 evaluatezforces)
>>> loc_origin= 1e-4 # Small offset in R to avoid numerical issues
>>> ax= lambda t: evaluateRforces(moving_lmcpot,loc_origin,0.,phi=0.,t=t,
                                  use_physical=False)
>>> ay= lambda t: evaluatephiforces(moving_lmcpot,loc_origin,0.,phi=0.,t=t,
                                    use_physical=False)/loc_origin
>>> az= lambda t: evaluatezforces(moving_lmcpot,loc_origin,0.,phi=0.,t=t,
                                  use_physical=False)
```

Directly using these accelerations in the `NonInertialFrameForce` is very slow (because they have to be evaluated *a lot* during orbit integration), so we build interpolated versions to speed things up:

```
>>> t_intunits= o.time(use_physical=False)[::-1] # need to reverse the order for␣
↪interp
>>> ax4int= [ax(t) for t in t_intunits]
>>> ax_int= lambda t: numpy.interp(t,t_intunits,ax4int)
>>> ay4int= [ay(t) for t in t_intunits]
>>> ay_int= lambda t: numpy.interp(t,t_intunits,ay4int)
>>> az4int= [az(t) for t in t_intunits]
>>> az_int= lambda t: numpy.interp(t,t_intunits,az4int)
```

Note that we use `numpy.interp` here as the interpolation function, because if numba is installed, `galpy` will

automatically use it to try to build fast, C versions of the functions of time in `NonInertialFrameForce`. For this, `numba` must be able to compile the function and it can do this for `numpy.interp` (but not for `scipy` interpolation functions).

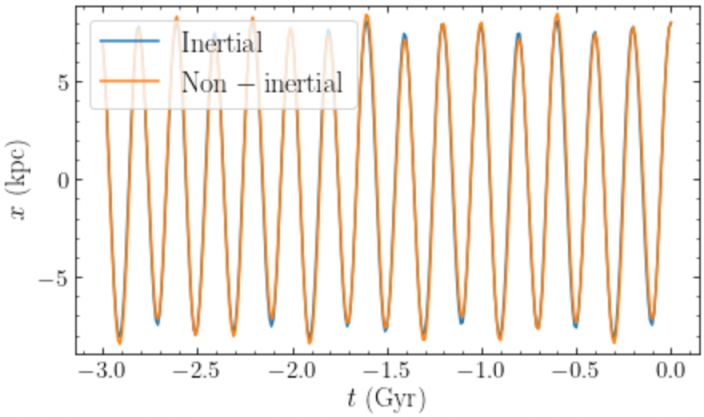
With these functions defined, we can then set up the `NonInertialFrameForce` with this acceleration of the origin

```
>>> nip= NonInertialFrameForce(a0=[ax_int,ay_int,az_int])
```

As an example, let's compute the past orbit of the Sun with and without taking the acceration of the origin into account. We'll look at how the x position changes in time

```
>>> sunts= numpy.linspace(0.,-3.,301)*units.Gyr
>>> osun_inertial= Orbit()
>>> osun_inertial.integrate(sunts,MWPotential2014)
>>> osun_inertial.plotx(label=r'$\mathrm{Inertial}$')
>>> osun_noninertial= Orbit()
>>> osun_noninertial.integrate(sunts,MWPotential2014+nip)
>>> osun_noninertial.plotx(overplot=True,label=r'$\mathrm{Non-inertial}$')
>>> plt.legend(fontsize=18.,loc='upper left',framealpha=0.8)
```

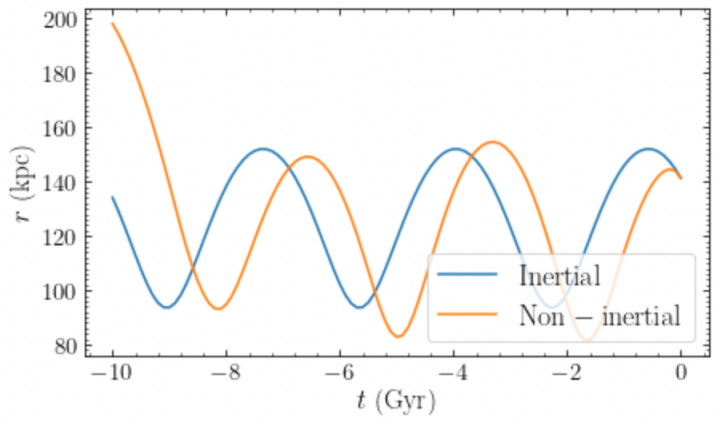
This gives



We see that there is only a small difference. This is because the acceleration of the origin due to the LMC is much smaller than the acceleration felt by the Sun during its orbit from the Milky Way. However, if we look at a dwarf galaxy orbiting far in the halo, we do notice small differences. For example, let's look at the orbit of Fornax over the past 10 Gyr

```
>>> fornaxts= numpy.linspace(0.,-10.,101)*units.Gyr
>>> ofornax_inertial= Orbit.from_name('Fornax')
>>> ofornax_inertial.integrate(fornaxts,MWPotential2014)
>>> ofornax_inertial.plotr(label=r'$\mathrm{Inertial}$')
>>> ofornax_noninertial= Orbit.from_name('Fornax')
>>> ofornax_noninertial.integrate(fornaxts,MWPotential2014+nip)
>>> ofornax_noninertial.plotr(overplot=True,label=r'$\mathrm{Non-inertial}$')
>>> plt.autoscale()
>>> plt.legend(fontsize=18.,loc='lower right',framealpha=0.8)
```

This gives



Now we see that there are significant differences in the past orbit when we take the acceleration of the Galactocentric reference frame into account. The reason that the orbit changes abruptly at -8 Gyr is because the LMC has a previous pericenter passage then in the orbit that we calculated for it, leading to a significant fictitious acceleration force at that time. Whether this is correct is of course highly uncertain.

To check whether the acceleration of the Milky Way's origin that we obtained using the simple approximation above is realistic, we can, for example, compare to the results shown in Figure 10 of Vasiliev et al. (2021). This figure displays the displacement of the Milky Way's origin and its velocity as a function of time, and also the fictitious force induced by the acceleration of the origin (this is minus the acceleration). To compute these quantities for the model above, we simply integrate the acceleration (starting 3 Gyr ago like Vasiliev et al.):

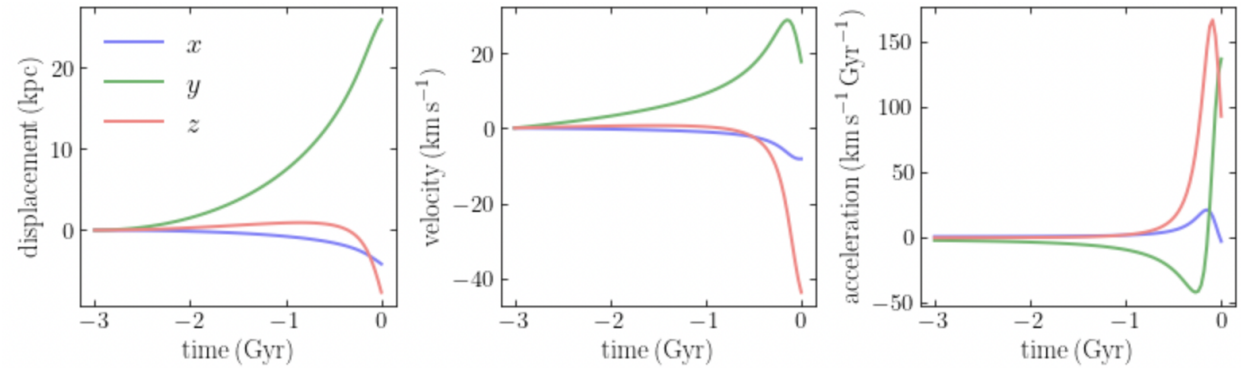
```
>>> from scipy import integrate
>>> from galpy.util import conversion
>>> vo, ro= 220., 8.
>>> int_ts_phys= numpy.linspace(-3.,0.,101)
>>> int_ts= int_ts_phys/conversion.time_in_Gyr(vo,ro)
>>> ax4plot= ax_int(int_ts)
>>> ay4plot= ay_int(int_ts)
>>> az4plot= az_int(int_ts)
>>> vx4plot= integrate.cumulative_trapezoid(ax4plot,x=int_ts,initial=0.)
>>> vy4plot= integrate.cumulative_trapezoid(ay4plot,x=int_ts,initial=0.)
>>> vz4plot= integrate.cumulative_trapezoid(az4plot,x=int_ts,initial=0.)
>>> xx4plot= integrate.cumulative_trapezoid(vx4plot,x=int_ts,initial=0.)
>>> xy4plot= integrate.cumulative_trapezoid(vy4plot,x=int_ts,initial=0.)
>>> xz4plot= integrate.cumulative_trapezoid(vz4plot,x=int_ts,initial=0.)
>>> plt.figure(figsize=(11,3.5))
>>> plt.subplot(1,3,1)
>>> plt.plot(int_ts_phys,xx4plot*ro,color=(0.5,0.5,247./256.),lw=2.,
             label=r'$x$')
>>> plt.plot(int_ts_phys,xy4plot*ro,color=(111./256,180./256,109./256),lw=2.,
             label=r'$y$')
>>> plt.plot(int_ts_phys,xz4plot*ro,color=(239./256,135./256,132./256),lw=2.,
             label=r'$z$')
>>> plt.xlabel(r'$\mathrm{time}\,(\mathrm{Gyr})$')
>>> plt.ylabel(r'$\mathrm{displacement}\,(\mathrm{kpc})$')
>>> plt.legend(frameon=False,fontsize=18.)
>>> plt.subplot(1,3,2)
>>> plt.plot(int_ts_phys,vx4plot*vo,color=(0.5,0.5,247./256.),lw=2.)
>>> plt.plot(int_ts_phys,vy4plot*vo,color=(111./256,180./256,109./256),lw=2.)
>>> plt.plot(int_ts_phys,vz4plot*vo,color=(239./256,135./256,132./256),lw=2.)
>>> plt.xlabel(r'$\mathrm{time}\,(\mathrm{Gyr})$')
>>> plt.ylabel(r'$\mathrm{velocity}\,(\mathrm{km\,s}^{-1})$')
```

(continues on next page)

(continued from previous page)

```
>>> plt.subplot(1,3,3)
>>> plt.plot(int_ts_phys,-ax4plot*conversion.force_in_kmsMyr(vo,ro)*1000.,
             color=(0.5,0.5,247./256.),lw=2.)
>>> plt.plot(int_ts_phys,-ay4plot*conversion.force_in_kmsMyr(vo,ro)*1000.,
             color=(111./256,180./256,109./256),lw=2.)
>>> plt.plot(int_ts_phys,-az4plot*conversion.force_in_kmsMyr(vo,ro)*1000.,
             color=(239./256,135./256,132./256),lw=2.)
>>> plt.xlabel(r'$\mathrm{time}\,(\mathrm{Gyr})$')
>>> plt.ylabel(r'$\mathrm{acceleration}\,(\mathrm{km\,s}^{-1}\,\mathrm{Gyr}^{-1})$')
>>> plt.tight_layout()
```

and we obtain



The main trends and magnitudes in this figure are the same as those in figure 10 of Vasiliev et al., so the acceleration of the origin that we computed here is reasonable. Note that Vasiliev et al. use a different LMC mass and that other aspects of their modeling differ (like the Milky Way's potential), so we don't expect an exact match.

1.6 Two-dimensional disk distribution functions

galpy contains various disk distribution functions, both in two and three dimensions. This section introduces the two-dimensional distribution functions, useful for studying the dynamics of stars that stay relatively close to the mid-plane of a galaxy. The vertical motions of these stars may be approximated as being entirely decoupled from the motion in the plane.

1.6.1 Types of disk distribution functions

galpy contains the following distribution functions for razor-thin disks: `galpy.df.dehnendf`, `galpy.df.shudf`, and `galpy.df.schwarzschilddf`. These are the distribution functions of Dehnen (1999AJ....118.1201D), Shu (1969ApJ...158..505S), and Schwarzschild (the Shu DF in the epicycle approximation, see Binney & Tremaine 2008). Everything shown below for `dehnendf` can also be done for `shudf` and `schwarzschilddf`. The Schwarzschild DF is primarily included as an educational tool; it is *not* a true steady-state DF, because it uses the approximate energy from the epicycle approximation rather than the true energy, and is fully superseded by the Shu DF, which *is* a good steady-state DF.

These disk distribution functions are functions of the energy and the angular momentum alone. They can be evaluated for orbits, or for a given energy and angular momentum. At this point, only power-law rotation curves are supported. A `dehnendf` instance is initialized as follows

```
>>> from galpy.df import dehnendf
>>> dfc= dehnendf(beta=0.)
```

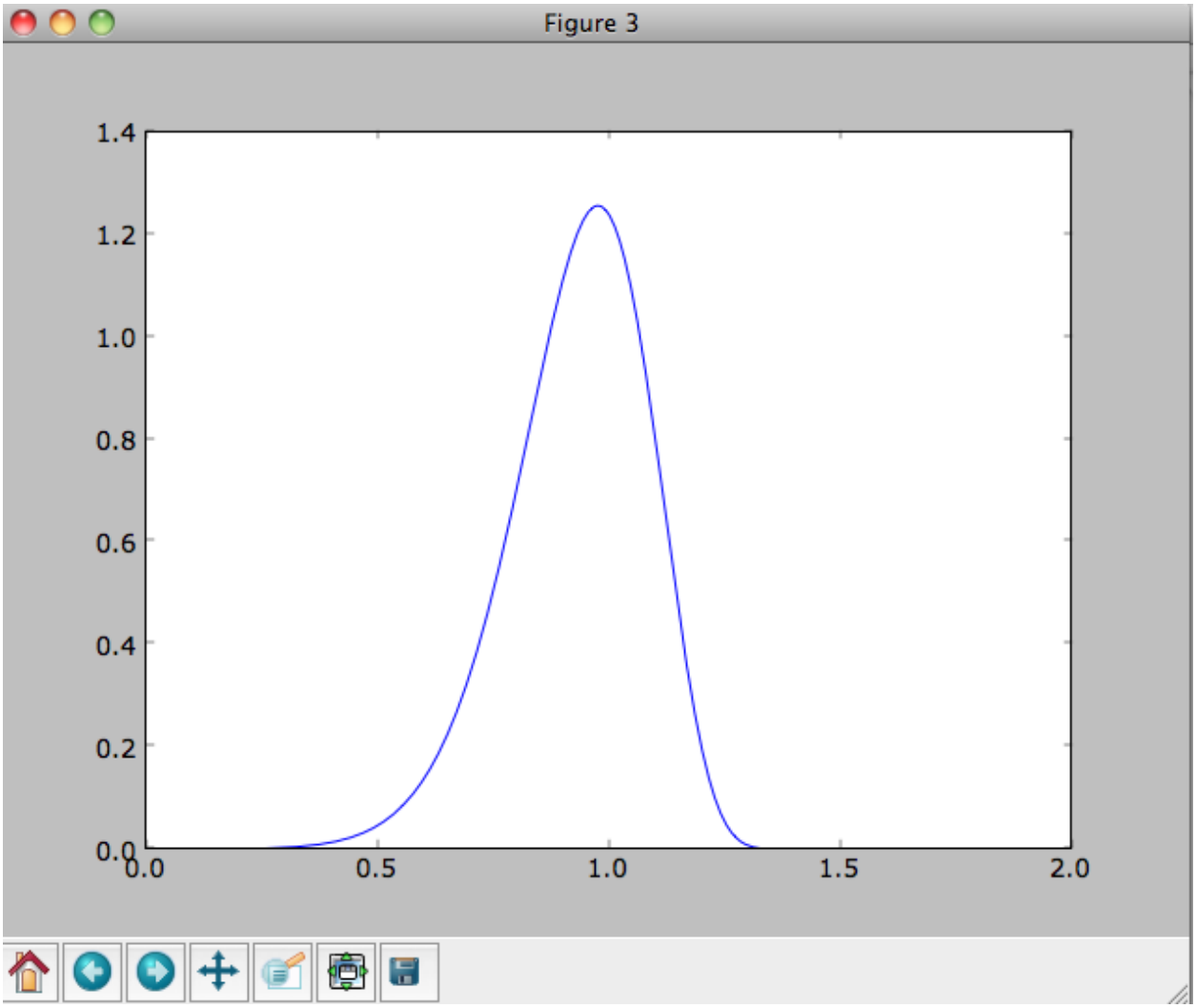
This initializes a `dehnendf` instance based on an exponential surface-mass profile with scale-length 1/3 and an exponential radial-velocity-dispersion profile with scale-length 1 and a value of 0.2 at R=1. Different parameters for these profiles can be provided as an initialization keyword. For example,

```
>>> dfc= dehnendf(beta=0.,profileParams=(1./4.,1.,0.2))
```

initializes the distribution function with a radial scale length of 1/4 instead.

We can show that these distribution functions have an asymmetric drift built-in by evaluating the DF at R=1. We first create a set of orbit-instances and then evaluate the DF at them

```
>>> from galpy.orbit import Orbit
>>> os= [Orbit([1.,0.,1.+-0.9+1.8/1000*ii]) for ii in range(1001)]
>>> dfro= [dfc(o) for o in os]
>>> plot([1.+-0.9+1.8/1000*ii for ii in range(1001)],dfro)
```



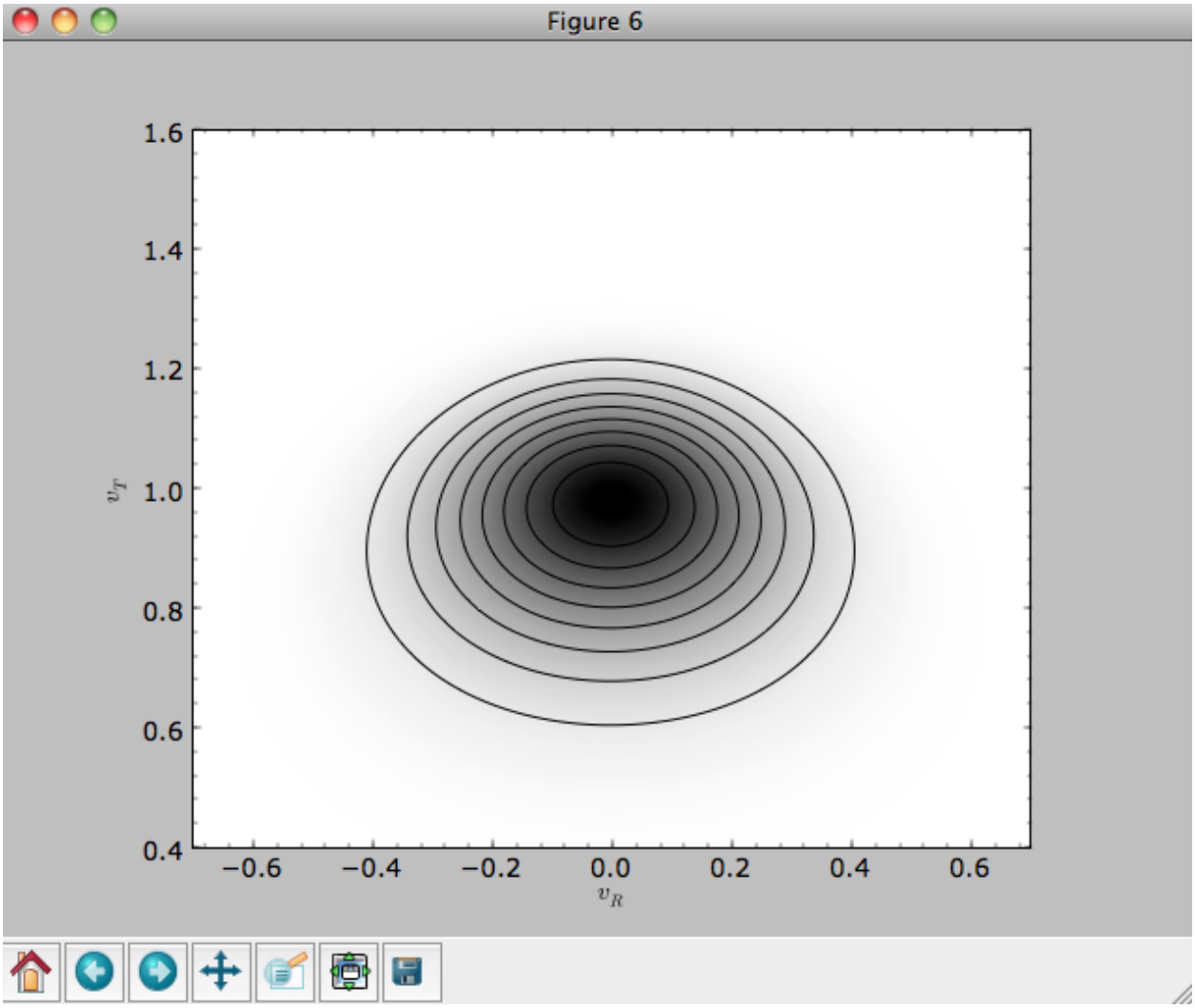
Or we can plot the two-dimensional density at R=1.

```
>>> dfro= [[dfc(Orbit([1.,-0.7+1.4/200*jj,1.-0.6+1.2/200*ii])) for jj in␣
↪range(201)]for ii in range(201)]
>>> dfro= numpy.array(dfro)
```

(continues on next page)

(continued from previous page)

```
>>> from galpy.util.plot import dens2d
>>> dens2d(dfro,origin='lower',cmap='gist_yarg',contours=True,xrange=[-0.7,0.7],
↪yrange=[0.4,1.6],xlabel=r'$v_R$',ylabel=r'$v_T$')
```



1.6.2 Evaluating moments of the DF

galpy can evaluate various moments of the disk distribution functions. For example, we can calculate the mean velocities (for the DF with a scale length of 1/3 above)

```
>>> dfc.meanvT(1.)
# 0.91715276979447324
>>> dfc.meanvR(1.)
# 0.0
```

and the velocity dispersions

```
>>> numpy.sqrt(dfc.sigmaR2(1.))
# 0.19321086259083936
```

(continues on next page)

(continued from previous page)

```
>>> numpy.sqrt(dfc.sigmaT2(1.))
# 0.15084122011271159
```

and their ratio

```
>>> dfc.sigmaR2(1.)/dfc.sigmaT2(1.)
# 1.6406766813028968
```

In the limit of zero velocity dispersion (the epicycle approximation) this ratio should be equal to 2, which we can check as follows

```
>>> dfccold= dehnendf(beta=0.,profileParams=(1./3.,1.,0.02))
>>> dfccold.sigmaR2(1.)/dfccold.sigmaT2(1.)
# 1.9947493895454664
```

We can also calculate higher order moments

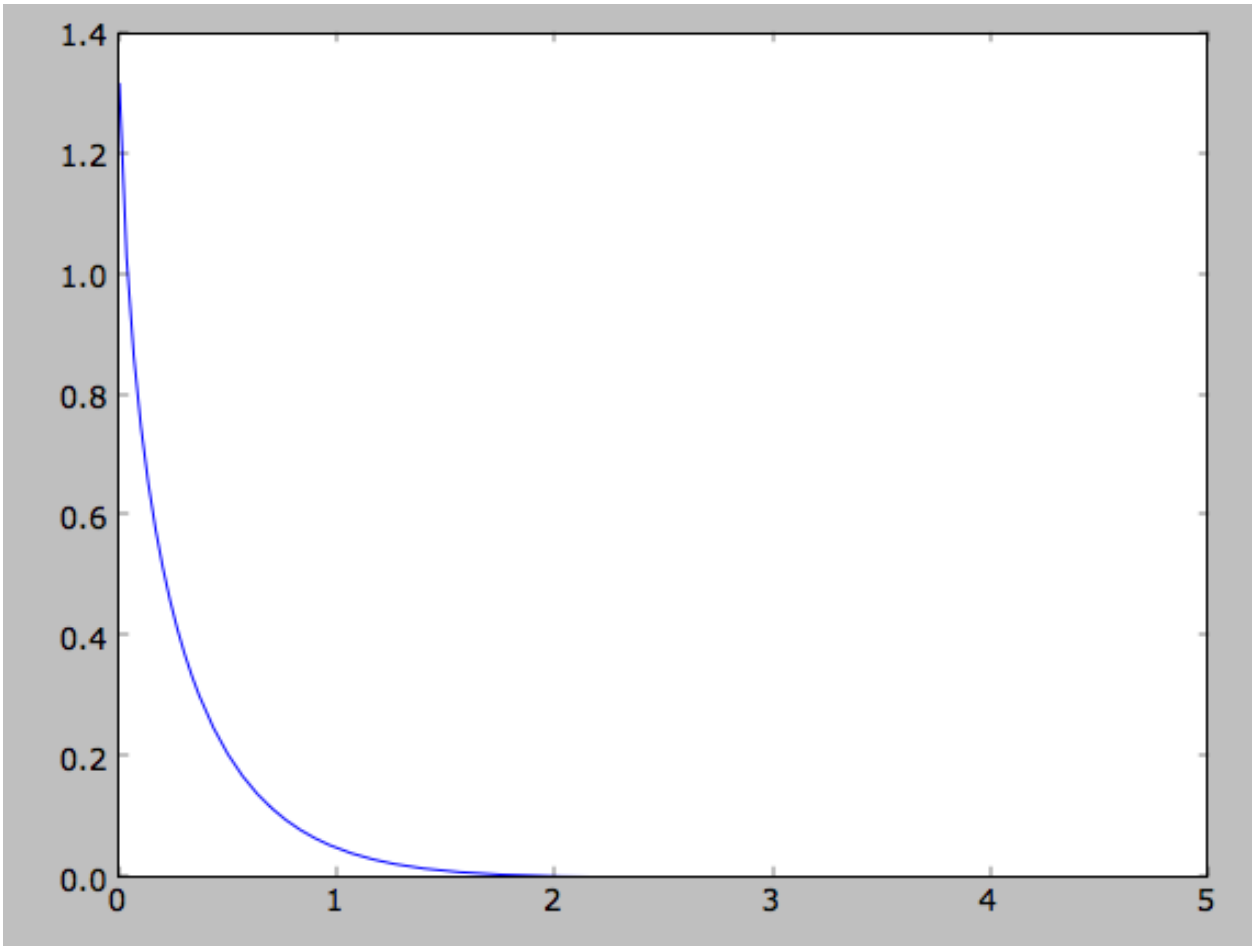
```
>>> dfc.skewvT(1.)
# -0.48617143862047763
>>> dfc.kurtosisvT(1.)
# 0.13338978593181494
>>> dfc.kurtosisvR(1.)
# -0.12159407676394096
```

We already saw above that the velocity dispersion at R=1 is not exactly equal to the input velocity dispersion (0.19321086259083936 vs. 0.2). Similarly, the whole surface-density and velocity-dispersion profiles are not quite equal to the exponential input profiles. We can calculate the resulting surface-mass density profile using `surfacemass`, `sigmaR2`, and `sigma2surfacemass`. The latter calculates the product of the velocity dispersion squared and the surface-mass density. E.g.,

```
>>> dfc.surfacemass(1.)
# 0.050820867101511534
```

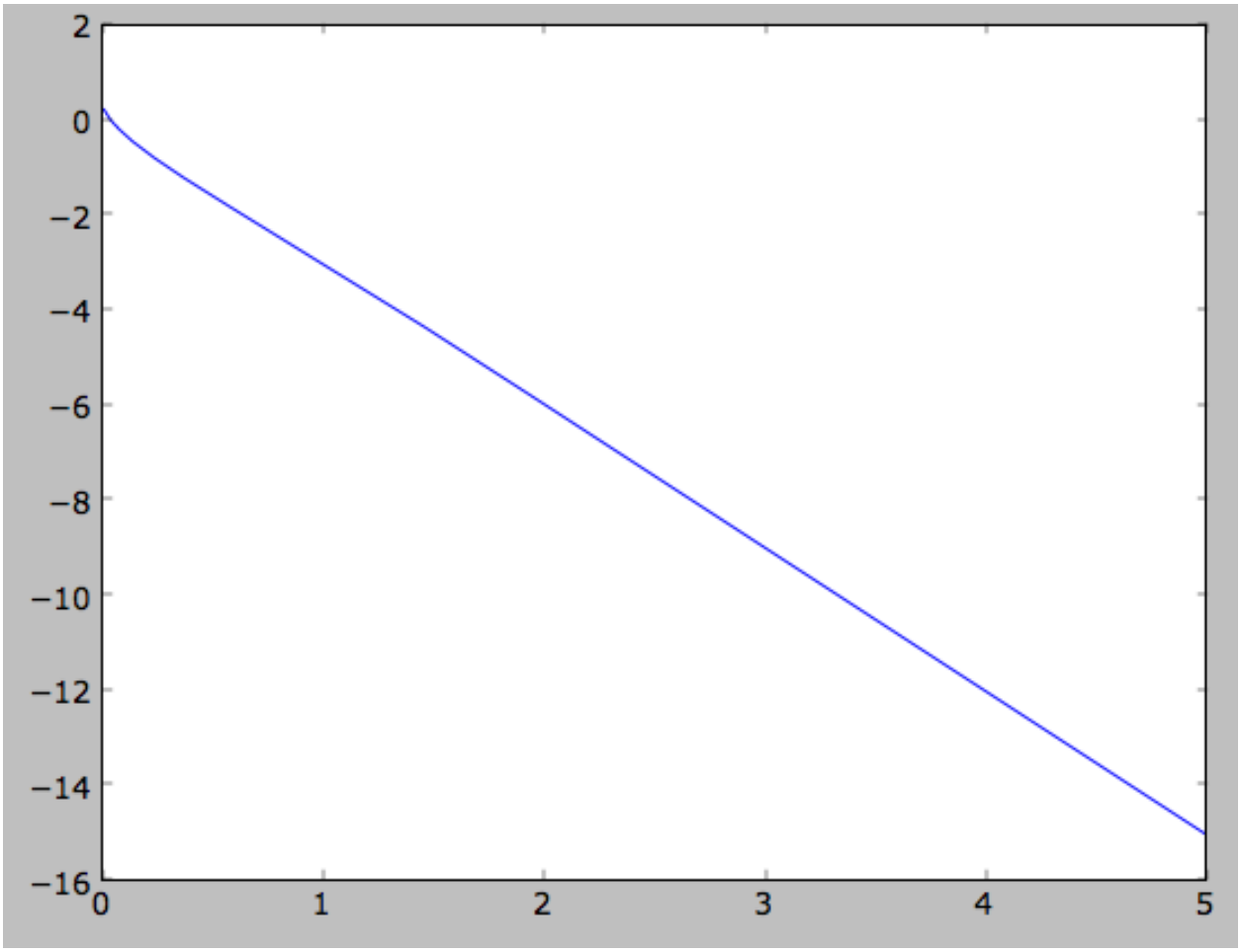
We can plot the surface-mass density as follows

```
>>> Rs= numpy.linspace(0.01,5.,151)
>>> out= [dfc.surfacemass(r) for r in Rs]
>>> plot(Rs, out)
```



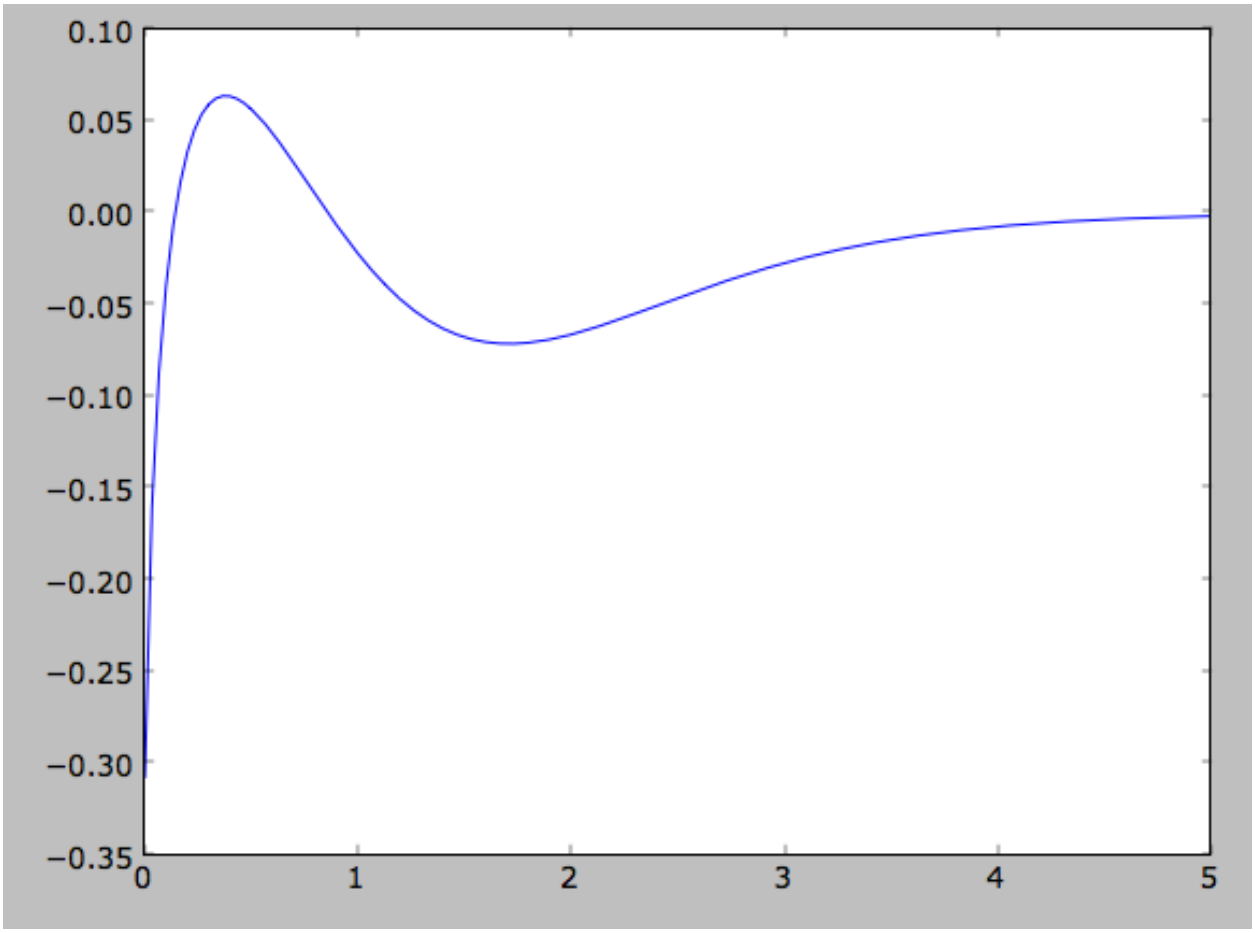
or

```
>>> plot(Rs,numpy.log(out))
```



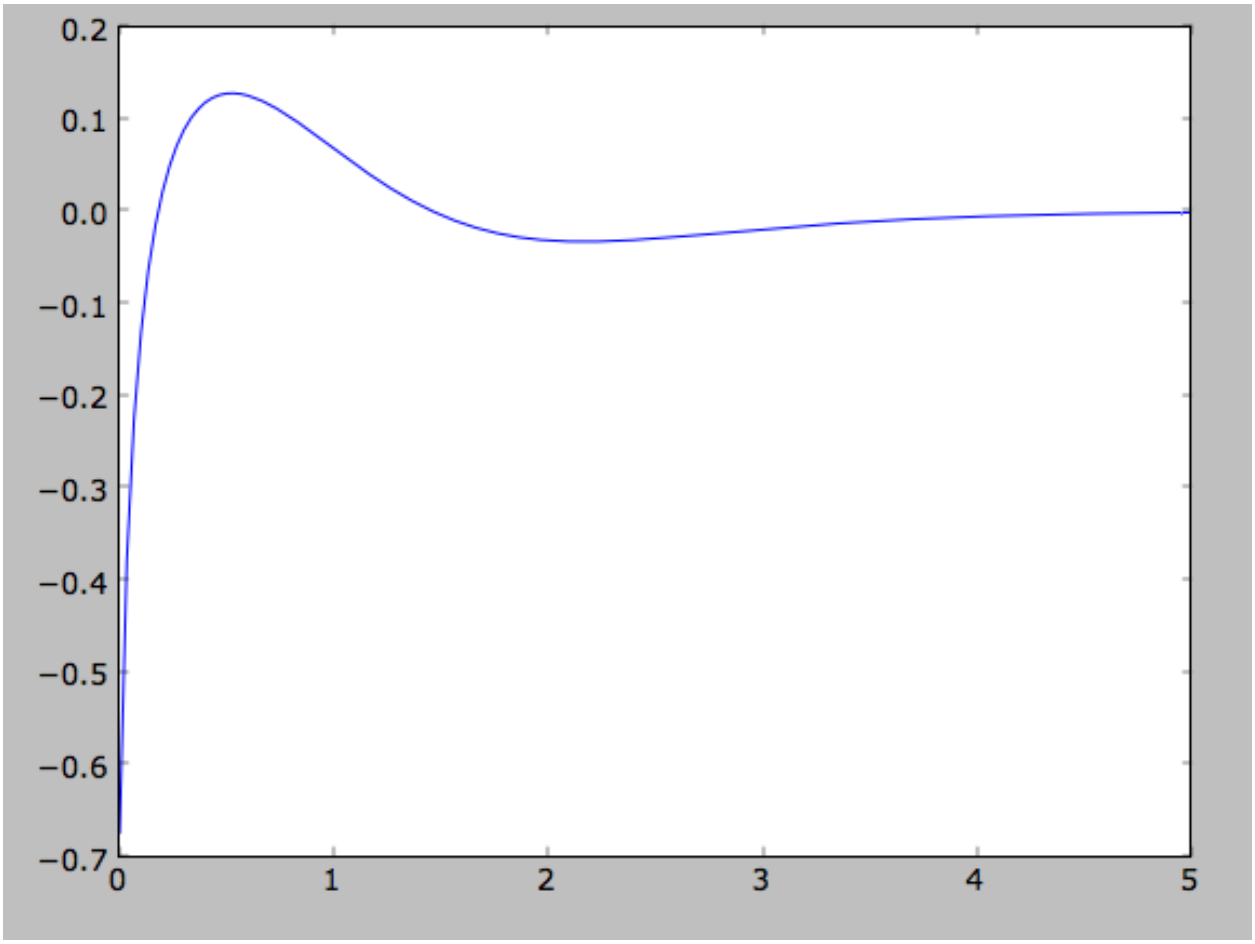
which shows the exponential behavior expected for an exponential disk. We can compare this to the input surface-mass density

```
>>> input_out= [dfc.targetSurfacemass(r) for r in Rs]
>>> plot(Rs,numpy.log(input_out)-numpy.log(out))
```



which shows that there are significant differences between the desired surface-mass density and the actual surface-mass density. We can do the same for the velocity-dispersion profile

```
>>> out= [dfc.sigmaR2(r) for r in Rs]
>>> input_out= [dfc.targetSigma2(r) for r in Rs]
>>> plot(Rs,numpy.log(input_out)-numpy.log(out))
```



That the input surface-density and velocity-dispersion profiles are not the same as the output profiles, means that estimates of DF properties based on these profiles will not be quite correct. Obviously this is the case for the surface-density and velocity-dispersion profiles themselves, which have to be explicitly calculated by integration over the DF rather than by evaluating the input profiles. This also means that estimates of the asymmetric drift based on the input profiles will be wrong. We can calculate the asymmetric drift at R=1 using the asymmetric drift equation derived from the Jeans equation (eq. 4.228 in Binney & Tremaine 2008), using the input surface-density and velocity dispersion profiles

```
>>> dfc.asymmetricdrift(1.)
# 0.090000000000000024
```

which should be equal to the circular velocity minus the mean rotational velocity

```
>>> 1.-dfc.meanvT(1.)
# 0.082847230205526756
```

These are not the same in part because of the difference between the input and output surface-density and velocity-dispersion profiles (and because the `asymmetricdrift` method assumes that the ratio of the velocity dispersions squared is two using the epicycle approximation; see above).

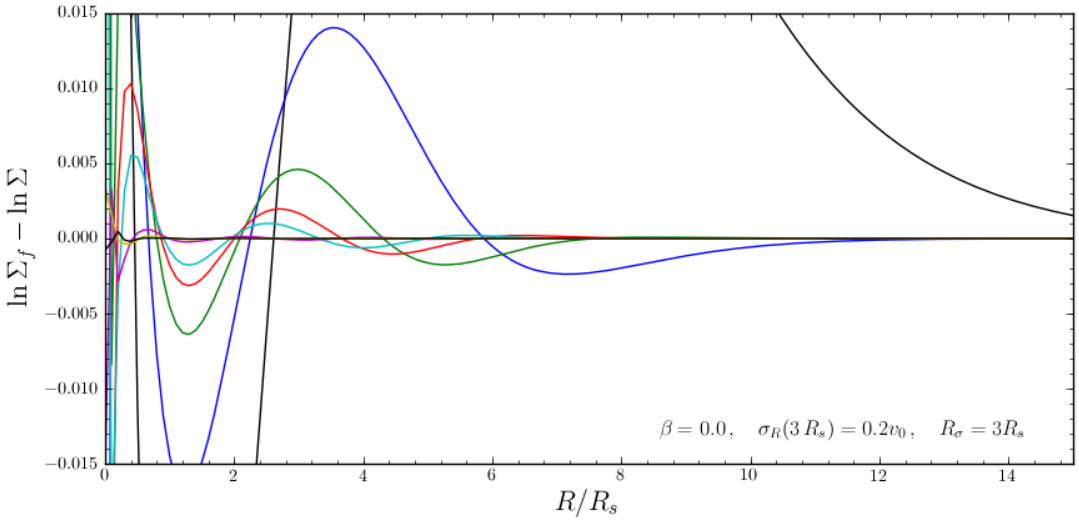
1.6.3 Using corrected disk distribution functions

As shown above, for a given surface-mass density and velocity dispersion profile, the two-dimensional disk distribution functions only do a poor job of reproducing the desired profiles. We can correct this by calculating a

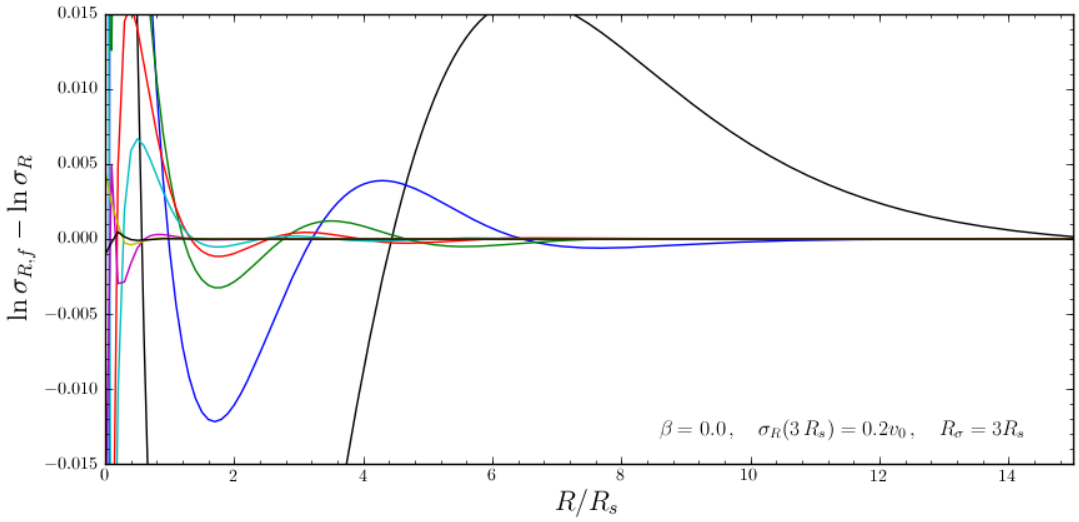
set of *corrections* to the input profiles such that the output profiles more closely resemble the desired profiles (see 1999AJ....118.1201D). galpy supports the calculation of these corrections, and comes with some pre-calculated corrections (these can be found here). For example, the following initializes a `dehnendf` with corrections up to 20th order (the default)

```
>>> dfc= dehnendf(beta=0.,correct=True)
```

The following figure shows the difference between the actual surface-mass density profile and the desired profile for 1, 2, 3, 4, 5, 10, 15, and 20 iterations



and the same for the velocity-dispersion profile



galpy will automatically save any new corrections that you calculate.

All of the methods for an uncorrected disk DF can be used for the corrected DFs as well. For example, the velocity dispersion is now

```
>>> numpy.sqrt(dfc.sigmaR2(1.))
# 0.19999985069451526
```

and the mean rotation velocity is

```
>>> dfc.meanvT(1.)
# 0.90355161181498711
```

and (correct) asymmetric drift

```
>>> 1.-dfc.meanvT(1.)
# 0.09644838818501289
```

That this still does not agree with the simple `dfc.asymmetricdrift` estimate is because of the latter's using the epicycle approximation for the ratio of the velocity dispersions.

1.6.4 Oort constants and functions

galpy also contains methods to calculate the Oort functions for two-dimensional disk distribution functions. These are known as the *Oort constants* when measured in the solar neighborhood. They are combinations of the mean velocities and derivatives thereof. galpy calculates these by direct integration over the DF and derivatives of the DF. Thus, we can calculate

```
>>> dfc= dehnendf(beta=0.)
>>> dfc.oortA(1.)
# 0.43190780889218749
>>> dfc.oortB(1.)
# -0.48524496090228575
```

The K and C Oort constants are zero for axisymmetric DFs

```
>>> dfc.oortC(1.)
# 0.0
>>> dfc.oortK(1.)
# 0.0
```

In the epicycle approximation, for a flat rotation curve A =- B = 0.5. The explicit calculates of A and B for warm DFs quantify how good (or bad) this approximation is

```
>>> dfc.oortA(1.)+dfc.oortB(1.)
# -0.053337152010098254
```

For the cold DF from above the approximation is much better

```
>>> dfccold= dehnendf(beta=0.,profileParams=(1./3.,1.,0.02))
>>> dfccold.oortA(1.), dfccold.oortB(1.)
# (0.49917556666144003, -0.49992824742490816)
```

1.6.5 Sampling data from the DF

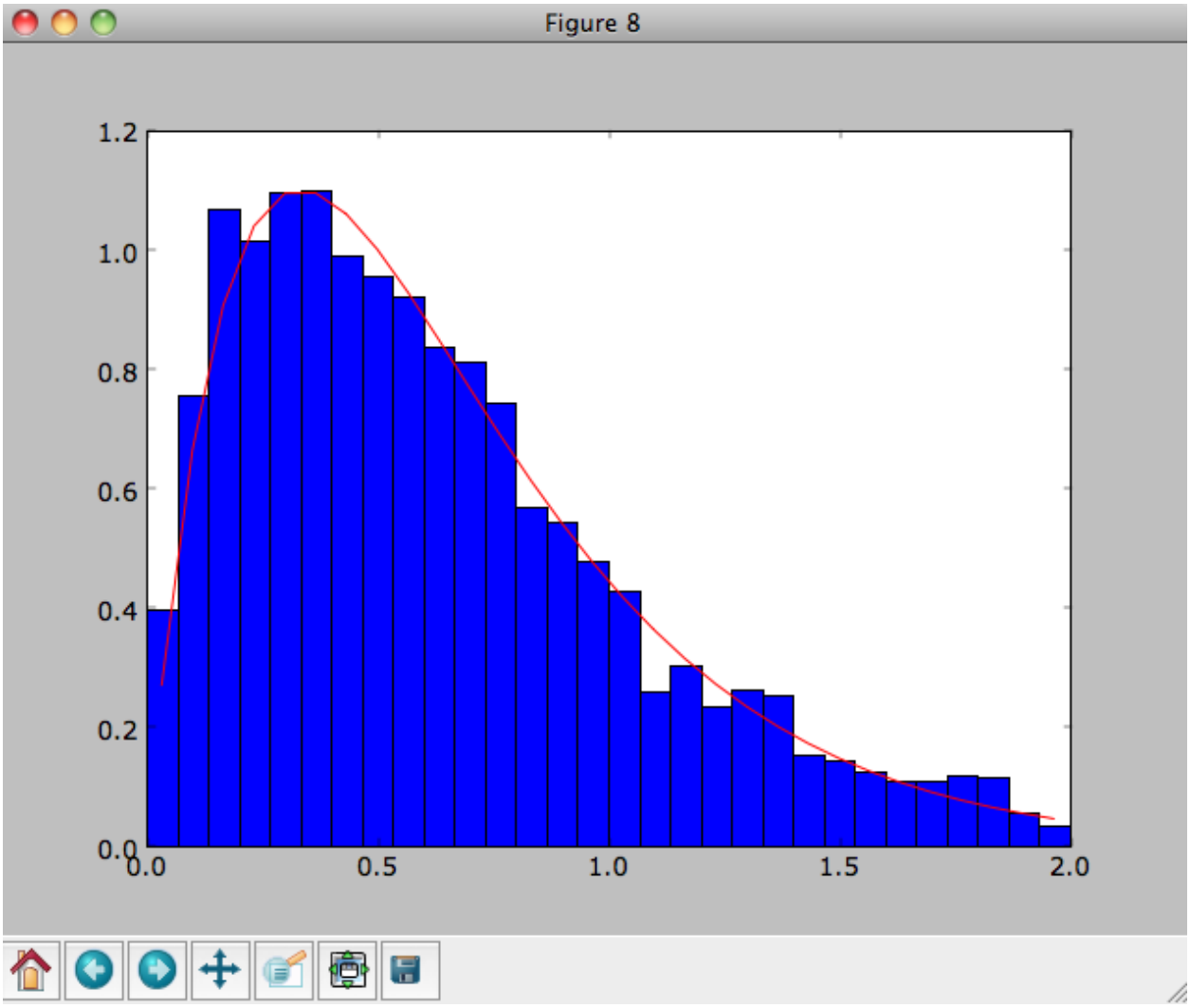
We can sample from the disk distribution functions using `sample`. `sample` can return either an energy–angular-momentum pair, or a full orbit initialization. We can sample 4000 orbits for example as (could take two minutes)

```
>>> o= dfc.sample(n=4000,returnOrbit=True,nphi=1)
```

We can then plot the histogram of the sampled radii and compare it to the input surface-mass density profile

```
>>> Rs= [e.R() for e in o]
>>> hists, bins, edges= hist(Rs,range=[0,2],normed=True,bins=30)
>>> xs= numpy.array([(bins[ii+1]+bins[ii])/2. for ii in range(len(bins)-1)])
>>> plot(xs, xs*exp(-xs*3.)*9.,'r-')
```

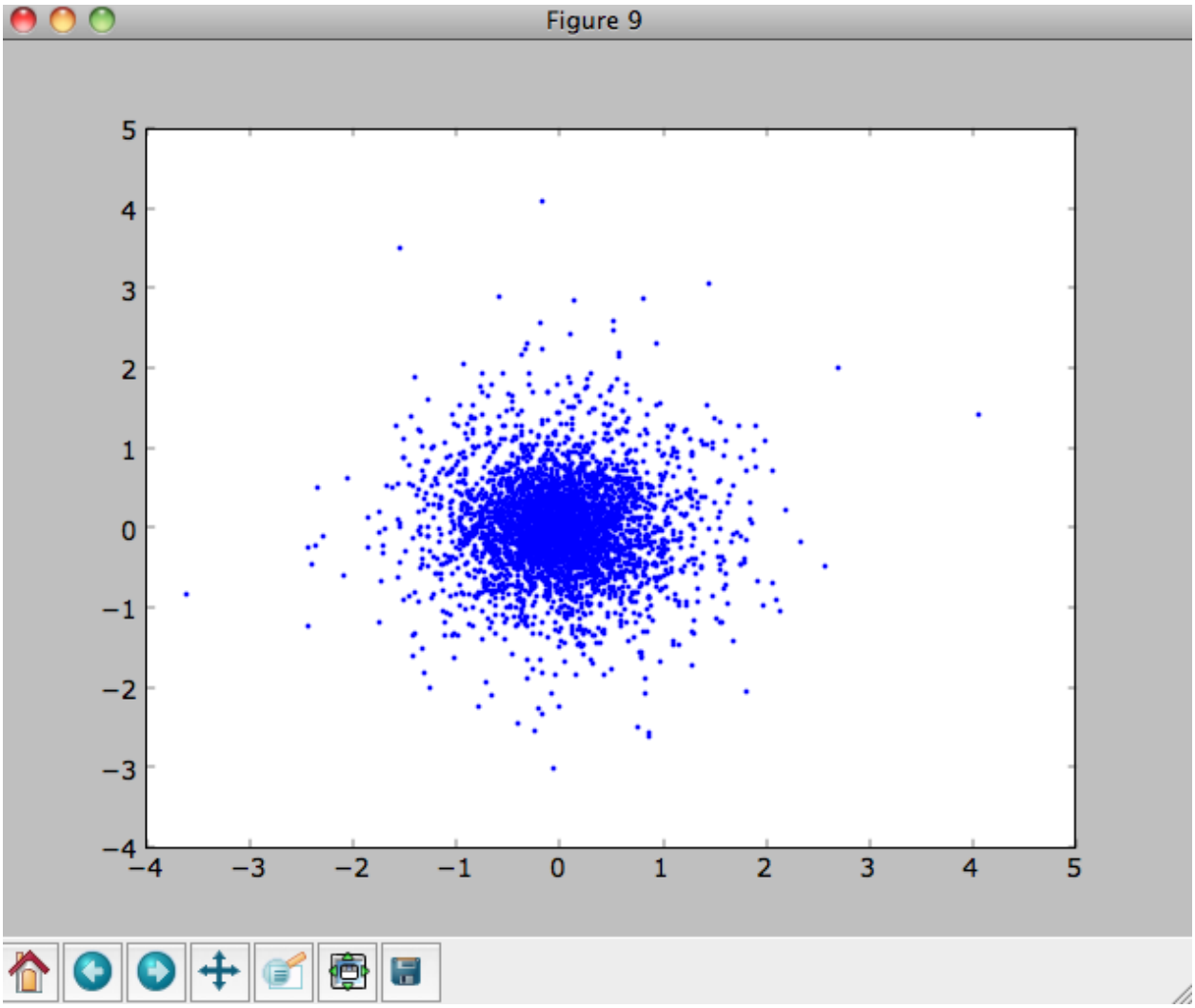
E.g.,



We can also plot the spatial distribution of the sampled disk

```
>>> xs= [e.x() for e in o]
>>> ys= [e.y() for e in o]
>>> figure()
>>> plot(xs,ys,',')
```

E.g.,

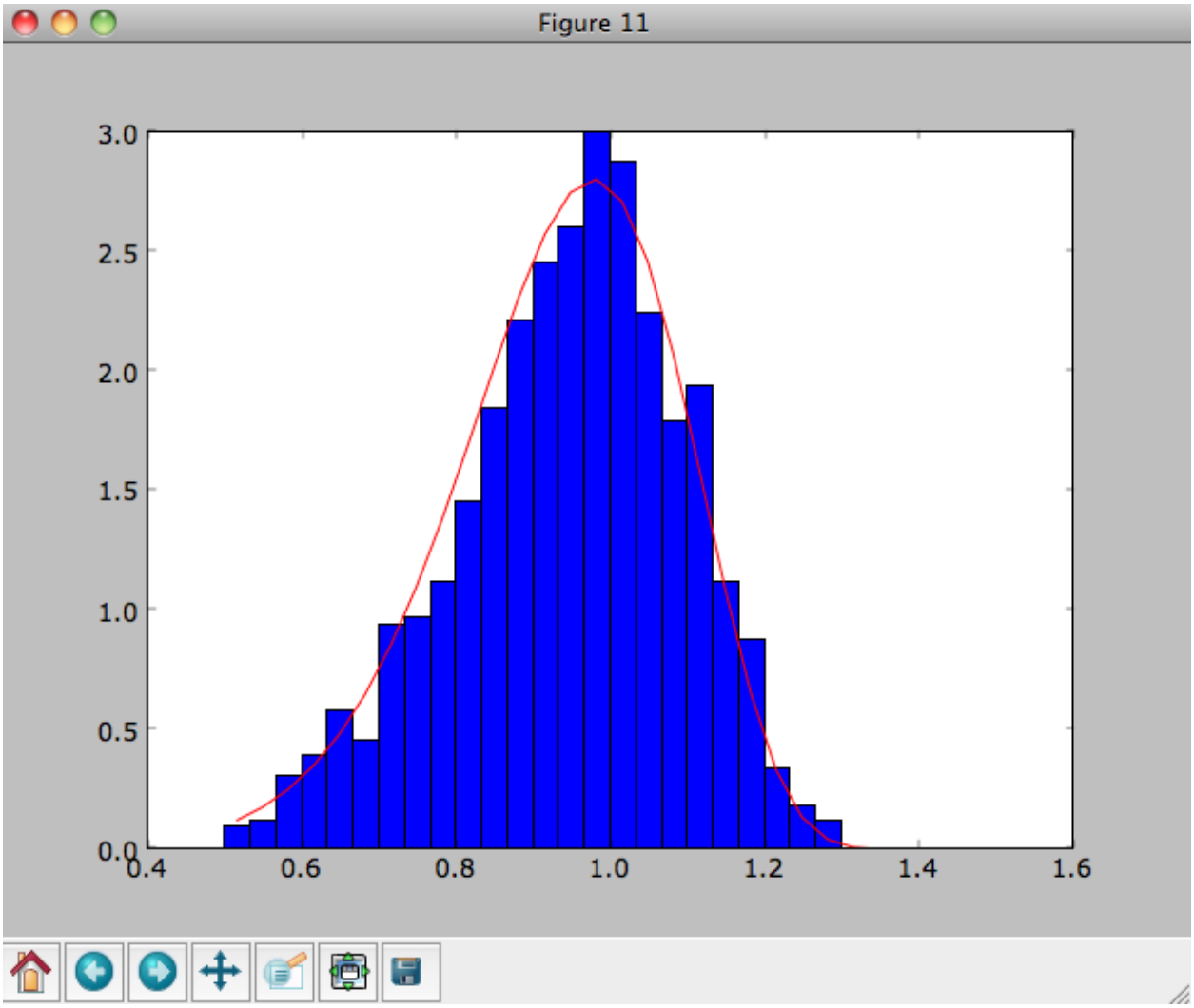


We can also sample points in a specific radial range (might take a few minutes)

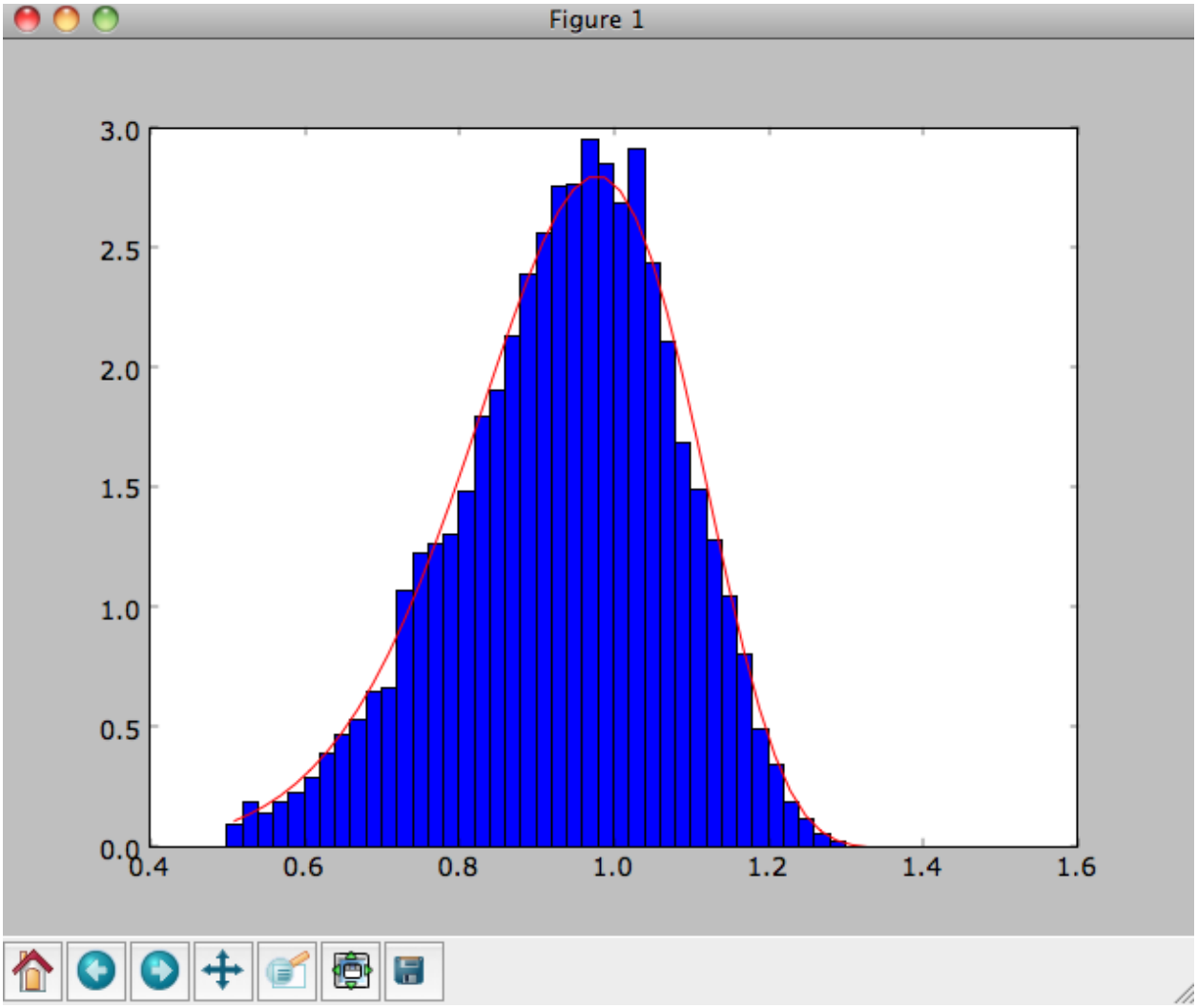
```
>>> o= dfc.sample(n=1000,returnOrbit=True,nphi=1,rrange=[0.8,1.2])
```

and we can plot the distribution of tangential velocities

```
>>> vTs= [e.vxvv[2] for e in o]
>>> hists, bins, edges= hist(vTs,range=[.5,1.5],normed=True,bins=30)
>>> xs= numpy.array([(bins[ii+1]+bins[ii])/2. for ii in range(len(bins)-1)])
>>> dfro= [dfc(Orbit([1.,0.,x]))/9./numpy.exp(-3.) for x in xs]
>>> plot(xs,dfro,'r-')
```

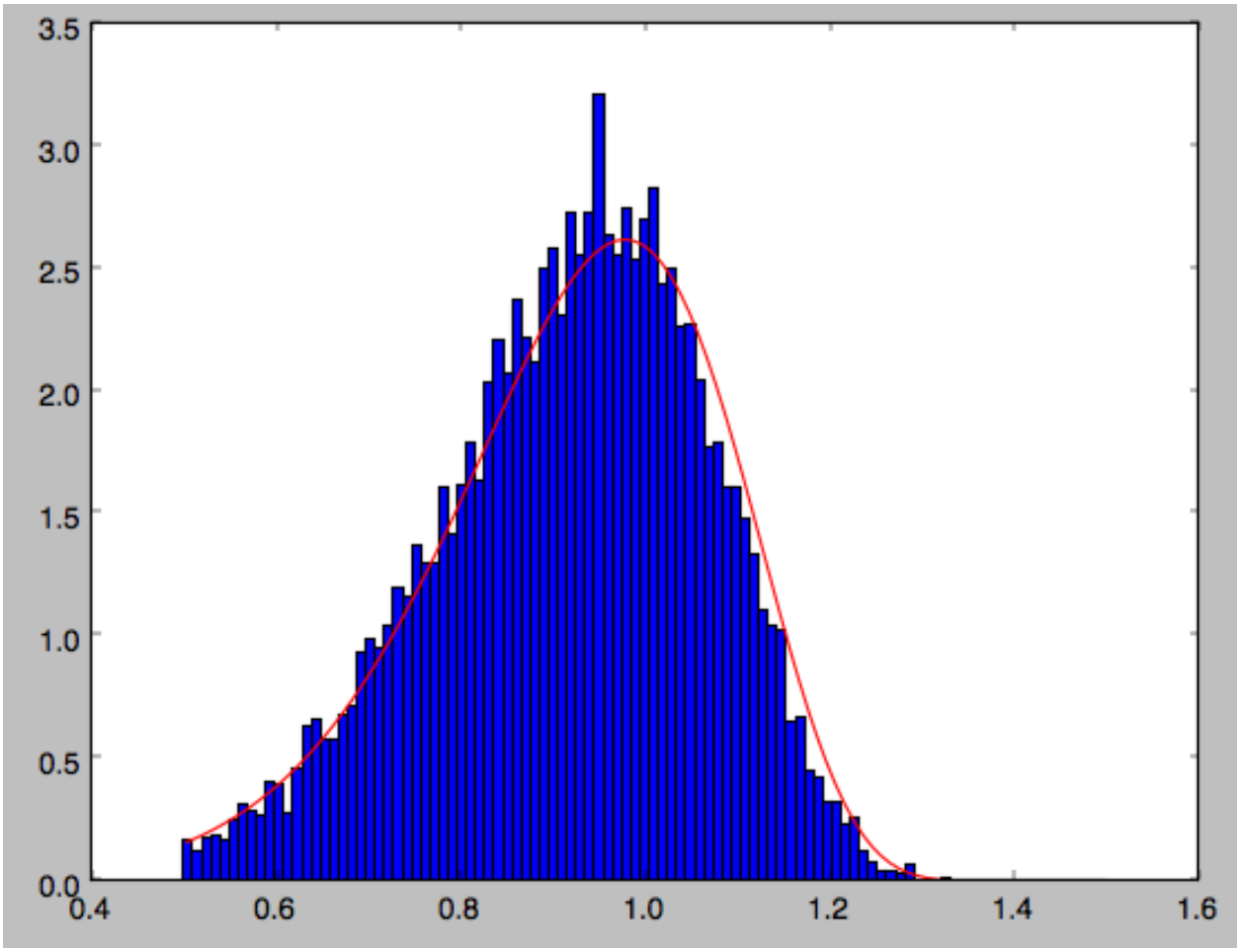


The agreement between the sampled distribution and the theoretical curve is not as good because the sampled distribution has a finite radial range. If we sample 10,000 points in `rrange=[0.95,1.05]` the agreement is better (this takes a long time):



We can also directly sample velocities at a given radius rather than in a range of radii. Doing this for a correct DF gives

```
>>> dfc= dehnendf(beta=0.,correct=True)
>>> vrvt= dfc.sampleVRVT(1.,n=10000)
>>> hists, bins, edges= hist(vrvt[:,1],range=[.5,1.5],normed=True,bins=101)
>>> xs= numpy.array([(bins[ii+1]+bins[ii])/2. for ii in range(len(bins)-1)])
>>> dfro= [dfc(Orbit([1.,0.,x])) for x in xs]
>>> plot(xs,dfro/numpy.sum(dfro)/(xs[1]-xs[0]),'r-')
```



galpy further has support for sampling along a given line of sight in the disk, which is useful for interpreting surveys consisting of a finite number of pointings. For example, we can sampled distances along a given line of sight

```
>>> ds= dfc.sampledSurfacemassLOS(30./180.*numpy.pi,n=10000)
```

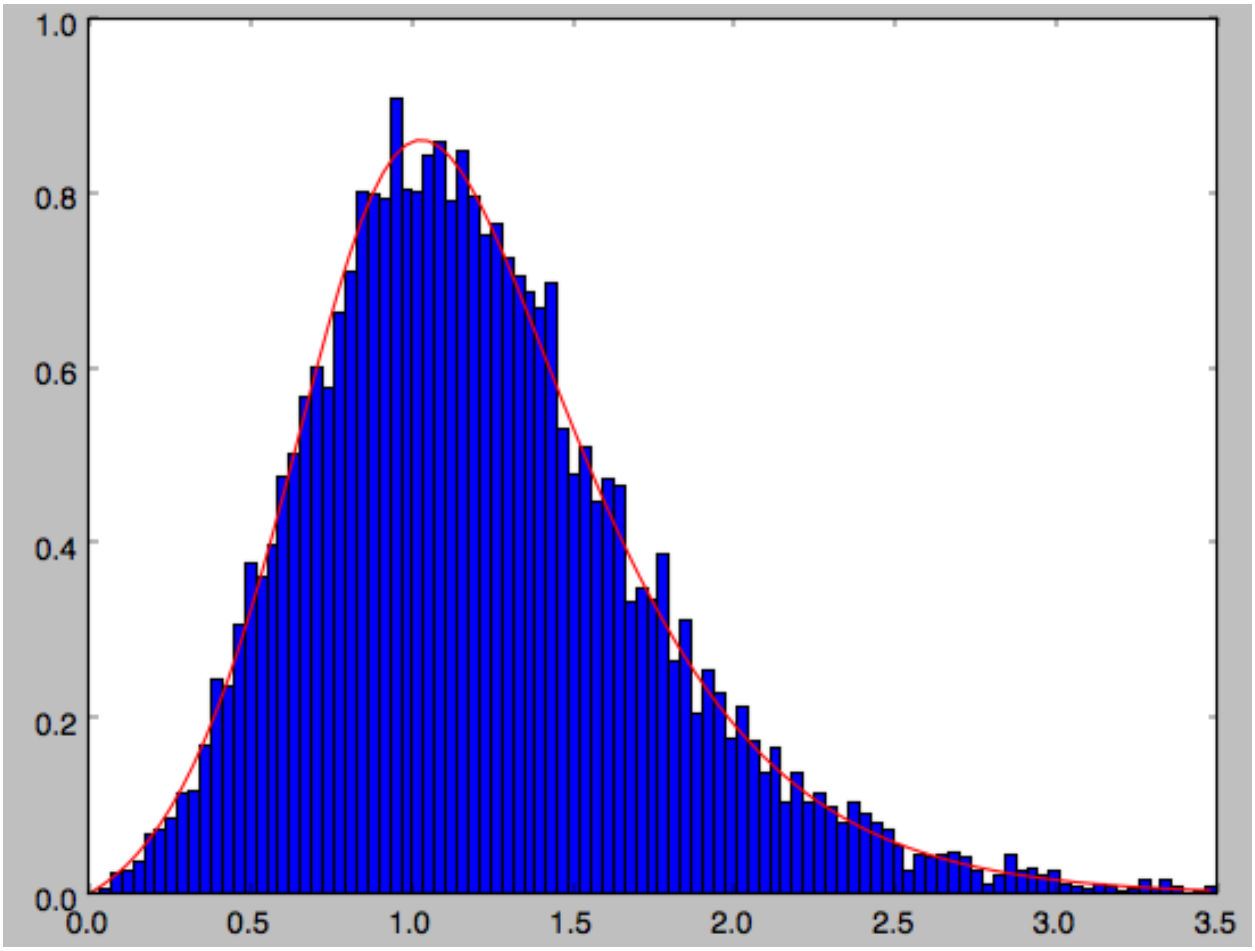
which is very fast. We can histogram these

```
>>> hists, bins, edges= hist(ds,range=[0.,3.5],normed=True,bins=101)
```

and compare it to the predicted distribution, which we can calculate as

```
>>> xs= numpy.array([(bins[ii+1]+bins[ii])/2. for ii in range(len(bins)-1)])
>>> fd= numpy.array([dfc.surfacemassLOS(d,30.) for d in xs])
>>> plot(xs,fd/numpy.sum(fd)/(xs[1]-xs[0]),'r-')
```

which shows very good agreement with the sampled distances



galpy can further sample full 4D phase–space coordinates along a given line of sight through `dfc.sampleLOS`.

1.6.6 Non-axisymmetric, time-dependent disk distribution functions

`galpy` also supports the evaluation of non-axisymmetric, time-dependent two-dimensional DFs. These specific DFs are constructed by assuming an initial axisymmetric steady state, described by a DF of the family discussed above, that is then acted upon by a non-axisymmetric, time-dependent perturbation. The DF at a given time and phase-space position is evaluated by integrating the orbit backwards in time in the non-axisymmetric potential until the time of the initial DF is reached. From Liouville's theorem, which states that phase-space volume is conserved along the orbit, we then know that we can evaluate the non-axisymmetric DF today as the initial DF at the initial point on the orbit. This procedure was first used by Dehnen (2000).

This is implemented in `galpy` as `galpy.df.evolveddiskdf`. Such a DF is setup by specifying the initial DF, the non-axisymmetric potential, and the time of the initial state. For example, we can look at the effect of an elliptical perturbation to the potential like that described by Kuijken & Tremaine. To do this, we set up an elliptical perturbation to a logarithmic potential that is grown slowly to minimize non-adiabatic effects

```
>>> from galpy.potential import LogarithmicHaloPotential, EllipticalDiskPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> ep= EllipticalDiskPotential(twophio=0.05,phib=0.,p=0.,tform=-150.,tsteady=125.)
```

This perturbation starts to be grown at `tform=-150` over a time period of `tsteady=125` time units. We will consider the effect of this perturbation on a very cold disk (velocity dispersion $\sigma_R = 0.0125\,v_c$) and a warm disk ($\sigma_R = 0.15\,v_c$). We set up these two initial DFs

```
>>> idfcold= dehnendf(beta=0.,profileParams=(1./3.,1.,0.0125))
>>> idfwarm= dehnendf(beta=0.,profileParams=(1./3.,1.,0.15))
```

and then set up the `evolveddiskdf`

```
>>> from galpy.df import evolveddiskdf
>>> edfcold= evolveddiskdf(idfcold,[lp,ep],to=-150.)
>>> edfwarm= evolveddiskdf(idfwarm,[lp,ep],to=-150.)
```

where we specify that the initial state is at `to=-150`.

We can now use these `evolveddiskdf` instances in much the same way as `diskdf` instances. One difference is that there is much more support for evaluating the DF on a grid (to help speed up the rather slow computations involved). Thus, we can evaluate the mean radial velocity at `R=0.9`, `phi=22.5` degree, and `t=0` by using a grid

```
>>> mvrcold, gridcold= edfcold.meanvR(0.9,phi=22.5,deg=True,t=0.,grid=True,
↪returnGrid=True,gridpoints=51,nsigma=6.)
>>> mvrwarm, gridwarm= edfwarm.meanvR(0.9,phi=22.5,deg=True,t=0.,grid=True,
↪returnGrid=True,gridpoints=51)
>>> print(mvrcold, mvrwarm)
# -0.0358753028951 -0.0294763627935
```

The cold response agrees well with the analytical calculation, which predicts that this is $-0.05/\sqrt{2}$:

```
>>> print(mvrcold+0.05/sqrt(2.))
# -0.000519963835811
```

The warm response is slightly smaller in amplitude

```
>>> print(mvrwarm/mvrcold)
# 0.821633837619
```

although the numerical uncertainty in `mvrwarm` is large, because the grid is not sufficiently fine.

We can then re-use this grid in calculations of other moments of the DF, e.g.,

```
>>> print(edfcold.meanvT(0.9,phi=22.5,deg=True,t=0.,grid=gridcold))
# 0.965058551359
>>> print(edfwarm.meanvT(0.9,phi=22.5,deg=True,t=0.,grid=gridwarm))
# 0.915397094614
```

which returns the mean rotational velocity, and

```
>>> print(edfcold.vertexdev(0.9,phi=22.5,deg=True,t=0.,grid=gridcold))
# 0.0560531474616
>>> print(edfwarm.vertexdev(0.9,phi=22.5,deg=True,t=0.,grid=gridwarm))
# 0.0739164830253
```

which gives the vertex deviation in rad. The reason we have to calculate the grid out to `6nsigma` for the cold response is that the response is much bigger than the velocity dispersion of the population. This velocity dispersion is used to automatically to set the grid edges, but sometimes has to be adjusted to contain the full DF.

`evolveddiskdf` can also calculate the Oort functions, by directly calculating the spatial derivatives of the DF. These can also be calculated on a grid, such that we can do

```
>>> oortacold, gridcold, gridrcold, gridphicold= edfcold.oortA(0.9,phi=22.5,deg=True,
↪t=0.,returnGrids=True,gridpoints=51,derivGridpoints=51,grid=True,derivphiGrid=True,
↪derivRGrid=True,nsigma=6.)
>>> oortawarm, gridwarm, gridrwarm, gridphiwarm= edfwarm.oortA(0.9,phi=22.5,deg=True,
↪t=0.,returnGrids=True,gridpoints=51,derivGridpoints=51,grid=True,derivphiGrid=True,
↪derivRGrid=True)
>>> print(oortacold, oortawarm)
# 0.575494559999 0.526389833249
```

It is clear that these are quite different. The cold calculation is again close to the analytical prediction, which says that $A = A_{\rm axi} + 0.05/(2\sqrt{2})$ where $A_{\rm axi} = 1/(2 \times 0.9)$ in this case:

```
>>> print(oortacold-(0.5/0.9+0.05/2./sqrt(2.)))
# 0.0022613349141670236
```

These grids can then be re-used for the other Oort functions, for example,

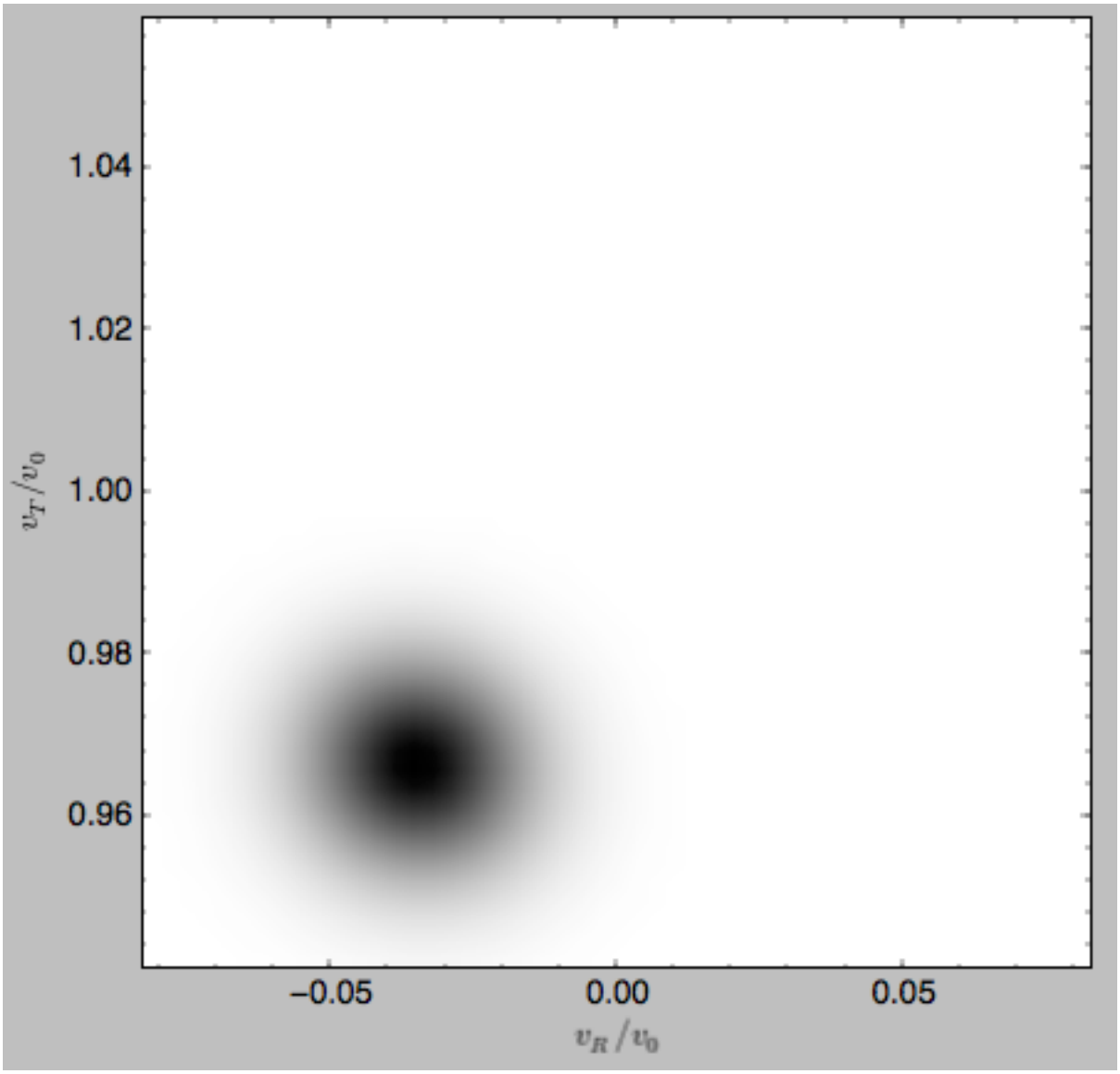
```
>>> print(edfcold.oortB(0.9,phi=22.5,deg=True,t=0.,grid=gridcold,
↪derivphiGrid=gridphicold,derivRGrid=gridrcold))
# -0.574674310521
>>> print(edfwarm.oortB(0.9,phi=22.5,deg=True,t=0.,grid=gridwarm,
↪derivphiGrid=gridphiwarm,derivRGrid=gridrwarm))
# -0.555546911144
```

and similar for `oortC` and `oortK`. These warm results should again be considered for illustration only, as the grid is not sufficiently fine to have a small numerical error.

The grids that have been calculated can also be plotted to show the full velocity DF. For example,

```
>>> gridcold.plot()
```

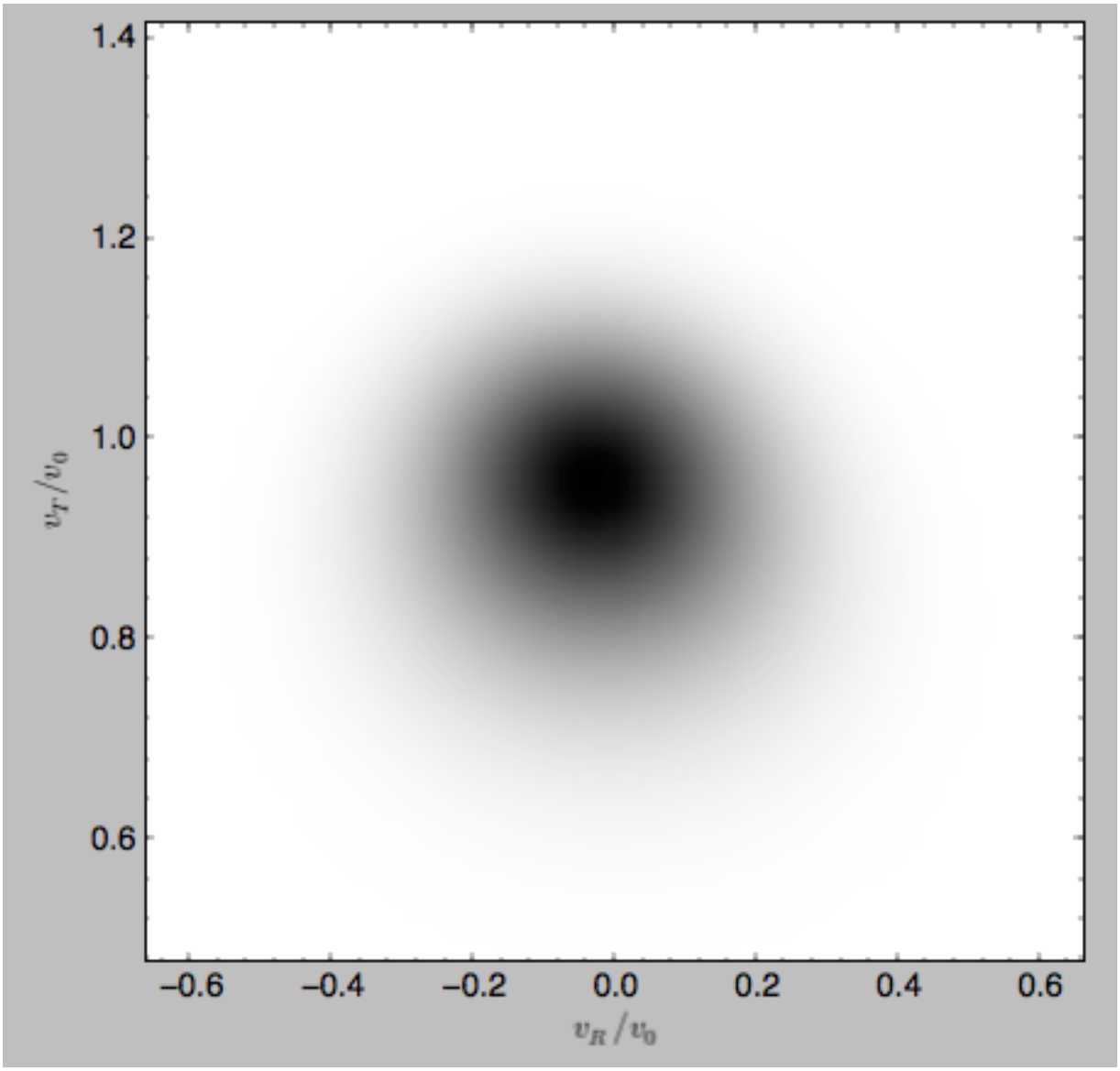
gives



which demonstrates that the DF is basically the initial DF that has been displaced (by a significant amount compared to the velocity dispersion). The warm velocityd distribution is given by

```
>>> gridwarm.plot()
```

which returns



The shift of the smooth DF here is much smaller than the velocity dispersion.

1.6.7 Example: The Hercules stream in the Solar neighborhood as a result of the Galactic bar

We can combine the orbit integration capabilities of galpy with the provided distribution functions and see the effect of the Galactic bar on stellar velocities. By backward integrating orbits starting at the Solar position in a potential that includes the Galactic bar we can evaluate what the velocity distribution is that we should see today if the Galactic bar stirred up a steady-state disk. For this we initialize a flat rotation curve potential and Dehnen's bar potential

```
>>> from galpy.potential import LogarithmicHaloPotential, DehnenBarPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> dp= DehnenBarPotential()
```

The Dehnen bar potential is initialized to start bar formation four bar periods before the present day and to have completely formed the bar two bar periods ago. We can integrate back to the time before bar-formation:

```
>>> ts= numpy.linspace(0,dp.tform(),1000)
```

where `dp.tform()` is the time of bar-formation (in the usual time-coordinates).

We initialize orbits on a grid in velocity space and integrate them

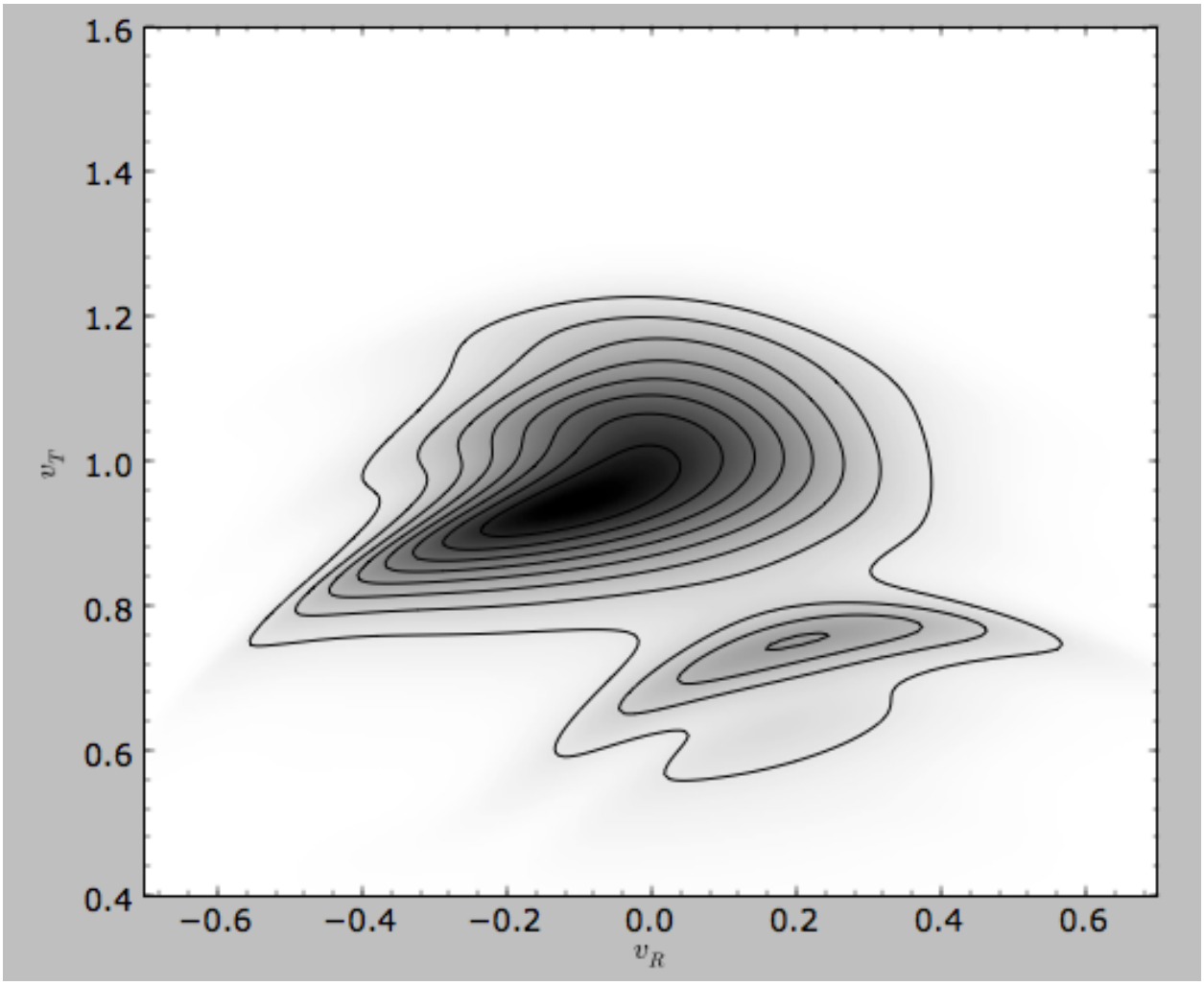
```
>>> ins= Orbit(numpy.array([[[1.,-0.7+1.4/100*jj,1.-0.6+1.2/100*ii,0.] for jj in␣
↪range(101)] for ii in range(101)]))
>>> ins.integrate(ts,[lp,dp])
```

We can then evaluate the weight of these orbits by assuming that the disk was in a steady-state before bar-formation with a Dehnen distribution function. We evaluate the Dehnen distribution function at `dp.tform()` for each of the orbits (evaluating the distribution function only works for an Orbit with a single object, so we need to unpack the Orbit instance that contains all orbits)

```
>>> dfc= dehnendf(beta=0.,correct=True)
>>> out= [[dfc(o(dp.tform())) for o in j] for j in ins]
>>> out= numpy.array(out)
```

This gives

```
>>> from galpy.util.plot import dens2d
>>> dens2d(out,origin='lower',cmap='gist_yarg',contours=True,xrange=[-0.7,0.7],
↪yrange=[0.4,1.6],xlabel=r'$v_R$',ylabel=r'$v_T$')
```

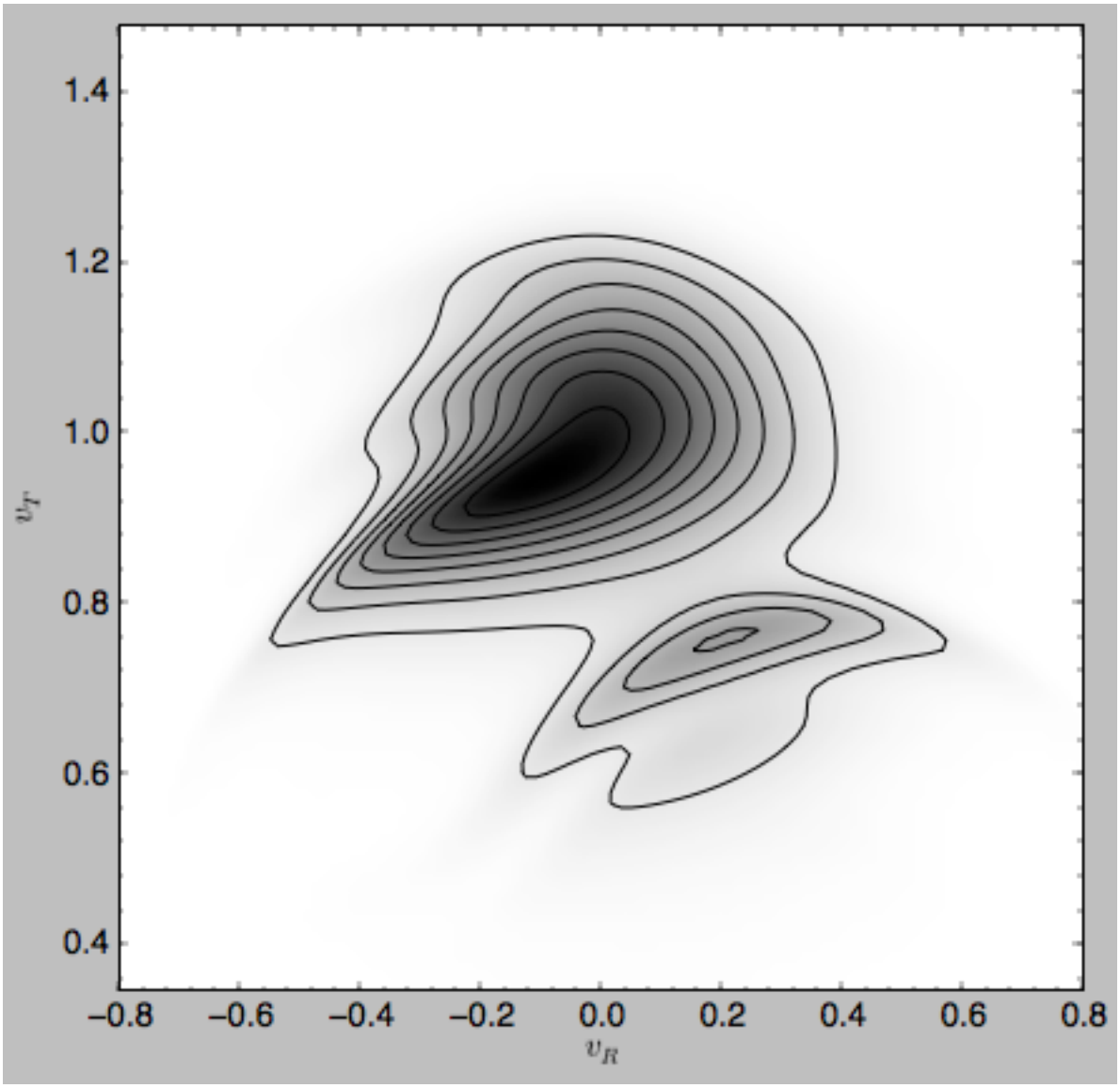


Now that `galpy` contains the `evolveddiskdf` described above, this whole calculation is encapsulated in this module and can be done much more easily as

```
>>> edf= evolveddiskdf(dfc,[lp,dp],to=dp.tform())
>>> mvr, grid= edf.meanvR(1.,grid=True,gridpoints=101,returnGrid=True)
```

The gridded DF can be accessed as `grid.df`, which we can plot as before

```
>>> dens2d(grid.df.T,origin='lower',cmap='gist_yarg',contours=True,xrange=[grid.
↪vRgrid[0],grid.vRgrid[-1]],yrange=[grid.vTgrid[0],grid.vTgrid[-1]],xlabel=r'$v_R$',
↪ylabel=r'$v_T$')
```



For more information see 2000AJ....119..800D and 2010ApJ...725.1676B. Note that the x-axis in the Figure above is defined as minus the x-axis in these papers.

1.7 Action-angle coordinates

galpy can calculate actions and angles for a large variety of potentials (any time-independent potential in principle). These are implemented in a separate module `galpy.actionAngle`, and the preferred method for accessing them is through the routines in this module. There is also some support for accessing the actionAngle routines as methods of the `Orbit` class.

Tip: If you want to quickly and easily compute actions, angles, or frequencies using the Staeckel approximation, using the `Orbit` interface as described in *this section* is recommended. Especially if you are starting from observed coordinates, as `Orbit` instances can easily be initialized using these.

Since v1.2, galpy can also compute positions and velocities corresponding to a given set of actions and angles for

axisymmetric potentials using the TorusMapper code of Binney & McMillan (2016). This is described in *this section* below. The interface for this is different than for the other action-angle classes, because the transformations are generally different.

Action-angle coordinates can be calculated for the following potentials/approximations:

- Isochrone potential
- Spherical potentials
- Adiabatic approximation
- Staeckel approximation
- A general orbit-integration-based technique

There are classes corresponding to these different potentials/approximations and actions, frequencies, and angles can typically be calculated using these three methods:

- __call__: returns the actions
- actionsFreqs: returns the actions and the frequencies
- actionsFreqsAngles: returns the actions, frequencies, and angles

These are not all implemented for each of the cases above yet.

The adiabatic and Staeckel approximation have also been implemented in C and using grid-based interpolation, for extremely fast action-angle calculations (see below).

1.7.1 Action-angle coordinates for the isochrone potential

The isochrone potential is the only potential for which all of the actions, frequencies, and angles can be calculated analytically. We can do this in galpy by doing

```
>>> from galpy.potential import IsochronePotential
>>> from galpy.actionAngle import actionAngleIsochrone
>>> ip= IsochronePotential(b=1.,normalize=1.)
>>> aAI= actionAngleIsochrone(ip=ip)
```

`aAI` is now an instance that can be used to calculate action-angle variables for the specific isochrone potential `ip`. Calling this instance returns (J_R, L_Z, J_Z)

```
>>> aAI(1.,0.1,1.1,0.1,0.) #inputs R,vR,vT,z,vz
# (array([ 0.00713759]), array([ 1.1]), array([ 0.00553155]))
```

or for a more eccentric orbit

```
>>> aAI(1.,0.5,1.3,0.2,0.1)
# (array([ 0.13769498]), array([ 1.3]), array([ 0.02574507]))
```

Note that we can also specify `phi`, but this is not necessary

```
>>> aAI(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.13769498]), array([ 1.3]), array([ 0.02574507]))
```

We can likewise calculate the frequencies as well

```
>>> aAI.actionsFreqs(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.13769498]),
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 1.29136096]),
#  array([ 0.79093738]),
#  array([ 0.79093738]))
```

The output is $(J_R, L_Z, J_Z, \Omega_R, \Omega_\phi, \Omega_Z)$. For any spherical potential, $\Omega_\phi = \mathrm{sgn}(L_Z)\Omega_Z$, such that the last two frequencies are the same.

We obtain the angles as well by calling

```
>>> aAI.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.13769498]),
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 1.29136096]),
#  array([ 0.79093738]),
#  array([ 0.79093738]),
#  array([ 0.57101518]),
#  array([ 5.96238847]),
#  array([ 1.24999949]))
```

The output here is $(J_R, L_Z, J_Z, \Omega_R, \Omega_\phi, \Omega_Z, \theta_R, \theta_\phi, \theta_Z)$.

To check that these are good action-angle variables, we can calculate them along an orbit

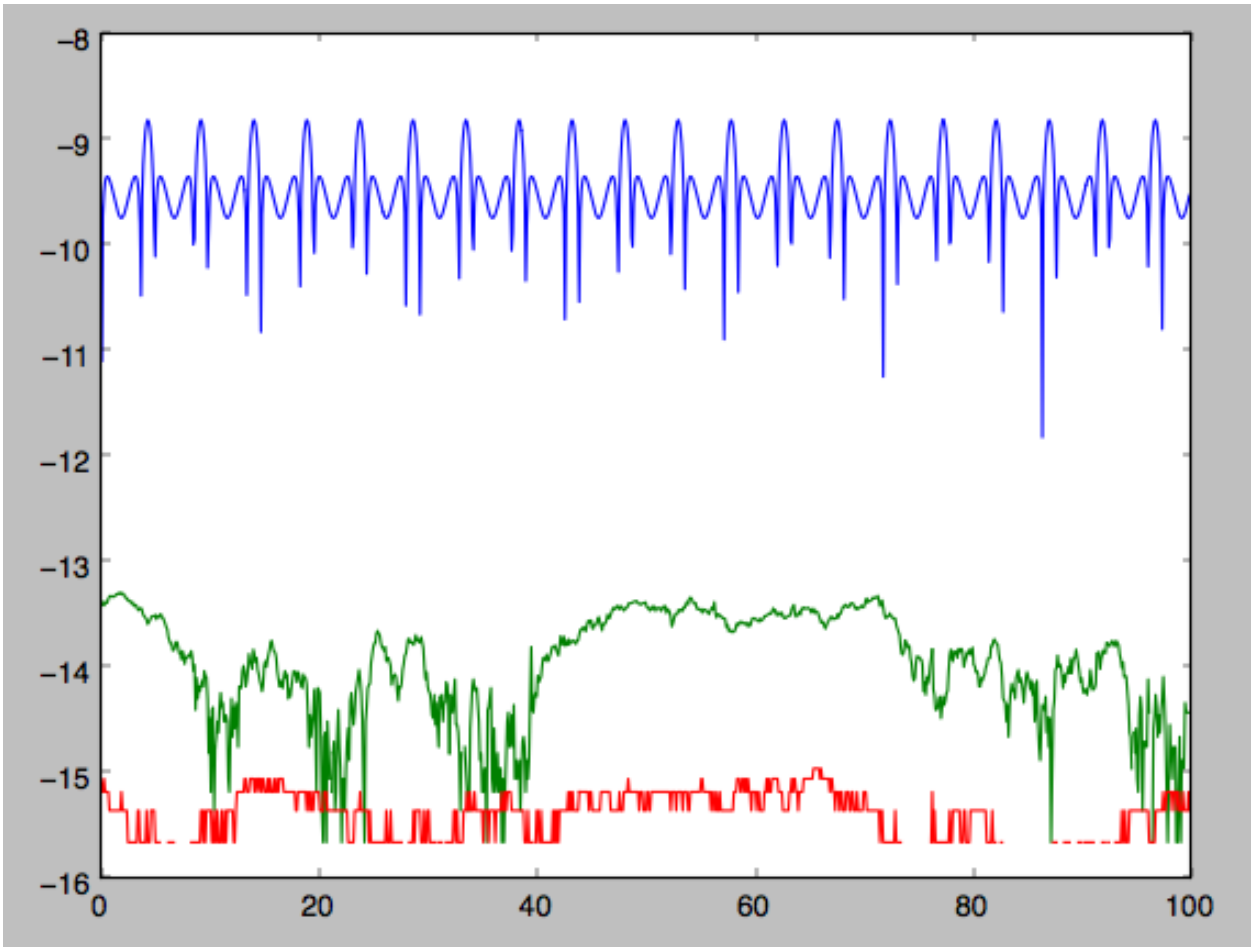
```
>>> from galpy.orbit import Orbit
>>> o= Orbit([1.,0.5,1.3,0.2,0.1,0.])
>>> ts= numpy.linspace(0.,100.,1001)
>>> o.integrate(ts,ip)
>>> jfa= aAI.actionsFreqsAngles(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),o.phi(ts))
```

which works because we can provide arrays for the `R` etc. inputs.

We can then check that the actions are constant over the orbit

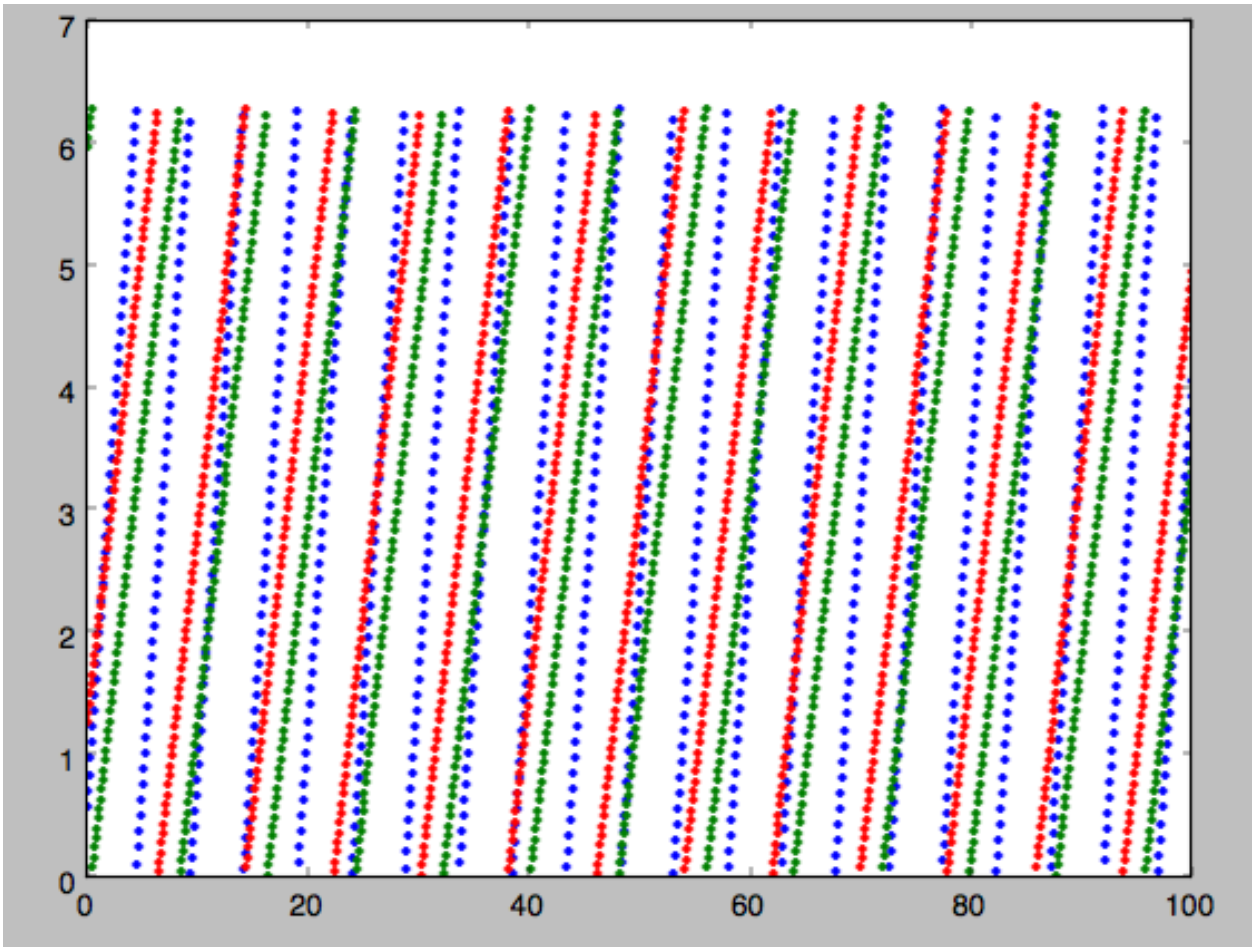
```
>>> plot(ts,numpy.log10(numpy.fabs((jfa[0]-numpy.mean(jfa[0])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[1]-numpy.mean(jfa[1])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[2]-numpy.mean(jfa[2])))))
```

which gives



The actions are all conserved. The angles increase linearly with time

```
>>> plot(ts,jfa[6],'b.')
>>> plot(ts,jfa[7],'g.')
>>> plot(ts,jfa[8],'r.')
```



1.7.2 Action-angle coordinates for spherical potentials

Action-angle coordinates for any spherical potential can be calculated using a few orbit integrations. These are implemented in galpy in the `actionAngleSpherical` module. For example, we can do

```
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> from galpy.actionAngle import actionAngleSpherical
>>> aAS= actionAngleSpherical(pot=lp)
```

For the same eccentric orbit as above we find

```
>>> aAS(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.22022112]), array([ 1.3]), array([ 0.02574507]))
>>> aAS.actionsFreqs(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.22022112]),
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 0.87630459]),
#  array([ 0.60872881]),
#  array([ 0.60872881]))
>>> aAS.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.22022112]),
```

(continues on next page)

(continued from previous page)

```
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 0.87630459]),
#  array([ 0.60872881]),
#  array([ 0.60872881]),
#  array([ 0.40443857]),
#  array([ 5.85965048]),
#  array([ 1.1472615]))
```

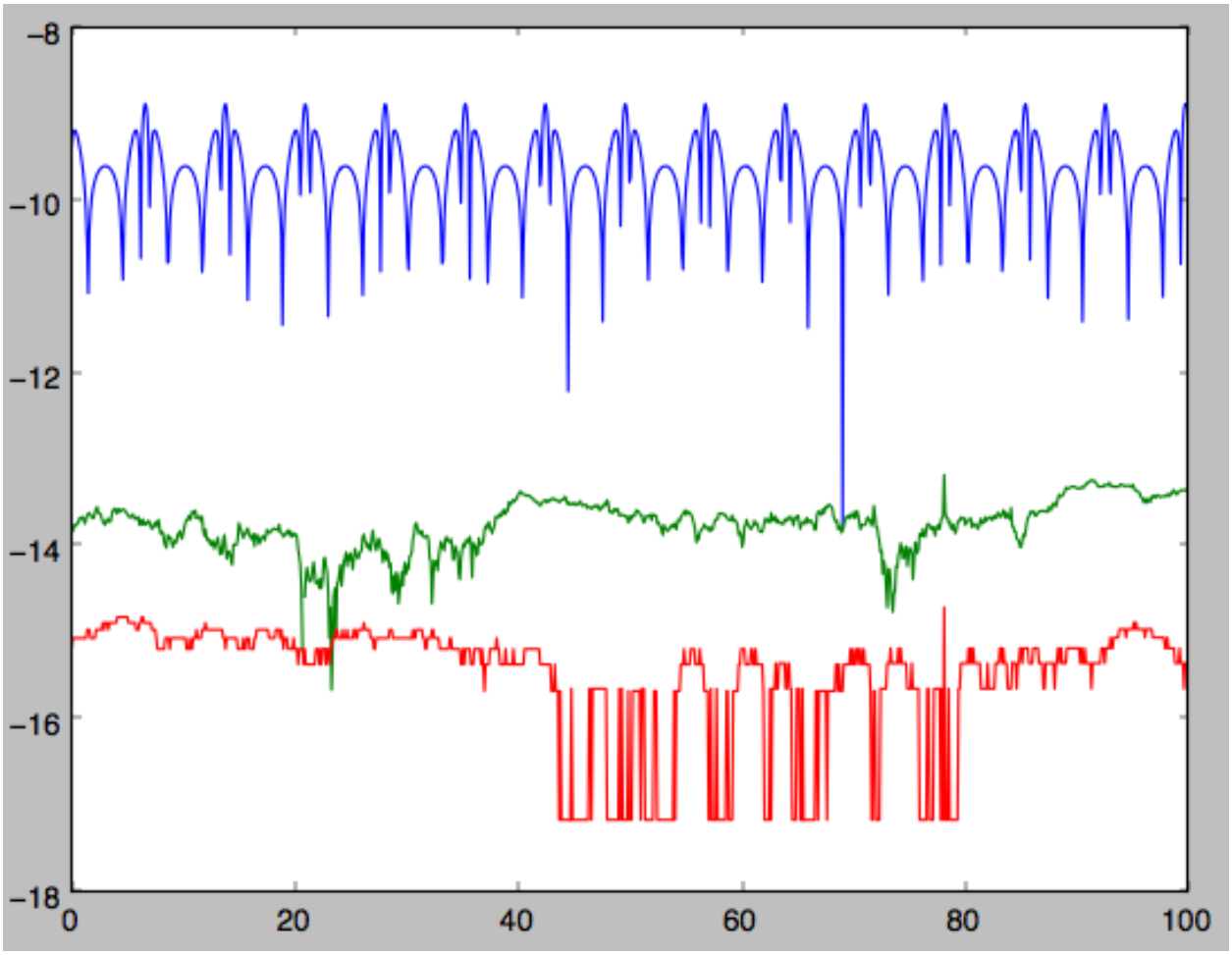
We can again check that the actions are conserved along the orbit and that the angles increase linearly with time:

```
>>> o.integrate(ts,lp)
>>> jfa= aAS.actionsFreqsAngles(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),o.phi(ts),
↪fixed_quad=True)
```

where we use `fixed_quad=True` for a faster evaluation of the required one-dimensional integrals using Gaussian quadrature. We then plot the action fluctuations

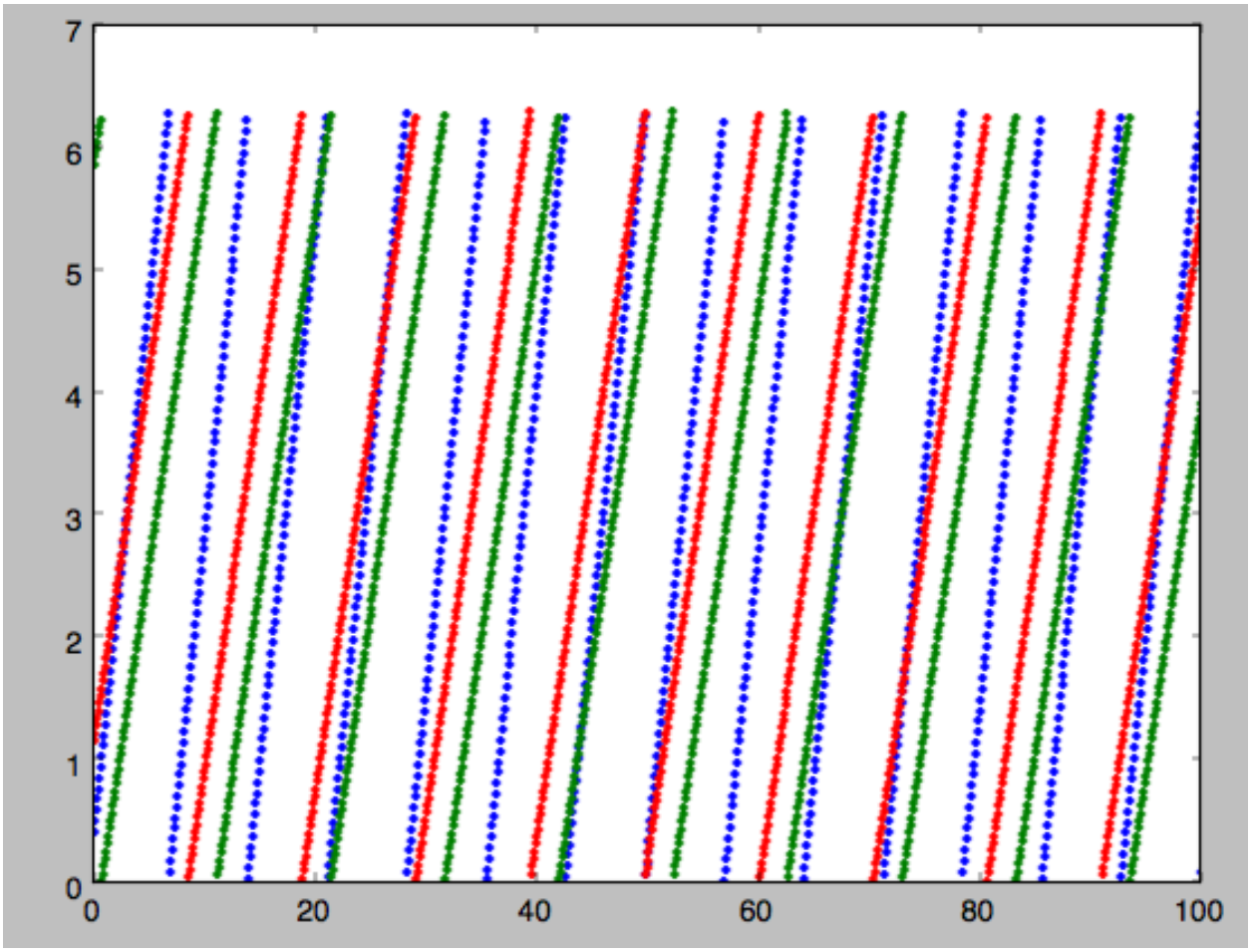
```
>>> plot(ts,numpy.log10(numpy.fabs((jfa[0]-numpy.mean(jfa[0])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[1]-numpy.mean(jfa[1])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[2]-numpy.mean(jfa[2])))))
```

which gives



showing that the actions are all conserved. The angles again increase linearly with time

```
>>> plot(ts,jfa[6],'b.')
>>> plot(ts,jfa[7],'g.')
>>> plot(ts,jfa[8],'r.')
```



We can check the spherical action-angle calculations against the analytical calculations for the isochrone potential. Starting again from the isochrone potential used in the previous section

```
>>> ip= IsochronePotential(b=1.,normalize=1.)
>>> aAI= actionAngleIsochrone(ip=ip)
>>> aAS= actionAngleSpherical(pot=ip)
```

we can compare the actions, frequencies, and angles computed using both

```
>>> aAI.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.13769498]),
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 1.29136096]),
#  array([ 0.79093738]),
#  array([ 0.79093738]),
#  array([ 0.57101518]),
#  array([ 5.96238847]),
#  array([ 1.24999949]))
```

(continues on next page)

(continued from previous page)

```
>>> aAS.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.13769498]),
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 1.29136096]),
#  array([ 0.79093738]),
#  array([ 0.79093738]),
#  array([ 0.57101518]),
#  array([ 5.96238838]),
#  array([ 1.2499994]))
```

or more explicitly comparing the two

```
>>> [r-s for r,s in zip(aAI.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.),aAS.
↪actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.))]
# [array([  6.66133815e-16]),
#  array([ 0.]),
#  array([ 0.]),
#  array([ -4.53851845e-10]),
#  array([  4.74775219e-10]),
#  array([  4.74775219e-10]),
#  array([ -1.65965242e-10]),
#  array([  9.04759645e-08]),
#  array([  9.04759649e-08])]
```

1.7.3 Action-angle coordinates using the adiabatic approximation

For non-spherical, axisymmetric potentials galpy contains multiple methods for calculating approximate action–angle coordinates. The simplest of those is the adiabatic approximation, which works well for disk orbits that do not go too far from the plane, as it assumes that the vertical motion is decoupled from that in the plane (e.g., 2010MNRAS.401.2318B).

Setup is similar as for other actionAngle objects

```
>>> from galpy.potential import MWPotential2014
>>> from galpy.actionAngle import actionAngleAdiabatic
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014)
```

and evaluation then proceeds similarly as before

```
>>> aAA(1.,0.1,1.1,0.,0.05)
# (0.01351896260559274, 1.1, 0.0004690133479435352)
```

We can again check that the actions are conserved along the orbit

```
>>> from galpy.orbit import Orbit
>>> ts=numpy.linspace(0.,100.,1001)
>>> o= Orbit([1.,0.1,1.1,0.,0.05])
>>> o.integrate(ts,MWPotential2014)
>>> js= aAA(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts))
```

This takes a while. The adiabatic approximation is also implemented in C, which leads to great speed-ups. Here is how to use it

```
>>> timeit(aAA(1.,0.1,1.1,0.,0.05))
# 10 loops, best of 3: 73.7 ms per loop
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
>>> timeit(aAA(1.,0.1,1.1,0.,0.05))
# 1000 loops, best of 3: 1.3 ms per loop
```

or about a *50 times* speed-up. For arrays the speed-up is even more impressive

```
>>> s= numpy.ones(100)
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 10 loops, best of 3: 37.8 ms per loop
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014) #back to no C
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 1 loops, best of 3: 7.71 s per loop
```

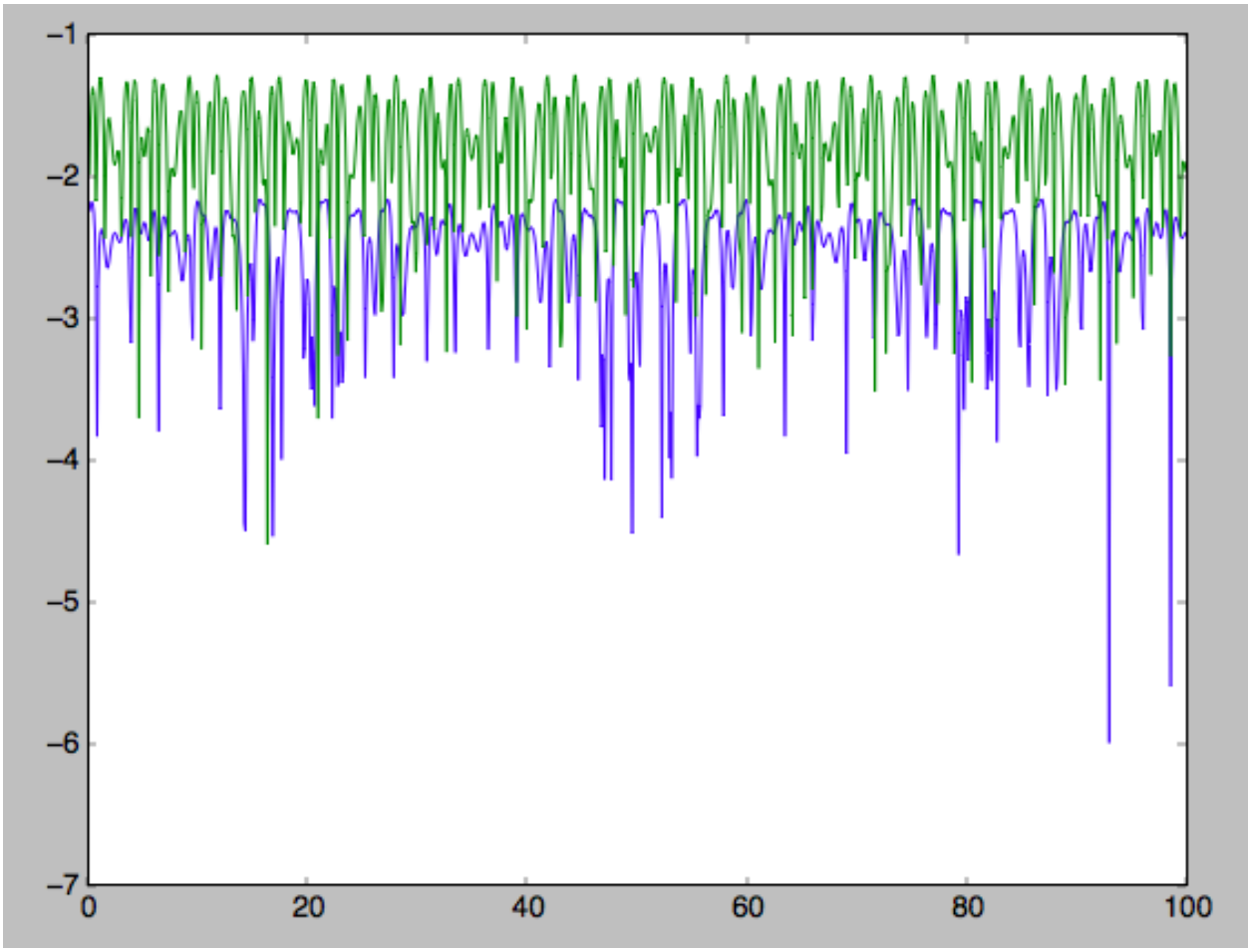
or a speed-up of 200! Back to the previous example, you can run it with `c=True` to speed up the computation

```
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
>>> js= aAA(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts))
```

We can plot the radial- and vertical-action fluctuation as a function of time

```
>>> plot(ts,numpy.log10(numpy.fabs((js[0]-numpy.mean(js[0]))/numpy.mean(js[0]))))
>>> plot(ts,numpy.log10(numpy.fabs((js[2]-numpy.mean(js[2]))/numpy.mean(js[2]))))
```

which gives



The radial action is conserved to about half a percent, the vertical action to two percent.

Another way to speed up the calculation of actions using the adiabatic approximation is to tabulate the actions on a grid in (approximate) integrals of the motion and evaluating new actions by interpolating on this grid. How this is done in practice is described in detail in the galpy paper. To setup this grid-based interpolation method, which is contained in `actionAngleAdiabaticGrid`, do

```
>>> from galpy.actionAngle import actionAngleAdiabaticGrid
>>> aAG= actionAngleAdiabaticGrid(pot=MWPotential2014,nR=31,nEz=31,nEr=51,nLz=51,
↪c=True)
```

where `c=True` specifies that we use the C implementation of `actionAngleAdiabatic` for speed. We can now evaluate in the same was as before, for example

```
>>> aAA(1.,0.1,1.1,0.,0.05), aAG(1.,0.1,1.1,0.,0.05)
# ((array([ 0.01352523]), array([ 1.1]), array([ 0.00046909])),
#  (0.013527010324238781, 1.1, 0.00047747359874375148))
```

which agree very well. To look at the timings, we first switch back to not using C and then list all of the relevant timings:

```
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=False)
# Not using C, direct calculation
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 1 loops, best of 3: 9.05 s per loop
```

(continues on next page)

(continued from previous page)

```
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
# Using C, direct calculation
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 10 loops, best of 3: 39.7 ms per loop
# Grid-based calculation
>>> timeit(aAG(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 1000 loops, best of 3: 1.09 ms per loop
```

Thus, in this example (and more generally) the grid-based calculation is significantly faster than even the direct implementation in C. The overall speed up between the direct Python version and the grid-based version is larger than 8,000; the speed up between the direct C version and the grid-based version is 36. For larger arrays of input phase-space positions, the latter speed up can increase to 150. For simpler, fully analytical potentials the speed up will be slightly less, but for `MWPotential2014` and other more complicated potentials (such as those involving a double-exponential disk), the overhead of setting up the grid is worth it when evaluating more than a few thousand actions.

The adiabatic approximation works well for orbits that stay close to the plane. The orbit we have been considering so far only reaches a height two percent of R_0, or about 150 pc for $R_0 = 8$ kpc.

```
>>> o.zmax()*8.
# 0.17903686455491979
```

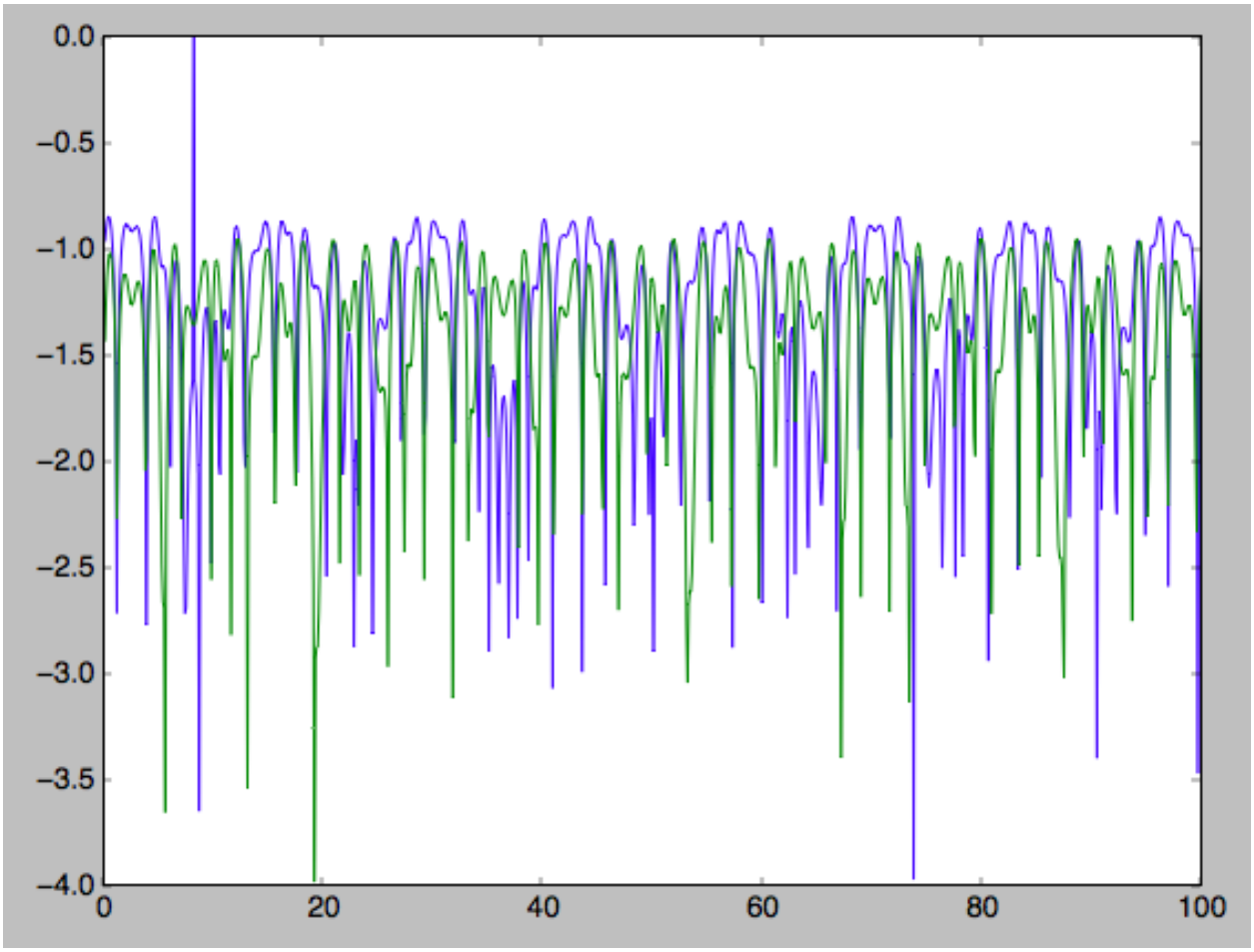
For orbits that reach distances of a kpc and more from the plane, the adiabatic approximation does not work as well. For example,

```
>>> o= Orbit([1.,0.1,1.1,0.,0.25])
>>> o.integrate(ts,MWPotential2014)
>>> o.zmax()*8.
# 1.3506059038621048
```

and we can again calculate the actions along the orbit

```
>>> js= aAA(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts))
>>> plot(ts,numpy.log10(numpy.fabs((js[0]-numpy.mean(js[0]))/numpy.mean(js[0]))))
>>> plot(ts,numpy.log10(numpy.fabs((js[2]-numpy.mean(js[2]))/numpy.mean(js[2]))))
```

which gives



The radial action is now only conserved to about ten percent and the vertical action to approximately five percent.

Warning: Frequencies and angles using the adiabatic approximation are not implemented at this time.

1.7.4 Action-angle coordinates using the Staeckel approximation

A better approximation than the adiabatic one is to locally approximate the potential as a Staeckel potential, for which actions, frequencies, and angles can be calculated through numerical integration. galpy contains an implementation of the algorithm of Binney (2012; 2012MNRAS.426.1324B), which accomplishes the Staeckel approximation for disk-like (i.e., oblate) potentials without explicitly fitting a Staeckel potential. For all intents and purposes the adiabatic approximation is made obsolete by this new method, which is as fast and more precise. The only advantage of the adiabatic approximation over the Staeckel approximation is that the Staeckel approximation requires the user to specify a *focal length* Δ to be used in the Staeckel approximation. However, this focal length can be easily estimated from the second derivatives of the potential (see Sanders 2012; 2012MNRAS.426..128S).

Starting from the second orbit example in the adiabatic section above, we first estimate a good focal length of the `MWPotential2014` to use in the Staeckel approximation. We do this by averaging (through the median) estimates at positions around the orbit (which we integrated in the example above)

```
>>> from galpy.actionAngle import estimateDeltaStaeckel
>>> estimateDeltaStaeckel(MWPotential2014,o.R(ts),o.z(ts))
# 0.40272708556203662
```

We will use $\Delta = 0.4$ in what follows. We set up the `actionAngleStaeckel` object

```
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=False) #c=True is the␣
↪default
```

and calculate the actions

```
>>> aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz())
# (0.019212848866725911, 1.1000000000000001, 0.015274597971510892)
```

The adiabatic approximation from above gives

```
>>> aAA(o.R(),o.vR(),o.vT(),o.z(),o.vz())
# (array([ 0.01686478]), array([ 1.1]), array([ 0.01590001]))
```

The actionAngleStaeckel calculations are sped up in two ways. First, the action integrals can be calculated using Gaussian quadrature by specifying `fixed_quad=True`

```
>>> aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz(),fixed_quad=True)
# (0.01922167296633687, 1.1000000000000001, 0.015276825017286706)
```

which in itself leads to a ten times speed up

```
>>> timeit(aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz(),fixed_quad=False))
# 10 loops, best of 3: 129 ms per loop
>>> timeit(aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz(),fixed_quad=True))
# 100 loops, best of 3: 10.3 ms per loop
```

Second, the actionAngleStaeckel calculations have also been implemented in C, which leads to even greater speed-ups, especially for arrays

```
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=True)
>>> s= numpy.ones(100)
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 10 loops, best of 3: 35.1 ms per loop
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=False) #back to no C
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s,fixed_quad=True))
# 1 loops, best of 3: 496 ms per loop
```

or a fifteen times speed up. The speed up is not that large because the bulge model in `MWPotential2014` requires expensive special functions to be evaluated. Computations could be sped up ten times more when using a simpler bulge model.

Similar to `actionAngleAdiabaticGrid`, we can also tabulate the actions on a grid of (approximate) integrals of the motion and interpolate over this look-up table when evaluating new actions. The details of how this look-up table is setup and used are again fully explained in the galpy paper. To use this grid-based Staeckel approximation, contained in `actionAngleStaeckelGrid`, do

```
>>> from galpy.actionAngle import actionAngleStaeckelGrid
>>> aASG= actionAngleStaeckelGrid(pot=MWPotential2014,delta=0.4,nE=51,npsi=51,nLz=61,
↪c=True)
```

where `c=True` makes sure that we use the C implementation of the Staeckel method to calculate the grid. Because this is a fully three-dimensional grid, setting up the grid takes longer than it does for the adiabatic method (which only uses two two-dimensional grids). We can then evaluate actions as before

```
>>> aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz()), aASG(o.R(),o.vR(),o.vT(),o.z(),o.vz())
# ((0.019212848866725911, 1.1000000000000001, 0.015274597971510892),
#  (0.019221119033345408, 1.1000000000000001, 0.015022528662310393))
```

These actions agree very well. We can compare the timings of these methods as above

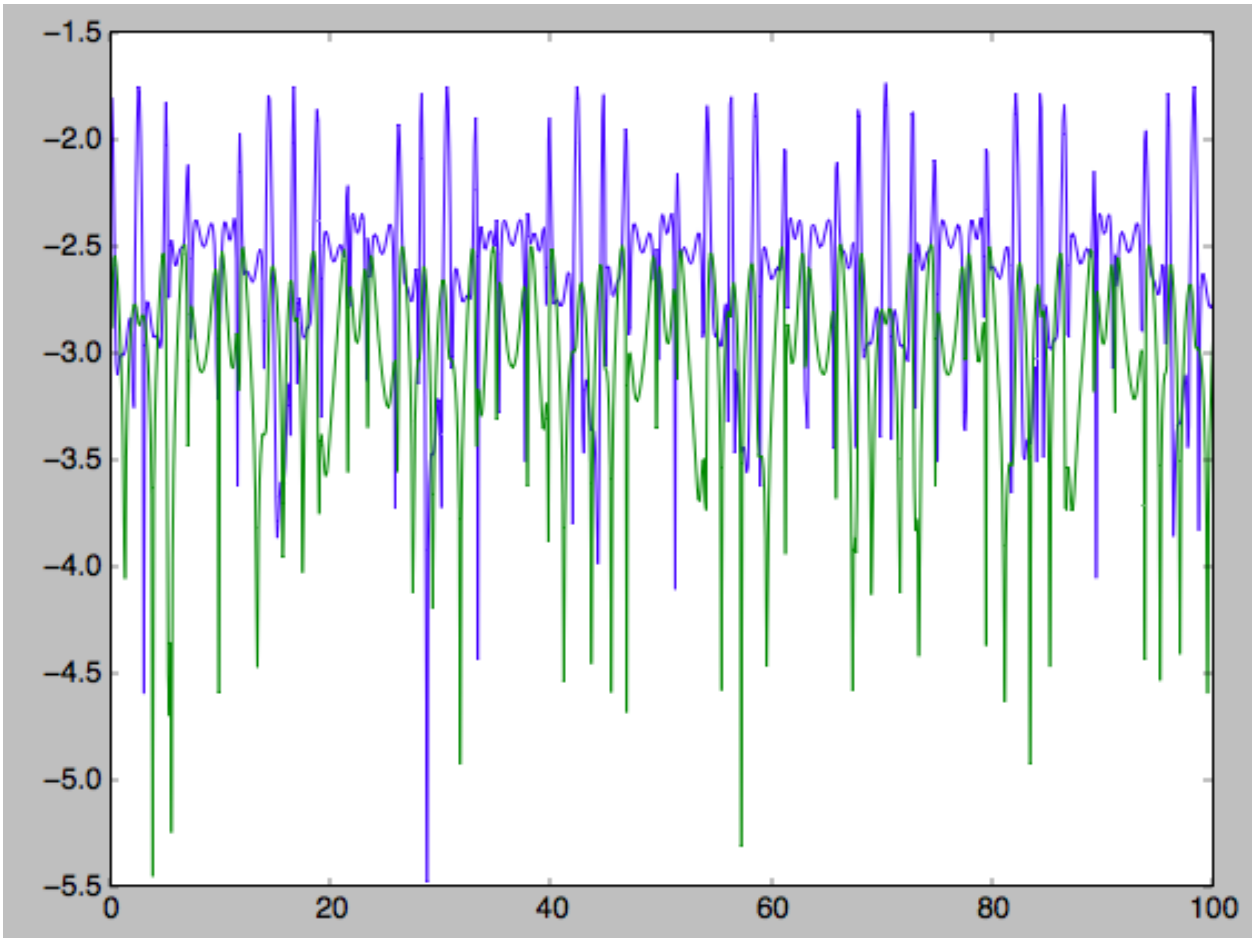
```
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s,fixed_quad=True))
# 1 loops, best of 3: 576 ms per loop # Not using C, direct calculation
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=True)
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 100 loops, best of 3: 17.8 ms per loop # Using C, direct calculation
>>> timeit(aASG(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 100 loops, best of 3: 3.45 ms per loop # Grid-based calculation
```

This demonstrates that the grid-based interpolation again leeds to a significant speed up, even over the C implementation of the direct calculation. This speed up becomes more significant for larger array input, although it saturates at about 25 times (at least for `MWPotential2014`).

We can now go back to checking that the actions are conserved along the orbit (going back to the `c=False` version of `actionAngleStaeckel`)

```
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=False)
>>> js= aAS(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),fixed_quad=True)
>>> plot(ts,numpy.log10(numpy.fabs((js[0]-numpy.mean(js[0]))/numpy.mean(js[0]))))
>>> plot(ts,numpy.log10(numpy.fabs((js[2]-numpy.mean(js[2]))/numpy.mean(js[2]))))
```

which gives



The radial action is now conserved to better than a percent and the vertical action to only a fraction of a percent. Clearly, this is much better than the five to ten percent errors found for the adiabatic approximation above.

For the Staeckel approximation we can also calculate frequencies and angles through the `actionsFreqs` and `actionsFreqsAngles` methods.

Warning: Frequencies and angles using the Staeckel approximation are *only* implemented in C. So use `c=True` in the setup of the actionAngleStaeckel object.

Warning: Angles using the Staeckel approximation in galpy are such that (a) the radial angle starts at zero at pericenter and increases then going toward apocenter; (b) the vertical angle starts at zero at $z=0$ and increases toward positive zmax. The latter is a different convention from that in Binney (2012), but is consistent with that in actionAngleIsochrone and actionAngleSpherical.

```
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=True)
>>> o= Orbit([1.,0.1,1.1,0.,0.25,0.]) #need to specify phi for angles
>>> aAS.actionsFreqsAngles(o.R(),o.vR(),o.vT(),o.z(),o.vz(),o.phi())
# (array([ 0.01922167]),
#  array([ 1.1]),
#  array([ 0.01527683]),
#  array([ 1.11317796]),
```

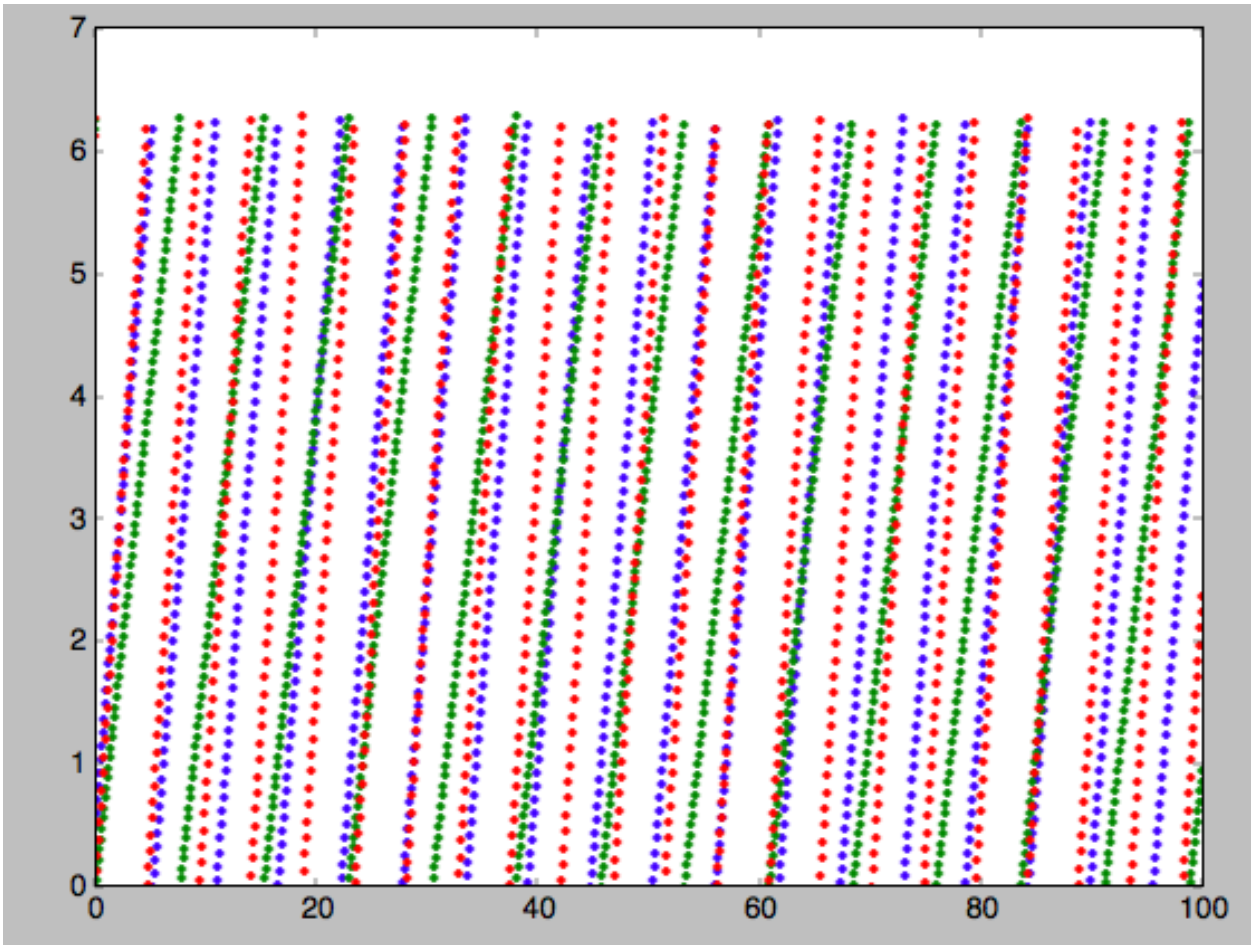
(continues on next page)

(continued from previous page)

```
#  array([ 0.82538032]),
#  array([ 1.34126138]),
#  array([ 0.37758087]),
#  array([ 6.17833493]),
#  array([ 6.13368239]))
```

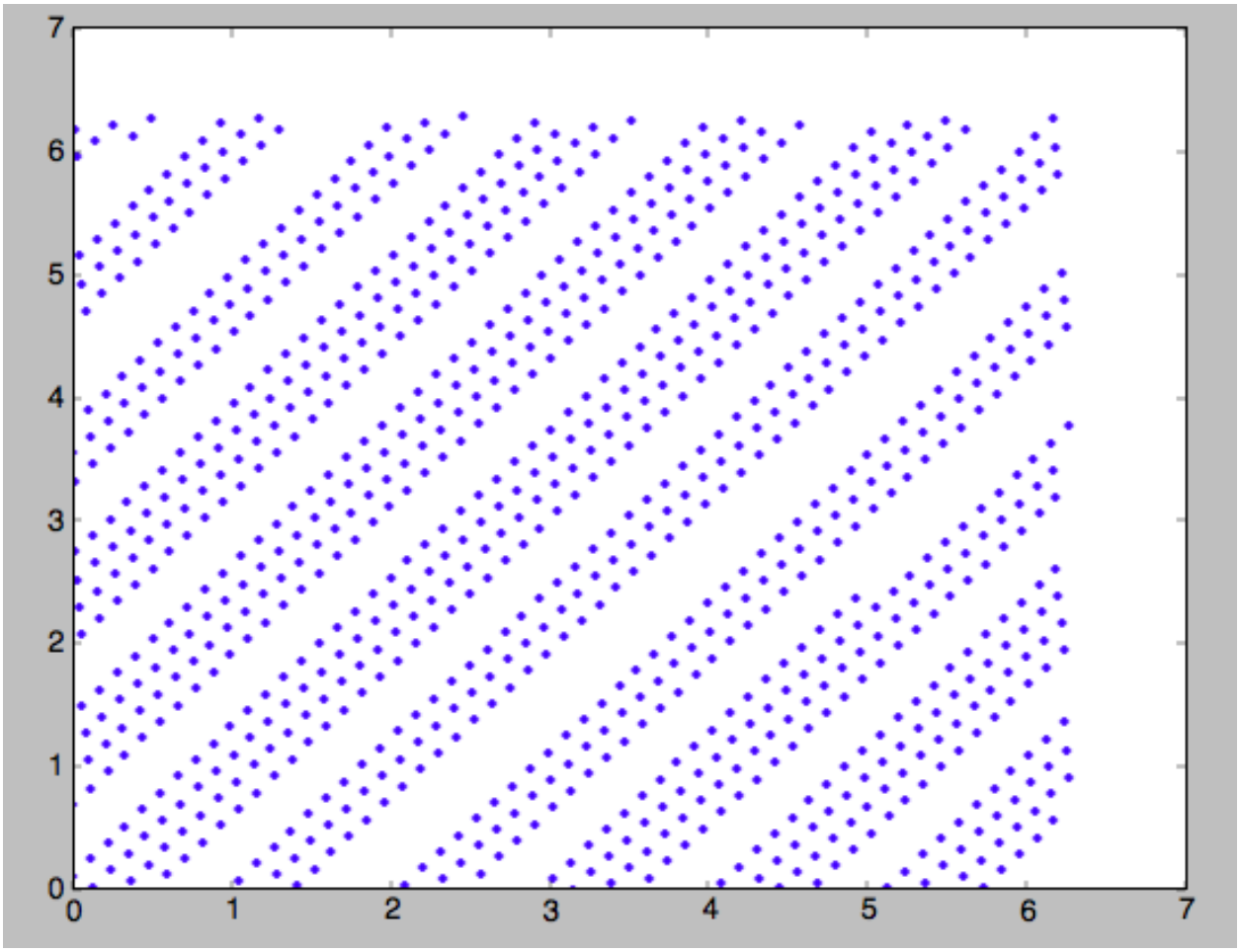
and we can check that the angles increase linearly along the orbit

```
>>> o.integrate(ts,MWPotential2014)
>>> jfa= aAS.actionsFreqsAngles(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),o.phi(ts))
>>> plot(ts,jfa[6],'b.')
>>> plot(ts,jfa[7],'g.')
>>> plot(ts,jfa[8],'r.')
```



or

```
>>> plot(jfa[6],jfa[8],'b.')
```



1.7.5 Action-angle coordinates using an orbit-integration-based approximation

The adiabatic and Staeckel approximations used above are good for stars on close-to-circular orbits, but they break down for more eccentric orbits (specifically, orbits for which the radial and/or vertical action is of a similar magnitude as the angular momentum). This is because the approximations made to the potential in these methods (that it is separable in R and z for the adiabatic approximation and that it is close to a Staeckel potential for the Staeckel approximation) break down for such orbits. Unfortunately, these methods cannot be refined to provide better approximations for eccentric orbits.

galpy contains a new method for calculating actions, frequencies, and angles that is completely general for any static potential. It can calculate the actions to any desired precision for any orbit in such potentials. The method works by employing an auxiliary isochrone potential and calculates action-angle variables by arithmetic operations on the actions and angles calculated in the auxiliary potential along an orbit (integrated in the true potential). Full details can be found in Appendix A of Bovy (2014).

We setup this method for a logarithmic potential as follows

```
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

`b=0.8` here sets the scale parameter of the auxiliary isochrone potential; this potential can also be specified as an

IsochronePotential instance through `ip=`). We can now calculate the actions for an orbit similar to that of the GD-1 stream

```
>>> obs= numpy.array([1.56148083,0.35081535,-1.15481504,0.88719443,-0.47713334,0.
↪12019596]) #orbit similar to GD-1
>>> aAIA(*obs)
# (array([ 0.16605011]), array([-1.80322155]), array([ 0.50704439]))
```

An essential requirement of this method is that the angles calculated in the auxiliary potential go through the full range $[0, 2\pi]$. If this is not the case, galpy will raise a warning

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=10.8)
>>> aAIA(*obs)
# galpyWarning: Full radial angle range not covered for at least one object; actions␣
↪are likely not reliable
# (array([ 0.08985167]), array([-1.80322155]), array([ 0.50849276]))
```

Therefore, some care should be taken to choosing a good auxiliary potential. galpy contains a method to estimate a decent scale parameter for the auxiliary scale parameter, which works similar to `estimateDeltaStaeckel` above except that it also gives a minimum and maximum b if multiple R and z are given

```
>>> from galpy.actionAngle import estimateBIsochrone
>>> from galpy.orbit import Orbit
>>> o= Orbit(obs)
>>> ts= numpy.linspace(0.,100.,1001)
>>> o.integrate(ts,lp)
>>> estimateBIsochrone(lp,o.R(ts),o.z(ts))
# (0.78065062339131952, 1.2265541473461612, 1.4899326335155412) #bmin,bmedian,bmax␣
↪over the orbit
```

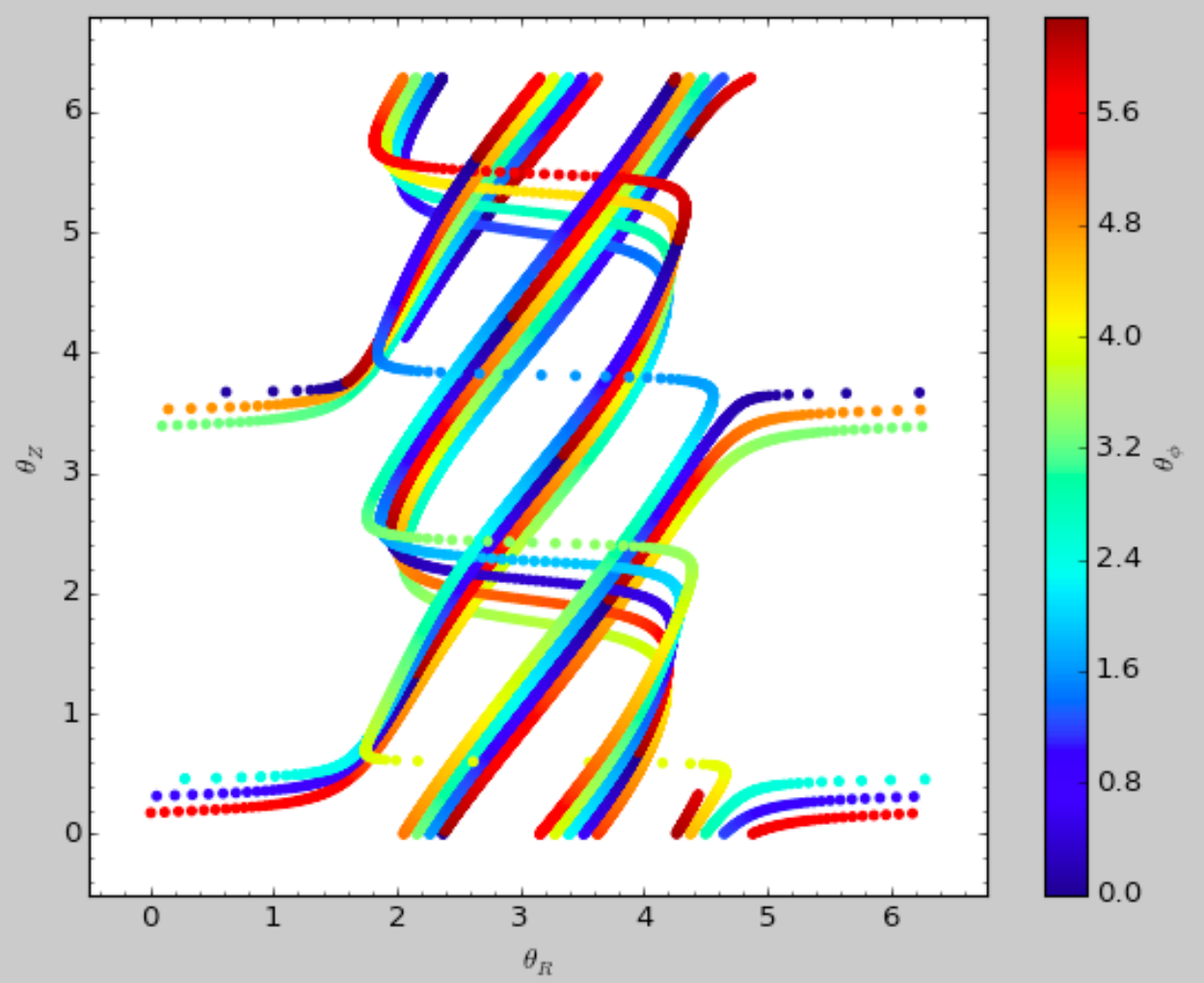
Experience shows that a scale parameter somewhere in the range returned by this function makes sure that the angles go through the full $[0, 2\pi]$ range. However, even if the angles go through the full range, the closer the angles increase to linear, the better the converenge of the algorithm is (and especially, the more accurate the calculation of the frequencies and angles is, see below). For example, for the scale parameter at the upper and of the range

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=1.5)
>>> aAIA(*obs)
# (array([ 0.01120145]), array([-1.80322155]), array([ 0.50788893]))
```

which does not agree with the previous calculation. We can inspect how the angles increase and how the actions converge by using the `aAIA.plot` function. For example, we can plot the radial versus the vertical angle in the auxiliary potential

```
>>> aAIA.plot(*obs,type='araz')
```

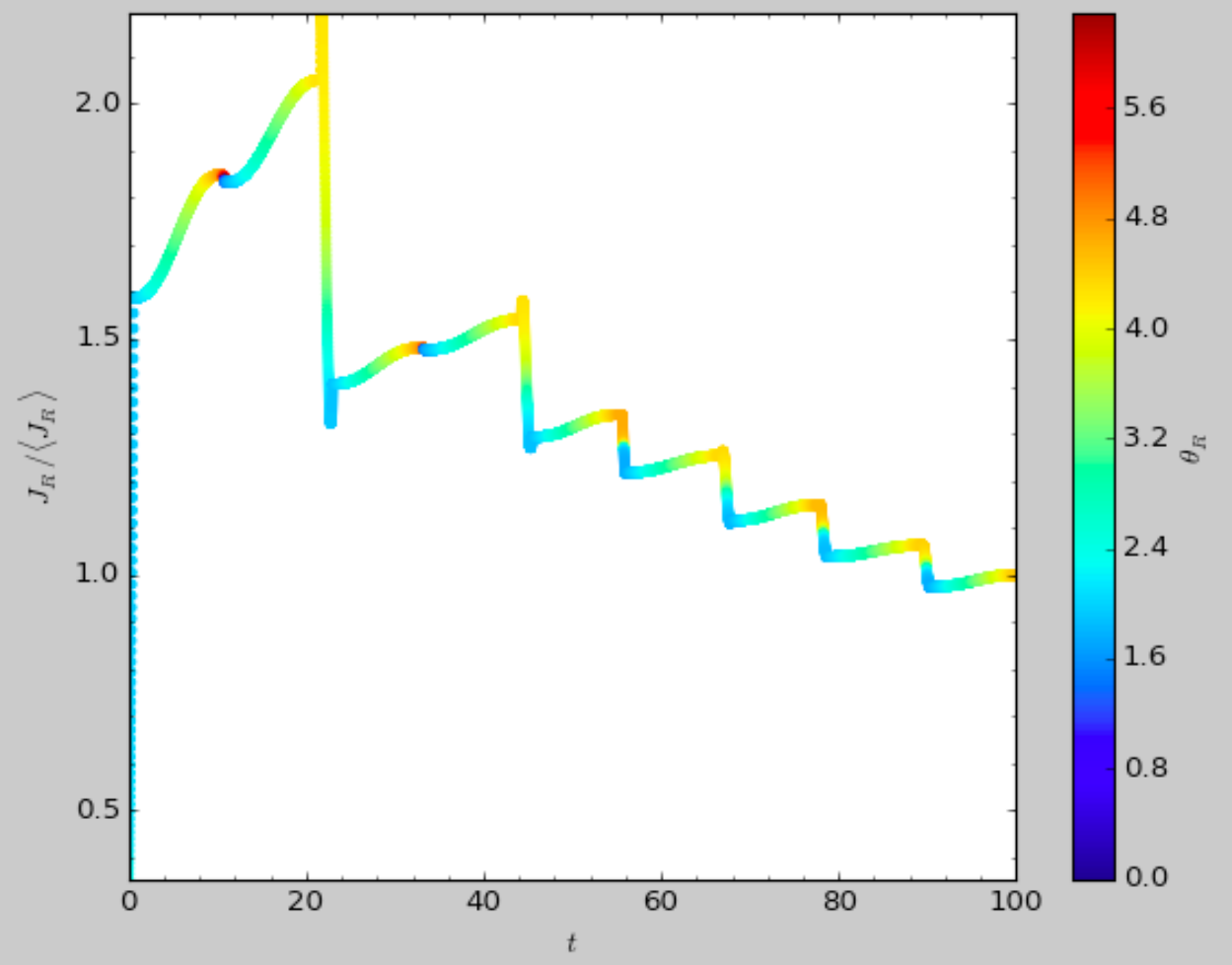
which gives



and this clearly shows that the angles increase *very* non-linearly, because the auxiliary isochrone potential used is too far from the real potential. This causes the actions to converge only very slowly. For example, for the radial action we can plot the converge as a function of integration time

```
>>> aAIA.plot(*obs,type='jr')
```

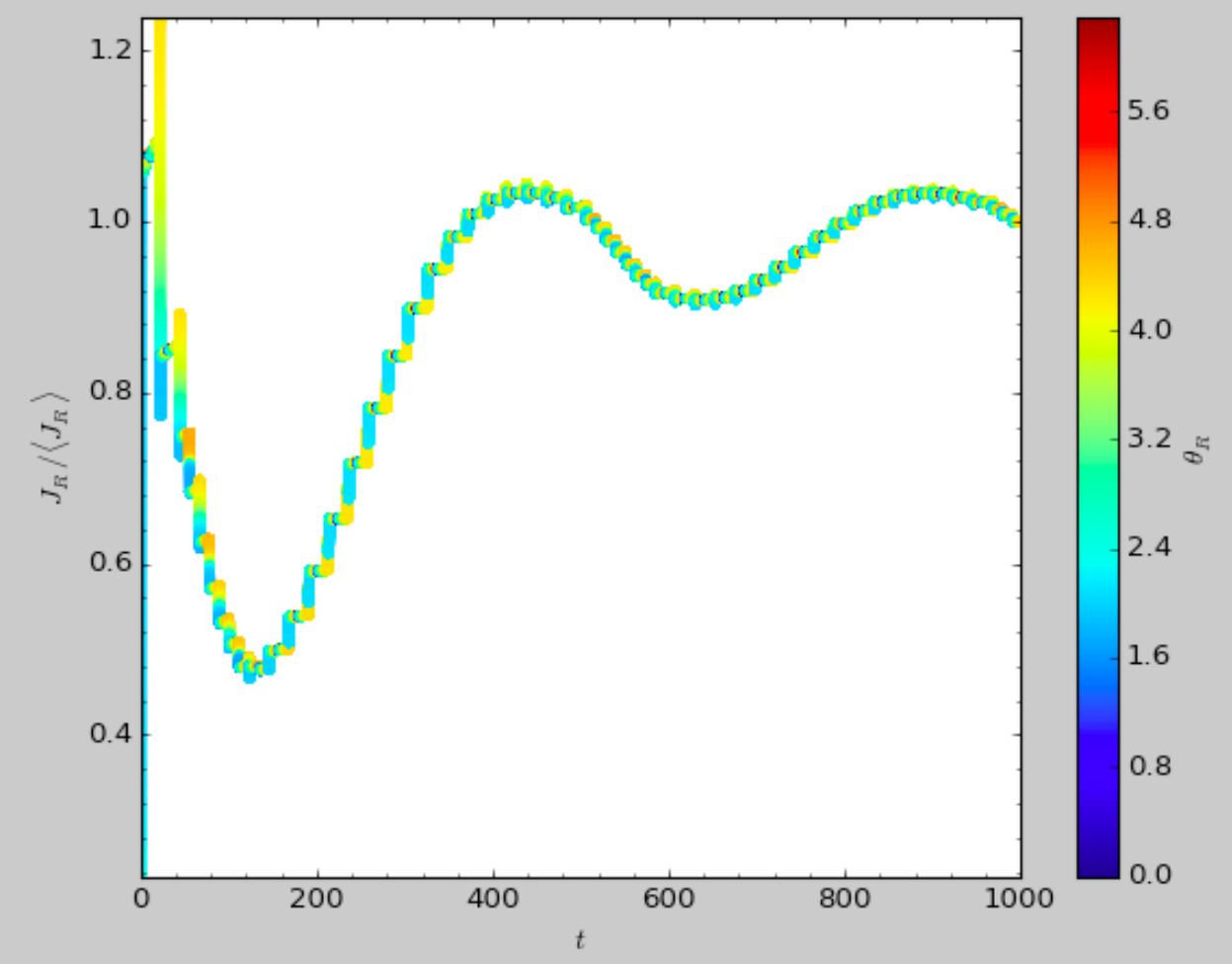
which gives



This Figure clearly shows that the radial action has not converged yet. We need to integrate *much* longer in this auxiliary potential to obtain convergence and because the angles increase so non-linearly, we also need to integrate the orbit much more finely:

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=1.5,tintJ=1000,ntintJ=800000)
>>> aAIA(*obs)
# (array([ 0.01711635]), array([-1.80322155]), array([ 0.51008058]))
>>> aAIA.plot(*obs,type='jr')
```

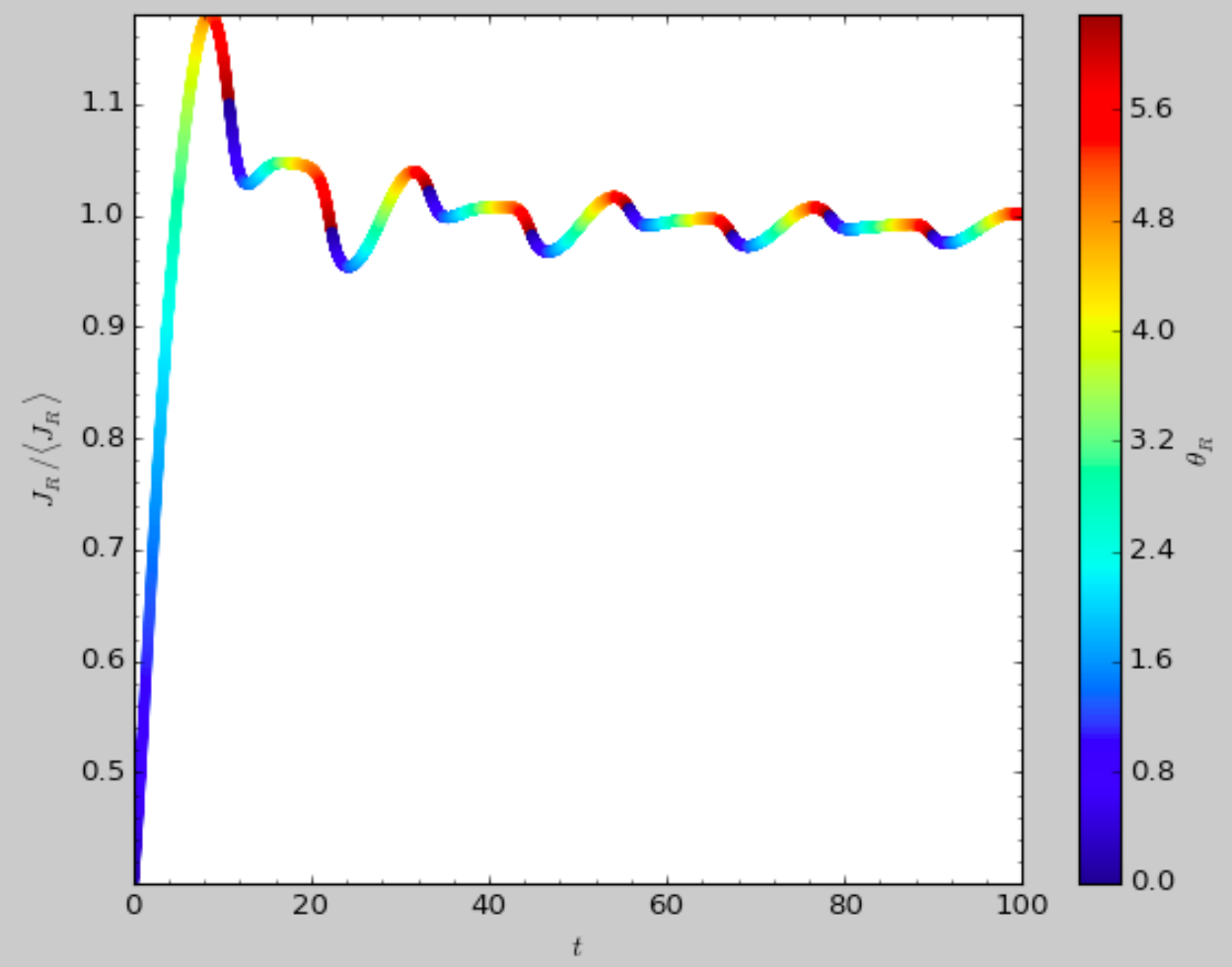
which shows slow convergence



Finding a better auxiliary potential makes convergence *much* faster and also allows the frequencies and the angles to be calculated by removing the small wiggles in the auxiliary angles vs. time (in the angle plot above, the wiggles are much larger, such that removing them is hard). The auxiliary potential used above had `b=0.8`, which shows very quick converenge and good behavior of the angles

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=0.8)
>>> aAIA.plot(*obs,type='jr')
```

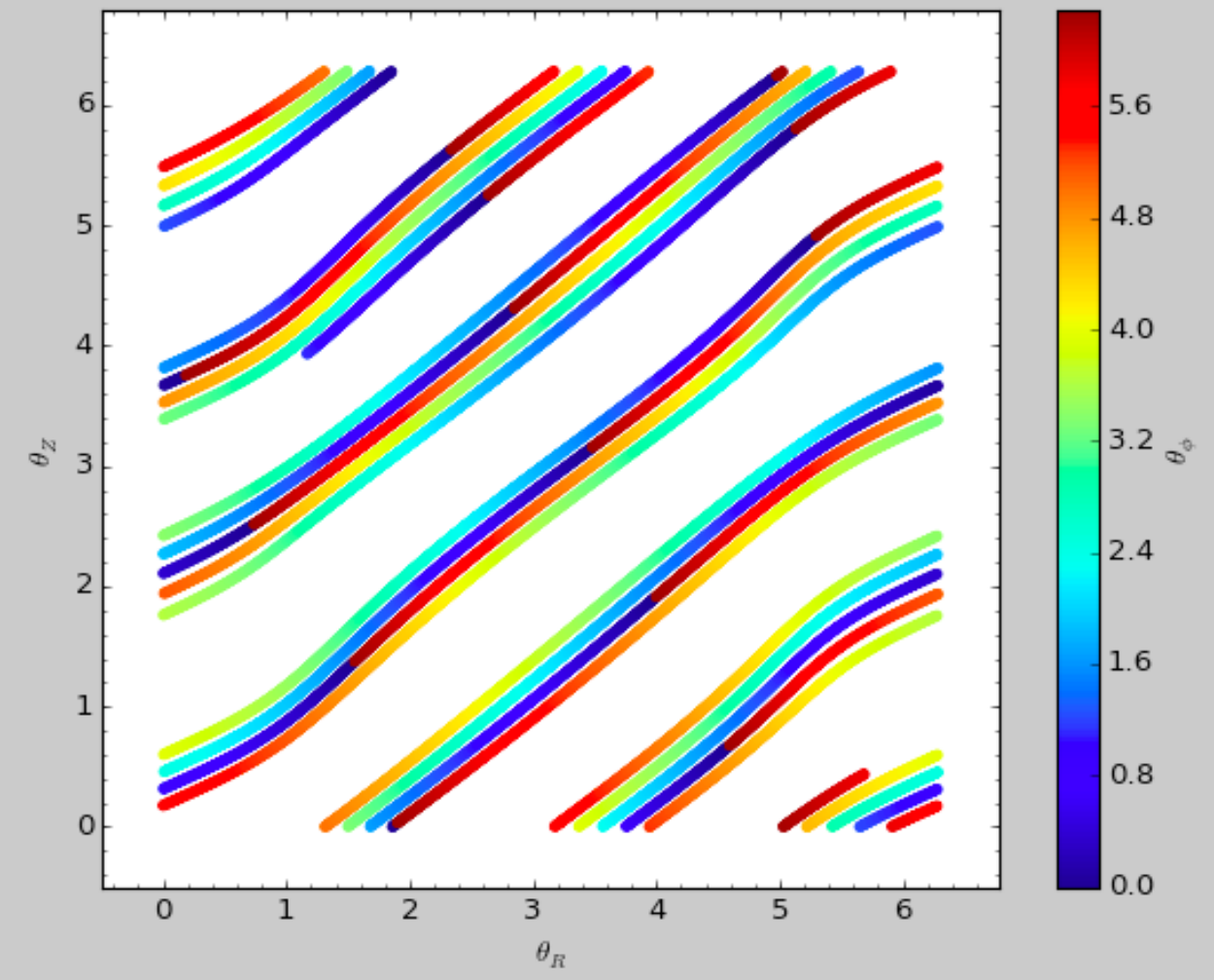
gives



and

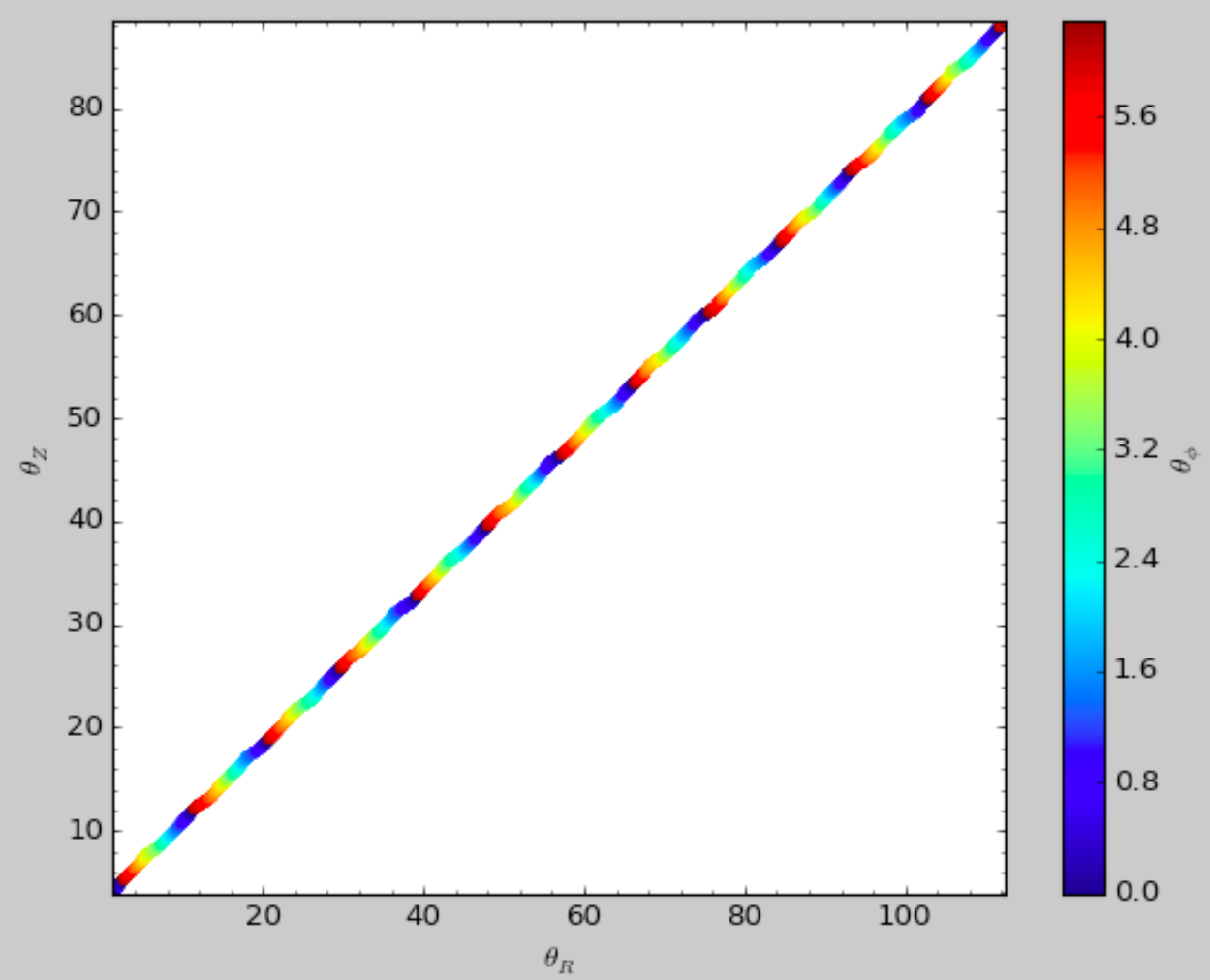
```
>>> aAIA.plot(*obs,type='araz')
```

gives



We can remove the periodic behavior from the angles, which clearly shows that they increase close-to-linear with time

```
>>> aAIA.plot(*obs,type='araz',deperiod=True)
```



We can then calculate the frequencies and the angles for this orbit as

```
>>> aAIA.actionsFreqsAngles(*obs)
# (array([ 0.16392384]),
#  array([-1.80322155]),
#  array([ 0.50999882]),
#  array([ 0.55808933]),
#  array([-0.38475753]),
#  array([ 0.42199713]),
#  array([ 0.18739688]),
#  array([ 0.3131815]),
#  array([ 2.18425661]))
```

This function takes as an argument `maxn=` the maximum n for which to remove sinusoidal wiggles. So we can raise this, for example to 4 from 3

```
>>> aAIA.actionsFreqsAngles(*obs,maxn=4)
# (array([ 0.16392384]),
#  array([-1.80322155]),
#  array([ 0.50999882]),
#  array([ 0.55808776]),
#  array([-0.38475733]),
#  array([ 0.4219968]),
```

(continues on next page)

(continued from previous page)

```
#  array([ 0.18732009]),
#  array([ 0.31318534]),
#  array([ 2.18421296]))
```

Clearly, there is very little change, as most of the wiggles are of low n.

This technique also works for triaxial potentials, but using those requires the code to also use the azimuthal angle variable in the auxiliary potential (this is unnecessary in axisymmetric potentials as the z component of the angular momentum is conserved). We can calculate actions for triaxial potentials by specifying that `nonaxi=True`:

```
>>> aAIA(*obs,nonaxi=True)
# (array([ 0.16605011]), array([-1.80322155]), array([ 0.50704439]))
```

1.7.6 Action-angle coordinates using the TorusMapper code

All of the methods described so far allow one to compute the actions, angles, and frequencies for a given phase-space location. `galpy` also contains some support for computing the inverse transformation by using an interface to the TorusMapper code. Currently, this is limited to axisymmetric potentials, because the TorusMapper code is limited to such potentials.

The basic use of this part of `galpy` is to compute an orbit $(R, v_R, v_T, z, v_z, \phi)$ for a given torus, specified by three actions (J_R, L_Z, J_Z) and as many angles along a torus as you want. First we set up an `actionAngleTorus` object

```
>>> from galpy.actionAngle import actionAngleTorus
>>> from galpy.potential import MWPotential2014
>>> aAT= actionAngleTorus(pot=MWPotential2014)
```

To compute an orbit, we first need to compute the frequencies, which we do as follows

```
>>> jr,lz,jz= 0.1,1.1,0.2
>>> Om= aAT.Freqs(jr,lz,jz)
```

This set consists of $(\Omega_R, \Omega_\phi, \Omega_Z, \mathrm{TMerr})$, where the last entry is the exit code of the TorusMapper code (will be printed as a warning when it is non-zero). Then we compute a set of angles that fall along an orbit as $\theta(t) = \theta_0 + \mathbf{\Omega}\, t$ for a set of times t

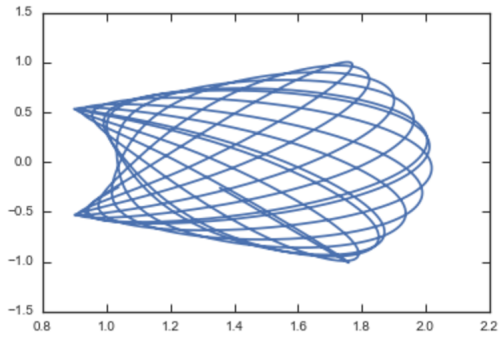
```
>>> times= numpy.linspace(0.,100.,10001)
>>> init_angle= numpy.array([1.,2.,3.])
>>> angles= numpy.tile(init_angle,(len(times),1))+Om[:3]*numpy.tile(times,(3,1)).T
```

Then we can compute the orbit by transforming the orbit in action-angle coordinates to configuration space as follows

```
>>> RvR,_,_,_,_= aAT.xvFreqs(jr,lz,jz,angles[:,0],angles[:,1],angles[:,2])
```

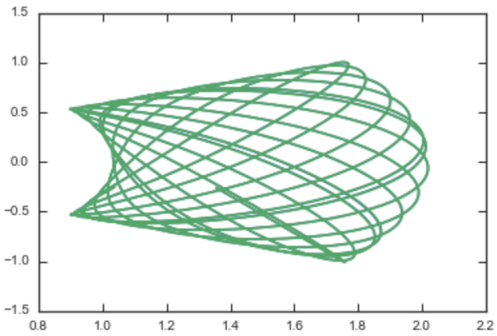
Note that the frequency is also always computed and returned by this method, because it can be obtained at zero cost. The `RvR` array has shape `(ntimes,6)` and the six phase-space coordinates are arranged in the usual `(R,vR,vT,z,vz,phi)` order. The orbit in (R, Z) is then given by

```
>>> plot(RvR[:,0],RvR[:,3])
```



We can compare this to the direct numerical orbit integration. We integrate the orbit, starting at the position and velocity of the initial angle `RvR[0]`

```
>>> from galpy.orbit import Orbit
>>> orb= Orbit(RvR[0])
>>> orb.integrate(times,MWPotential2014)
>>> orb.plot(overplot=True)
```

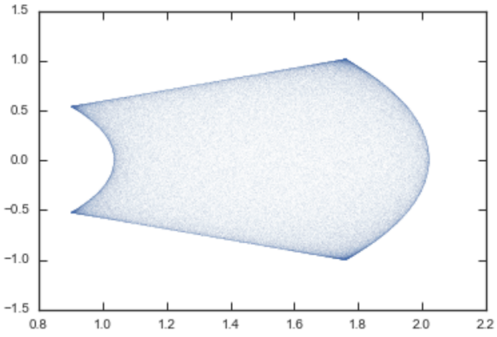


The two orbits are exactly the same.

Of course, we do not have to follow the path of an orbit to map the entire orbital torus and thus reveal the orbital building blocks of galaxies. To directly map a torus, we can do (don't worry, this doesn't take very long)

```
>>> nangles= 200001
>>> angler= numpy.random.uniform(size=nangles)*2.*numpy.pi
>>> anglep= numpy.random.uniform(size=nangles)*2.*numpy.pi
>>> anglez= numpy.random.uniform(size=nangles)*2.*numpy.pi
>>> RvR,_,_,_,_= aAT.xvFreqs(jr,lz,jz,angler,anglep,anglez)
>>> plot(RvR[:,0],RvR[:,3],',',alpha=0.02)
```

which directly shows where the orbit spends most of its time:



`actionAngleTorus` has additional methods documented on the action-angle API page for computing Hessians and Jacobians of the transformation between action-angle and configuration space coordinates.

1.7.7 Accessing action-angle coordinates for Orbit instances

While the most flexible way to access the actionAngle routines is through the methods in the `galpy.actionAngle` modules, action-angle coordinates can also be calculated for `galpy.orbit.Orbit` instances and this is often more convenient. This is illustrated here briefly. We initialize an Orbit instance

```
>>> from galpy.orbit import Orbit
>>> from galpy.potential import MWPotential2014
>>> o= Orbit([1.,0.1,1.1,0.,0.25,0.])
```

and we can then calculate the actions (default is to use the staeckel approximation with an automatically-estimated delta parameter, but this can be adjusted)

```
>>> o.jr(pot=MWPotential2014), o.jp(pot=MWPotential2014), o.jz(pot=MWPotential2014)
# (0.018194068808944613,1.1,0.01540155584446606)
```

`o.jp` here gives the azimuthal action (which is the z component of the angular momentum for axisymmetric potentials). We can also use the other methods described above or adjust the parameters of the approximation (see above):

```
>>> o.jr(pot=MWPotential2014,type='staeckel',delta=0.4), o.jp(pot=MWPotential2014,
↪type='staeckel',delta=0.4), o.jz(pot=MWPotential2014,type='staeckel',delta=0.4)
# (0.019221672966336707, 1.1, 0.015276825017286827)
>>> o.jr(pot=MWPotential2014,type='adiabatic'), o.jp(pot=MWPotential2014,type=
↪'adiabatic'), o.jz(pot=MWPotential2014,type='adiabatic')
# (0.016856430059017123, 1.1, 0.015897730620467752)
>>> o.jr(pot=MWPotential2014,type='isochroneApprox',b=0.8), o.jp(pot=MWPotential2014,
↪type='isochroneApprox',b=0.8), o.jz(pot=MWPotential2014,type='isochroneApprox',b=0.
↪8)
# (0.019066091295488922, 1.1, 0.015280492319332751)
```

These two methods give very precise actions for this orbit (both are converged to about 1%) and they agree very well

```
>>> (o.jr(pot=MWPotential2014,type='staeckel',delta=0.4)-o.jr(pot=MWPotential2014,
↪type='isochroneApprox',b=0.8))/o.jr(pot=MWPotential2014,type='isochroneApprox',b=0.
↪8)
# 0.00816012408818143
>>> (o.jz(pot=MWPotential2014,type='staeckel',delta=0.4)-o.jz(pot=MWPotential2014,
↪type='isochroneApprox',b=0.8))/o.jz(pot=MWPotential2014,type='isochroneApprox',b=0.
↪8)
# 0.00023999894566772273
```

We can also calculate the frequencies and the angles. This requires using the Staeckel or Isochrone approximations, because frequencies and angles are currently not supported for the adiabatic approximation. For example, the radial frequency

```
>>> o.Or(pot=MWPotential2014,type='staeckel',delta=0.4)
# 1.1131779637307115
>>> o.Or(pot=MWPotential2014,type='isochroneApprox',b=0.8)
# 1.1134635974560649
```

and the radial angle

```
>>> o.wr(pot=MWPotential2014,type='staeckel',delta=0.4)
# 0.37758086786371969
>>> o.wr(pot=MWPotential2014,type='isochroneApprox',b=0.8)
# 0.38159809018175395
```

which again agree to 1%. We can also calculate the other frequencies, angles, as well as periods using the functions `o.Op`, `o.oz`, `o.wp`, `o.wz`, `o.Tr`, `o.Tp`, `o.Tz`.

All of the functions above also work for `Orbit` instances that contain multiple objects. This is particularly convenient if you have data in observed coordinates (e.g., RA, Dec, etc.), for example,

```
>>> numpy.random.seed(1)
>>> nrand= 30
>>> ras= numpy.random.uniform(size=nrand)*360.*u.deg
>>> decs= 90.*(2.*numpy.random.uniform(size=nrand)-1.)*u.deg
>>> dists= numpy.random.uniform(size=nrand)*10.*u.kpc
>>> pmras= 2.*(2.*numpy.random.uniform(size=nrand)-1.)*u.mas/u.yr
>>> pmdecs= 2.*(2.*numpy.random.uniform(size=nrand)-1.)*u.mas/u.yr
>>> vloss= 200.*(2.*numpy.random.uniform(size=nrand)-1.)*u.km/u.s
>>> co= SkyCoord(ra=ras,dec=decs,distance=dists,
                 pm_ra_cosdec=pmras,pm_dec=pmdecs,
                 radial_velocity=vloss,
                 frame='icrs')
>>> orbits= Orbit(co)
>>> orbits.jr(pot=MWPotential2014)
# [2363.7957, 360.12445, 690.32238, 1046.2924, 132.9572, 86.989812, 272.06487, 360.
↪73566, 55.568238, 698.18447, 24.783574, 21.889352, 16.148216, 3870.4286, 743.63456,␣
↪317.66551, 325.93816, 183.86429, 56.087796, 180.42838, 1121.8019, 8700.8335, 977.
↪8525, 7.569396, 8.2847477, 210.72127, 160.9785, 680.63864, 1093.7413, 87.
↪629873]kmkpcs
```

1.7.8 Example: Evidence for a Lindblad resonance in the Solar neighborhood

We can use galpy to calculate action-angle coordinates for a set of stars in the Solar neighborhood and look for unexplained features. For this we download the data from the Geneva-Copenhagen Survey (2009A&A...501..941H; data available at viZier). Since the velocities in this catalog are given as U,V, and W, we use the `radec` and `UVW` keywords to initialize the orbits from the raw data. For each object `ii`

```
>>> o= Orbit(vxvv[ii,:],radec=True,uvw=True,vo=220.,ro=8.)
```

We then calculate the actions and angles for each object in a flat rotation curve potential

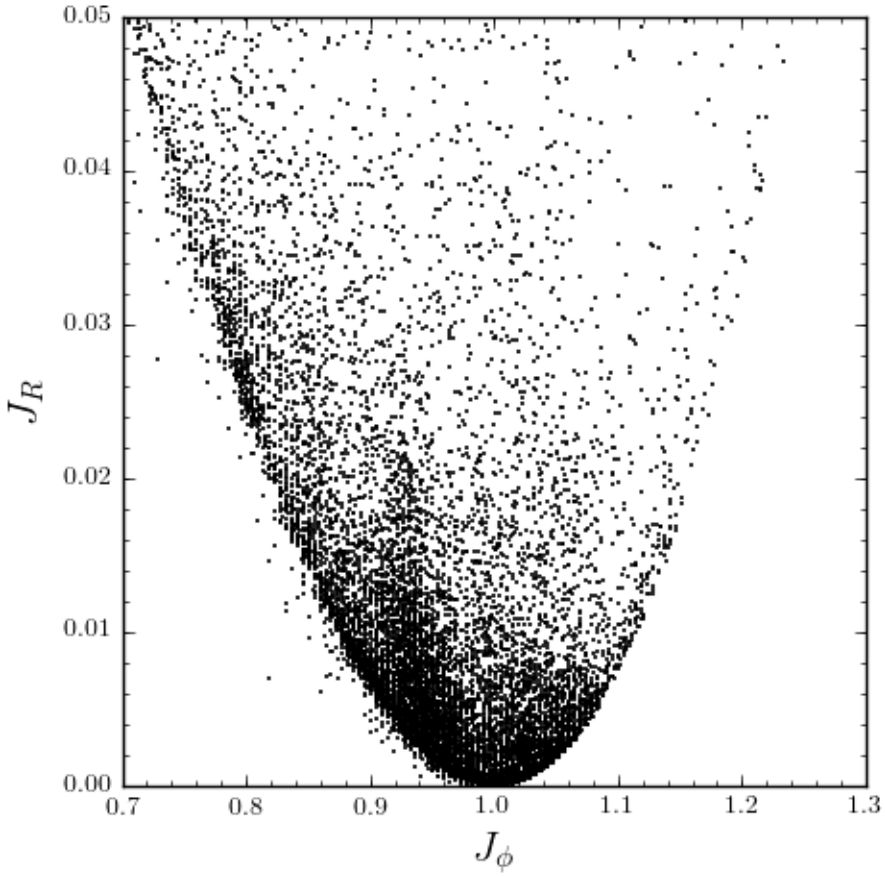
```
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> myjr[ii]= o.jr(lp)
```

etc.

Plotting the radial action versus the angular momentum

```
>>> import galpy.util.plot as galpy_plot
>>> galpy_plot.plot(myjp,myjr,'k.',ms=2.,xlabel=r'$J_{\phi}$',ylabel=r'$J_R$',
↪xrange=[0.7,1.3],yrange=[0.,0.05])
```

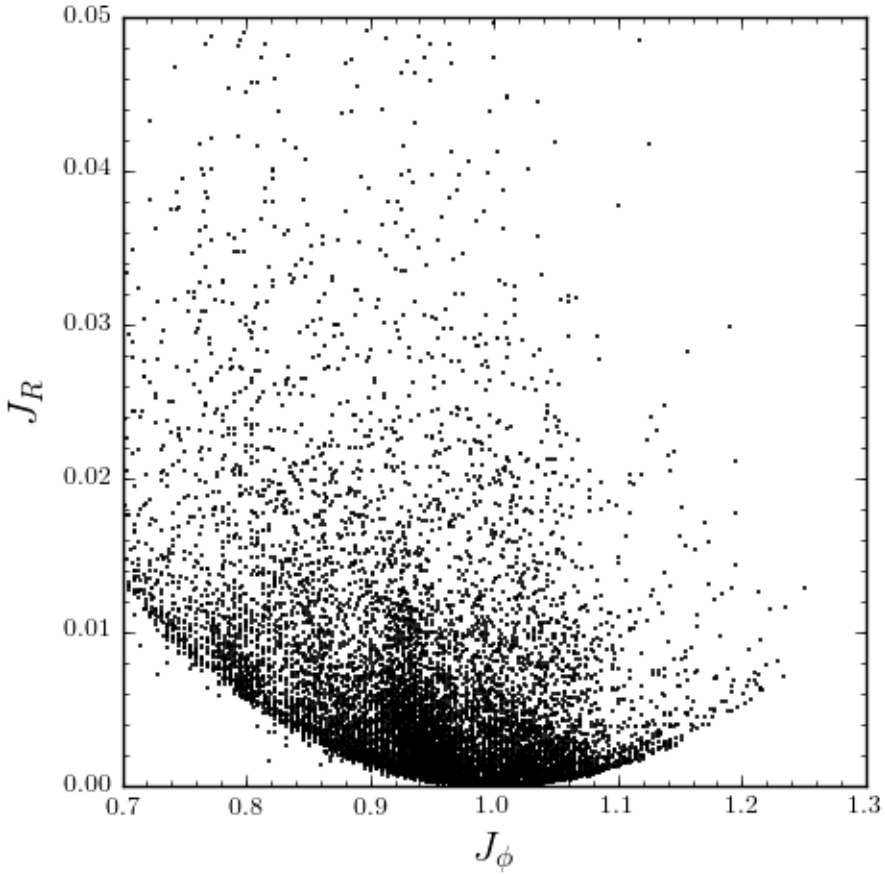
shows a feature in the distribution



If instead we use a power-law rotation curve with power-law index 1

```
>>> pp= PowerSphericalPotential(normalize=1.,alpha=-2.)
>>> myjr[ii]= o.jr(pp)
```

We find that the distribution is stretched, but the feature remains



Code for this example can be found `here` (note that this code uses a particular download of the GCS data set; if you use your own version, you will need to modify the part of the code that reads the data). For more information see 2010MNRAS.409..145S.

1.7.9 Example: actions in an N-body simulation

To illustrate how we can use `galpy` to calculate actions in a snapshot of an N-body simulation, we again look at the `g15784` snapshot in the `pynbody` test suite, discussed in *The potential of N-body simulations*. Please look at that section for information on how to setup the potential of this snapshot in `galpy`. One change is that we should set `enable_c=True` in the instantiation of the `InterpSnapshotRZPotential` object

```
>>> spi= InterpSnapshotRZPotential(h1,rgrid=(numpy.log(0.01),numpy.log(20.),101),
↪logR=True,zgrid=(0.,10.,101),interpPot=True,zsym=True,enable_c=True)
>>> spi.normalize(R0=10.)
```

where we again normalize the potential to use galpy's *natural units*.

We first load a pristine copy of the simulation (because the normalization above leads to some inconsistent behavior in pynbody)

```
>>> sc = pynbody.load('Repos/pynbody-testdata/g15784.lr.01024.gz'); hc = sc.halos();␣
↪hc1= hc[1]; pynbody.analysis.halo.center(hc1,mode='hyb'); pynbody.analysis.angmom.
↪faceon(hc1, cen=(0,0,0),mode='ssc'); sc.physical_units()
```

and then select particles near R=*8* kpc by doing

```
>>> sn= pynbody.filt.BandPass('rxy','7 kpc','9 kpc')
>>> R,vR,vT,z,vz = [numpy.ascontiguousarray(hc1.s[sn][x]) for x in ('rxy','vr','vt','z
↪','vz')]
```

These have physical units, so we normalize them (the velocity normalization is the circular velocity at R=*10* kpc, see *here*).

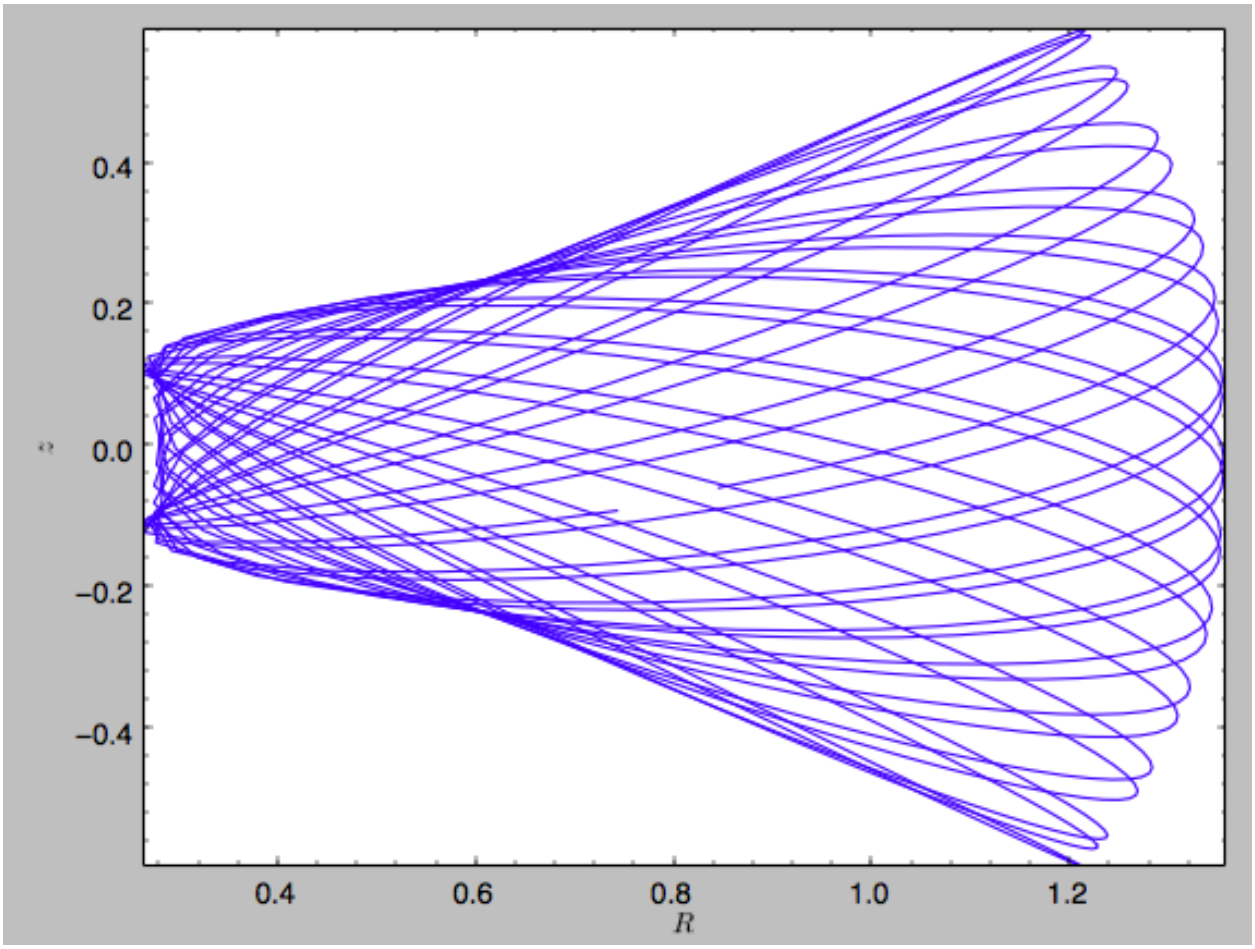
```
>>> ro, vo= 10., 294.62723076942245
>>> R/= ro
>>> z/= ro
>>> vR/= vo
>>> vT/= vo
>>> vz/= vo
```

We will calculate actions using `actionAngleStaeckel` above. We can first integrate a random orbit in this potential

```
>>> from galpy.orbit import Orbit
>>> numpy.random.seed(1)
>>> ii= numpy.random.permutation(len(R))[0]
>>> o= Orbit([R[ii],vR[ii],vT[ii],z[ii],vz[ii]])
>>> ts= numpy.linspace(0.,100.,1001)
>>> o.integrate(ts,spi)
```

This orbit looks like this

```
>>> o.plot()
```



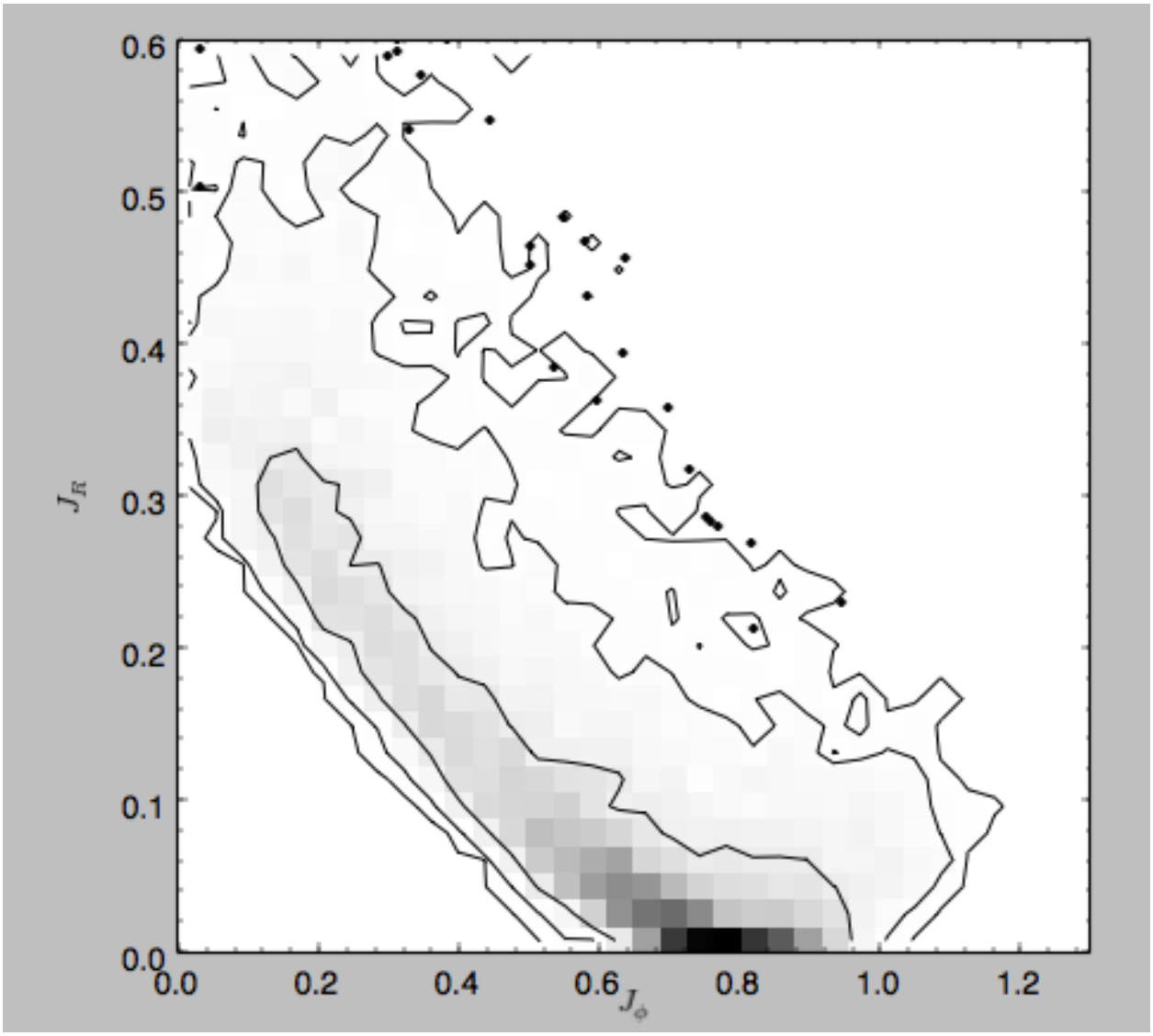
We can now calculate the actions by doing

```
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=spi,delta=0.45,c=True)
>>> jr,lz,jz= aAS(R,vR,vT,z,vz)
```

These actions are also in *natural units*; you can obtain physical units by multiplying with *ro*vo*. We can now plot these actions

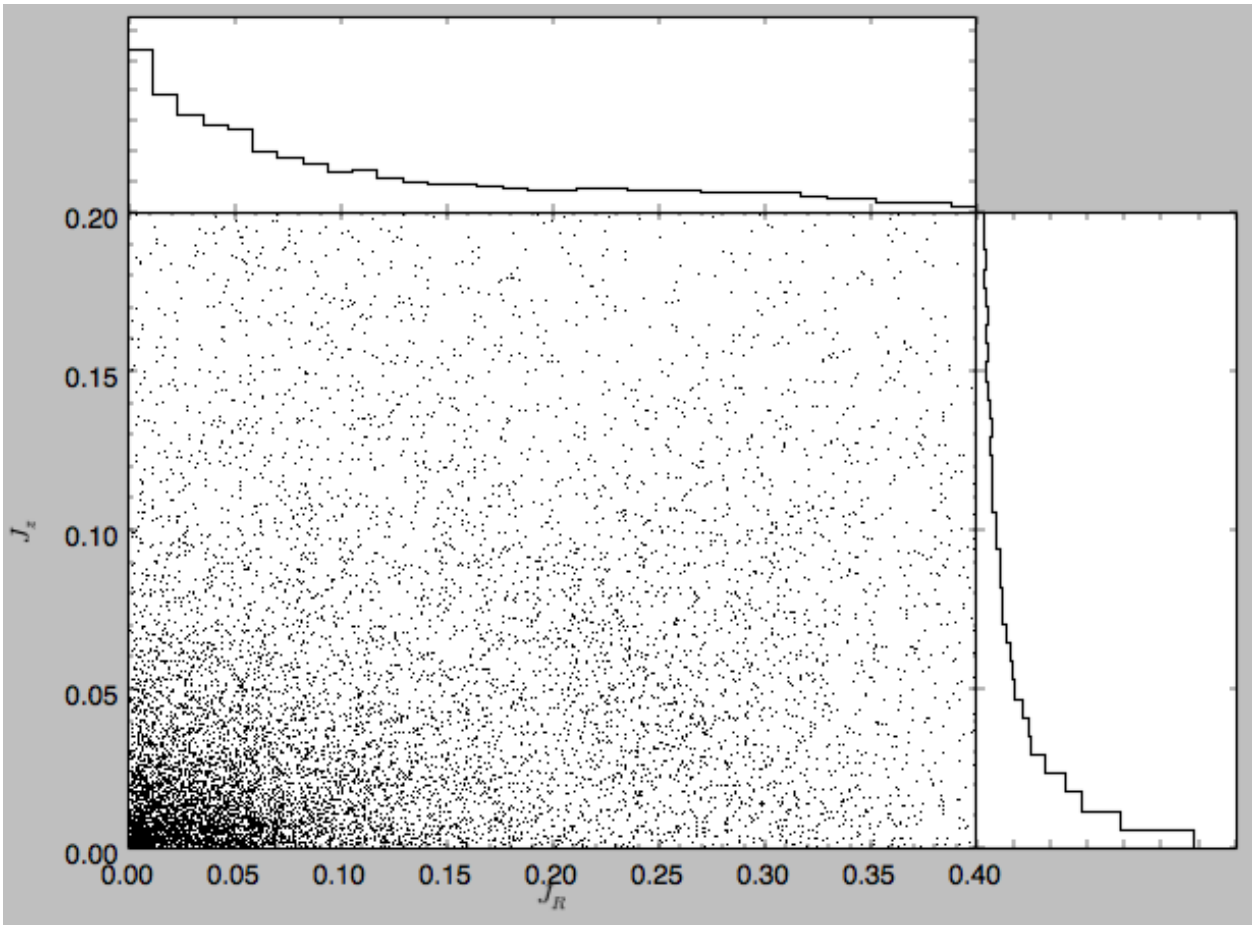
```
>>> from galpy.util import plot as galpy_plot
>>> galpy_plot.scatterplot(lz,jr,'k.',xlabel=r'$J_\phi$',ylabel=r'$J_R$',xrange=[0.,1.
↪3],yrange=[0.,.6])
```

which gives



Note the similarity between this figure and the GCS figure above. The curve shape is due to the selection (low angular momentum stars can only enter the selected radial ring if they are very elliptical and therefore have large radial action) and the density gradient in angular momentum is due to the falling surface density of the disk. We can also look at the distribution of radial and vertical actions.

```
>>> galpy_plot.plot(jr,jz,'k,',xlabel=r'$J_R$',ylabel=r'$J_z$',xrange=[0.,.4],
↪yrange=[0.,0.2],onedhists=True)
```



With the other methods in the actionAngle module we can also calculate frequencies and angles.

1.8 Three-dimensional disk distribution functions

galpy contains a fully three-dimensional disk distribution: `galpy.df.quasiisothermaldf`, which is an approximately isothermal distribution function expressed in terms of action–angle variables (see 2010MNRAS.401.2318B and 2011MNRAS.413.1889B). Recent research shows that this distribution function provides a good model for the DF of mono-abundance sub-populations (MAPs) of the Milky Way disk (see 2013MNRAS.434..652T and 2013ApJ...779..115B). This distribution function family requires action-angle coordinates to evaluate the DF, so `galpy.df.quasiisothermaldf` makes heavy use of the routines in `galpy.actionAngle` (in particular those in `galpy.actionAngleAdiabatic` and `galpy.actionAngle.actionAngleStaeckel`).

1.8.1 Setting up the DF and basic properties

The quasi-isothermal DF is defined by a gravitational potential and a set of parameters describing the radial surface-density profile and the radial and vertical velocity dispersion as a function of radius. In addition, we have to provide an instance of a `galpy.actionAngle` class to calculate the actions for a given position and velocity. For example, for a `galpy.potential.MWPotential2014` potential using the adiabatic approximation for the actions, we import and define the following

```
>>> from galpy.potential import MWPotential2014
>>> from galpy.actionAngle import actionAngleAdiabatic
>>> from galpy.df import quasiisothermaldf
>>> aA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
```

and then setup the `quasiisothermaldf` instance

```
>>> qdf= quasiisothermaldf(1./3.,0.2,0.1,1.,1.,pot=MWPotential2014,aA=aA,
↪cutcounter=True)
```

which sets up a DF instance with a radial scale length of $R_0/3$, a local radial and vertical velocity dispersion of $0.2\,V_c(R_0)$ and $0.1\,V_c(R_0)$, respectively, and a radial scale lengths of the velocity dispersions of R_0. `cutcounter=True` specifies that counter-rotating stars are explicitly excluded (normally these are just exponentially suppressed). As for the two-dimensional disk DFs, these parameters are merely input (or target) parameters; the true density and velocity dispersion profiles calculated by evaluating the relevant moments of the DF (see below) are not exactly exponential and have scale lengths and local normalizations that deviate slightly from these input parameters. We can estimate the DF's actual radial scale length near R_0 as

```
>>> qdf.estimate_hr(1.)
# 0.32908034635647182
```

which is quite close to the input value of 1/3. Similarly, we can estimate the scale lengths of the dispersions

```
>>> qdf.estimate_hsr(1.)
# 1.1913935820372923
>>> qdf.estimate_hsz(1.)
# 1.0506918075359255
```

The vertical profile is fully specified by the velocity dispersions and radial density / dispersion profiles under the assumption of dynamical equilibrium. We can estimate the scale height of this DF at a given radius and height as follows

```
>>> qdf.estimate_hz(1.,0.125)
# 0.021389597757156088
```

Near the mid-plane this vertical scale height becomes very large because the vertical profile flattens, e.g.,

```
>>> qdf.estimate_hz(1.,0.125/100.)
# 1.006386030587223
```

or even

```
>>> qdf.estimate_hz(1.,0.)
# 187649.98447377066
```

which is basically infinity.

1.8.2 Evaluating moments

We can evaluate various moments of the DF giving the density, mean velocities, and velocity dispersions. For example, the mean radial velocity is again everywhere zero because the potential and the DF are axisymmetric

```
>>> qdf.meanvR(1.,0.)
# 0.0
```

Likewise, the mean vertical velocity is everywhere zero

```
>>> qdf.meanvz(1.,0.)
# 0.0
```

The mean rotational velocity has a more interesting dependence on position. Near the plane, this is the same as that calculated for a similar two-dimensional disk DF (see *Evaluating moments of the DF*)

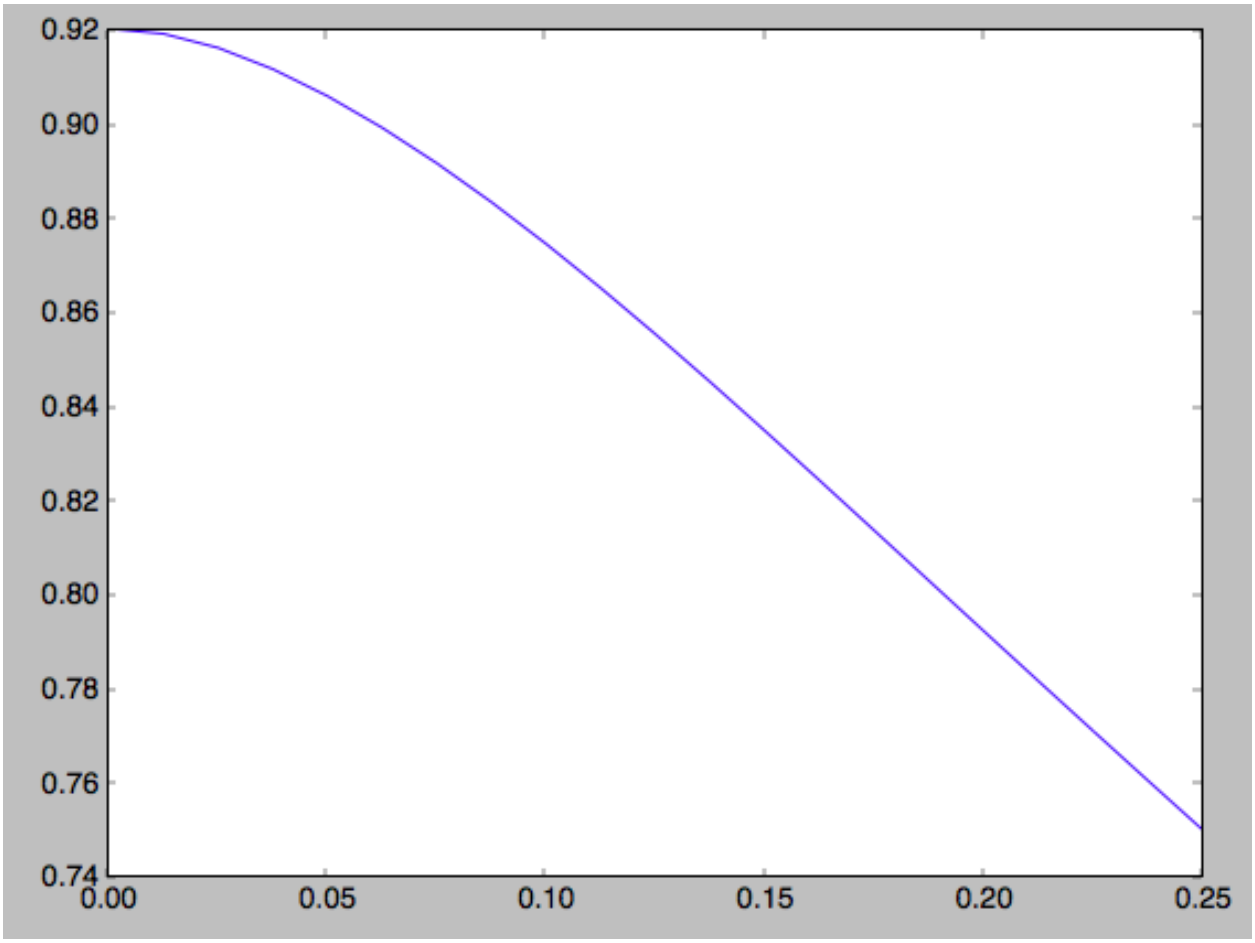
```
>>> qdf.meanvT(1.,0.)
# 0.91988346380781227
```

However, this value decreases as one moves further from the plane. The `quasiisothermaldf` allows us to calculate the average rotational velocity as a function of height above the plane. For example,

```
>>> zs= numpy.linspace(0.,0.25,21)
>>> mvts= numpy.array([qdf.meanvT(1.,z) for z in zs])
```

which gives

```
>>> plot(zs,mvts)
```



We can also calculate the second moments of the DF. We can check whether the radial and velocity dispersions at R_0 are close to their input values

```
>>> numpy.sqrt(qdf.sigmaR2(1.,0.))
# 0.20807112565801389
```

(continues on next page)

(continued from previous page)

```
>>> numpy.sqrt(qdf.sigmaz2(1.,0.))
# 0.090453510526130904
```

and they are pretty close. We can also calculate the mixed R and z moment, for example,

```
>>> qdf.sigmaRz(1.,0.125)
# 0.0
```

or expressed as an angle (the *tilt of the velocity ellipsoid*)

```
>>> qdf.tilt(1.,0.125)
# 0.0
```

This tilt is zero because we are using the adiabatic approximation. As this approximation assumes that the motions in the plane are decoupled from the vertical motions of stars, the mixed moment is zero. However, this approximation is invalid for stars that go far above the plane. By using the Staeckel approximation to calculate the actions, we can model this coupling better. Setting up a `quasiisothermaldf` instance with the Staeckel approximation

```
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.45,c=True)
>>> qdfS= quasiisothermaldf(1./3.,0.2,0.1,1.,1.,pot=MWPotential2014,aA=aAS,
↪cutcounter=True)
```

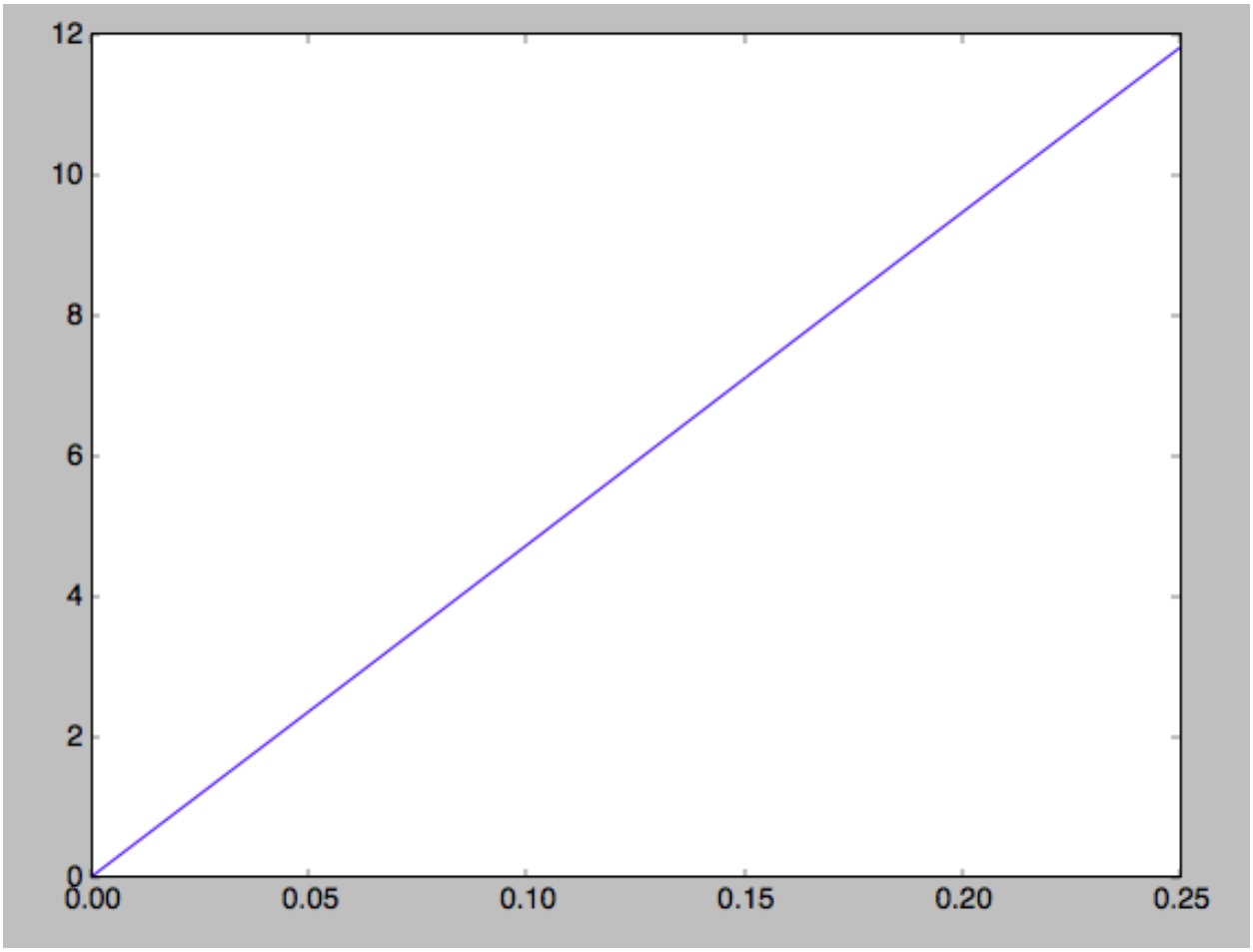
we can similarly calculate the tilt

```
>>> qdfS.tilt(1.,0.125)
# 0.10314272868452541
```

or about 5 degrees (the returned value has units of rad). As a function of height, we find

```
>>> tilts= numpy.array([qdfS.tilt(1.,z) for z in zs])
>>> plot(zs,tilts*180./numpy.pi)
```

which gives



We can also calculate the density and surface density (the zero-th velocity moments). For example, the vertical density

```
>>> densz= numpy.array([qdf.density(1.,z) for z in zs])
```

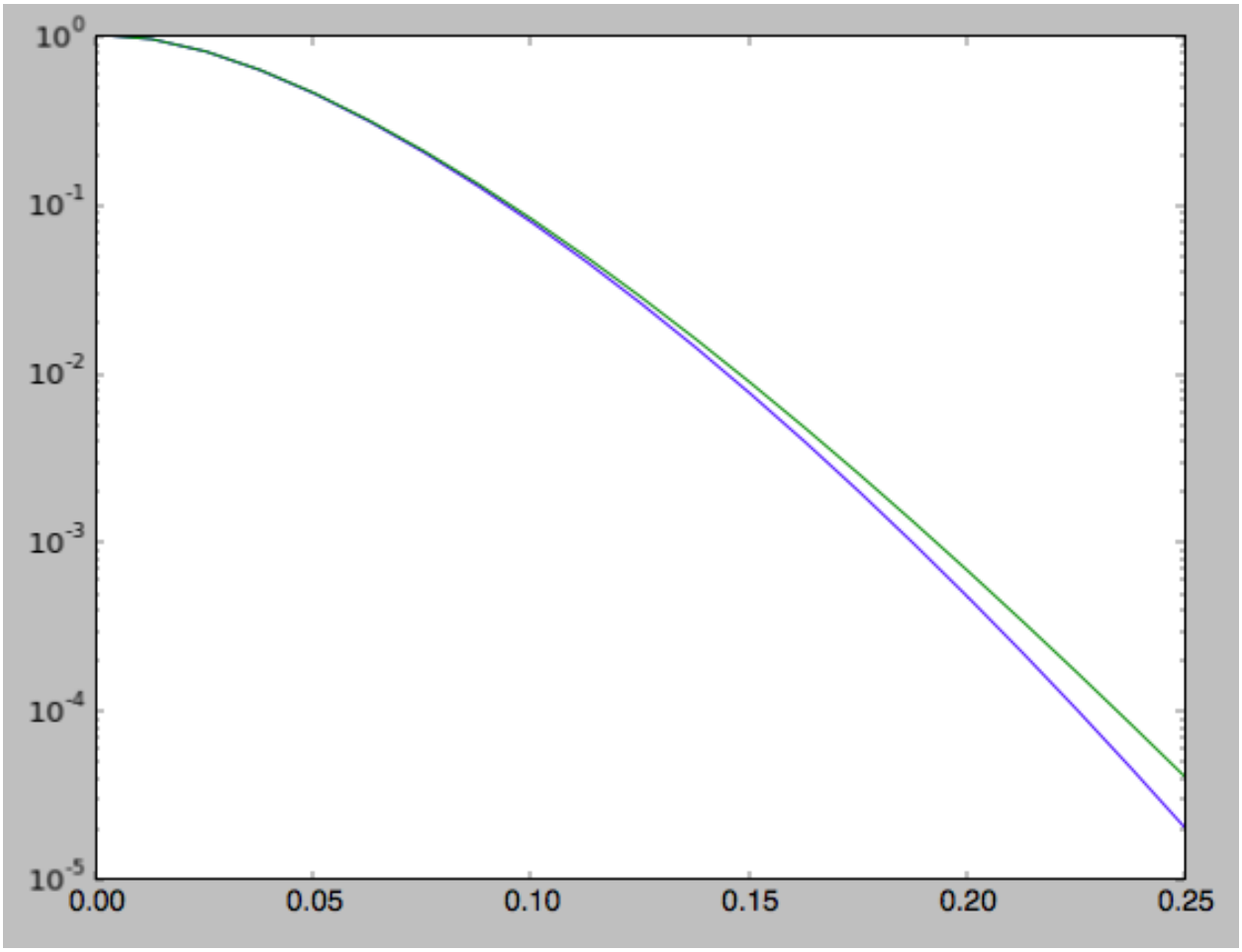
and

```
>>> denszS= numpy.array([qdfS.density(1.,z) for z in zs])
```

We can compare the vertical profiles calculated using the adiabatic and Staeckel action-angle approximations

```
>>> semilogy(zs,densz/densz[0])
>>> semilogy(zs,denszS/denszS[0])
```

which gives



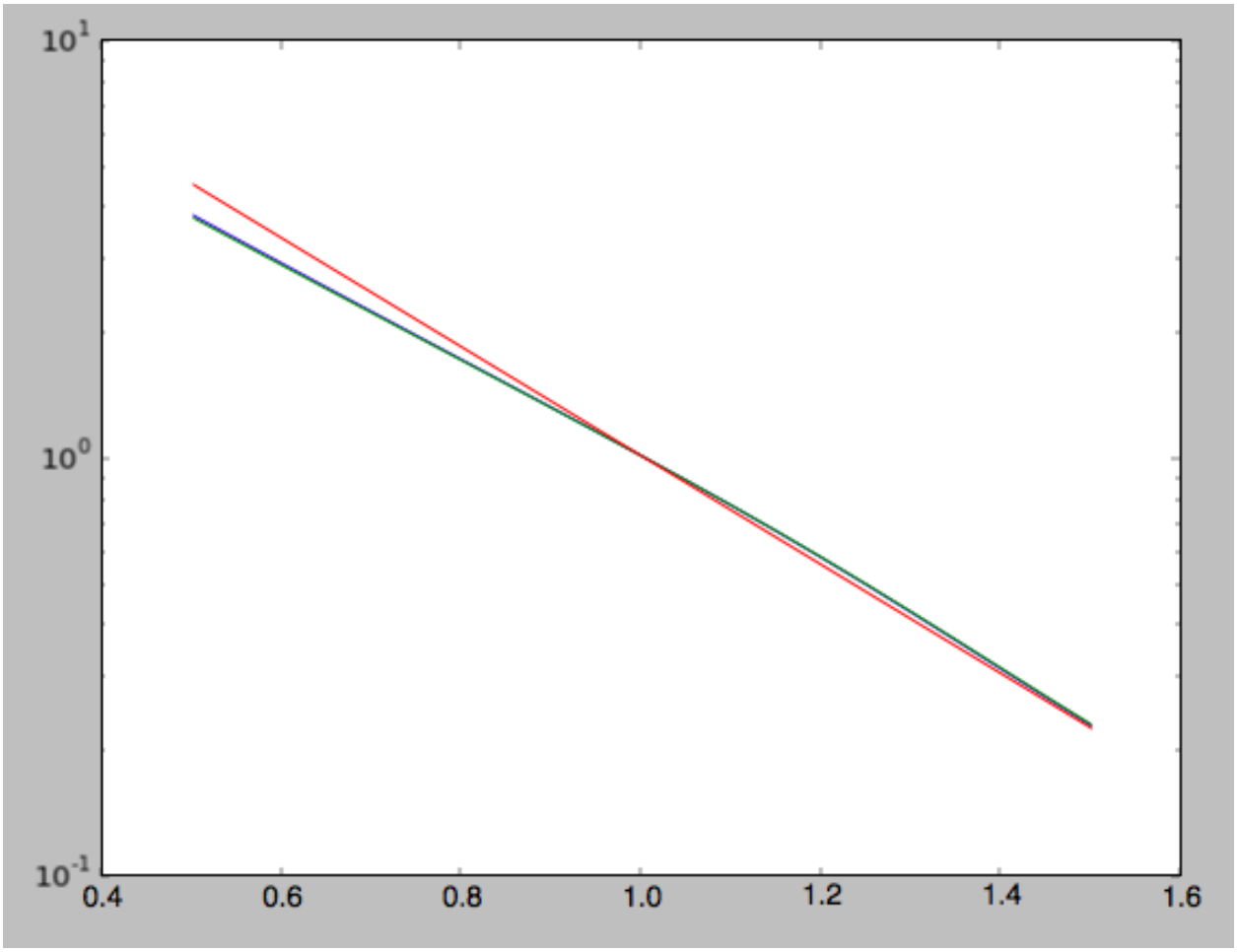
Similarly, we can calculate the radial profile of the surface density

```
>>> rs= numpy.linspace(0.5,1.5,21)
>>> surfr= numpy.array([qdf.surfacemass_z(r) for r in rs])
>>> surfrS= numpy.array([qdfS.surfacemass_z(r) for r in rs])
```

and compare them with each other and an exponential with scale length 1/3

```
>>> semilogy(rs,surfr/surfr[10])
>>> semilogy(rs,surfrS/surfrS[10])
>>> semilogy(rs,numpy.exp(-(rs-1.)/(1./3.)))
```

which gives



The two radial profiles are almost indistinguishable and are very close, if somewhat shallower, than the pure exponential profile.

General velocity moments, including all higher order moments, are implemented in `quasiisothermaldf.vmomentdensity`.

1.8.3 Evaluating and sampling the full probability distribution function

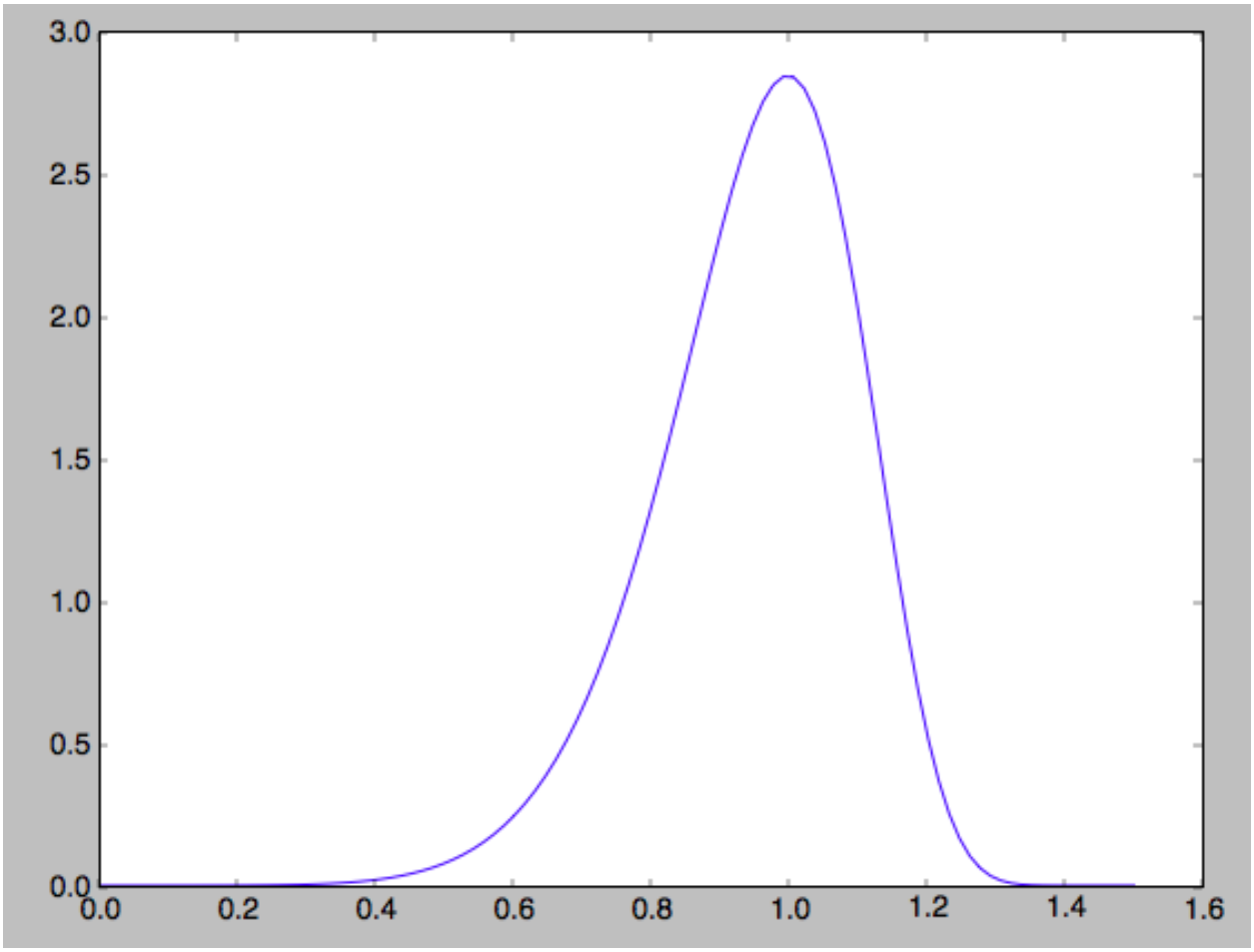
We can evaluate the distribution itself by calling the object, e.g.,

```
>>> qdf(1.,0.1,1.1,0.1,0.) #input: R,vR,vT,z,vz
# array([ 16.86790643])
```

or as a function of rotational velocity, for example in the mid-plane

```
>>> vts= numpy.linspace(0.,1.5,101)
>>> pvt= numpy.array([qdfS(1.,0.,vt,0.,0.) for vt in vts])
>>> plot(vts,pvt/numpy.sum(pvt)/(vts[1]-vts[0]))
```

which gives



This is, however, not the true distribution of rotational velocities at R =0 and z =0, because it is conditioned on zero radial and vertical velocities. We can calculate the distribution of rotational velocities marginalized over the radial and vertical velocities as

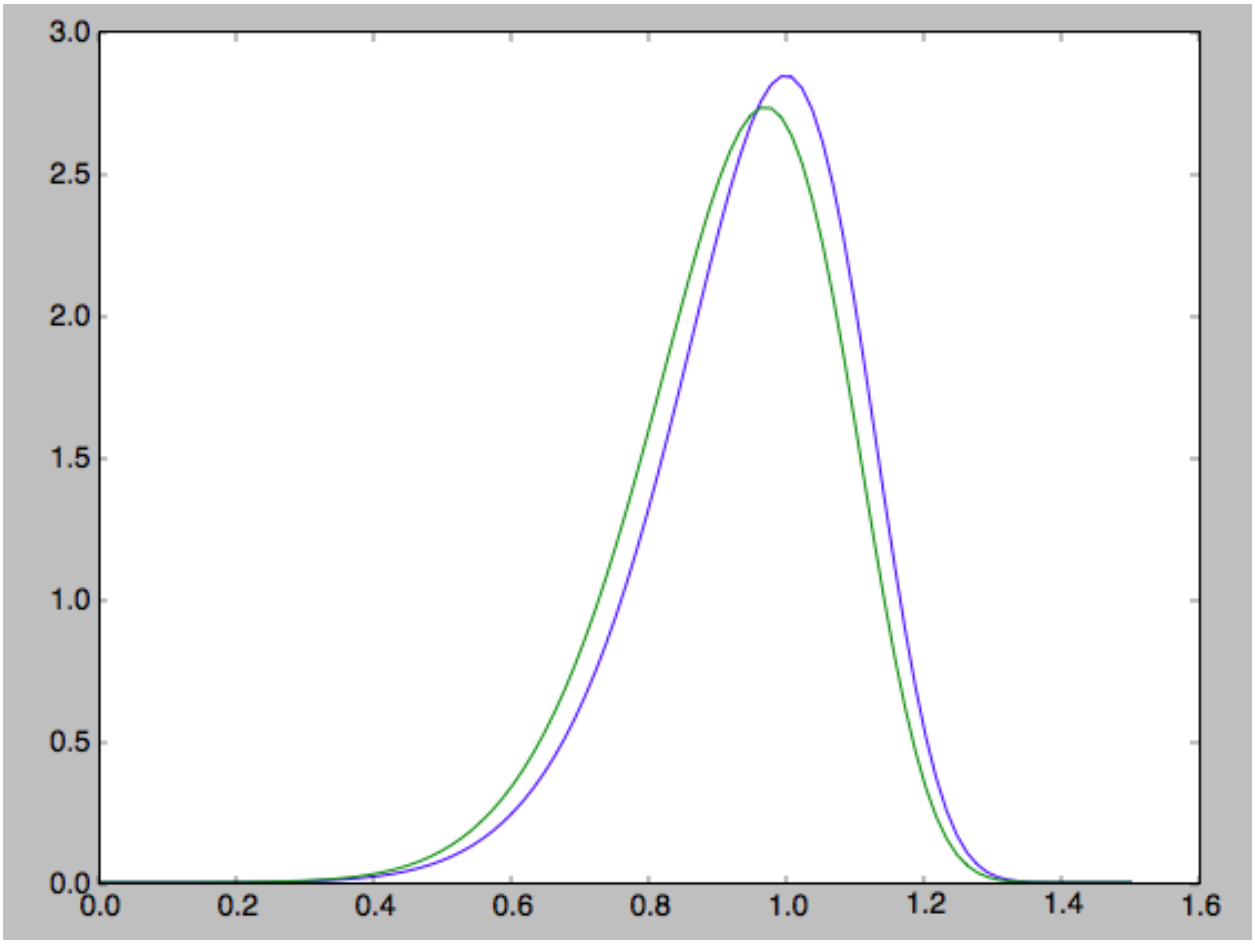
```
>>> qdfS.pvT(1.,1.,0.) #input vT,R,z
# 14.677231196899195
```

or as a function of rotational velocity

```
>>> pvt= numpy.array([qdfS.pvT(vt,1.,0.) for vt in vts])
```

overplotting this over the previous distribution gives

```
>>> plot(vts,pvt/numpy.sum(pvt)/(vts[1]-vts[0]))
```

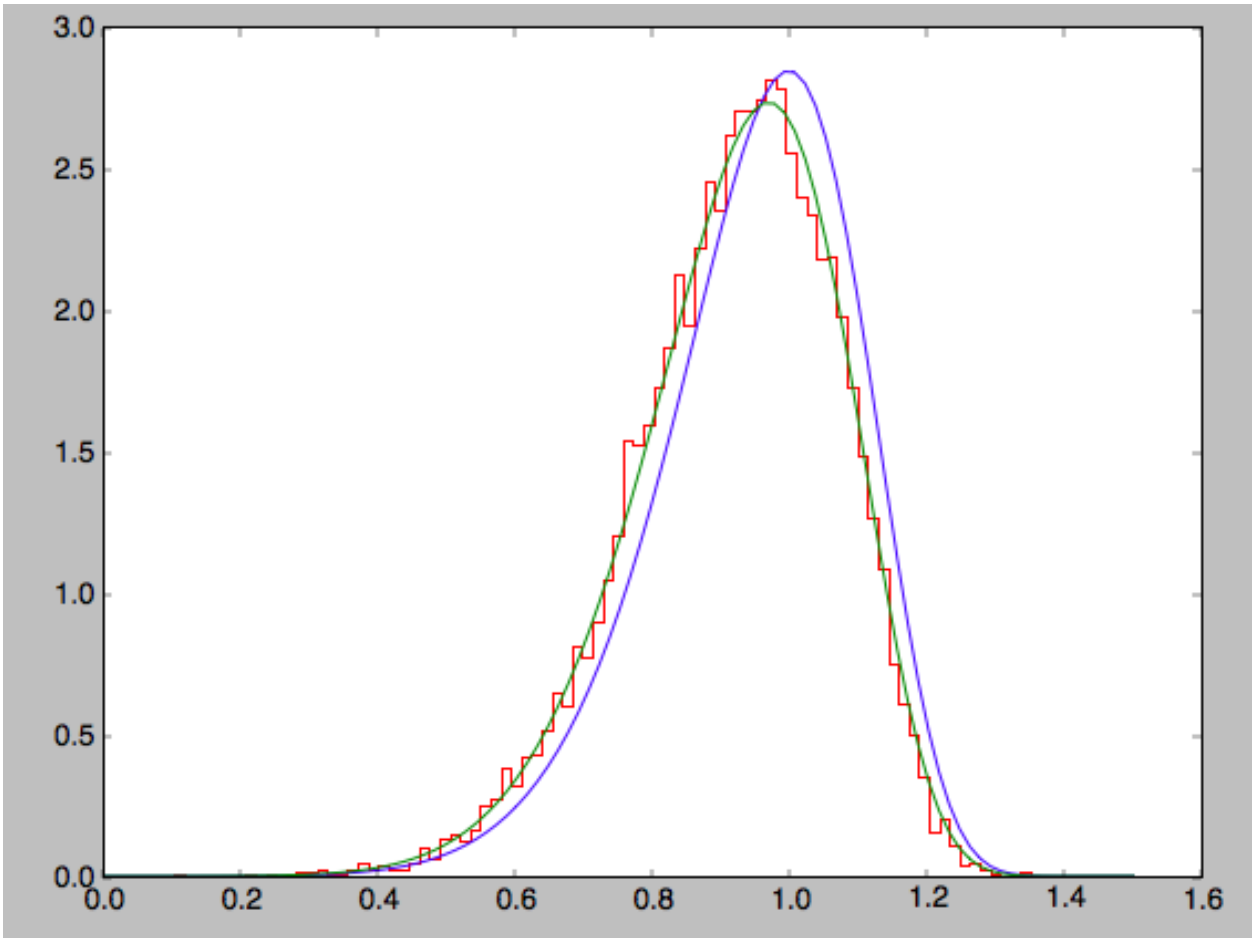


which is slightly different from the conditioned distribution. Similarly, we can calculate marginalized velocity probabilities `pvR`, `pvz`, `pvRvT`, `pvRvz`, and `pvTvz`. These are all multiplied with the density, such that marginalizing these over the remaining velocity component results in the density.

We can sample velocities at a given location using `quasiisothermaldf.sampleV` (there is currently no support for sampling locations from the density profile, although that is rather trivial):

```
>>> vs= qdfS.sampleV(1.,0.,n=10000)
>>> hist(vs[:,1],normed=True,histtype='step',bins=101,range=[0.,1.5])
```

gives



which shows very good agreement with the green (marginalized over vR and vz) curve (as it should).

1.9 Dynamical modeling of tidal streams

galpy contains tools to model the dynamics of tidal streams, making extensive use of action-angle variables. As an example, we can model the dynamics of the following tidal stream (that of Bovy 2014; 2014ApJ...795...95B). This movie shows the disruption of a cluster on a GD-1-like orbit around the Milky Way:

The blue line is the orbit of the progenitor cluster and the black points are cluster members. The disruption is shown in an approximate orbital plane and the movie is comoving with the progenitor cluster.

Streams can be represented by simple dynamical models in action-angle coordinates. In action-angle coordinates, stream members are stripped from the progenitor cluster onto orbits specified by a set of actions (J_R, J_ϕ, J_Z), which remain constant after the stars have been stripped. This is shown in the following movie, which shows the generation of the stream in action space

The color-coding gives the angular momentum J_ϕ and the black dot shows the progenitor orbit. These actions were calculated using `galpy.actionAngle.actionAngleIsochroneApprox`. The points move slightly because of small errors in the action calculation (these are correlated, so the cloud of points moves coherently because of calculation errors). The same movie that also shows the actions of stars in the cluster can be found here. This shows that the actions of stars in the cluster are not conserved (because the self-gravity of the cluster is important), but that the actions of stream members freeze once they are stripped. The angle difference between stars in a stream and the progenitor increases linearly with time, which is shown in the following movie:

where the radial and vertical angle difference with respect to the progenitor (co-moving at $(\theta_R, \theta_\phi, \theta_Z) = (\pi, \pi, \pi)$) is shown for each snapshot (the color-coding gives θ_ϕ).

One last movie provides further insight in how a stream evolves over time. The following movie shows the evolution of the stream in the two dimensional plane of frequency and angle along the stream (that is, both are projections of the three dimensional frequencies or angles onto the angle direction along the stream). The points are color-coded by the time at which they were removed from the progenitor cluster.

It is clear that disruption happens in bursts (at pericenter passages) and that the initial frequency distribution at the time of removal does not change (much) with time. However, stars removed at larger frequency difference move away from the cluster faster, such that the end of the stream is primarily made up of stars with large frequency differences with respect to the progenitor. This leads to a gradient in the typical orbit in the stream, and the stream is on average *not* on a single orbit.

1.9.1 Modeling streams in galpy

In galpy we can model streams using the tools in `galpy.df.streamdf`. We setup a streamdf instance by specifying the host gravitational potential `pot=`, an actionAngle method (typically `galpy.actionAngle.actionAngleIsochroneApprox`), a `galpy.orbit.Orbit` instance with the position of the progenitor, a parameter related to the velocity dispersion of the progenitor, and the time since disruption began. We first import all of the necessary modules

```
>>> from galpy.df import streamdf
>>> from galpy.orbit import Orbit
>>> from galpy.potential import LogarithmicHaloPotential
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.util import conversion #for unit conversions
```

setup the potential and actionAngle instances

```
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAI= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

define a progenitor Orbit instance

```
>>> obs= Orbit([1.56148083,0.35081535,-1.15481504,0.88719443,-0.47713334,0.12019596])
```

and instantiate the streamdf model

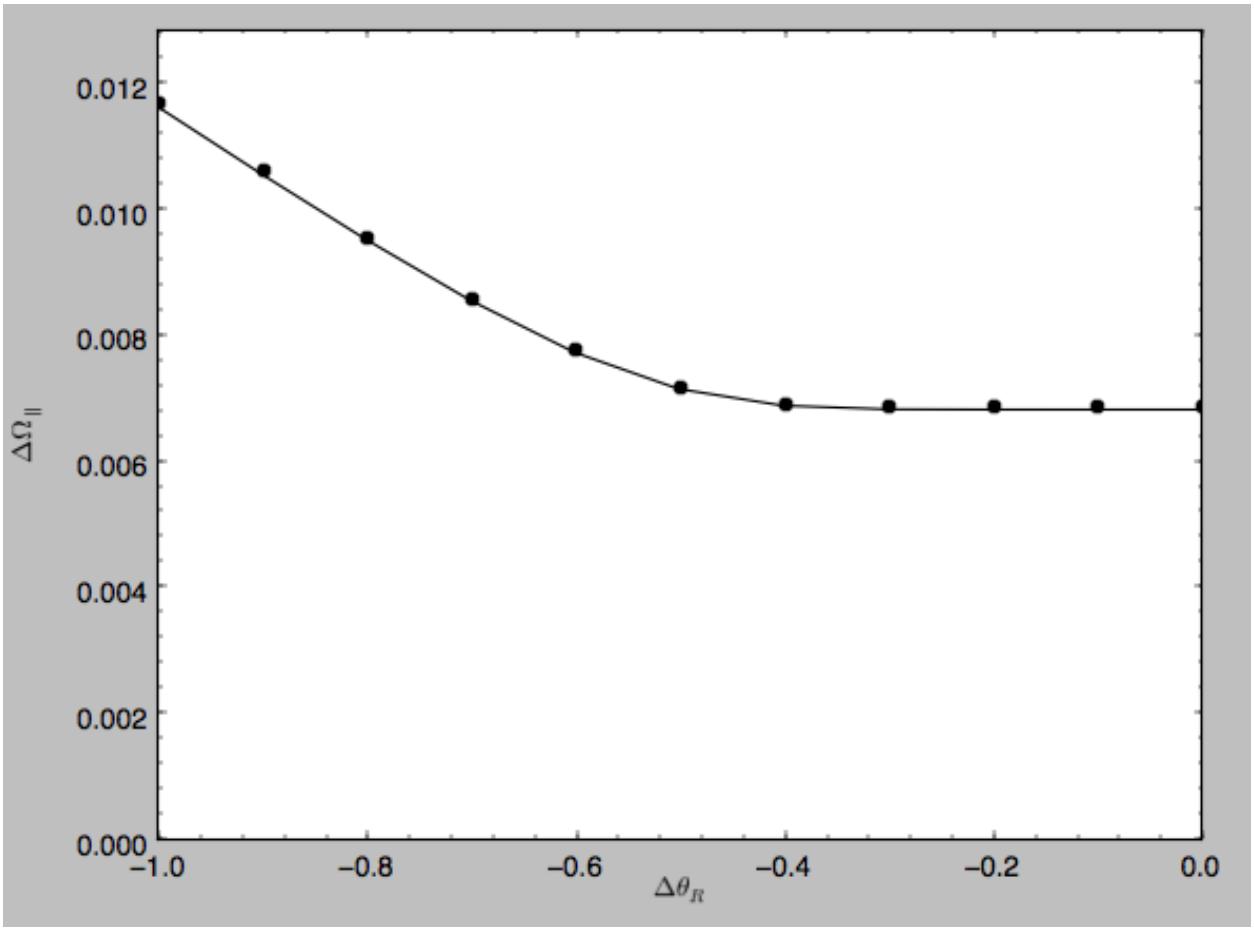
```
>>> sigv= 0.365 #km/s
>>> sdf= streamdf(sigv/220.,progenitor=obs,pot=lp,aA=aAI,leading=True,nTrackChunks=11,
↪tdisrupt=4.5/conversion.time_in_Gyr(220.,8.))
```

for a leading stream. This runs in about half a minute on a 2011 Macbook Air.

Bovy (2014) discusses how the calculation of the track needs to be iterated for potentials where there is a large offset between the track and a single orbit. One can increase the default number of iterations by specifying `nTrackIterations=` in the streamdf initialization (the default is set based on the angle between the track's frequency vector and the progenitor orbit's frequency vector; you can access the number of iterations used as `sdf.nTrackIterations`). To check whether the track is calculated accurately, one can use the following

```
>>> sdf.plotCompareTrackAAModel()
```

which in this case gives



This displays the stream model's track in frequency offset (y axis) versus angle offset (x axis) as the solid line; this is the track that the model should have if it is calculated correctly. The points are the frequency and angle offset calculated from the calculated track's $(\mathbf{x}, \mathbf{v})$. For a properly computed track these should line up, as they do in this figure. If they do not line up, increasing `nTrackIterations` is necessary.

We can calculate some simple properties of the stream, such as the ratio of the largest and second-to-largest eigenvalue of the Hessian $\partial\mathbf{\Omega}/\partial\mathbf{J}$

```
>>> sdf.freqEigvalRatio(isotropic=True)
# 34.450028399901434
```

or the model's ratio of the largest and second-to-largest eigenvalue of the model frequency variance matrix

```
>>> sdf.freqEigvalRatio()
# 29.625538344985291
```

The fact that this ratio is so large means that an approximately one dimensional stream will form.

Similarly, we can calculate the angle between the frequency vector of the progenitor and of the model mean frequency vector

```
>>> sdf.misalignment()
# 0.0086441947505973005
```

which returns this angle in radians. We can also calculate the angle between the frequency vector of the progenitor and the principal eigenvector of $\partial\mathbf{\Omega}/\partial\mathbf{J}$

```
>>> sdf.misalignment(isotropic=True)
# 0.02238411611147997
```

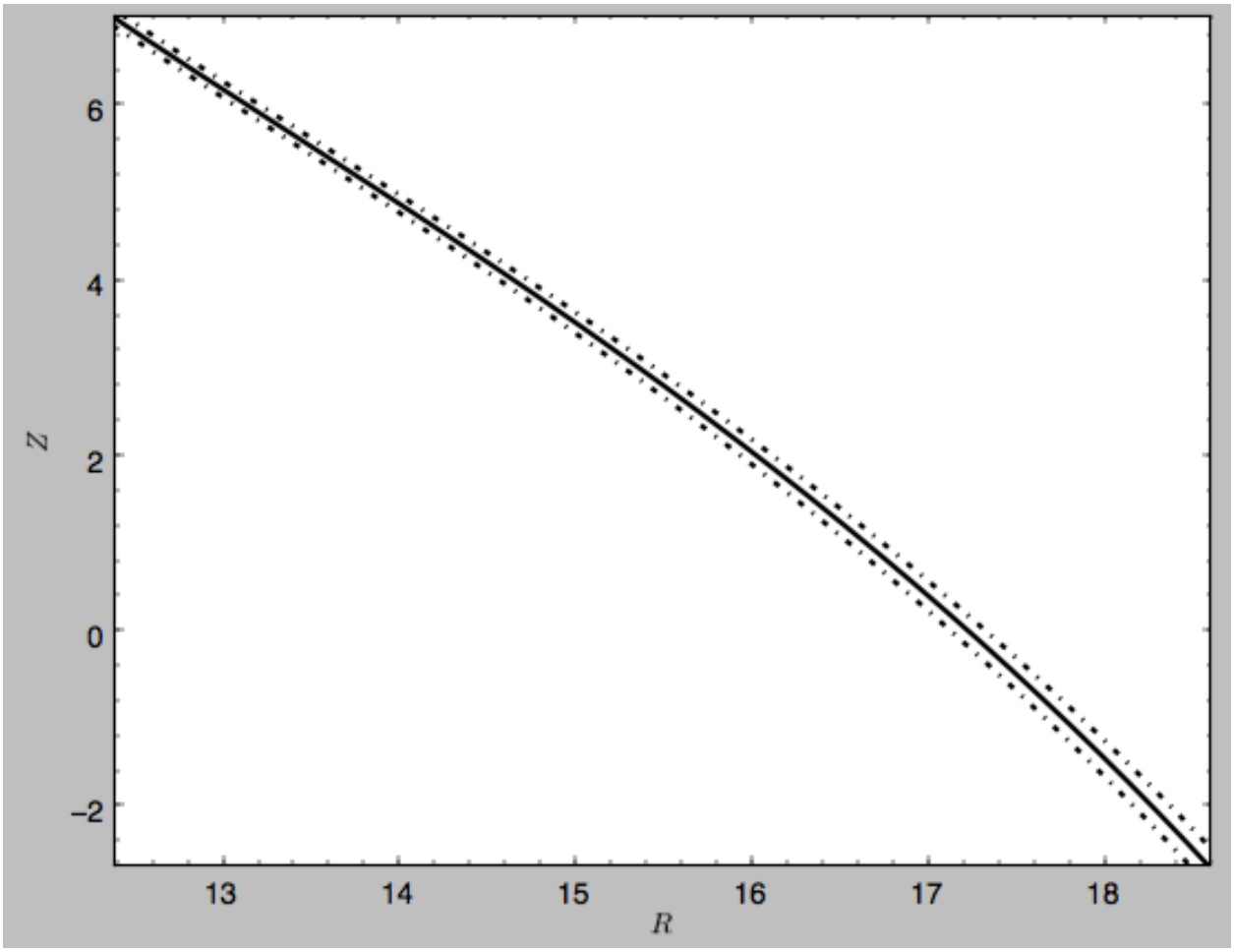
(the reason these are obtained by specifying `isotropic=True` is that these would be the ratio of the eigenvalues or the angle if we assumed that the disrupted materials action distribution were isotropic).

1.9.2 Calculating the average stream location (track)

We can display the stream track in various coordinate systems as follows

```
>>> sdf.plotTrack(d1='r',d2='z',interp=True,color='k',spread=2,overplot=False,lw=2.,
↪scaleToPhysical=True)
```

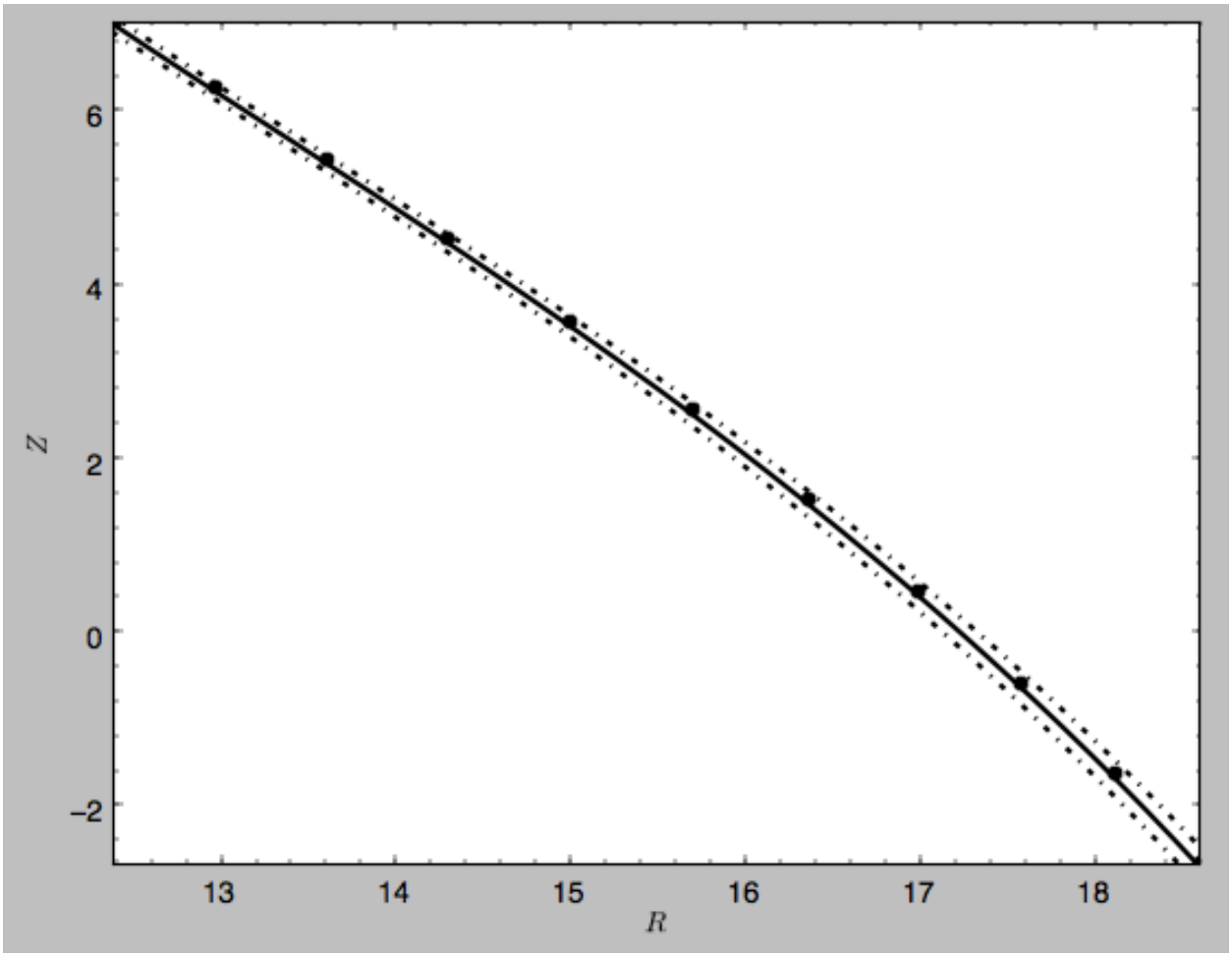
which gives



which shows the track in Galactocentric R and Z coordinates as well as an estimate of the spread around the track as the dash-dotted line. We can overplot the points along the track along which the $(\mathbf{x},\mathbf{v}) \rightarrow (\mathbf{\Omega},\theta)$ transformation and the track position is explicitly calculated, by turning off the interpolation

```
>>> sdf.plotTrack(d1='r',d2='z',interp=False,color='k',spread=0,overplot=True,ls='none
↪',marker='o',scaleToPhysical=True)
```

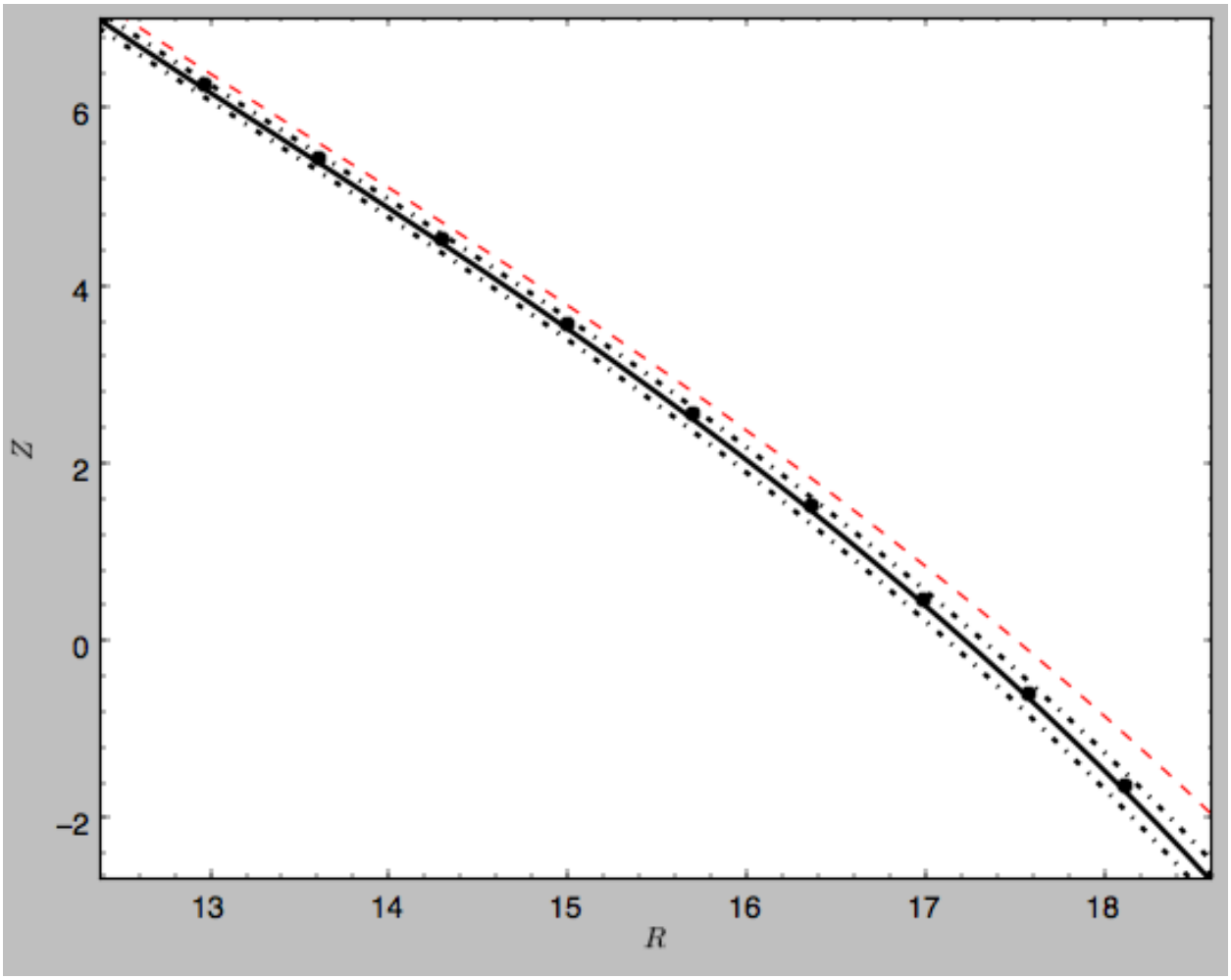
which gives



We can also overplot the orbit of the progenitor

```
>>> sdf.plotProgenitor(d1='r',d2='z',color='r',overplot=True,ls='--',
↪scaleToPhysical=True)
```

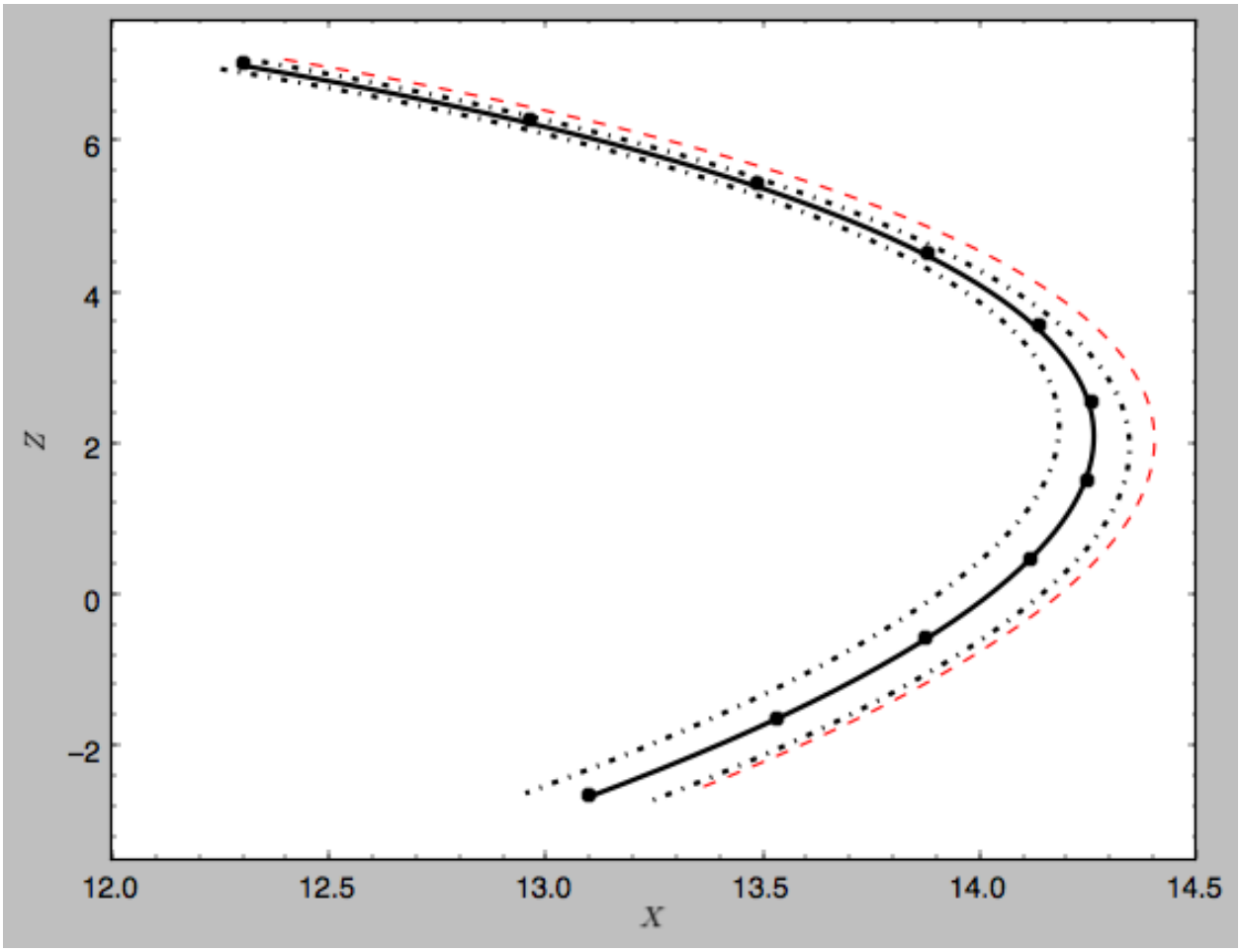
to give



We can do the same in other coordinate systems, for example X and Z (as in Figure 1 of Bovy 2014)

```
>>> sdf.plotTrack(d1='x',d2='z',interp=True,color='k',spread=2,overplot=False,lw=2.,
↪scaleToPhysical=True)
>>> sdf.plotTrack(d1='x',d2='z',interp=False,color='k',spread=0,overplot=True,ls='none
↪',marker='o',scaleToPhysical=True)
>>> sdf.plotProgenitor(d1='x',d2='z',color='r',overplot=True,ls='--',
↪scaleToPhysical=True)
>>> xlim(12.,14.5); ylim(-3.5,7.6)
```

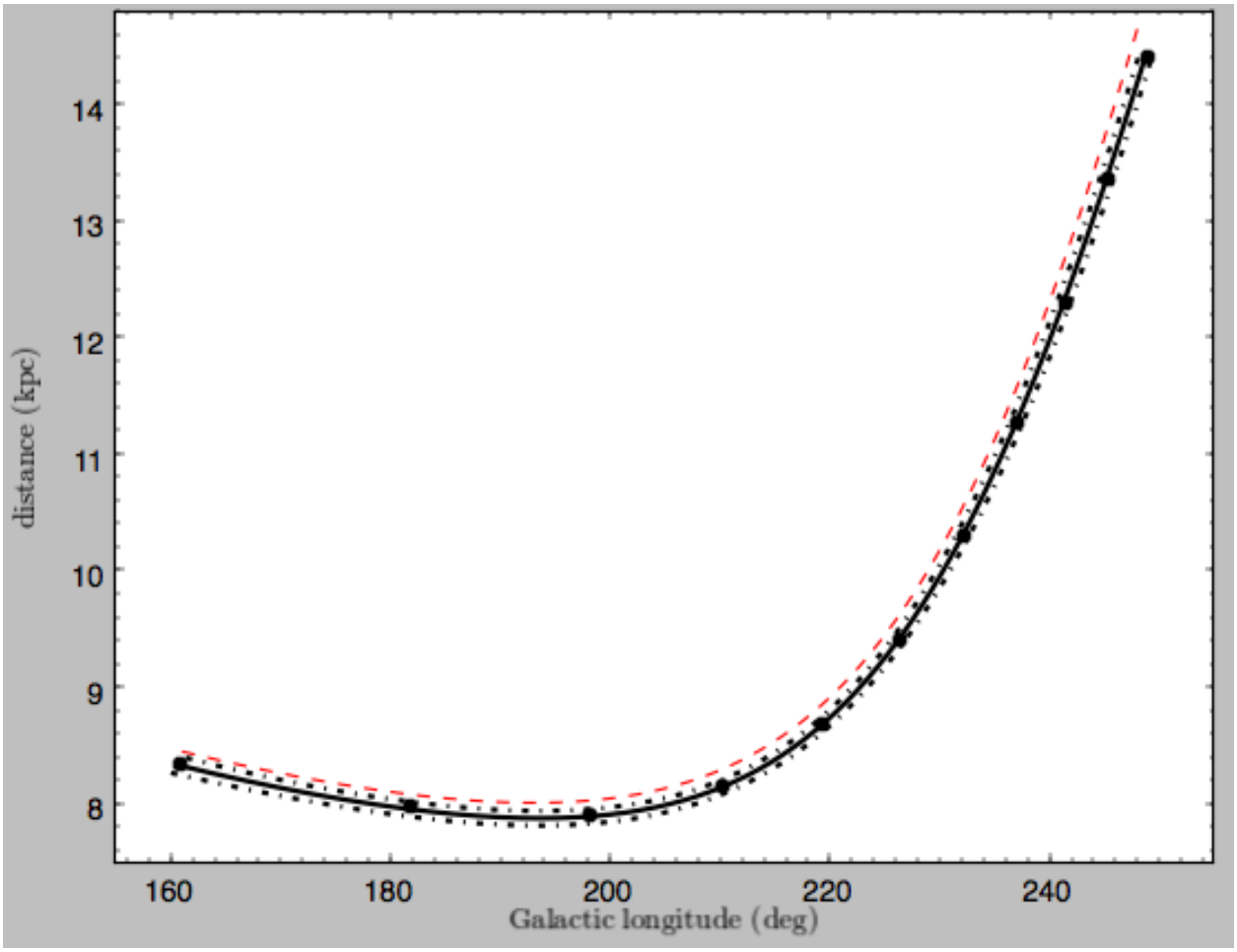
which gives



or we can calculate the track in observable coordinates, e.g.,

```
>>> sdf.plotTrack(d1='ll',d2='dist',interp=True,color='k',spread=2,overplot=False,
↪lw=2.)
>>> sdf.plotTrack(d1='ll',d2='dist',interp=False,color='k',spread=0,overplot=True,ls=
↪'none',marker='o')
>>> sdf.plotProgenitor(d1='ll',d2='dist',color='r',overplot=True,ls='--')
>>> xlim(155.,255.); ylim(7.5,14.8)
```

which displays



Coordinate transformations to physical coordinates are done using parameters set when initializing the `sdf` instance. See the help for `?streamdf` for a complete list of initialization parameters.

1.9.3 Mock stream data generation

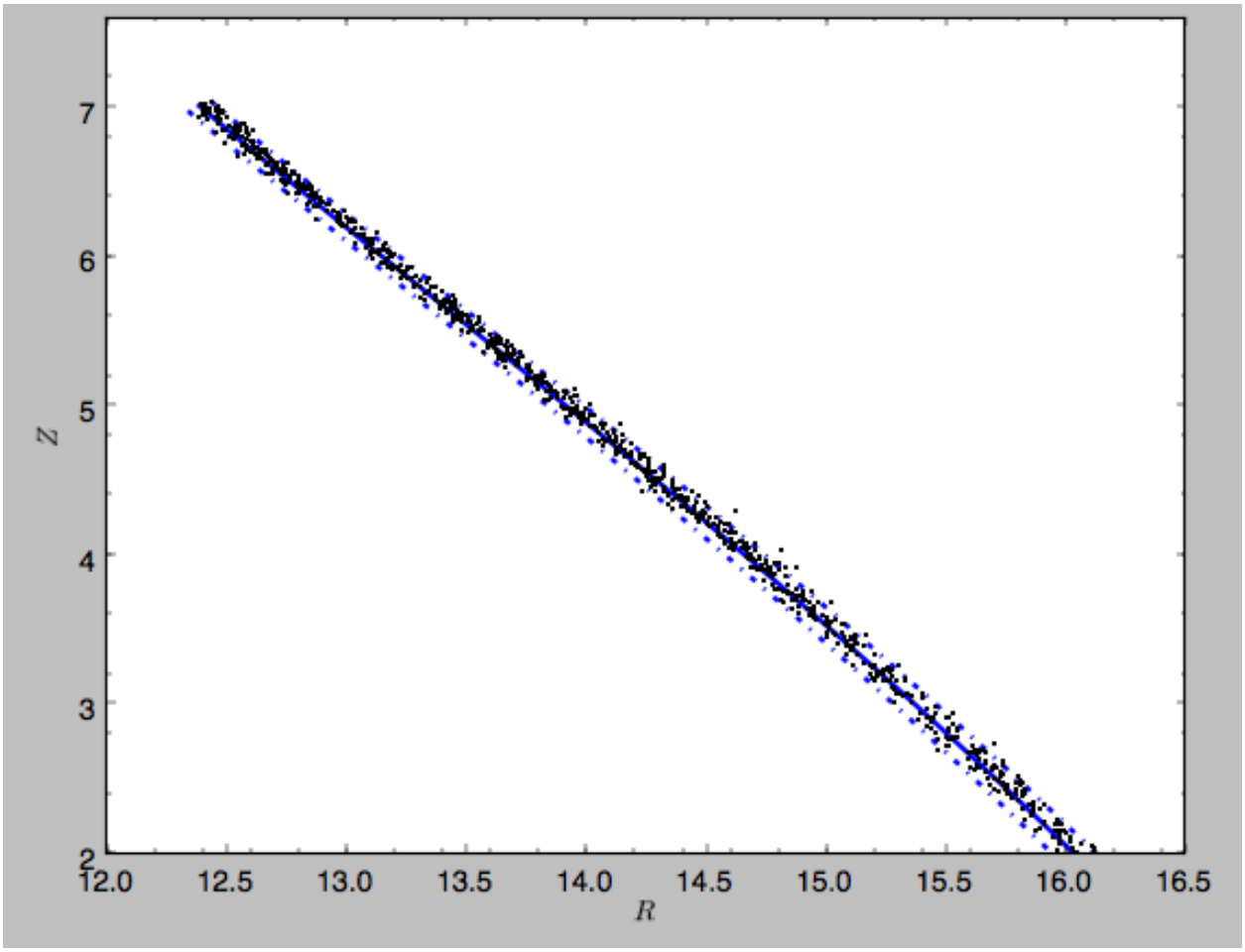
We can also easily generate mock data from the stream model. This uses `streamdf.sample`. For example,

```
>>> RvR= sdf.sample(n=1000)
```

which returns the sampled points as a set $(R, v_R, v_T, Z, v_Z, \phi)$ in natural galpy coordinates. We can plot these and compare them to the track location

```
>>> sdf.plotTrack(d1='r',d2='z',interp=True,color='b',spread=2,overplot=False,lw=2.,
↪scaleToPhysical=True)
>>> plot(RvR[0]*8.,RvR[3]*8.,'k.',ms=2.) #multiply by the physical distance scale
>>> xlim(12.,16.5); ylim(2.,7.6)
```

which gives



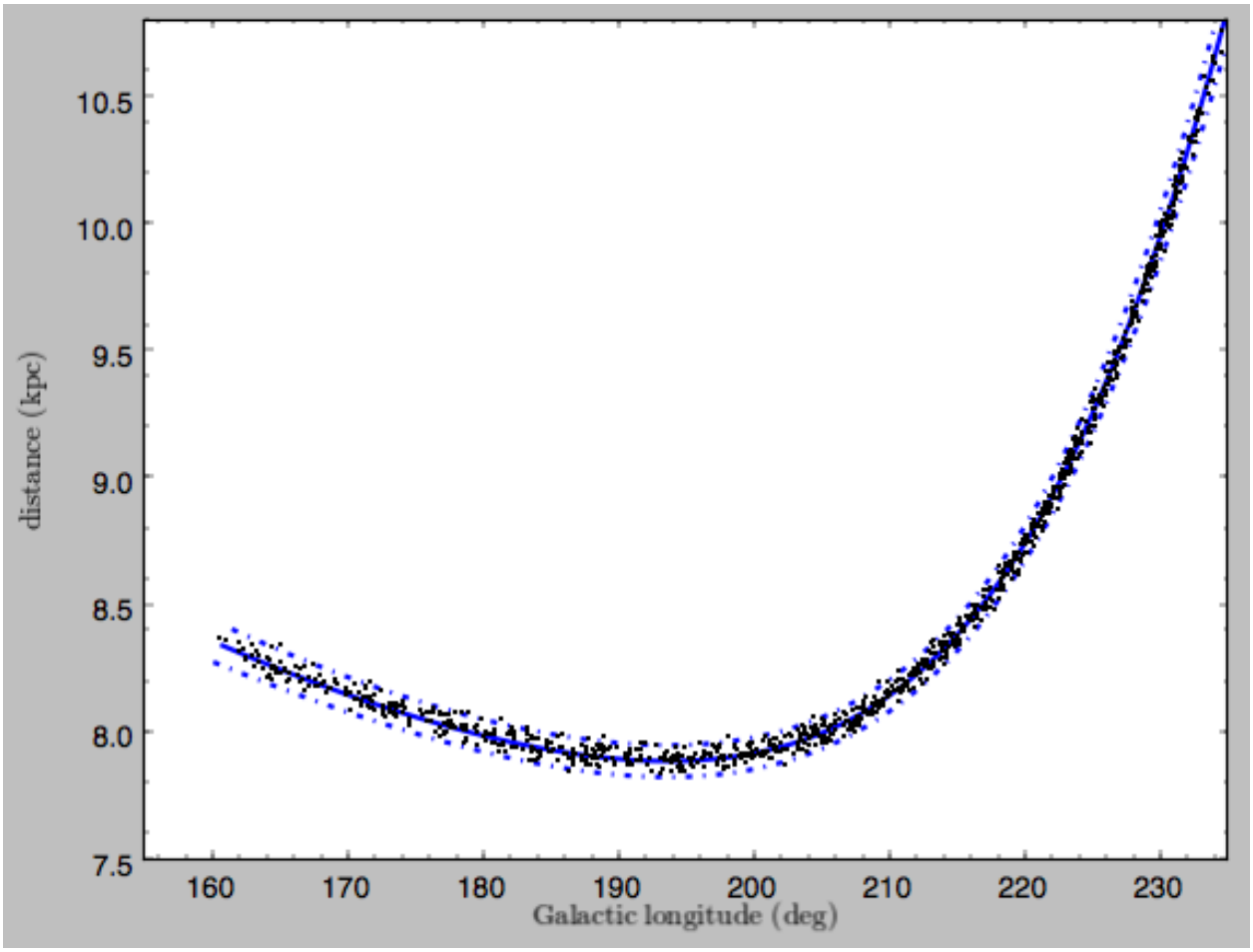
Similarly, we can generate mock data in observable coordinates

```
>>> lb= sdf.sample(n=1000,lb=True)
```

and plot it

```
>>> sdf.plotTrack(d1='ll',d2='dist',interp=True,color='b',spread=2,overplot=False,
↪lw=2.)
>>> plot(lb[0],lb[2],'k.',ms=2.)
>>> xlim(155.,235.); ylim(7.5,10.8)
```

which displays

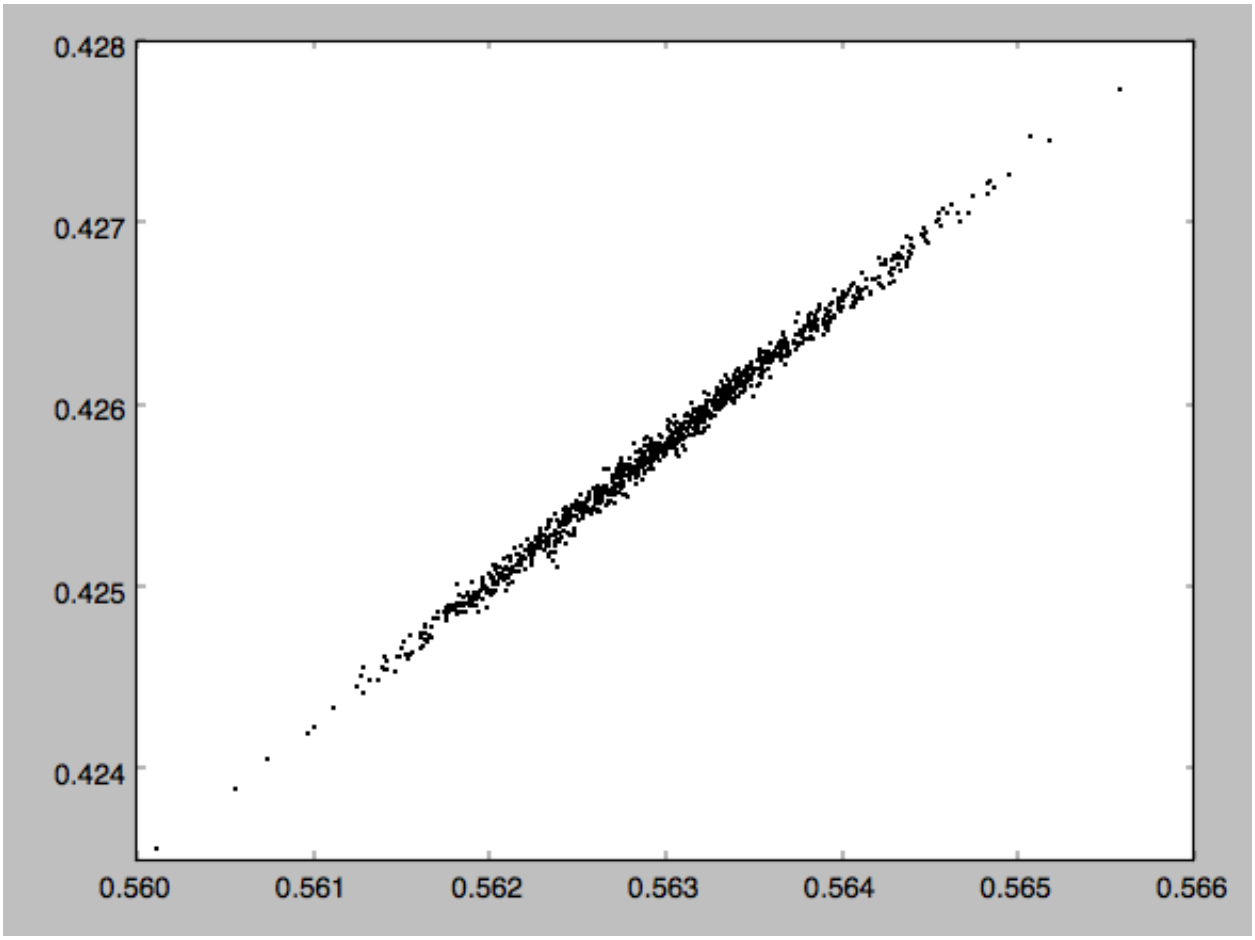


We can also just generate mock stream data in frequency-angle coordinates

```
>>> mockaA= sdf.sample(n=1000,returnaAdt=True)
```

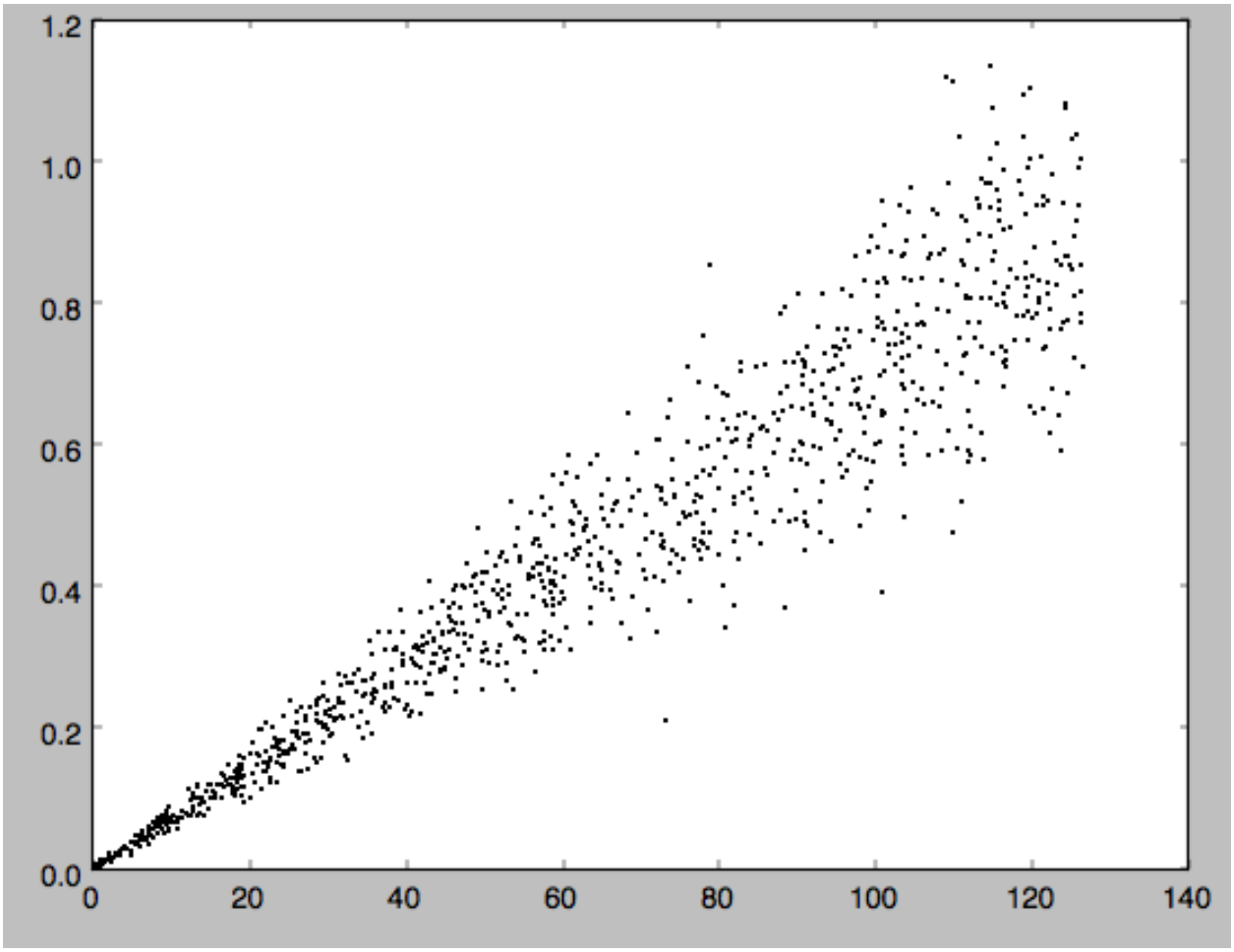
which returns a tuple with three components: an array with shape [3,N] of frequency vectors $(\Omega_R, \Omega_\phi, \Omega_Z)$, an array with shape [3,N] of angle vectors $(\theta_R, \theta_\phi, \theta_Z)$ and t_s, the stripping time. We can plot the vertical versus the radial frequency

```
>>> plot(mockaA[0][0],mockaA[0][2],'k.',ms=2.)
```



or we can plot the magnitude of the angle offset as a function of stripping time

```
>>> plot(mockaA[2],numpy.sqrt(numpy.sum((mockaA[1]-numpy.tile(sdf._progenitor_angle,
↪(1000,1)).T)**2.,axis=0)),'k.',ms=2.)
```



1.9.4 Evaluating and marginalizing the full PDF

We can also evaluate the stream PDF, the probability of a $(\mathbf{x}, \mathbf{v})$ phase-space position in the stream. We can evaluate the PDF, for example, at the location of the progenitor

```
>>> sdf(obs.R(),obs.vR(),obs.vT(),obs.z(),obs.vz(),obs.phi())
# array([-33.16985861])
```

which returns the natural log of the PDF. If we go to slightly higher in Z and slightly smaller in R, the PDF becomes zero

```
>>> sdf(obs.R()-0.1,obs.vR(),obs.vT(),obs.z()+0.1,obs.vz(),obs.phi())
# array([-inf])
```

because this phase-space position cannot be reached by a leading stream star. We can also marginalize the PDF over unobserved directions. For example, similar to Figure 10 in Bovy (2014), we can evaluate the PDF $p(X|Z)$ near a point on the track, say near Z =2 kpc (=0.25 in natural units. We first find the approximate Gaussian PDF near this point, calculated from the stream track and dispersion (see above)

```
>>> meanp, varp= sdf.gaussApprox([None,None,2./8.,None,None,None])
```

where the input is a array with entries [X,Y,Z,vX,vY,vZ] and we substitute None for directions that we want to establish the approximate PDF for. So the above expression returns an approximation to $p(X, Y, v_X, v_Y, v_Z|Z)$. This

approximation allows us to get a sense of where the PDF peaks and what its width is

```
>>> meanp[0]*8.
# 14.267559400127833
>>> numpy.sqrt(varp[0,0])*8.
# 0.04152968631186698
```

We can now evaluate the PDF $p(X|Z)$ as a function of X near the peak

```
>>> xs= numpy.linspace(-3.*numpy.sqrt(varp[0,0]),3.*numpy.sqrt(varp[0,0]),21)+meanp[0]
>>> logps= numpy.array([sdf.callMarg([x,None,2./8.,None,None,None]) for x in xs])
>>> ps= numpy.exp(logps)
```

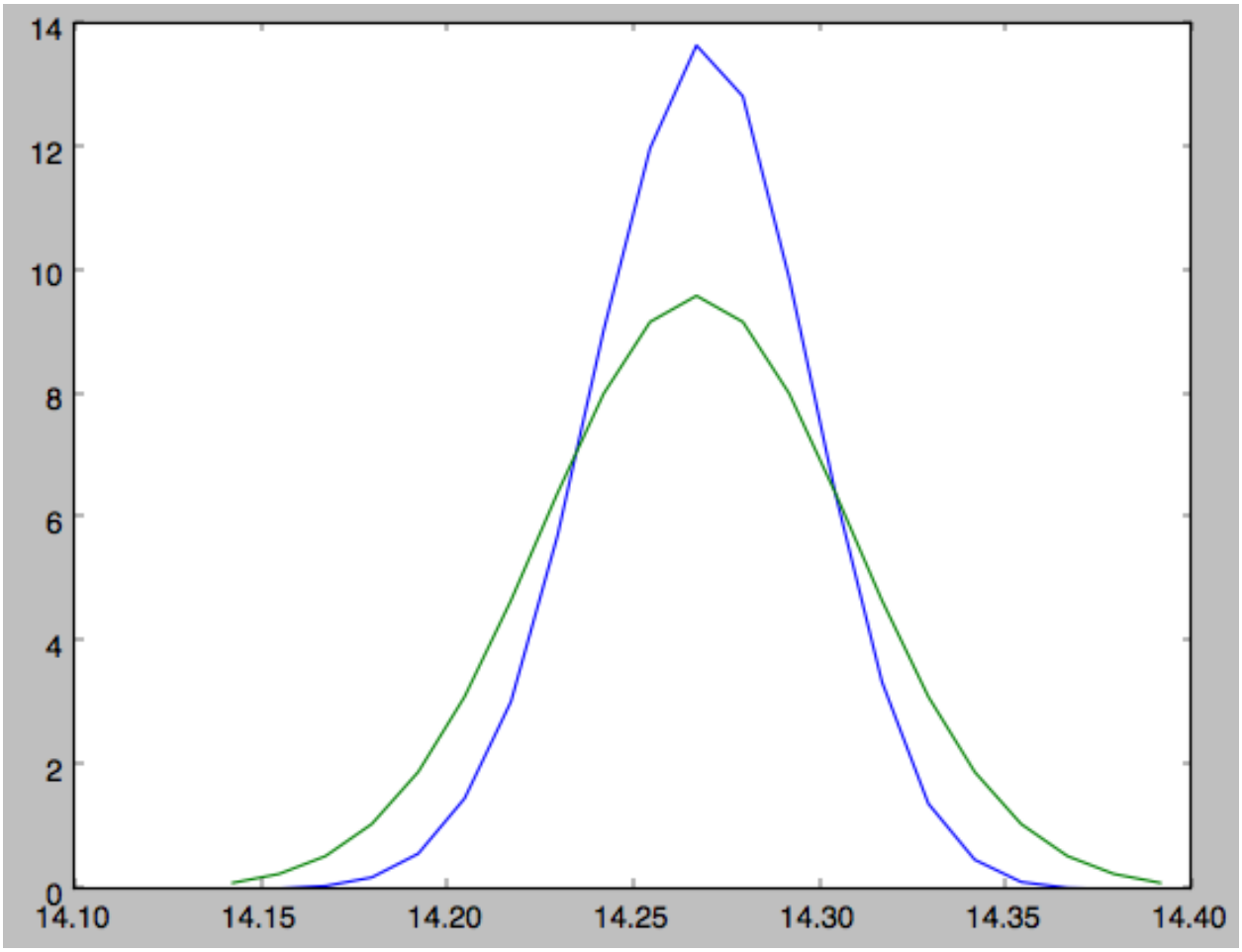
and we normalize the PDF

```
>>> ps/= numpy.sum(ps)*(xs[1]-xs[0])*8.
```

and plot it together with the Gaussian approximation

```
>>> plot(xs*8.,ps)
>>> plot(xs*8.,1./numpy.sqrt(2.*numpy.pi)/numpy.sqrt(varp[0,0])/8.*numpy.exp(-0.5*(xs-
↪meanp[0])**2./varp[0,0]))
```

which gives



Sometimes it is hard to automatically determine the closest point on the calculated track if only one phase-space

coordinate is given. For example, this happens when evaluating $p(Z|X)$ for $X > 13$ kpc here, where there are two branches of the track in Z (see the figure of the track above). In that case, we can determine the closest track point on one of the branches by hand and then provide this closest point as the basis of PDF calculations. The following example shows how this is done for the upper Z branch at $X = 13.5$ kpc, which is near $Z = 5$ kpc (Figure 10 in Bovy 2014).

```
>>> cindx= sdf.find_closest_trackpoint(13.5/8.,None,5.32/8.,None,None,None,xy=True)
```

gives the index of the closest point on the calculated track. This index can then be given as an argument for the PDF functions:

```
>>> meanp, varp= meanp, varp= sdf.gaussApprox([13.5/8.,None,None,None,None,None],
↪cindx=cindx)
```

computes the approximate $p(Y, Z, v_X, v_Y, v_Z|X)$ near the upper Z branch. In Z, this PDF has mean and dispersion

```
>>> meanp[1]*8.
# 5.4005530328542077
>>> numpy.sqrt(varp[1,1])*8.
# 0.05796023309510244
```

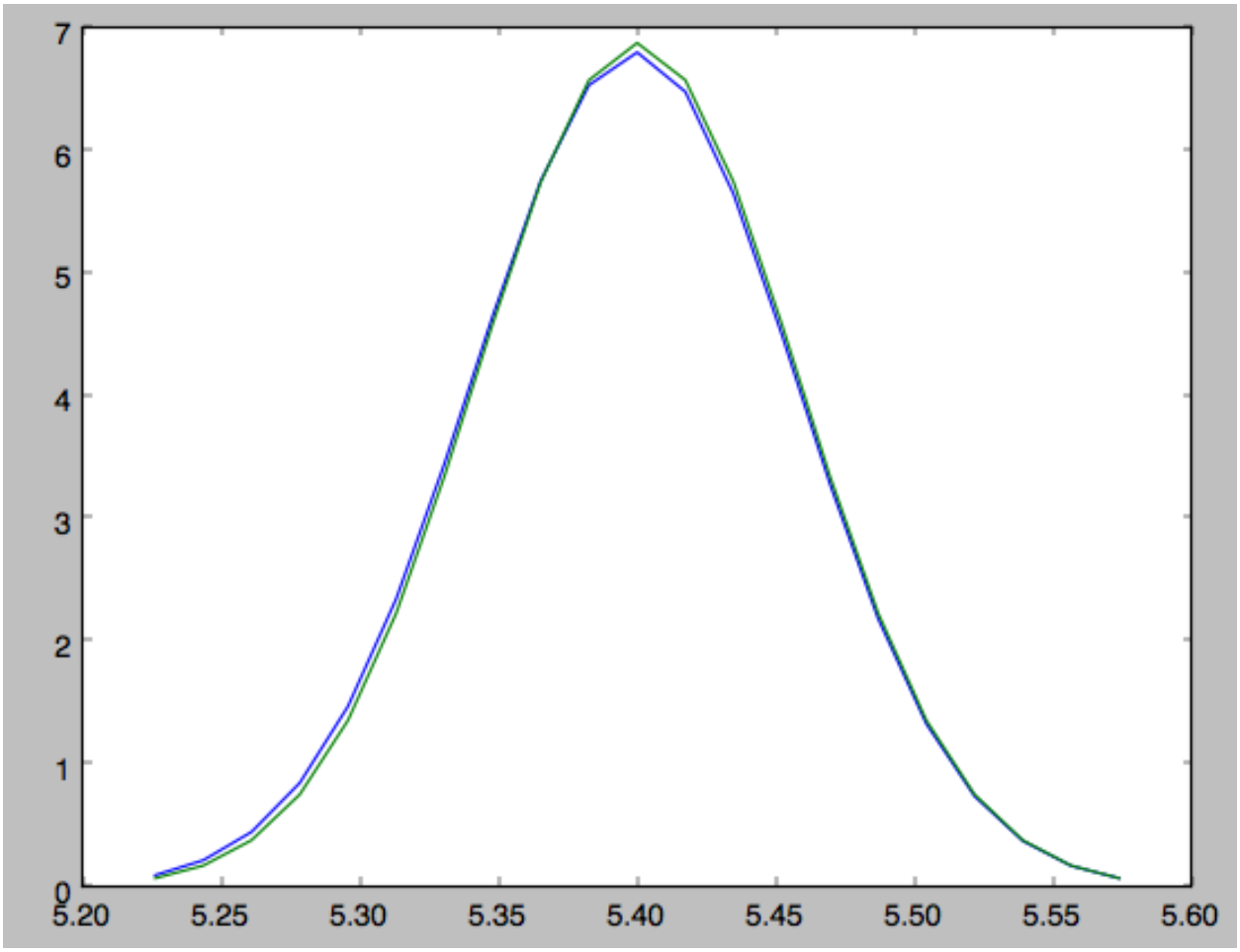
We can then evaluate $p(Z|X)$ for the upper branch as

```
>>> zs= numpy.linspace(-3.*numpy.sqrt(varp[1,1]),3.*numpy.sqrt(varp[1,1]),21)+meanp[1]
>>> logps= numpy.array([sdf.callMarg([13.5/8.,None,z,None,None,None],cindx=cindx) for␣
↪z in zs])
>>> ps= numpy.exp(logps)
>>> ps/= numpy.sum(ps)*(zs[1]-zs[0])*8.
```

and we can again plot this and the approximation

```
>>> plot(zs*8.,ps)
>>> plot(zs*8.,1./numpy.sqrt(2.*numpy.pi)/numpy.sqrt(varp[1,1])/8.*numpy.exp(-0.5*(zs-
↪meanp[1])**2./varp[1,1]))
```

which gives



The approximate PDF in this case is very close to the correct PDF. When supplying the closest track point, care needs to be taken that this really is the closest track point. Otherwise the approximate PDF will not be quite correct.

1.9.5 Modeling gaps in streams

`galpy` also contains tools to model the effect of impacts due to dark-matter subhalos on streams (see Sanders, Bovy, & Erkal 2015). This is implemented as a subclass `streamgapdf` of `streamdf`, because they share many of the same methods. Setting up a `streamgapdf` object requires the same arguments and keywords as setting up a `streamdf` instance (to specify the smooth underlying stream model and the Galactic potential) as well as parameters that specify the impact (impact parameter and velocity, location and time of closest approach, mass and structure of the subhalo, and helper keywords that specify how the impact should be calculated). An example used in the paper (but not that with the modifications in Sec. 6.1) is as follows. Imports:

```
>>> from galpy.df import streamdf, streamgapdf
>>> from galpy.orbit import Orbit
>>> from galpy.potential import LogarithmicHaloPotential
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.util import conversion
```

Parameters for the smooth stream and the potential:

```
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAI= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

(continues on next page)

(continued from previous page)

```
>>> prog_unp_peri= Orbit([2.6556151742081835,
                          0.2183747276300308,
                          0.67876510797240575,
                          -2.0143395648974671,
                          -0.3273737682604374,
                          0.24218273922966019])
>>> V0, R0= 220., 8.
>>> sigv= 0.365*(10./2.)**(1./3.) # km/s
>>> tdisrupt= 10.88/conversion.time_in_Gyr(V0,R0)
```

and the parameters of the impact

```
>>> GM= 10.**-2./conversion.mass_in_1010msol(V0,R0)
>>> rs= 0.625/R0
>>> impactb= 0.
>>> subhalovel= numpy.array([6.82200571,132.7700529,149.4174464])/V0
>>> timpact= 0.88/conversion.time_in_Gyr(V0,R0)
>>> impact_angle= -2.34
```

The setup is then

```
>>> sdf_sanders15= streamgapdf(sigv/V0,progenitor=prog_unp_peri,pot=lp,aA=aAI,
                               leading=False,nTrackChunks=26,
                               nTrackIterations=1,
                               sigMeanOffset=4.5,
                               tdisrupt=tdisrupt,
                               Vnorm=V0,Rnorm=R0,
                               impactb=impactb,
                               subhalovel=subhalovel,
                               timpact=timpact,
                               impact_angle=impact_angle,
                               GM=GM,rs=rs)
```

The `streamgapdf` implementation is currently not entirely complete (for example, one cannot yet evaluate the full phase-space PDF), but the model can be sampled as in the paper above. To compare the perturbed model to the unperturbed model, we also set up an unperturbed model of the same stream

```
>>> sdf_sanders15_unp= streamdf(sigv/V0,progenitor=prog_unp_peri,pot=lp,aA=aAI,
                               leading=False,nTrackChunks=26,
                               nTrackIterations=1,
                               sigMeanOffset=4.5,
                               tdisrupt=tdisrupt,
                               Vnorm=V0,Rnorm=R0)
```

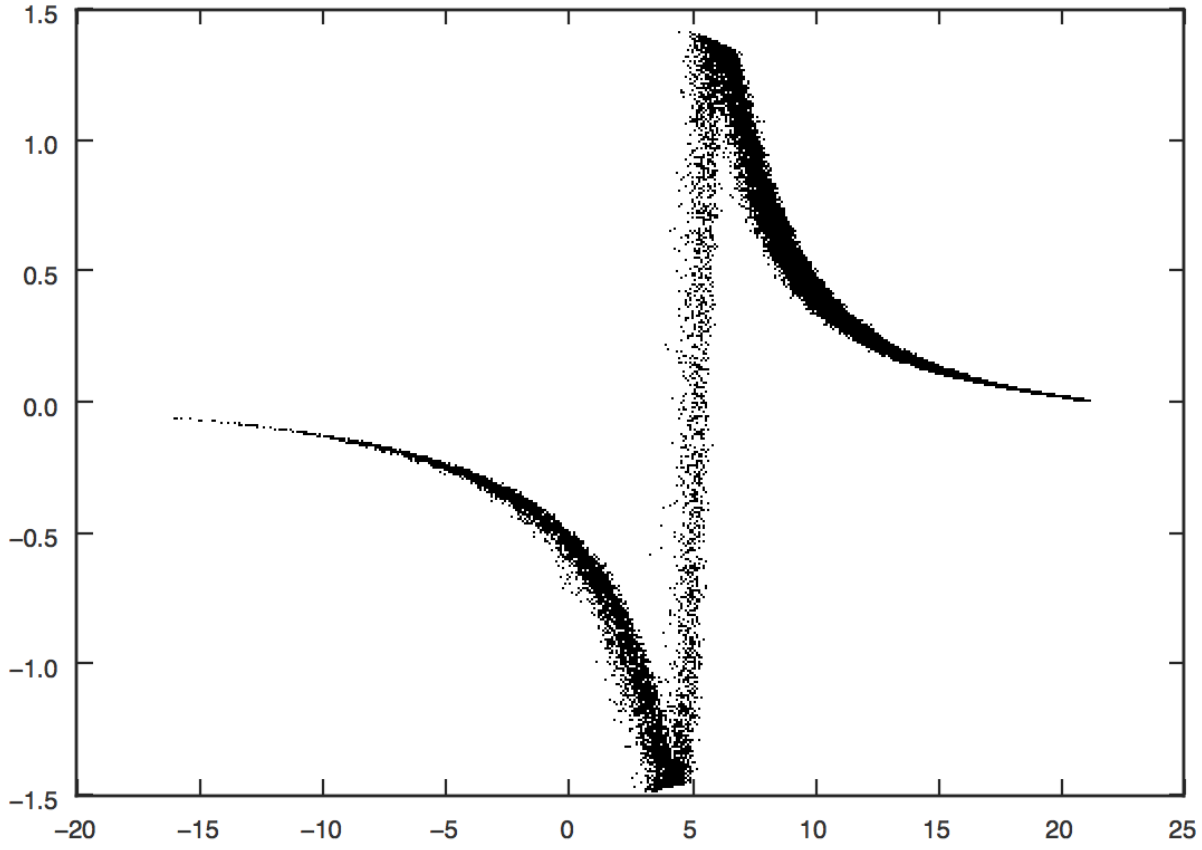
We can then sample positions and velocities for the perturbed and unperturbed prediction for the *same* particle by using the same random seed:

```
>>> numpy.random.seed(1)
>>> xv_mock_per= sdf_sanders15.sample(n=100000,xy=True).T
>>> numpy.random.seed(1) # should give same points
>>> xv_mock_unp= sdf_sanders15_unp.sample(n=100000,xy=True).T
```

and we can plot the offset due to the perturbation, for example,

```
>>> plot(xv_mock_unp[:,0]*R0,(xv_mock_per[:,0]-xv_mock_unp[:,0])*R0,'k,')
```

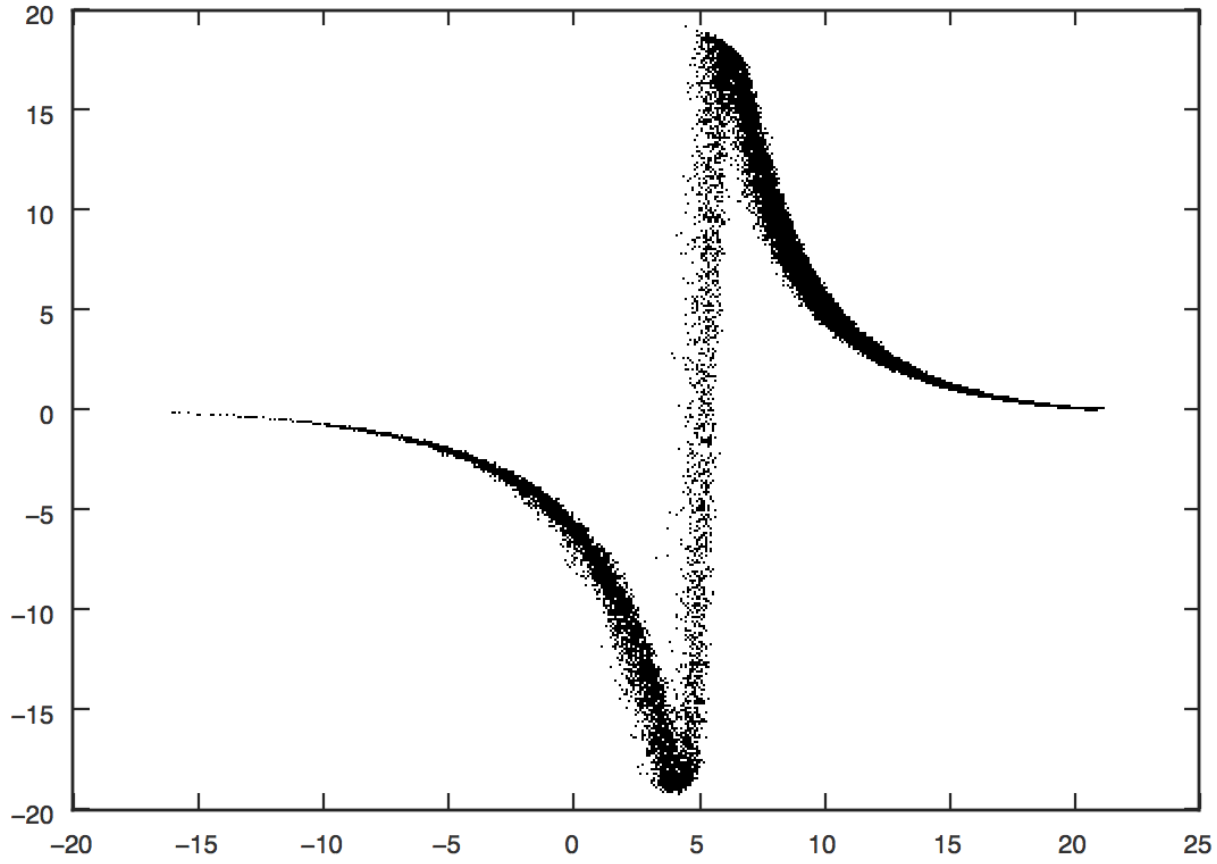
for the difference in X as a function of unperturbed X:



or

```
>>> plot(xv_mock_unp[:,0]*R0,(xv_mock_per[:,4]-xv_mock_unp[:,4])*V0,'k,')
```

for the difference in v_Y as a function of unperturbed X:


20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20
25

CHAPTER 2

Library reference

2.1 Orbit (`galpy.orbit`)

See *Orbit initialization* for a detailed explanation on how to set up Orbit instances.

2.1.1 Initialization

galpy.orbit.Orbit

`Orbit.__init__`(*vxvv=None*, *ro=None*, *vo=None*, *zo=None*, *solarmotion=None*, *radec=False*, *uvw=False*, *lb=False*)

NAME:

__init__

PURPOSE:

Initialize an Orbit instance

INPUT:

vxvv - initial conditions (must all have the same phase-space dimension); can be either

a) astropy (>v3.0) SkyCoord with arbitrary shape, including velocities (note that this turns *on* physical output even if ro and vo are not given)

b) array of arbitrary shape (shape,phasedim) (shape of the orbits, followed by the phase-space dimension of the orbit); shape information is retained and used in outputs; elements can be either

1) in Galactocentric cylindrical coordinates with phase-space coordinates arranged as [R,vR,vT(,z,vz,phi)]; needs to be in internal units (for Quantity input; see 'list' option below)

2) [ra,dec,d,mu_ra, mu_dec,vlos] in [deg,deg,kpc,mas/yr,mas/yr,km/s] (ICRS; mu_ra = mu_ra * cos dec); (for Quantity input, see 'list' option below);

4) [ra,dec,d,U,V,W] in [deg,deg,kpc,km/s,km/s,kms]; (for Quantity input; see 'list' option below); ICRS frame

5) (l,b,d,mu_l, mu_b, vlos) in [deg,deg,kpc,mas/yr,mas/yr,km/s) (mu_l = mu_l * cos b); (for Quantity input; see 'list' option below)

6) [l,b,d,U,V,W] in [deg,deg,kpc,km/s,km/s,kms]; (for Quantity input; see 'list' option below)

5) and 6) also work when leaving out b and mu_b/W

c) lists of initial conditions, entries can be

1) individual Orbit instances (of single objects)

2) Quantity arrays arranged as in section 2) above (so things like [R,vR,vT,z,vz,phi], where R, vR, ... can be arbitrary shape Quantity arrays)

3) list of Quantities (so things like [R1,vR1,..,], where R1, vR1, ... are scalar Quantities

4) None: assumed to be the Sun; if None occurs in a list it is assumed to be the Sun *and all other items in the list are assumed to be [ra,dec,...]*; cannot be combined with Quantity lists (2 and 3 above)

5) lists of scalar phase-space coordinates arranged as in b) (so things like [R,vR,...] where R,vR are scalars in internal units

OPTIONAL INPUTS:

ro - distance from vantage point to GC (kpc; can be Quantity)

vo - circular velocity at ro (km/s; can be Quantity)

zo - offset toward the NGP of the Sun wrt the plane (kpc; can be Quantity; default = 20.8 pc from Bennett & Bovy 2019)

solarmotion - 'hogg' or 'dehnen', or 'schoenrich', or value in [-U,V,W]; can be Quantity

OUTPUT:

instance

HISTORY:

2018-10-13 - Written - Mathew Bub (UofT)

2019-01-01 - Better handling of unit/coordinate-conversion parameters and consistency checks - Bovy (UofT)

2019-02-01 - Handle array of SkyCoords in a faster way by making use of the fact that array of SkyCoords is processed correctly by Orbit

2019-02-18 - Don't support radec, lb, or uvw keywords to avoid slow coordinate transformations that would require ugly code to fix - Bovy (UofT)

2019-03-19 - Allow array vxvv and arbitrary shapes - Bovy (UofT)

galpy.orbit.Orbit.from_fit

classmethod Orbit.**from_fit**(*init_vxvv*, *vxvv*, *vxvv_err=None*, *pot=None*, *radec=False*, *lb=False*, *customsky=False*, *lb_to_customsky=None*, *pmllpmbb_to_customsky=None*, *tintJ=10*, *ntintJ=1000*, *integrate_method='dopr54_c'*, *ro=None*, *vo=None*, *zo=None*, *solarmotion=None*, *disp=False*)

NAME:

from_fit

PURPOSE:

Initialize an Orbit using a fit to data

INPUT:

init_vxvv - initial guess for the fit (same representation [e.g.,radec=True] as vxvv data, except when customsky, then init_vxvv is assumed to be ra,dec)

vxvv - [:,6] array of positions and velocities along the orbit (if not lb=True or radec=True, these need to be in natural units [/ro,/vo], cannot be Quantities)

vxvv_err= [:,6] array of errors on positions and velocities along the orbit (if None, these are set to 0.01) (if not lb=True or radec=True, these need to be in natural units [/ro,/vo], cannot be Quantities)

pot= Potential to fit the orbit in

Keywords related to the input data:

radec= if True, input vxvv and vxvv are [ra,dec,d,mu_ra, mu_dec,vlos] in [deg,deg,kpc,mas/yr,mas/yr,km/s] (all J2000.0; mu_ra = mu_ra * cos dec); the attributes of the current Orbit are used to convert between these coordinates and Galactocentric coordinates

lb= if True, input vxvv and vxvv are [long,lat,d,mu_ll, mu_bb,vlos] in [deg,deg,kpc,mas/yr,mas/yr,km/s] (mu_ll = mu_ll * cos lat); the attributes of the current Orbit are used to convert between these coordinates and Galactocentric coordinates

customsky= if True, input vxvv and vxvv_err are [custom long,custom lat,d,mu_customll, mu_custombb,vlos] in [deg,deg,kpc,mas/yr,mas/yr,km/s] (mu_ll = mu_ll * cos lat) where custom longitude and custom latitude are a custom set of sky coordinates (e.g., ecliptic) and the proper motions are also expressed in these coordinats; you need to provide the functions lb_to_customsky and pmllpmbb_to_customsky to convert to the custom sky coordinates (these should have the same inputs and outputs as lb_to_radec and pmllpmbb_to_pmrapmdec); the attributes of the current Orbit are used to convert between these coordinates and Galactocentric coordinates

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s; entries can be Quantity) (default=Object-wide default) Cannot be an Orbit instance with the orbit of the reference point, as w/ the ra etc. functions Y is ignored and always assumed to be zero

lb_to_customsky= function that converts l,b,degree=False to the custom sky coordinates (like lb_to_radec); needs to be given when customsky=True

pmllpmbb_to_customsky= function that converts pmll,pmbb,l,b,degree=False to proper motions in the custom sky coordinates (like pmllpmbb_to_pmrapmdec); needs to be given when customsky=True

Keywords related to the orbit integrations:

tintJ= (default: 10) time to integrate orbits for fitting the orbit (can be Quantity)

ntintJ= (default: 1000) number of time-integration points

integrate_method= (default: 'dopr54_c') integration method to use

Keywords related to the coordinate transformation:

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

zo= offset toward the NGP of the Sun wrt the plane (kpc; can be Quantity; default = 20.8 pc from Bennett & Bovy 2019)

solarmotion= 'hogg' or 'dehnen', or 'schoenrich', or value in [-U,V,W]; can be Quantity

disp= (False) display the optimizer's convergence message

OUTPUT:

Orbit instance

HISTORY:

2014-06-17 - Written - Bovy (IAS)

2019-05-22 - Incorporated into new Orbit class as from_fit - Bovy (UofT)

galpy.orbit.Orbit.from_name

classmethod Orbit.**from_name**(**args*, ***kwargs*)

NAME:

from_name

PURPOSE:

given the name of an object or a list of names, retrieve coordinate information for that object from SIMBAD and return a corresponding orbit

INPUT:

name - the name of the object or list of names; when loading a collection of objects (like 'mwglobularclusters'), lists are not allowed

+standard Orbit initialization keywords:

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

zo= offset toward the NGP of the Sun wrt the plane (kpc; can be Quantity; default = 20.8 pc from Bennett & Bovy 2019)

solarmotion= 'hogg' or 'dehnen', or 'schoenrich', or value in [-U,V,W]; can be Quantity

OUTPUT:

orbit containing the phase space coordinates of the named object

HISTORY:

2018-07-15 - Written - Mathew Bub (UofT)

2019-05-21 - Generalized to multiple objects and incorporated into Orbits - Bovy (UofT)

2.1.2 Plotting

galpy.orbit.Orbit.animate

Orbit.**animate**(**args*, ***kwargs*)

NAME:

animate

PURPOSE:

animate a previously calculated orbit (with reasonable defaults)

INPUT:

d1= first dimension to plot ('x', 'y', 'R', 'vR', 'vT', 'z', 'vz', ...); can be list with up to three entries for three subplots; each entry can also be a user-defined function of time (e.g., lambda t: o.R(t) for R)

d2= second dimension to plot; can be list with up to three entries for three subplots; each entry can also be a user-defined function of time (e.g., lambda t: o.R(t) for R)

width= (600) width of output div in px

height= (400) height of output div in px

xlabel= (pre-defined labels) label for the first dimension (or list of labels if d1 is a list); should only have to be specified when using a function as d1 and can then specify as, e.g., [None,'YOUR LABEL',None] if d1 is a list of three xs and the first and last are standard entries)

ylabel= (pre-defined labels) label for the second dimension (or list of labels if d2 is a list); should only have to be specified when using a function as d2 and can then specify as, e.g., [None,'YOUR LABEL',None] if d1 is a list of three xs and the first and last are standard entries)

json_filename= (None) if set, save the data necessary for the figure in this filename (e.g., json_filename= 'orbit_data/orbit.json'); this path is also used in the output HTML, so needs to be accessible

staticPlot= (False) if True, create a static plot that doesn't allow zooming, panning, etc.

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

IPython.display.HTML object with code to animate the orbit; can be directly shown in jupyter notebook or embedded in HTML pages; get a text version of the HTML using the _repr_html_() function

HISTORY:

2017-09-17-24 - Written - Bovy (UofT)

2019-03-11 - Adapted for multiple orbits - Bovy (UofT)

galpy.orbit.Orbit.plot

Orbit.**plot**(**args*, ***kwargs*)

NAME:

plot

PURPOSE:

plot a previously calculated orbit (with reasonable defaults)

INPUT:

d1= first dimension to plot ('x', 'y', 'R', 'vR', 'vT', 'z', 'vz', ...); can also be an expression, like 'R*vR', or a user-defined function of time (e.g., lambda t: o.R(t) for R)

d2= second dimension to plot; can also be an expression, like 'R*vR', or a user-defined function of time (e.g., lambda t: o.R(t) for R)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

matplotlib.plot inputs+galpy.util.plot.plot inputs

OUTPUT:

sends plot to output device

HISTORY:

2010-07-26 - Written - Bovy (NYU)

2010-09-22 - Adapted to more general framework - Bovy (NYU)

2013-11-29 - added ra,dec kwargs and other derived quantities - Bovy (IAS)

2014-06-11 - Support for plotting in physical coordinates - Bovy (IAS)

2017-11-28 - Allow arbitrary functions of time to be plotted - Bovy (UofT)

2019-04-13 - Edited for multiple Orbits - Bovy (UofT)

galpy.orbit.Orbit.plot3d

Orbit.**plot3d**(**args*, ***kwargs*)

NAME:

plot3d

PURPOSE:

plot 3D aspects of an Orbit

INPUT:

d1= first dimension to plot ('x', 'y', 'R', 'vR', 'vT', 'z', 'vz', ...); can also be an expression, like 'R*vR', or a user-defined function of time (e.g., lambda t: o.R(t) for R)

d2= second dimension to plot

d3= third dimension to plot

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

galpy.util.plot.plot3d args and kwargs

OUTPUT:

plot

HISTORY:

2010-07-26 - Written - Bovy (NYU)

2010-09-22 - Adapted to more general framework - Bovy (NYU)

2010-01-08 - Adapted to 3D - Bovy (NYU)

2013-11-29 - added ra,dec kwargs and other derived quantities - Bovy (IAS)

2014-06-11 - Support for plotting in physical coordinates - Bovy (IAS)

2017-11-28 - Allow arbitrary functions of time to be plotted - Bovy (UofT)

2019-04-13 - Adapated for multiple orbits - Bovy (UofT)

In addition to these methods, any calculable attribute listed below can be plotted versus other attributes using `plotATTR`, where `ATTR` is an attribute like `R`, `ll`, etc. In this case, the y axis will have `ATTR` and the overrideable x axis default is time. For example, `o.plotR()` will plot the orbit's R vs time.

2.1.3 Attributes

- `Orbit.`**`name Name(s) of objects initialized using ``Orbit.from_name```**
- `Orbit.`**`shape Tuple of Orbit dimensions`**
- `Orbit.`**`size Total number of elements in the Orbit instance`**

2.1.4 Methods

galpy.orbit.Orbit.__call__

`Orbit.`**`__call__`**(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

return the orbits at time t

INPUT:

t - desired time (can be Quantity)

OUTPUT:

an Orbit instance with initial conditions set to the phase-space at time t; shape of new Orbit is (shape_old,nt)

HISTORY:

2019-03-05 - Written - Bovy (UofT)

2019-03-20 - Implemented multiple times –> Orbits - Bovy (UofT)

galpy.orbit.Orbit.__getitem__

Orbit.**__getitem__**(*key*)

NAME:

__getitem__

PURPOSE:

get a subset of this instance's orbits

INPUT:

key - slice

OUTPUT:

For single item: Orbit instance, for multiple items: another Orbit instance

HISTORY:

2018-12-31 - Written - Bovy (UofT)

galpy.orbit.Orbit.bb

Orbit.**bb**(**args*, ***kwargs*)

NAME:

bb

PURPOSE:

return Galactic latitude

INPUT:

t - (optional) time at which to get bb

obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

b(t) [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT

galpy.orbit.Orbit.dec

Orbit.**dec**(**args*, ***kwargs*)

NAME:

dec

PURPOSE:

return the declination

INPUT:

t - (optional) time at which to get dec

obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

dec(t) [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.dim

Orbit.**dim**()

NAME:

dim

PURPOSE:

return the dimension of the Orbit

INPUT:

(none)

OUTPUT:

dimension

HISTORY:

2011-02-03 - Written - Bovy (NYU)

galpy.orbit.Orbit.dist

Orbit.**dist**(*args, **kwargs)

NAME:

dist

PURPOSE:

return distance from the observer in kpc

INPUT:

t - (optional) time at which to get dist

obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

dist(t) in kpc [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.E

Orbit.**E**(**args*, ***kwargs*)

NAME:

E

PURPOSE:

calculate the energy

INPUT:

t - (optional) time at which to get the energy (can be Quantity)

pot= Potential instance or list of such instances

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

energy [*input_shape,nt]

HISTORY:

2019-03-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.e

Orbit.**e**(*analytic=False*, *pot=None*, ***kwargs*)

NAME:

e

PURPOSE:

calculate the eccentricity, either numerically from the numerical orbit integration or using analytical means

INPUT:

analytic(= False) compute this analytically

pot - potential to use for analytical calculation

For 3D orbits different approximations for analytic=True are available (see the EccZmaxRperiRap method of actionAngle modules):

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic': assuming motion splits into R and z
2) 'staeckel': assuming motion splits into u and v of prolate spheroidal coordinate system, exact for Staeckel potentials (incl. all spherical potentials)
3) 'spherical': for spherical potentials, exact

+actionAngle module setup kwargs for the corresponding actionAngle modules (actionAngleAdiabatic, actionAngleStaeckel, and actionAngleSpherical)

OUTPUT:

eccentricity [*input_shape]

HISTORY:

2019-02-25 - Written - Bovy (UofT)

galpy.orbit.Orbit.ER

Orbit.**ER**(**args*, ***kwargs*)

NAME:

ER

PURPOSE:

calculate the radial energy

INPUT:

t - (optional) time at which to get the radial energy (can be Quantity)

pot= Potential instance or list of such instances

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output (can be Quantity)

OUTPUT:

radial energy [*input_shape,nt]

HISTORY:

2019-03-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.Ez

Orbit.**Ez**(**args*, ***kwargs*)

NAME:

Ez

PURPOSE:

calculate the vertical energy

INPUT:

t - (optional) time at which to get the vertical energy (can be Quantity)

pot= Potential instance or list of such instances

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output (can be Quantity)

OUTPUT:

vertical energy [*input_shape,nt]

HISTORY:

2019-03-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.flip

Orbit.**flip**(*inplace=False*)

NAME:

flip

PURPOSE:

'flip' an orbit's initial conditions such that the velocities are minus the original velocities; useful for quick backward integration; returns a new Orbit instance

INPUT:

inplace= (False) if True, flip the orbit in-place, that is, without returning a new instance and also flip the velocities of the integrated orbit (if it exists)

OUTPUT:

Orbit instance that has the velocities of the current orbit flipped (inplace=False) or just flips all velocities of current instance (inplace=True)

HISTORY:

2019-03-02 - Written - Bovy (UofT)

galpy.orbit.Orbit.integrate

Orbit.**integrate**(*t*, *pot*, *method='symplec4_c'*, *progressbar=True*, *dt=None*, *numcores=2*, *force_map=False*)

NAME:

integrate

PURPOSE:

integrate this Orbit instance with multiprocessing

INPUT:

t - list of times at which to output (0 has to be in this!) (can be Quantity)

pot - potential instance or list of instances

method = 'odeint' for scipy's odeint 'leapfrog' for a simple leapfrog implementation 'leapfrog_c' for a simple leapfrog implementation in C 'symplec4_c' for a 4th order symplectic integrator in C 'symplec6_c' for a 6th order symplectic integrator in C 'rk4_c' for a 4th-order Runge-Kutta integrator in C 'rk6_c' for a 6-th order Runge-Kutta integrator in C 'dopr54_c' for a 5-4 Dormand-Prince integrator in C 'dop853' for a 8-5-3 Dormand-Prince integrator in Python 'dop853_c' for a 8-5-3 Dormand-Prince integrator in C

progressbar= (True) if True, display a tqdm progress bar when integrating multiple orbits (requires tqdm to be installed!)

dt - if set, force the integrator to use this basic stepsize; must be an integer divisor of output stepsize (only works for the C integrators that use a fixed stepsize) (can be Quantity)

numcores - number of cores to use for Python-based multiprocessing (pure Python or using force_map=True); default = OMP_NUM_THREADS

force_map= (False) if True, force use of Python-based multiprocessing (not recommended)

OUTPUT:

None (get the actual orbit using getOrbit())

HISTORY:

2018-10-13 - Written as parallel_map applied to regular Orbit integration - Mathew Bub (UofT)

2018-12-26 - Written to use OpenMP C implementation - Bovy (UofT)

galpy.orbit.Orbit.integrate_dxdv

Currently only supported for `planarOrbit` instances.

`Orbit.`**`integrate_dxdv`**(*dxdv*, *t*, *pot*, *method='dopr54_c'*, *progressbar=True*, *dt=None*, *numcores=2*, *force_map=False*, *rectIn=False*, *rectOut=False*)

NAME:

integrate_dxdv

PURPOSE:

integrate the orbit and a small area of phase space

INPUT:

dxdv - [dR,dvR,dvT,dphi], shape=(*input_shape,4)

t - list of times at which to output (0 has to be in this!) (can be Quantity)

pot - potential instance or list of instances

progressbar= (True) if True, display a tqdm progress bar when integrating multiple orbits (requires tqdm to be installed!)

dt - if set, force the integrator to use this basic stepsize; must be an integer divisor of output stepsize (only works for the C integrators that use a fixed stepsize) (can be Quantity)

method = 'odeint' for scipy's odeint 'rk4_c' for a 4th-order Runge-Kutta integrator in C 'rk6_c' for a 6-th order Runge-Kutta integrator in C 'dopr54_c' for a 5-4 Dormand-Prince integrator in C 'dopr853_c' for a 8-5-3 Dormand-Prince integrator in C

rectIn= (False) if True, input dxdv is in rectangular coordinates

rectOut= (False) if True, output dxdv (that in orbit_dxdv) is in rectangular coordinates

numcores - number of cores to use for Python-based multiprocessing (pure Python or using force_map=True); default = OMP_NUM_THREADS

force_map= (False) if True, force use of Python-based multiprocessing (not recommended)

OUTPUT:

(none) (get the actual orbit using getOrbit_dxdv(), the orbit that is integrated alongside with dxdv is stored as usual, any previous regular orbit integration will be erased!)

HISTORY:

2011-10-17 - Written - Bovy (IAS)

2014-06-29 - Added rectIn and rectOut - Bovy (IAS)

2019-05-21 - Parallelized and incorporated into new Orbits class - Bovy (UofT)

galpy.orbit.Orbit.getOrbit

Orbit.**getOrbit**()

NAME:

getOrbit

PURPOSE:

return previously calculated orbits

INPUT:

(none)

OUTPUT:

array orbit[*input_shape,nt,nphasedim]

HISTORY:

2019-03-02 - Written - Bovy (UofT)

galpy.orbit.Orbit.getOrbit_dxdv

`integrate_dxdv` is currently only supported for `planarOrbit` instances. `getOrbit_dxdv` is therefore also only supported for those types of `Orbit`.

Orbit.**getOrbit_dxdv**()

NAME:

getOrbit_dxdv

PURPOSE:

return a previously calculated integration of a small phase-space volume (with integrate_dxdv)

INPUT:

(none)

OUTPUT:

array orbit[*input_shape,nt,nphasedim]

HISTORY:

2019-05-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.helioX

Orbit.**helioX**(**args*, ***kwargs*)

NAME:

helioX

PURPOSE:

return Heliocentric Galactic rectangular x-coordinate (aka "X")

INPUT:

t - (optional) time at which to get X

obs=[X,Y,Z] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

helioX(t) in kpc [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.helioY

Orbit.**helioY**(**args*, ***kwargs*)

NAME:

helioY

PURPOSE:

return Heliocentric Galactic rectangular y-coordinate (aka "Y")

INPUT:

t - (optional) time at which to get Y

obs=[X,Y,Z] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

helioY(t) in kpc [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.helioZ

Orbit.**helioZ**(**args*, ***kwargs*)

NAME:

helioZ

PURPOSE:

return Heliocentric Galactic rectangular z-coordinate (aka "Z")

INPUT:

t - (optional) time at which to get Z

obs=[X,Y,Z] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

helioZ(t) in kpc [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.Jacobi

Orbit.**Jacobi**(**args*, ***kwargs*)

NAME:

Jacobi

PURPOSE:

calculate the Jacobi integral E - Omega L

INPUT:

t - (optional) time at which to get the Jacobi integral (can be Quantity)

OmegaP= pattern speed (can be Quantity)

pot= potential instance or list of such instances

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Jacobi integral [*input_shape,nt]

HISTORY:

2019-03-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.jp

Orbit.**jp**(*pot=None*, ***kwargs*)

NAME:

jp

PURPOSE:

calculate the azimuthal action

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

jp [*input_shape]

HISTORY:

2019-02-26 - Written - Bovy (UofT)

galpy.orbit.Orbit.jr

Orbit.**jr**(*pot=None*, ***kwargs*)

NAME:

jr

PURPOSE:

calculate the radial action

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

jr [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.jz

Orbit.**jz**(*pot=None*, ***kwargs*)

NAME:

jz

PURPOSE:

calculate the vertical action

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

jz [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.ll

Orbit.**ll**(**args*, ***kwargs*)

NAME:

ll

PURPOSE:

return Galactic longitude

INPUT:

t - (optional) time at which to get ll

obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

l(t) [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.L

Orbit.**L**(**args*, ***kwargs*)

NAME:

L

PURPOSE:

calculate the angular momentum at time t

INPUT:

t - (optional) time at which to get the angular momentum (can be Quantity)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

angular momentum [*input_shape,nt,3]

HISTORY:

2019-03-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.LcE

Orbit.**LcE**(**args*, ***kwargs*)

NAME:

LcE

PURPOSE:

calculate the angular momentum of a circular orbit with the same energy

INPUT:

pot= potential instance or list of such instances

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

L_c(E) [*input_shape,nt]

HISTORY:

2022-04-07 - Written - Bovy (UofT)

galpy.orbit.Orbit.Lz

Orbit.**Lz**(**args*, ***kwargs*)

NAME:

Lz

PURPOSE:

calculate the z-component of the angular momentum at time t

INPUT:

t - (optional) time at which to get the angular momentum (can be Quantity)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

z-component of the angular momentum [*input_shape,nt]

HISTORY:

2019-03-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.Op

Orbit.**Op**(*pot=None*, ***kwargs*)

NAME:

Op

PURPOSE:

calculate the azimuthal frequency

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Op [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.Or

Orbit.**Or**(*pot=None*, ***kwargs*)

NAME:

Or

PURPOSE:

calculate the radial frequency

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Or [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.Oz

Orbit.**Oz**(*pot=None*, ***kwargs*)

NAME:

Oz

PURPOSE:

calculate the vertical frequency

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Oz [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.phasedim

Orbit.**phasedim**()

NAME:

phasedim

PURPOSE:

return the phase-space dimension of the problem (2 for 1D, 3 for 2D-axi, 4 for 2D, 5 for 3D-axi, 6 for 3D)

INPUT:

(none)

OUTPUT:

phase-space dimension

HISTORY:

2018-12-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.phi

Orbit.**phi**(**args*, ***kwargs*)

NAME:

phi

PURPOSE:

return azimuth

INPUT:

t - (optional) time at which to get the azimuth

OUTPUT:

phi(t) [*input_shape,nt] in [-pi,pi]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.pmbb

Orbit.**pmbb**(**args*, ***kwargs*)

NAME:

pmbb

PURPOSE:

return proper motion in Galactic latitude (in mas/yr)

INPUT:

t - (optional) time at which to get pmbb

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

pm_b(t) in mas/yr [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.pmdec

Orbit.**pmdec**(**args*, ***kwargs*)

NAME:

pmdec

PURPOSE:

return proper motion in declination (in mas/yr)

INPUT:

t - (optional) time at which to get pmdec

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

pm_dec(t) in mas/yr [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.pmll

Orbit.**pmll**(**args*, ***kwargs*)

NAME:

pmll

PURPOSE:

return proper motion in Galactic longitude (in mas/yr)

INPUT:

t - (optional) time at which to get pmll

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

pm_l(t) in mas/yr [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.pmra

Orbit.**pmra**(**args*, ***kwargs*)

NAME:

pmra

PURPOSE:

return proper motion in right ascension (in mas/yr)

INPUT:

t - (optional) time at which to get pmra

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

pm_ra(t) in mas / yr [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.r

Orbit.**r**(**args*, ***kwargs*)

NAME:

r

PURPOSE:

return spherical radius at time t

INPUT:

t - (optional) time at which to get the radius

ro= (Object-wide default) physical scale for distances to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

r(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.R

Orbit.**R**(**args*, ***kwargs*)

NAME:

R

PURPOSE:

return cylindrical radius at time t

INPUT:

t - (optional) time at which to get the radius (can be Quantity)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

R(t) [*input_shape,nt]

HISTORY:

2019-02-01 - Written - Bovy (UofT)

galpy.orbit.Orbit.ra

Orbit.**ra**(**args*, ***kwargs*)

NAME:

ra

PURPOSE:

return the right ascension

INPUT:

t - (optional) time at which to get ra

obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

OUTPUT:

ra(t) [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.rap

`Orbit.rap`(*analytic=False*, *pot=None*, ***kwargs*)

NAME:

rap

PURPOSE:

calculate the apocenter radius, either numerically from the numerical orbit integration or using analytical means

INPUT:

analytic(= False) compute this analytically

pot - potential to use for analytical calculation

For 3D orbits different approximations for analytic=True are available (see the EccZmaxRperiRap method of actionAngle modules):

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic': assuming motion splits into R and z
2) 'staeckel': assuming motion splits into u and v of prolate spheroidal coordinate system, exact for Staeckel potentials (incl. all spherical potentials)
3) 'spherical': for spherical potentials, exact

+actionAngle module setup kwargs for the corresponding actionAngle modules (actionAngleAdiabatic, actionAngleStaeckel, and actionAngleSpherical)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

R_ap [*input_shape]

HISTORY:

2019-02-25 - Written - Bovy (UofT)

galpy.orbit.Orbit.rE

Orbit.**rE**(**args*, ***kwargs*)

NAME:

rE

PURPOSE:

calculate the radius of a circular orbit with the same energy

INPUT:

pot= potential instance or list of such instances

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

r_E [*input_shape,nt]

HISTORY:

2022-04-07 - Written as thin wrapper around Potential.rE - Bovy (UofT)

galpy.orbit.Orbit.reshape

Orbit.**reshape**(*newshape*)

NAME:

reshape

PURPOSE:

Change the shape of the Orbit instance

INPUT:

newshape - new shape (int or tuple of ints; see numpy.reshape)

OUTPUT:

(none; re-shaping is done in-place)

HISTORY:

2019-03-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.rguiding

Orbit.**rguiding**(**args*, ***kwargs*)

NAME:

rguiding

PURPOSE:

calculate the guiding-center radius (the radius of a circular orbit with the same angular momentum)

INPUT:

pot= potential instance or list of such instances

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

R_guiding [*input_shape,nt]

HISTORY:

2019-03-02 - Written as thin wrapper around Potential.rl - Bovy (UofT)

galpy.orbit.Orbit.rperi

`Orbit.`**`rperi`**(*analytic=False*, *pot=None*, ***kwargs*)

NAME:

rperi

PURPOSE:

calculate the pericenter radius, either numerically from the numerical orbit integration or using analytical means

INPUT:

analytic(= False) compute this analytically

pot - potential to use for analytical calculation

For 3D orbits different approximations for analytic=True are available (see the EccZmaxRperiRap method of actionAngle modules):

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic': assuming motion splits into R and z
2) 'staeckel': assuming motion splits into u and v of prolate spheroidal coordinate system, exact for Staeckel potentials (incl. all spherical potentials)
3) 'spherical': for spherical potentials, exact

+actionAngle module setup kwargs for the corresponding actionAngle modules (actionAngleAdiabatic, actionAngleStaeckel, and actionAngleSpherical)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

R_peri [*input_shape]

HISTORY:

2019-02-25 - Written - Bovy (UofT)

galpy.orbit.Orbit.SkyCoord

Orbit.**SkyCoord**(**args*, ***kwargs*)

NAME:

SkyCoord

PURPOSE:

return the positions and velocities as an astropy SkyCoord

INPUT:

t - (optional) time at which to get the position

obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

SkyCoord(t) [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.theta

Orbit.**theta**(**args*, ***kwargs*)

NAME:

theta

PURPOSE:

return spherical polar angle

INPUT:

t - (optional) time at which to get the angle

OUTPUT:

theta(t) [*input_shape,nt]

HISTORY:

2020-07-01 - Written - James Lane (UofT)

galpy.orbit.Orbit.time

Orbit.**time**(**args*, ***kwargs*)

NAME:

time

PURPOSE:

return the times at which the orbit is sampled

INPUT:

t - (default: integration times) time at which to get the time (for consistency reasons); default is to return the list of times at which the orbit is sampled

ro= (Object-wide default) physical scale for distances to use to convert

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

t(t)

HISTORY:

2019-02-28 - Written - Bovy (UofT)

galpy.orbit.Orbit.toLinear

`Orbit.toLinear()`

NAME:

toLinear

PURPOSE:

convert 3D orbits into 1D orbits (z)

INPUT:

(none)

OUTPUT:

linear Orbit instance

HISTORY:

2019-03-02 - Written - Bovy (UofT)

galpy.orbit.Orbit.toPlanar

`Orbit.toPlanar()`

NAME:

toPlanar

PURPOSE:

convert 3D orbits into 2D orbits

INPUT:

(none)

OUTPUT:

planar Orbit instance

HISTORY:

2019-03-02 - Written - Bovy (UofT)

galpy.orbit.Orbit.Tp

Orbit.**Tp**(*pot=None*, ***kwargs*)

NAME:

Tp

PURPOSE:

calculate the azimuthal period

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Tp [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.Tr

Orbit.**Tr**(*pot=None*, ***kwargs*)

NAME:

Tr

PURPOSE:

calculate the radial period

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Tr [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.TrTp

Orbit.**TrTp**(*pot=None*, ***kwargs*)

NAME:

TrTp

PURPOSE:

the 'ratio' between the radial and azimuthal period Tr/Tphi*pi

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

OUTPUT:

Tr/Tp*pi [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.turn_physical_off

Orbit.**turn_physical_off**()

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2019-02-28 - Written - Bovy (UofT)

galpy.orbit.Orbit.turn_physical_on

Orbit.**turn_physical_on**(*ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2019-02-28 - Written - Bovy (UofT)

2020-04-22 - Don't turn on a parameter when it is False - Bovy (UofT)

galpy.orbit.Orbit.Tz

Orbit.**Tz**(*pot=None*, ***kwargs*)

NAME:

Tz

PURPOSE:

calculate the vertical period

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Tz [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.U

Orbit.**U**(*args, **kwargs)

NAME:

U

PURPOSE:

return Heliocentric Galactic rectangular x-velocity (aka "U")

INPUT:

t - (optional) time at which to get U

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

U(t) in km/s [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.V

Orbit.**V**(*args, **kwargs)

NAME:

V

PURPOSE:

return Heliocentric Galactic rectangular y-velocity (aka "V")

INPUT:

t - (optional) time at which to get U

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

V(t) in km/s [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.vbb

Orbit.**vbb**(**args*, ***kwargs*)

NAME:

vbb

PURPOSE:

return velocity in Galactic latitude (km/s)

INPUT:

t - (optional) time at which to get vbb (can be Quantity)

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer in the Galactocentric frame (in kpc and km/s) (default=[8.0,0.,0.,0.,220.,0.]; entries can be Quantity) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

OUTPUT:

v_b(t) in km/s [*input_shape]

HISTORY:

2019-02-28 - Written - Bovy (UofT)

galpy.orbit.Orbit.vdec

Orbit.**vdec**(**args*, ***kwargs*)

NAME:

vdec

PURPOSE:

return velocity in declination (km/s)

INPUT:

t - (optional) time at which to get vdec (can be Quantity)

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer in the Galactocentric frame (in kpc and km/s) (default=[8.0,0.,0.,0.,220.,0.]; entries can be Quantity) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

OUTPUT:

v_dec(t) in km/s [*input_shape]

HISTORY:

2019-02-28 - Written - Bovy (UofT)

galpy.orbit.Orbit.vll

Orbit.**vll**(**args*, ***kwargs*)

NAME:

vll

PURPOSE:

return the velocity in Galactic longitude (km/s)

INPUT:

t - (optional) time at which to get vll (can be Quantity)

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer in the Galactocentric frame (in kpc and km/s) (default=[8.0,0.,0.,0.,220.,0.]; entries can be Quantity) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

OUTPUT:

v_l(t) in km/s [*input_shape]

HISTORY:

2019-02-28 - Written - Bovy (UofT)

galpy.orbit.Orbit.vlos

Orbit.**vlos**(**args*, ***kwargs*)

NAME:

vlos

PURPOSE:

return the line-of-sight velocity (in km/s)

INPUT:

t - (optional) time at which to get vlos

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

vlos(t) in km/s [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.vphi

Orbit.**vphi**(**args*, ***kwargs*)

NAME:

vphi

PURPOSE:

return angular velocity

INPUT:

t - (optional) time at which to get the angular velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vphi(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.vr

Orbit.**vr**(**args*, ***kwargs*)

NAME:

vr

PURPOSE:

return spherical radial velocity. For < 3 dimensions returns vR

INPUT:

t - (optional) time at which to get the radial velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vr(t) [*input_shape,nt]

HISTORY:

2020-07-01 - Written - James Lane (UofT)

galpy.orbit.Orbit.vR

Orbit.**vR**(**args*, ***kwargs*)

NAME:

vR

PURPOSE:

return radial velocity at time t

INPUT:

t - (optional) time at which to get the radial velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vR(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.vra

Orbit.**vra**(**args*, ***kwargs*)

NAME:

vra

PURPOSE:

return velocity in right ascension (km/s)

INPUT:

t - (optional) time at which to get vra (can be Quantity)

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer in the Galactocentric frame (in kpc and km/s) (default=[8.0,0.,0.,0.,220.,0.]; entries can be Quantity) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

vo= (Object-wide default) physical scale for velocities to use to convert (can be Quantity)

OUTPUT:

v_ra(t) in km/s [*input_shape]

HISTORY:

2019-02-28 - Written - Bovy (UofT)

galpy.orbit.Orbit.vtheta

Orbit.**vtheta**(**args*, ***kwargs*)

NAME:

vtheta

PURPOSE:

return spherical polar velocity

INPUT:

t - (optional) time at which to get the theta velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vtheta(t) [*input_shape,nt]

HISTORY:

2020-07-01 - Written - James Lane (UofT)

galpy.orbit.Orbit.vT

Orbit.**vT**(**args*, ***kwargs*)

NAME:

vT

PURPOSE:

return tangential velocity at time t

INPUT:

t - (optional) time at which to get the tangential velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vT(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.vx

Orbit.**vx**(**args*, ***kwargs*)

NAME:

vx

PURPOSE:

return x velocity at time t

INPUT:

t - (optional) time at which to get the velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vx(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.vy

Orbit.**vy**(**args*, ***kwargs*)

NAME:

vy

PURPOSE:

return y velocity at time t

INPUT:

t - (optional) time at which to get the velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vy(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.vz

Orbit.**vz**(**args*, ***kwargs*)

NAME:

vz

PURPOSE:

return vertical velocity

INPUT:

t - (optional) time at which to get the vertical velocity

vo= (Object-wide default) physical scale for velocities to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

vz(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.W

Orbit.**W**(**args*, ***kwargs*)

NAME:

W

PURPOSE:

return Heliocentric Galactic rectangular z-velocity (aka "W")

INPUT:

t - (optional) time at which to get W

obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=Object-wide default) OR Orbit object that corresponds to the orbit of the observer; Note that when Y is non-zero, the coordinate system is rotated around z such that Y'=0

ro= distance in kpc corresponding to R=1. (default=Object-wide default)

vo= velocity in km/s corresponding to v=1. (default=Object-wide default)

OUTPUT:

W(t) in km/s [*input_shape,nt]

HISTORY:

2019-02-21 - Written - Bovy (UofT)

galpy.orbit.Orbit.wp

`Orbit.wp(pot=None, **kwargs)`

NAME:

wp

PURPOSE:

calculate the azimuthal angle

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

OUTPUT:

wp [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.wr

`Orbit.wr(pot=None, **kwargs)`

NAME:

wr

PURPOSE:

calculate the radial angle

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

OUTPUT:

wr [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.wz

Orbit.**wz**(*pot=None*, ***kwargs*)

NAME:

wz

PURPOSE:

calculate the vertical angle

INPUT:

pot - potential

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic'
2) 'staeckel'
3) 'isochroneApprox'
4) 'spherical'

+actionAngle module setup kwargs

OUTPUT:

wz [*input_shape]

HISTORY:

2019-02-27 - Written - Bovy (UofT)

galpy.orbit.Orbit.x

Orbit.**x**(**args*, ***kwargs*)

NAME:

x

PURPOSE:

return x

INPUT:

t - (optional) time at which to get x

ro= (Object-wide default) physical scale for distances to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

x(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.y

Orbit.**y**(**args*, ***kwargs*)

NAME:

y

PURPOSE:

return y

INPUT:

t - (optional) time at which to get y

ro= (Object-wide default) physical scale for distances to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

y(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.z

Orbit.**z**(**args*, ***kwargs*)

NAME:

z

PURPOSE:

return vertical height

INPUT:

t - (optional) time at which to get the vertical height

ro= (Object-wide default) physical scale for distances to use to convert

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

z(t) [*input_shape,nt]

HISTORY:

2019-02-20 - Written - Bovy (UofT)

galpy.orbit.Orbit.zmax

`Orbit.`**`zmax`**(*analytic=False*, *pot=None*, ***kwargs*)

NAME:

zmax

PURPOSE:

calculate the maximum vertical height, either numerically from the numerical orbit integration or using analytical means

INPUT:

analytic(= False) compute this analytically

pot - potential to use for analytical calculation

For 3D orbits different approximations for analytic=True are available (see the EccZmaxRperiRap method of actionAngle modules):

type= ('staeckel') type of actionAngle module to use

1) 'adiabatic': assuming motion splits into R and z
2) 'staeckel': assuming motion splits into u and v of prolate spheroidal coordinate system, exact for Staeckel potentials (incl. all spherical potentials)
3) 'spherical': for spherical potentials, exact

+actionAngle module setup kwargs for the corresponding actionAngle modules (actionAngleAdiabatic, actionAngleStaeckel, and actionAngleSpherical)

ro= (Object-wide default) physical scale for distances to use to convert (can be Quantity)

use_physical= use to override Object-wide default for using a physical scale for output

OUTPUT:

Z_max [*input_shape]

HISTORY:

2019-02-25 - Written - Bovy (UofT)

2.2 Potential (galpy.potential)

2.2.1 3D potentials

General instance routines

Use as `Potential-instance.method(...)`

galpy.potential.Potential.__add__

`Potential.__add__(b)`

NAME:

__add__

PURPOSE:

Add Force or Potential instances together to create a multi-component potential (e.g., pot= pot1+pot2+pot3)

INPUT:

b - Force or Potential instance or a list thereof

OUTPUT:

List of Force or Potential instances that represents the combined potential

HISTORY:

2019-01-27 - Written - Bovy (UofT)

2020-04-22 - Added check that unit systems of combined potentials are compatible - Bovy (UofT)

galpy.potential.Potential.__mul__

`Potential.__mul__(b)`

NAME:

__mul__

PURPOSE:

Multiply a Force or Potential's amplitude by a number

INPUT:

b - number

OUTPUT:

New instance with amplitude = (old amplitude) x b

HISTORY:

2019-01-27 - Written - Bovy (UofT)

galpy.potential.Potential.__call__

Warning: `galpy` potentials do *not* necessarily approach zero at infinity. To compute, for example, the escape velocity or whether or not an orbit is unbound, you need to take into account the value of the potential at infinity. E.g., $v_{\rm esc}(r) = \sqrt{2[\Phi(\infty) - \Phi(r)]}$.

`Potential.__call__`(*R*, *z*, *phi=0.0*, *t=0.0*, *dR=0*, *dphi=0*)

NAME:

__call__

PURPOSE:

evaluate the potential at (R,z,phi,t)

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

Phi(R,z,t)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

galpy.potential.Potential.dens

`Potential.dens`(*R*, *z*, *phi=0.0*, *t=0.0*, *forcepoisson=False*)

NAME:

dens

PURPOSE:

evaluate the density rho(R,z,t)

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

KEYWORDS:

forcepoisson= if True, calculate the density through the Poisson equation, even if an explicit expression for the density exists

OUTPUT:

rho (R,z,phi,t)

HISTORY:

2010-08-08 - Written - Bovy (NYU)

galpy.potential.Potential.dvcircdR

Potential.**dvcircdR**(*R*, *phi=None*, *t=0.0*)

NAME:

dvcircdR

PURPOSE:

calculate the derivative of the circular velocity at R wrt R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

phi= (None) azimuth to use for non-axisymmetric potentials

t - time (optional; can be Quantity)

OUTPUT:

derivative of the circular rotation velocity wrt R

HISTORY:

2013-01-08 - Written - Bovy (IAS)

2016-06-28 - Added phi= keyword for non-axisymmetric potential - Bovy (UofT)

galpy.potential.Potential.epifreq

Potential.**epifreq**(*R*, *t=0.0*)

NAME:

epifreq

PURPOSE:

calculate the epicycle frequency at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

epicycle frequency

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.Potential.flattening

Potential.**flattening**(*R*, *z*, *t=0.0*)

NAME:

flattening

PURPOSE:

calculate the potential flattening, defined as sqrt(fabs(z/R F_R/F_z))

INPUT:

R - Galactocentric radius (can be Quantity)

z - height (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

flattening

HISTORY:

2012-09-13 - Written - Bovy (IAS)

galpy.potential.Potential.LcE

Potential.**LcE**(*E*, *t=0.0*)

NAME:

LcE

PURPOSE:

calculate the angular momentum of a circular orbit with energy E

INPUT:

E - Energy (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

Lc(E)

HISTORY:

2022-04-06 - Written - Bovy (UofT)

galpy.potential.Potential.lindbladR

Potential.**lindbladR**(*OmegaP*, *m=2*, *t=0.0*, ***kwargs*)

NAME:

lindbladR

PURPOSE:

calculate the radius of a Lindblad resonance

INPUT:

OmegaP - pattern speed (can be Quantity)

m= order of the resonance (as in m(O-Op)=kappa (negative m for outer) use m='corotation' for corotation +scipy.optimize.brentq xtol,rtol,maxiter kwargs

t - time (optional; can be Quantity)

OUTPUT:

radius of Linblad resonance, None if there is no resonance

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.Potential.mass

Potential.**mass**(*R*, *z=None*, *t=0.0*, *forceint=False*)

NAME:

mass

PURPOSE:

evaluate the mass enclosed

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z= (None) vertical height up to which to integrate (can be Quantity)

t - time (optional; can be Quantity)

forceint= if True, calculate the mass through integration of the density, even if an explicit expression for the mass exists

OUTPUT:

Mass enclosed within the spherical shell with radius R if z is None else mass in the slab <R and between -z and z; except: potentials inheriting from EllipsoidalPotential, which if z is None return the mass within the ellipsoidal shell with semi-major axis R

HISTORY:

2014-01-29 - Written - Bovy (IAS)

2019-08-15 - Added spherical warning - Bovy (UofT)

2021-03-15 - Changed to integrate to spherical shell for z is None slab otherwise - Bovy (UofT)

2021-03-18 - Switched to using Gauss' theorem - Bovy (UofT)

galpy.potential.Potential.nemo_accname

Potential.**nemo_accname**()

NAME:

nemo_accname

PURPOSE:

return the accname potential name for use of this potential with NEMO

INPUT:

(none)

OUTPUT:

Acceleration name

HISTORY:

2014-12-18 - Written - Bovy (IAS)

galpy.potential.Potential.nemo_accpars

Potential.**nemo_accpars**(*vo*, *ro*)

NAME:

nemo_accpars

PURPOSE:

return the accpars potential parameters for use of this potential with NEMO

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

accpars string

HISTORY:

2014-12-18 - Written - Bovy (IAS)

galpy.potential.Potential.omegac

Potential.**omegac**(*R*, *t=0.0*)

NAME:

omegac

PURPOSE:

calculate the circular angular speed at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

circular angular speed

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.Potential.phiforce

Potential.**phiforce**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

phiforce

PURPOSE:

evaluate the azimuthal force F_phi = -d Phi / d phi (R,z,phi,t) [note that this is a torque, not a force!)

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (rad; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

F_phi (R,z,phi,t)

HISTORY:

2010-07-10 - Written - Bovy (NYU)

galpy.potential.Potential.phizderiv

Potential.**phizderiv**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

phizderiv

PURPOSE:

evaluate the mixed azimuthal,vertical derivative

INPUT:

R - Galactocentric radius (can be Quantity)

Z - vertical height (can be Quantity)

phi - Galactocentric azimuth (can be Quantity)

t - time (can be Quantity)

OUTPUT:

d2Phi/dphidz

HISTORY:

2021-04-30 - Written - Bovy (UofT)

galpy.potential.Potential.phi2deriv

Potential.**phi2deriv**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

phi2deriv

PURPOSE:

evaluate the second azimuthal derivative

INPUT:

R - Galactocentric radius (can be Quantity)

Z - vertical height (can be Quantity)

phi - Galactocentric azimuth (can be Quantity)

t - time (can be Quantity)

OUTPUT:

d2Phi/dphi2

HISTORY:

2013-09-24 - Written - Bovy (IAS)

galpy.potential.Potential.plot

Potential.**plot**(*t=0.0*, *rmin=0.0*, *rmax=1.5*, *nrs=21*, *zmin=-0.5*, *zmax=0.5*, *nzs=21*, *effective=False*, *Lz=None*, *phi=None*, *xy=False*, *xrange=None*, *yrange=None*, *justcontours=False*, *levels=None*, *cntrcolors=None*, *ncontours=21*, *savefilename=None*)

NAME:

plot

PURPOSE:

plot the potential

INPUT:

t= time to plot potential at

rmin= minimum R (can be Quantity) [xmin if xy]

rmax= maximum R (can be Quantity) [ymax if xy]

nrs= grid in R

zmin= minimum z (can be Quantity) [ymin if xy]

zmax= maximum z (can be Quantity) [ymax if xy]

nzs= grid in z

phi= (None) azimuth to use for non-axisymmetric potentials

xy= (False) if True, plot the potential in X-Y

effective= (False) if True, plot the effective potential Phi + Lz^2/2/R^2

Lz= (None) angular momentum to use for the effective potential when effective=True

justcontours= (False) if True, just plot contours

savefilename - save to or restore from this savefile (pickle)

xrange, yrange= can be specified independently from rmin,zmin, etc.

levels= (None) contours to plot

ncontours - number of contours when levels is None

cntrcolors= (None) colors of the contours (single color or array with length ncontours)

OUTPUT:

plot to output device

HISTORY:

2010-07-09 - Written - Bovy (NYU)

2014-04-08 - Added effective= - Bovy (IAS)

galpy.potential.Potential.plotDensity

Potential.**plotDensity**(*t=0.0*, *rmin=0.0*, *rmax=1.5*, *nrs=21*, *zmin=-0.5*, *zmax=0.5*, *nzs=21*, *phi=None*, *xy=False*, *ncontours=21*, *savefilename=None*, *aspect=None*, *log=False*, *justcontours=False*, ***kwargs*)

NAME:

plotDensity

PURPOSE:

plot the density of this potential

INPUT:

t= time to plot potential at

rmin= minimum R (can be Quantity) [xmin if xy]

rmax= maximum R (can be Quantity) [ymax if xy]

nrs= grid in R

zmin= minimum z (can be Quantity) [ymin if xy]

zmax= maximum z (can be Quantity) [ymax if xy]

nzs= grid in z

phi= (None) azimuth to use for non-axisymmetric potentials

xy= (False) if True, plot the density in X-Y

ncontours= number of contours

justcontours= (False) if True, just plot contours

savefilename= save to or restore from this savefile (pickle)

log= if True, plot the log density

OUTPUT:

plot to output device

HISTORY:

2014-01-05 - Written - Bovy (IAS)

galpy.potential.Potential.plotEscapecurve

Potential.**plotEscapecurve**(**args*, ***kwargs*)

NAME:

plotEscapecurve

PURPOSE:

plot the escape velocity curve for this potential (in the z=0 plane for non-spherical potentials)

INPUT:

Rrange - range (can be Quantity)

grid= number of points to plot

savefilename= save to or restore from this savefile (pickle)

+galpy.util.plot.plot(*args,**kwargs)

OUTPUT:

plot to output device

HISTORY:

2010-08-08 - Written - Bovy (NYU)

galpy.potential.Potential.plotRotcurve

Potential.**plotRotcurve**(**args*, ***kwargs*)

NAME:

plotRotcurve

PURPOSE:

plot the rotation curve for this potential (in the z=0 plane for non-spherical potentials)

INPUT:

Rrange - range (can be Quantity)

grid= number of points to plot

savefilename=- save to or restore from this savefile (pickle)

+galpy.util.plot.plot(*args,**kwargs)

OUTPUT:

plot to output device

HISTORY:

2010-07-10 - Written - Bovy (NYU)

galpy.potential.Potential.plotSurfaceDensity

Potential.**plotSurfaceDensity**(*t=0.0*, *z=inf*, *xmin=0.0*, *xmax=1.5*, *nxs=21*, *ymin=-0.5*, *ymax=0.5*, *nys=21*, *ncontours=21*, *savefilename=None*, *aspect=None*, *log=False*, *justcontours=False*, ***kwargs*)

NAME:

plotSurfaceDensity

PURPOSE:

plot the surface density of this potential

INPUT:

t= time to plot potential at

z= (inf) height between which to integrate the density (from -z to z; can be a Quantity)

xmin= minimum x (can be Quantity)

xmax= maximum x (can be Quantity)

nxs= grid in x

ymin= minimum y (can be Quantity)

ymax= maximum y (can be Quantity)

nys= grid in y

ncontours= number of contours

justcontours= (False) if True, just plot contours

savefilename= save to or restore from this savefile (pickle)

log= if True, plot the log density

OUTPUT:

plot to output device

HISTORY:

2020-08-19 - Written - Bovy (UofT)

galpy.potential.Potential.R2deriv

`Potential.`**`R2deriv`**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

R2deriv

PURPOSE:

evaluate the second radial derivative

INPUT:

R - Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - Galactocentric azimuth (can be Quantity)

t - time (can be Quantity)

OUTPUT:

d2phi/dR2

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.Potential.r2deriv

Potential.**r2deriv**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

r2deriv

PURPOSE:

evaluate the second spherical radial derivative

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2phi/dr2

HISTORY:

2018-03-21 - Written - Webb (UofT)

galpy.potential.Potential.rE

Potential.**rE**(*E*, *t=0.0*)

NAME:

rE

PURPOSE:

calculate the radius of a circular orbit with energy E

INPUT:

E - Energy (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

radius

HISTORY:

2022-04-06 - Written - Bovy (UofT)

NOTE:

An efficient way to call this function on many objects is provided as the Orbit method rE

galpy.potential.Potential.Rzderiv

Potential.**Rzderiv**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

Rzderiv

PURPOSE:

evaluate the mixed R,z derivative

INPUT:

R - Galactocentric radius (can be Quantity)

Z - vertical height (can be Quantity)

phi - Galactocentric azimuth (can be Quantity)

t - time (can be Quantity)

OUTPUT:

d2phi/dz/dR

HISTORY:

2013-08-26 - Written - Bovy (IAS)

galpy.potential.Potential.Rforce

`Potential.`**`Rforce`**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

Rforce

PURPOSE:

evaluate cylindrical radial force F_R (R,z)

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

F_R (R,z,phi,t)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

galpy.potential.Potential.rforce

`Potential.`**`rforce`**(*R*, *z*, ***kwargs*)

NAME:

rforce

PURPOSE:

evaluate spherical radial force F_r (R,z)

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

v - current velocity in cylindrical coordinates (optional, but required when including dissipative forces; can be a Quantity)

OUTPUT:

F_r (R,z,phi,t)

HISTORY:

2016-06-20 - Written - Bovy (UofT)

galpy.potential.Potential.rhalf

Potential.**rhalf**(*t=0.0*, *INF=inf*)

NAME:

rhalf

PURPOSE:

calculate the half-mass radius, the radius of the spherical shell that contains half the total mass

INPUT:

t= (0.) time (optional; can be Quantity)

INF= (numpy.inf) radius at which the total mass is calculated (internal units, just set this to something very large)

OUTPUT:

half-mass radius

HISTORY:

2021-03-18 - Written - Bovy (UofT)

galpy.potential.Potential.rl

Potential.**rl**(*lz*, *t=0.0*)

NAME:

rl

PURPOSE:

calculate the radius of a circular orbit of Lz

INPUT:

lz - Angular momentum (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

radius

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

NOTE:

An efficient way to call this function on many objects is provided as the Orbit method rguiding

galpy.potential.Potential.Rphideriv

Potential.**Rphideriv**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

Rphideriv

PURPOSE:

evaluate the mixed radial, azimuthal derivative

INPUT:

R - Galactocentric radius (can be Quantity)

Z - vertical height (can be Quantity)

phi - Galactocentric azimuth (can be Quantity)

t - time (can be Quantity)

OUTPUT:

d2Phi/dphidR

HISTORY:

2014-06-30 - Written - Bovy (IAS)

galpy.potential.Potential.rtide

Potential.**rtide**(*R*, *z*, *phi=0.0*, *t=0.0*, *M=None*)

NAME:

rtide

PURPOSE:

Calculate the tidal radius for object of mass M assuming a circular orbit as

$$r_t^3 = \frac{GM_s}{\Omega^2 - \mathrm{d}^2\Phi/\mathrm{d}r^2}$$

where M_s is the cluster mass, Ω is the circular frequency, and Φ is the gravitational potential. For non-spherical potentials, we evaluate $\Omega^2 = (1/r)(\mathrm{d}\Phi/\mathrm{d}r)$ and evaluate the derivatives at the given position of the cluster.

INPUT:

R - Galactocentric radius (can be Quantity)

z - height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

M - (default = None) Mass of object (can be Quantity)

OUTPUT:

Tidal Radius

HISTORY:

2018-03-21 - Written - Webb (UofT)

galpy.potential.Potential.surfdens

Potential.**surfdens**(*R*, *z*, *phi=0.0*, *t=0.0*, *forcepoisson=False*)

NAME:

surfdens

PURPOSE:

evaluate the surface density $\Sigma(R, z, \phi, t) = \int_{-z}^{+z} dz' \rho(R, z', \phi, t)$

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

KEYWORDS:

forcepoisson= if True, calculate the surface density through the Poisson equation, even if an explicit expression for the surface density exists

OUTPUT:

Sigma (R,z,phi,t)

HISTORY:

2018-08-19 - Written - Bovy (UofT)

2021-04-19 - Adjusted for non-z-symmetric densities - Bovy (UofT)

galpy.potential.Potential.tdyn

Potential.**tdyn**(*R*, *t=0.0*)

NAME:

tdyn

PURPOSE:

calculate the dynamical time from tdyn^2 = 3pi/[G<rho>]

INPUT:

R - Galactocentric radius (can be Quantity)

t= (0.) time (optional; can be Quantity)

OUTPUT:

Dynamical time

HISTORY:

2021-03-18 - Written - Bovy (UofT)

galpy.planar.Potential.toPlanar

Potential.**toPlanar**()

NAME:

toPlanar

PURPOSE:

convert a 3D potential into a planar potential in the mid-plane

INPUT:

(none)

OUTPUT:

planarPotential

HISTORY:

unknown

galpy.potential.Potential.toVertical

Potential.**toVertical**(*R*, *phi=None*, *t0=0.0*)

NAME:

toVertical

PURPOSE:

convert a 3D potential into a linear (vertical) potential at R

INPUT:

R - Galactocentric radius at which to create the vertical potential (can be Quantity)

phi= (None) Galactocentric azimuth at which to create the vertical potential (can be Quantity); required for non-axisymmetric potential

t0= (0.) time at which to create the vertical potential (can be Quantity)

OUTPUT:

linear (vertical) potential: Phi(z,phi,t) = Phi(R,z,phi,t)-Phi(R,0.,phi0,t0) where phi0 and t0 are the phi and t inputs

HISTORY

unknown

galpy.potential.Potential.ttensor

Potential.**ttensor**(*R*, *z*, *phi=0.0*, *t=0.0*, *eigenval=False*)

NAME:

ttensor

PURPOSE:

Calculate the tidal tensor Tij=-d(Psi)(dxidxj)

INPUT:

R - Galactocentric radius (can be Quantity)

z - height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

eigenval - return eigenvalues if true (optional; boolean)

OUTPUT:

Tidal Tensor

HISTORY:

2018-03-21 - Written - Webb (UofT)

galpy.potential.Potential.turn_physical_off

Potential.**turn_physical_off**()

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

galpy.potential.Potential.turn_physical_on

Potential.**turn_physical_on**(*ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

2020-04-22 - Don't turn on a parameter when it is False - Bovy (UofT)

galpy.potential.Potential.vcirc

Potential.**vcirc**(*R*, *phi=None*, *t=0.0*)

NAME:

vcirc

PURPOSE:

calculate the circular velocity at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

phi= (None) azimuth to use for non-axisymmetric potentials

t - time (optional; can be Quantity)

OUTPUT:

circular rotation velocity

HISTORY:

2011-10-09 - Written - Bovy (IAS)

2016-06-15 - Added phi= keyword for non-axisymmetric potential - Bovy (UofT)

galpy.potential.Potential.verticalfreq

Potential.**verticalfreq**(*R*, *t=0.0*)

NAME:

verticalfreq

PURPOSE:

calculate the vertical frequency at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

vertical frequency

HISTORY:

2012-07-25 - Written - Bovy (IAS@MPIA)

galpy.potential.Potential.vesc

Potential.**vesc**(*R*, *t=0.0*)

NAME:

vesc

PURPOSE:

calculate the escape velocity at R for this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

escape velocity

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.Potential.vterm

Potential.**vterm**(*l*, *t=0.0*, *deg=True*)

NAME:

vterm

PURPOSE:

calculate the terminal velocity at l in this potential

INPUT:

l - Galactic longitude [deg/rad; can be Quantity)

t - time (optional; can be Quantity)

deg= if True (default), l in deg

OUTPUT:

terminal velocity

HISTORY:

2013-05-31 - Written - Bovy (IAS)

galpy.potential.Potential.z2deriv

Potential.**z2deriv**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

z2deriv

PURPOSE:

evaluate the second vertical derivative

INPUT:

R - Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - Galactocentric azimuth (can be Quantity)

t - time (can be Quantity)

OUTPUT:

d2phi/dz2

HISTORY:

2012-07-25 - Written - Bovy (IAS@MPIA)

galpy.potential.Potential.zforce

`Potential.`**`zforce`**(*R*, *z*, *phi=0.0*, *t=0.0*)

NAME:

zforce

PURPOSE:

evaluate the vertical force F_z (R,z,t)

INPUT:

R - Cylindrical Galactocentric radius (can be Quantity)

z - vertical height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

F_z (R,z,phi,t)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

galpy.potential.Potential.zvc

`Potential.`**`zvc`**(*R*, *E*, *Lz*, *phi=0.0*, *t=0.0*)

NAME:

zvc

PURPOSE:

Calculate the zero-velocity curve: z such that Phi(R,z) + Lz/[2R^2] = E (assumes that F_z(R,z) = negative at positive z such that there is a single solution)

INPUT:

R - Galactocentric radius (can be Quantity)

E - Energy (can be Quantity)

Lz - Angular momentum (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

z such that Phi(R,z) + Lz/[2R^2] = E

HISTORY:

2020-08-20 - Written - Bovy (UofT)

galpy.potential.Potential.zvc_range

`Potential.`**`zvc_range`**(*E*, *Lz*, *phi=0.0*, *t=0.0*)

NAME:

zvc_range

PURPOSE:

Calculate the minimum and maximum radius for which the zero-velocity curve exists for this energy and angular momentum (R such that Phi(R,0) + Lz/[2R^2] = E)

INPUT:

E - Energy (can be Quantity)

Lz - Angular momentum (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

Solutions R such that Phi(R,0) + Lz/[2R^2] = E

HISTORY:

2020-08-20 - Written - Bovy (UofT)

In addition to these, the `NFWPotential` also has methods to calculate virial quantities

galpy.potential.Potential.conc

`Potential.`**`conc`**(*H=70.0*, *Om=0.3*, *t=0.0*, *overdens=200.0*, *wrtcrit=False*, *ro=None*, *vo=None*)

NAME:

conc

PURPOSE:

return the concentration

INPUT:

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

t - time (optional; can be Quantity)

overdens= (200) overdensity which defines the virial radius

wrtcrit= (False) if True, the overdensity is wrt the critical density rather than the mean matter density

ro= distance scale in kpc or as Quantity (default: object-wide, which if not set is 8 kpc))

vo= velocity scale in km/s or as Quantity (default: object-wide, which if not set is 220 km/s))

OUTPUT:

concentration (scale/rvir)

HISTORY:

2014-04-03 - Written - Bovy (IAS)

galpy.potential.Potential.mvir

Potential.**mvir**(*H=70.0*, *Om=0.3*, *t=0.0*, *overdens=200.0*, *wrtcrit=False*, *forceint=False*, *ro=None*, *vo=None*, *use_physical=False*)

NAME:

mvir

PURPOSE:

calculate the virial mass

INPUT:

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

overdens= (200) overdensity which defines the virial radius

wrtcrit= (False) if True, the overdensity is wrt the critical density rather than the mean matter density

ro= distance scale in kpc or as Quantity (default: object-wide, which if not set is 8 kpc))

vo= velocity scale in km/s or as Quantity (default: object-wide, which if not set is 220 km/s))

KEYWORDS:

forceint= if True, calculate the mass through integration of the density, even if an explicit expression for the mass exists

OUTPUT:

M(<rvir)

HISTORY:

2014-09-12 - Written - Bovy (IAS)

galpy.potential.NFWPotential.rmax

NFWPotential.**rmax**()

NAME:

rmax

PURPOSE:

calculate the radius at which the rotation curve peaks

INPUT:

(none)

OUTPUT:

Radius at which the rotation curve peaks

HISTORY:

2020-02-05 - Written - Bovy (UofT)

galpy.potential.NFWPotential.rvir

NFWPotential.**rvir**(*H=70.0*, *Om=0.3*, *t=0.0*, *overdens=200.0*, *wrtcrit=False*, *ro=None*, *vo=None*, *use_physical=False*)

NAME:

rvir

PURPOSE:

calculate the virial radius for this density distribution

INPUT:

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

overdens= (200) overdensity which defines the virial radius

wrtcrit= (False) if True, the overdensity is wrt the critical density rather than the mean matter density

ro= distance scale in kpc or as Quantity (default: object-wide, which if not set is 8 kpc))

vo= velocity scale in km/s or as Quantity (default: object-wide, which if not set is 220 km/s))

OUTPUT:

virial radius

HISTORY:

2014-01-29 - Written - Bovy (IAS)

galpy.potential.NFWPotential.vmax

NFWPotential.**vmax**()

NAME:

vmax

PURPOSE:

calculate the maximum rotation curve velocity

INPUT:

(none)

OUTPUT:

Peak velocity in the rotation curve

HISTORY:

2020-02-05 - Written - Bovy (UofT)

General 3D potential routines

Use as `method(...)`

galpy.potential.dvcircdR

`galpy.potential.`**`dvcircdR`**(*Pot*, *R*, *phi=None*, *t=0.0*)

NAME:

dvcircdR

PURPOSE:

calculate the derivative of the circular velocity wrt R at R in potential Pot

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius (can be Quantity)

phi= (None) azimuth to use for non-axisymmetric potentials

t= time (optional; can be Quantity)

OUTPUT:

derivative of the circular rotation velocity wrt R

HISTORY:

2013-01-08 - Written - Bovy (IAS)

2016-06-28 - Added phi= keyword for non-axisymmetric potential - Bovy (UofT)

galpy.potential.epifreq

`galpy.potential.`**`epifreq`**(*Pot*, *R*, *t=0.0*)

NAME:

epifreq

PURPOSE:

calculate the epicycle frequency at R in the potential Pot

INPUT:

Pot - Potential instance or list thereof

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

epicycle frequency

HISTORY:

2012-07-25 - Written - Bovy (IAS)

galpy.potential.evaluateDensities

galpy.potential.**evaluateDensities**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *forcepoisson=False*)

NAME:

evaluateDensities

PURPOSE:

convenience function to evaluate a possible sum of densities

INPUT:

Pot - potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (can be Quantity)

t - time (can be Quantity)

forcepoisson= if True, calculate the density through the Poisson equation, even if an explicit expression for the density exists

OUTPUT:

rho(R,z)

HISTORY:

2010-08-08 - Written - Bovy (NYU)

2013-12-28 - Added forcepoisson - Bovy (IAS)

galpy.potential.evaluatephiforces

galpy.potential.**evaluatephiforces**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *v=None*)

NAME:

evaluatephiforces

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT: Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

v - current velocity in cylindrical coordinates (optional, but required when including dissipative forces; can be a Quantity)

OUTPUT:

F_phi(R,z,phi,t)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

2018-03-16 - Added velocity input for dissipative forces - Bovy (UofT)

galpy.potential.evaluatePotentials

Warning: `galpy` potentials do *not* necessarily approach zero at infinity. To compute, for example, the escape velocity or whether or not an orbit is unbound, you need to take into account the value of the potential at infinity. E.g., $v_{\rm esc}(r) = \sqrt{2[\Phi(\infty) - \Phi(r)]}$.

`galpy.potential.`**`evaluatePotentials`**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *dR=0*, *dphi=0*)

NAME:

evaluatePotentials

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (can be Quantity)

t - time (can be Quantity)

dR= dphi=, if set to non-zero integers, return the dR, dphi't derivative instead

OUTPUT:

Phi(R,z)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

galpy.potential.evaluatephizderivs

galpy.potential.**evaluatephizderivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluatephizderivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2Phi/dphi/dz(R,z,phi,t)

HISTORY:

2021-04-30 - Written - Bovy (UofT)

galpy.potential.evaluatephi2derivs

galpy.potential.**evaluatephi2derivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluatephi2derivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2Phi/d2phi(R,z,phi,t)

HISTORY:

2018-03-28 - Written - Bovy (UofT)

galpy.potential.evaluateRphiderivs

galpy.potential.**evaluateRphiderivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluateRphiderivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2Phi/dRdphi(R,z,phi,t)

HISTORY:

2014-06-30 - Written - Bovy (IAS)

galpy.potential.evaluateR2derivs

galpy.potential.**evaluateR2derivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluateR2derivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2Phi/d2R(R,z,phi,t)

HISTORY:

2012-07-25 - Written - Bovy (IAS)

galpy.potential.evaluater2derivs

galpy.potential.**evaluater2derivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluater2derivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2phi/dr2(R,z,phi,t)

HISTORY:

2018-03-28 - Written - Bovy (UofT)

galpy.potential.evaluateRzderivs

galpy.potential.**evaluateRzderivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluateRzderivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2Phi/dz/dR(R,z,phi,t)

HISTORY:

2013-08-28 - Written - Bovy (IAS)

galpy.potential.evaluateRforces

galpy.potential.**evaluateRforces**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *v=None*)

NAME:

evaluateRforce

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity))

t - time (optional; can be Quantity)

v - current velocity in cylindrical coordinates (optional, but required when including dissipative forces; can be a Quantity)

OUTPUT:

F_R(R,z,phi,t)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

2018-03-16 - Added velocity input for dissipative forces - Bovy (UofT)

galpy.potential.evaluaterforces

galpy.potential.**evaluaterforces**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *v=None*)

NAME:

evaluaterforces

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

v - current velocity in cylindrical coordinates (optional, but required when including dissipative forces; can be a Quantity)

OUTPUT:

F_r(R,z,phi,t)

HISTORY:

2016-06-10 - Written - Bovy (UofT)

galpy.potential.evaluateSurfaceDensities

galpy.potential.**evaluateSurfaceDensities**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *forcepoisson=False*)

NAME:

evaluateSurfaceDensities

PURPOSE:

convenience function to evaluate a possible sum of surface densities

INPUT:

Pot - potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (can be Quantity)

t - time (can be Quantity)

forcepoisson= if True, calculate the surface density through the Poisson equation, even if an explicit expression for the surface density exists

OUTPUT:

Sigma(R,z)

HISTORY:

2018-08-20 - Written - Bovy (UofT)

galpy.potential.evaluatez2derivs

galpy.potential.**evaluatez2derivs**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*)

NAME:

evaluatez2derivs

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

d2Phi/d2z(R,z,phi,t)

HISTORY:

2012-07-25 - Written - Bovy (IAS)

galpy.potential.evaluatezforces

galpy.potential.**evaluatezforces**(*Pot*, *R*, *z*, *phi=None*, *t=0.0*, *v=None*)

NAME:

evaluatezforces

PURPOSE:

convenience function to evaluate a possible sum of potentials

INPUT:

Pot - a potential or list of potentials

R - cylindrical Galactocentric distance (can be Quantity)

z - distance above the plane (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

v - current velocity in cylindrical coordinates (optional, but required when including dissipative forces; can be a Quantity)

OUTPUT:

F_z(R,z,phi,t)

HISTORY:

2010-04-16 - Written - Bovy (NYU)

2018-03-16 - Added velocity input for dissipative forces - Bovy (UofT)

galpy.potential.flatten

galpy.potential.**flatten**(*Pot*)

NAME:

flatten

PURPOSE:

flatten a possibly nested list of Potential instances into a flat list

INPUT:

Pot - list (possibly nested) of Potential instances

OUTPUT:

Flattened list of Potential instances

HISTORY:

2018-03-14 - Written - Bovy (UofT)

galpy.potential.flattening

galpy.potential.**flattening**(*Pot*, *R*, *z*, *t=0.0*)

NAME:

flattening

PURPOSE:

calculate the potential flattening, defined as sqrt(fabs(z/R F_R/F_z))

INPUT:

Pot - Potential instance or list thereof

R - Galactocentric radius (can be Quantity)

z - height (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

flattening

HISTORY:

2012-09-13 - Written - Bovy (IAS)

galpy.potential.LcE

galpy.potential.**LcE**(*Pot*, *E*, *t=0.0*)

NAME:

LcE

PURPOSE:

calculate the angular momentum of a circular orbit with energy E

INPUT:

Pot - Potential instance or list thereof

E - Energy (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

Lc(E)

HISTORY:

2022-04-06 - Written - Bovy (UofT)

galpy.potential.lindbladR

galpy.potential.**lindbladR**(*Pot*, *OmegaP*, *m=2*, *t=0.0*, ***kwargs*)

NAME:

lindbladR

PURPOSE:

calculate the radius of a Lindblad resonance

INPUT:

Pot - Potential instance or list of such instances

OmegaP - pattern speed (can be Quantity)

m= order of the resonance (as in m(O-Op)=kappa (negative m for outer) use m='corotation' for corotation

+scipy.optimize.brentq xtol,rtol,maxiter kwargs

t - time (optional; can be Quantity)

OUTPUT:

radius of Linblad resonance, None if there is no resonance

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.mass

galpy.potential.**mass**(*Pot*, *R*, *z=None*, *t=0.0*, *forceint=False*)

NAME:

mass

PURPOSE:

convenience function to evaluate a possible sum of masses

INPUT:

Pot - potential or list of potentials (dissipative forces in such a list are ignored)

R - cylindrical Galactocentric distance (can be Quantity)

z= (None) vertical height up to which to integrate (can be Quantity)

t - time (can be Quantity)

forceint= if True, calculate the mass through integration of the density, even if an explicit expression for the mass exists

OUTPUT:

Mass enclosed within the spherical shell with radius R if z is None else mass in the slab <R and between -z and z

HISTORY:

2021-02-07 - Written - Bovy (UofT)

2021-03-15 - Changed to integrate to spherical shell for z is None slab otherwise - Bovy (UofT)

galpy.potential.nemo_accname

galpy.potential.**nemo_accname**(*Pot*)

NAME:

nemo_accname

PURPOSE:

return the accname potential name for use of this potential or list of potentials with NEMO

INPUT:

Pot - Potential instance or list of such instances

OUTPUT:

Acceleration name in the correct format to give to accname=

HISTORY:

2014-12-18 - Written - Bovy (IAS)

galpy.potential.nemo_accpars

galpy.potential.**nemo_accpars**(*Pot*, *vo*, *ro*)

NAME:

nemo_accpars

PURPOSE:

return the accpars potential parameters for use of this potential or list of potentials with NEMO

INPUT:

Pot - Potential instance or list of such instances

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

accpars string in the corrct format to give to accpars

HISTORY:

2014-12-18 - Written - Bovy (IAS)

galpy.potential.omegac

galpy.potential.**omegac**(*Pot*, *R*, *t=0.0*)

NAME:

omegac

PURPOSE:

calculate the circular angular speed velocity at R in potential Pot

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

circular angular speed

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.plotDensities

galpy.potential.**plotDensities**(*Pot*, *rmin=0.0*, *rmax=1.5*, *nrs=21*, *zmin=-0.5*, *zmax=0.5*, *nzs=21*, *phi=None*, *xy=False*, *t=0.0*, *ncontours=21*, *savefilename=None*, *aspect=None*, *log=False*, *justcontours=False*, ***kwargs*)

NAME:

plotDensities

PURPOSE:

plot the density a set of potentials

INPUT:

Pot - Potential or list of Potential instances

rmin= minimum R (can be Quantity) [xmin if xy]

rmax= maximum R (can be Quantity) [ymax if xy]

nrs= grid in R

zmin= minimum z (can be Quantity) [ymin if xy]

zmax= maximum z (can be Quantity) [ymax if xy]

nzs= grid in z

phi= (None) azimuth to use for non-axisymmetric potentials

t= (0.) time to use to evaluate potential

xy= (False) if True, plot the density in X-Y

ncontours= number of contours

justcontours= (False) if True, just plot contours

savefilename= save to or restore from this savefile (pickle)

log= if True, plot the log density

OUTPUT:

plot to output device

HISTORY:

2013-07-05 - Written - Bovy (IAS)

galpy.potential.plotEscapecurve

galpy.potential.**plotEscapecurve**(*Pot*, **args*, ***kwargs*)

NAME:

plotEscapecurve

PURPOSE:

plot the escape velocity curve for this potential (in the z=0 plane for non-spherical potentials)

INPUT:

Pot - Potential or list of Potential instances

Rrange= Range in R to consider (can be Quantity)

grid= grid in R

savefilename= save to or restore from this savefile (pickle)

+galpy.util.plot.plot args and kwargs

OUTPUT:

plot to output device

HISTORY:

2010-08-08 - Written - Bovy (NYU)

galpy.potential.plotPotentials

galpy.potential.**plotPotentials**(*Pot*, *rmin=0.0*, *rmax=1.5*, *nrs=21*, *zmin=-0.5*, *zmax=0.5*, *nzs=21*, *phi=None*, *xy=False*, *t=0.0*, *effective=False*, *Lz=None*, *ncontours=21*, *savefilename=None*, *aspect=None*, *justcontours=False*, *levels=None*, *cntrcolors=None*)

NAME:

plotPotentials

PURPOSE:

plot a set of potentials

INPUT:

Pot - Potential or list of Potential instances

rmin= minimum R (can be Quantity) [xmin if xy]

rmax= maximum R (can be Quantity) [ymax if xy]

nrs= grid in R

zmin= minimum z (can be Quantity) [ymin if xy]

zmax= maximum z (can be Quantity) [ymax if xy]

nzs= grid in z

phi= (None) azimuth to use for non-axisymmetric potentials

t= (0.) time to use to evaluate potential

xy= (False) if True, plot the potential in X-Y

effective= (False) if True, plot the effective potential Phi + Lz^2/2/R^2

Lz= (None) angular momentum to use for the effective potential when effective=True

justcontours= (False) if True, just plot contours

levels= (None) contours to plot

ncontours - number of contours when levels is None

cntrcolors= (None) colors of the contours (single color or array with length ncontours)

savefilename= save to or restore from this savefile (pickle)

OUTPUT:

plot to output device

HISTORY:

2010-07-09 - Written - Bovy (NYU)

galpy.potential.plotRotcurve

galpy.potential.**plotRotcurve**(*Pot*, **args*, ***kwargs*)

NAME:

plotRotcurve

PURPOSE:

plot the rotation curve for this potential (in the z=0 plane for non-spherical potentials)

INPUT:

Pot - Potential or list of Potential instances

Rrange - Range in R to consider (needs to be in the units that you are plotting; can be Quantity)

grid= grid in R

phi= (None) azimuth to use for non-axisymmetric potentials

savefilename= save to or restore from this savefile (pickle)

+galpy.util.plot.plot args and kwargs

OUTPUT:

plot to output device

HISTORY:

2010-07-10 - Written - Bovy (NYU)

2016-06-15 - Added phi= keyword for non-axisymmetric potential - Bovy (UofT)

galpy.potential.plotSurfaceDensities

galpy.potential.**plotSurfaceDensities**(*Pot*, *xmin=-1.5*, *xmax=1.5*, *nxs=21*, *ymin=-1.5*, *ymax=1.5*, *nys=21*, *z=inf*, *t=0.0*, *ncontours=21*, *savefilename=None*, *aspect=None*, *log=False*, *justcontours=False*, ***kwargs*)

NAME:

plotSurfaceDensities

PURPOSE:

plot the surface density a set of potentials

INPUT:

Pot - Potential or list of Potential instances

xmin= minimum x (can be Quantity)

xmax= maximum x (can be Quantity)

nxs= grid in x

ymin= minimum y (can be Quantity)

ymax= maximum y (can be Quantity)

nys= grid in y

z= (inf) height between which to integrate the density (from -z to z; can be a Quantity)

t= (0.) time to use to evaluate potential

ncontours= number of contours

justcontours= (False) if True, just plot contours

savefilename= save to or restore from this savefile (pickle)

log= if True, plot the log density

OUTPUT:

plot to output device

HISTORY:

2020-08-19 - Written - Bovy (UofT)

galpy.potential.rE

galpy.potential.**rE**(*Pot*, *E*, *t=0.0*)

NAME:

rE

PURPOSE:

calculate the radius of a circular orbit with energy E

INPUT:

Pot - Potential instance or list thereof

E - Energy (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

radius

HISTORY:

2022-04-06 - Written - Bovy (UofT)

NOTE:

An efficient way to call this function on many objects is provided as the Orbit method rE

galpy.potential.rhalf

galpy.potential.**rhalf**(*Pot*, *t=0.0*, *INF=inf*)

NAME:

rhalf

PURPOSE:

calculate the half-mass radius, the radius of the spherical shell that contains half the total mass

INPUT:

Pot - Potential instance or list thereof

t= (0.) time (optional; can be Quantity)

INF= (numpy.inf) radius at which the total mass is calculated (internal units, just set this to something very large)

OUTPUT:

half-mass radius

HISTORY:

2021-03-18 - Written - Bovy (UofT)

galpy.potential.rl

galpy.potential.**rl**(*Pot*, *lz*, *t=0.0*)

NAME:

rl

PURPOSE:

calculate the radius of a circular orbit of Lz

INPUT:

Pot - Potential instance or list thereof

lz - Angular momentum (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

radius

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

NOTE:

An efficient way to call this function on many objects is provided as the Orbit method rguiding

galpy.potential.rtide

galpy.potential.**rtide**(*Pot*, *R*, *z*, *phi=0.0*, *t=0.0*, *M=None*)

NAME:

rtide

PURPOSE:

Calculate the tidal radius for object of mass M assuming a circular orbit as

$$r_t^3 = \frac{GM_s}{\Omega^2 - \mathrm{d}^2\Phi/\mathrm{d}r^2}$$

where M_s is the cluster mass, Ω is the circular frequency, and Φ is the gravitational potential. For non-spherical potentials, we evaluate $\Omega^2 = (1/r)(\mathrm{d}\Phi/\mathrm{d}r)$ and evaluate the derivatives at the given position of the cluster.

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius (can be Quantity)

z - height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

M - (default = None) Mass of object (can be Quantity)

OUTPUT:

Tidal Radius

HISTORY:

2018-03-21 - Written - Webb (UofT)

galpy.potential.tdyn

galpy.potential.**tdyn**(*Pot*, *R*, *t=0.0*)

NAME:

tdyn

PURPOSE:

calculate the dynamical time from tdyn^2 = 3pi/[G<rho>]

INPUT:

Pot - Potential instance or list thereof

R - Galactocentric radius (can be Quantity)

t= (0.) time (optional; can be Quantity)

OUTPUT:

Dynamical time

HISTORY:

2021-03-18 - Written - Bovy (UofT)

galpy.potential.to_amuse

galpy.potential.**to_amuse**(*Pot*, *t=0.0*, *tgalpy=0.0*, *reverse=False*, *ro=None*, *vo=None*)

NAME:

to_amuse

PURPOSE:

Return an AMUSE representation of a galpy Potential or list of Potentials

INPUT:

Pot - Potential instance or list of such instances

t= (0.) Initial time in AMUSE (can be in internal galpy units or AMUSE units)

tgalpy= (0.) Initial time in galpy (can be in internal galpy units or AMUSE units); because AMUSE initial times have to be positive, this is useful to set if the initial time in galpy is negative

reverse= (False) set whether the galpy potential evolves forwards or backwards in time (default: False); because AMUSE can only integrate forward in time, this is useful to integrate backward in time in AMUSE

ro= (default taken from Pot) length unit in kpc

vo= (default taken from Pot) velocity unit in km/s

OUTPUT:

AMUSE representation of Pot

HISTORY:

2019-08-04 - Written - Bovy (UofT)

2019-08-12 - Implemented actual function - Webb (UofT)

galpy.potential.ttensor

galpy.potential.**ttensor**(*Pot*, *R*, *z*, *phi=0.0*, *t=0.0*, *eigenval=False*)

NAME:

ttensor

PURPOSE:

Calculate the tidal tensor Tij=-d(Psi)(dxidxj)

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius (can be Quantity)

z - height (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

eigenval - return eigenvalues if true (optional; boolean)

OUTPUT:

Tidal Tensor

HISTORY:

2018-03-21 - Written - Webb (UofT)

galpy.potential.turn_physical_off

galpy.potential.**turn_physical_off**(*Pot*)

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

galpy.potential.turn_physical_on

galpy.potential.**turn_physical_on**(*Pot*, *ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

galpy.potential.vcirc

galpy.potential.**vcirc**(*Pot*, *R*, *phi=None*, *t=0.0*)

NAME:

vcirc

PURPOSE:

calculate the circular velocity at R in potential Pot

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius (can be Quantity)

phi= (None) azimuth to use for non-axisymmetric potentials

t= time (optional; can be Quantity)

OUTPUT:

circular rotation velocity

HISTORY:

2011-10-09 - Written - Bovy (IAS)

2016-06-15 - Added phi= keyword for non-axisymmetric potential - Bovy (UofT)

galpy.potential.verticalfreq

galpy.potential.**verticalfreq**(*Pot*, *R*, *t=0.0*)

NAME:

verticalfreq

PURPOSE:

calculate the vertical frequency at R in the potential Pot

INPUT:

Pot - Potential instance or list thereof

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

vertical frequency

HISTORY:

2012-07-25 - Written - Bovy (IAS@MPIA)

galpy.potential.vesc

galpy.potential.**vesc**(*Pot*, *R*, *t=0.0*)

NAME:

vesc

PURPOSE:

calculate the escape velocity at R for potential Pot

INPUT:

Pot - Potential instances or list thereof

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

escape velocity

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.vterm

galpy.potential.**vterm**(*Pot*, *l*, *t=0.0*, *deg=True*)

NAME:

vterm

PURPOSE:

calculate the terminal velocity at l in this potential

INPUT:

Pot - Potential instance

l - Galactic longitude [deg/rad; can be Quantity)

t - time (optional; can be Quantity)

deg= if True (default), l in deg

OUTPUT:

terminal velocity

HISTORY:

2013-05-31 - Written - Bovy (IAS)

galpy.potential.zvc

galpy.potential.**zvc**(*Pot*, *R*, *E*, *Lz*, *phi=0.0*, *t=0.0*)

NAME:

zvc

PURPOSE:

Calculate the zero-velocity curve: z such that Phi(R,z) + Lz/[2R^2] = E (assumes that F_z(R,z) = negative at positive z such that there is a single solution)

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius (can be Quantity)

E - Energy (can be Quantity)

Lz - Angular momentum (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

z such that Phi(R,z) + Lz/[2R^2] = E

HISTORY:

2020-08-20 - Written - Bovy (UofT)

galpy.potential.zvc_range

galpy.potential.**zvc_range**(*Pot*, *E*, *Lz*, *phi=0.0*, *t=0.0*)

NAME:

zvc_range

PURPOSE:

Calculate the minimum and maximum radius for which the zero-velocity curve exists for this energy and angular momentum (R such that Phi(R,0) + Lz/[2R^2] = E)

INPUT:

Pot - Potential instance or list of such instances

E - Energy (can be Quantity)

Lz - Angular momentum (can be Quantity)

phi - azimuth (optional; can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

Solutions R such that Phi(R,0) + Lz/[2R^2] = E

HISTORY:

2020-08-20 - Written - Bovy (UofT)

In addition to these, the following methods are available to compute expansion coefficients for the `SCFPotential` class for a given density

galpy.potential.scf_compute_coeffs

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos and Asin by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$ and $Asin = 0$.

For a given $\rho(R, z, \phi)$ we can compute $Acos$ and $Asin$ through the following equation

$$\begin{bmatrix} Acos \\ Asin \end{bmatrix}_{nlm} = \frac{4a^3}{I_{nl}} \int_{\xi=0}^{\infty} \int_{\cos(\theta)=-1}^{1} \int_{\phi=0}^{2\pi} (1+\xi)^2 (1-\xi)^{-4} \rho(R,z,\phi) \Phi_{nlm}(\xi, \cos(\theta), \phi) d\phi d\cos(\theta) d\xi$$

Where

$$\Phi_{nlm}(\xi, \cos(\theta), \phi) = -\frac{\sqrt{2l+1}}{a 2^{2l+1}} \sqrt{\frac{(l-m)!}{(l+m)!}} (1+\xi)^l (1-\xi)^{l+1} C_n^{2l+3/2}(\xi) P_{lm}(\cos(\theta)) \begin{bmatrix} \cos(m\phi) \\ \sin(m\phi) \end{bmatrix}$$

$$I_{nl} = -K_{nl} \frac{4\pi}{a 2^{8l+6}} \frac{\Gamma(n+4l+3)}{n!(n+2l+3/2)[\Gamma(2l+3/2)]^2} \qquad K_{nl} = \frac{1}{2} n(n+4l+3) + (l+1)(2l+1)$$

P_{lm} is the Associated Legendre Polynomials whereas C_n^{α} is the Gegenbauer polynomial.

Also note $\xi = \frac{r-a}{r+a}$, and n, l and m are integers bounded by $0 <= n < N$, $0 <= l < L$, and $0 <= m <= l$

galpy.potential.**scf_compute_coeffs**(*dens*, *N*, *L*, *a=1.0*, *radial_order=None*, *costheta_order=None*, *phi_order=None*)

NAME:

scf_compute_coeffs

PURPOSE:

Numerically compute the expansion coefficients for a given triaxial density

INPUT:

dens - A density function that takes a parameter R, z and phi

N - size of the Nth dimension of the expansion coefficients

L - size of the Lth and Mth dimension of the expansion coefficients

a - parameter used to shift the basis functions

radial_order - Number of sample points of the radial integral. If None, radial_order=max(20, N + 3/2L + 1)

costheta_order - Number of sample points of the costheta integral. If None, If costheta_order=max(20, L + 1)

phi_order - Number of sample points of the phi integral. If None, If costheta_order=max(20, L + 1)

OUTPUT:

(Acos,Asin) - Expansion coefficients for density dens that can be given to SCFPotential.__init__

HISTORY:

2016-05-27 - Written - Aladdin Seaifan (UofT)

galpy.potential.scf_compute_coeffs_axi

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$.

Further note that this function is a specification of *scf_compute_coeffs* where $Acos_{nlm} = 0$ at $m \neq 0$ and $Asin_{nlm} = None$

For a given $\rho(R, z)$ we can compute $Acos$ and $Asin$ through the following equation

$$Acos_{nlm} = \frac{8\pi a^3}{I_{nl}} \int_{\xi=0}^{\infty} \int_{\cos(\theta)=-1}^{1} (1+\xi)^2 (1-\xi)^{-4} \rho(R,z) \Phi_{nlm}(\xi, \cos(\theta)) d\cos(\theta) d\xi \qquad Asin_{nlm} = None$$

Where

$$\Phi_{nlm}(\xi, \cos(\theta)) = -\frac{\sqrt{2l+1}}{a2^{2l+1}} (1+\xi)^l (1-\xi)^{l+1} C_n^{2l+3/2}(\xi) P_{l0}(\cos(\theta)) \delta_{m0}$$

$$I_{nl} = -K_{nl} \frac{4\pi}{a2^{8l+6}} \frac{\Gamma(n+4l+3)}{n!(n+2l+3/2)[\Gamma(2l+3/2)]^2} \qquad K_{nl} = \frac{1}{2} n(n+4l+3) + (l+1)(2l+1)$$

P_{lm} is the Associated Legendre Polynomials whereas C_n^{α} is the Gegenbauer polynomial.

Also note $\xi = \frac{r-a}{r+a}$, and n, l and m are integers bounded by $0 <= n < N$, $0 <= l < L$, and $m = 0$

galpy.potential.**scf_compute_coeffs_axi**(*dens*, *N*, *L*, *a=1.0*, *radial_order=None*, *costheta_order=None*)

NAME:

scf_compute_coeffs_axi

PURPOSE:

Numerically compute the expansion coefficients for a given axi-symmetric density

INPUT:

dens - A density function that takes a parameter R and z

N - size of the Nth dimension of the expansion coefficients

L - size of the Lth dimension of the expansion coefficients

a - parameter used to shift the basis functions

radial_order - Number of sample points of the radial integral. If None, radial_order=max(20, N + 3/2L + 1)

costheta_order - Number of sample points of the costheta integral. If None, If costheta_order=max(20, L + 1)

OUTPUT:

(Acos,Asin) - Expansion coefficients for density dens that can be given to SCFPotential.__init__

HISTORY:

2016-05-20 - Written - Aladdin Seaifan (UofT)

galpy.potential.scf_compute_coeffs_axi_nbody

This function is the equivalent to *galpy.potential.scf_compute_coeffs_axi* but computing the coefficients based on an N-body representation of the density.

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$.

Further note that this function is a specification of *galpy.potential.scf_compute_coeffs_nbody* where $Acos_{nlm} = 0$ at $m \neq 0$ and $Asin_{nlm} = None$.

galpy.potential.**scf_compute_coeffs_axi_nbody**(*pos*, *N*, *L*, *mass=1.0*, *a=1.0*)

NAME:

scf_compute_coeffs_axi_nbody

PURPOSE:

Numerically compute the expansion coefficients for a given N-body set of points assuming that the density is axisymmetric

INPUT:

pos - positions of particles in rectangular coordinates with shape [3,n]

N - size of the Nth dimension of the expansion coefficients

L - size of the Lth dimension of the expansion coefficients

mass= (1.) mass of particles (scalar or array with size n)

a= (1.) parameter used to scale the radius

OUTPUT:

(Acos,Asin) - Expansion coefficients for density dens that can be given to SCFPotential.__init__

HISTORY:

2021-02-22 - Written based on general code - Bovy (UofT)

galpy.potential.scf_compute_coeffs_nbody

This function is the equivalent to *galpy.potential.scf_compute_coeffs* but computing the coefficients based on an N-body representation of the density.

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos and Asin by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$ and $Asin = 0$.

`galpy.potential.`**`scf_compute_coeffs_nbody`**(*pos*, *N*, *L*, *mass=1.0*, *a=1.0*)

NAME:

scf_compute_coeffs_nbody

PURPOSE:

Numerically compute the expansion coefficients for a given N-body set of points

INPUT:

pos - positions of particles in rectangular coordinates with shape [3,n]

N - size of the Nth dimension of the expansion coefficients

L - size of the Lth and Mth dimension of the expansion coefficients

mass= (1.) mass of particles (scalar or array with size n)

a= (1.) parameter used to scale the radius

OUTPUT:

(Acos,Asin) - Expansion coefficients for density dens that can be given to SCFPotential.__init__

HISTORY:

2020-11-18 - Written - Morgan Bennett (UofT)

galpy.potential.scf_compute_coeffs_spherical

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$.

Futher note that this function is a specification of *scf_compute_coeffs_axi* where $Acos_{nlm} = 0$ at $l \neq 0$

For a given $\rho(r)$ we can compute $Acos$ and $Asin$ through the following equation

$$Acos_{nlm} = \frac{16\pi a^3}{I_{nl}} \int_{\xi=0}^{\infty} (1+\xi)^2 (1-\xi)^{-4} \rho(r) \Phi_{nlm}(\xi) d\xi \qquad Asin_{nlm} = None$$

Where

$$\Phi_{nlm}(\xi, \cos(\theta)) = -\frac{1}{2a}(1-\xi) C_n^{3/2}(\xi) \delta_{l0}\delta_{m0}$$

$$I_{n0} = -K_{n0}\frac{1}{4a}\frac{(n+2)(n+1)}{(n+3/2)} \qquad K_{nl} = \frac{1}{2}n(n+3)+1$$

C_n^{α} is the Gegenbauer polynomial.

Also note $\xi = \frac{r-a}{r+a}$, and n, l and m are integers bounded by $0 <= n < N$, $l = m = 0$

galpy.potential.**scf_compute_coeffs_spherical**(*dens*, *N*, *a=1.0*, *radial_order=None*)

NAME:

scf_compute_coeffs_spherical

PURPOSE:

Numerically compute the expansion coefficients for a given spherical density

INPUT:

dens - A density function that takes a parameter R

N - size of expansion coefficients

a= (1.) parameter used to scale the radius

radial_order - Number of sample points of the radial integral. If None, radial_order=max(20, N + 1)

OUTPUT:

(Acos,Asin) - Expansion coefficients for density dens that can be given to SCFPotential.__init__

HISTORY:

2016-05-18 - Written - Aladdin Seaifan (UofT)

galpy.potential.scf_compute_coeffs_spherical_nbody

This function is the equivalent to *galpy.potential.scf_compute_coeffs_spherical* but computing the coefficients based on an N-body representation of the density.

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$.

Futher note that this function is a specification of *galpy.potential.scf_compute_coeffs_nbody* where $Acos_{nlm} = 0$ at $l \neq 0$

galpy.potential.**scf_compute_coeffs_spherical_nbody**(*pos*, *N*, *mass=1.0*, *a=1.0*)

NAME:

scf_compute_coeffs_spherical_nbody

PURPOSE:

Numerically compute the expansion coefficients for a spherical expansion for a given N-body set of points

INPUT:

pos - positions of particles in rectangular coordinates with shape [3,n]

N - size of the Nth dimension of the expansion coefficients

mass= (1.) mass of particles (scalar or array with size n)

a= (1.) parameter used to scale the radius

OUTPUT:

(Acos,Asin) - Expansion coefficients for density dens that can be given to SCFPotential.__init__

HISTORY:

2020-11-18 - Written - Morgan Bennett (UofT)

2021-02-22 - Sped-up - Bovy (UofT)

Specific potentials

All of the following potentials can also be modified by the specific `WrapperPotentials` listed *below*.

Spherical potentials

Spherical potentials in `galpy` can be implemented in two ways: a) directly by inheriting from `Potential` and implementing the usual methods (`_evaluate`, `_Rforce`, etc.) or b) by inheriting from the general SphericalPotential class and implementing the functions `_revaluate(self,r,t=0.)`, `_rforce(self,r,t=0.)`, `_r2deriv(self,r,t=0.)`, and `_rdens(self,r,t=0.)` that evaluate the potential, radial force, (minus the) radial force derivative, and density as a function of the (here natural) spherical radius. For adding a C implementation when using method b), follow similar steps in C (use `interpSphericalPotential` as an example to follow). For historical reasons, most spherical potentials in `galpy` are directly implemented (option a above), but for new spherical potentials it is typically easier to follow option b).

Additional spherical potentials can be obtained by setting the axis ratios equal for the triaxial potentials listed in the section on ellipsoidal triaxial potentials below.

Arbitrary spherical density potential

class galpy.potential.**AnySphericalPotential**(*dens=<function AnySphericalPotential.<lambda>>*, *amp=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the potential of an arbitrary spherical density distribution $\rho(r)$

__init__(*dens=<function AnySphericalPotential.<lambda>>*, *amp=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize the potential of an arbitrary spherical density distribution

INPUT:

dens= (0.64/r/(1+r)**3) function of a single variable that gives the density as a function of radius (can return a Quantity)

amp= (1.) amplitude to be applied to the potential

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2021-01-05 - Written - Bovy (UofT)

Burkert potential

class galpy.potential.**BurkertPotential**(*amp=1.0*, *a=2.0*, *normalize=False*, *ro=None*, *vo=None*)

BurkertPotential.py: Potential with a Burkert density

$$\rho(r) = \frac{\mathrm{amp}}{(1+r/a)\,(1+[r/a]^2)}$$

__init__(*amp=1.0*, *a=2.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Burkert-density potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass density or Gxmass density

a = scale radius (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2013-04-10 - Written - Bovy (IAS)

2020-03-30 - Re-implemented using SphericalPotential - Bovy (UofT)

Double power-law density spherical potential

class galpy.potential.**TwoPowerSphericalPotential**(*amp=1.0*, *a=5.0*, *alpha=1.5*, *beta=3.5*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements spherical potentials that are derived from two-power density models

$$\rho(r) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(r/a)^\alpha\,(1+r/a)^{\beta-\alpha}}$$

__init__(*amp=1.0*, *a=5.0*, *alpha=1.5*, *beta=3.5*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a two-power-density potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

alpha - inner power

beta - outer power

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Started - Bovy (NYU)

Spherical Cored Dehnen potential

class galpy.potential.**DehnenCoreSphericalPotential**(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Dehnen Spherical Potential from Dehnen (1993) with alpha=0 (corresponding to an inner core)

$$\rho(r) = \frac{\mathrm{amp}}{12\,\pi\,a^3}\,\frac{1}{(1+r/a)^4}$$

__init__(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a cored Dehnen Spherical Potential; note that the amplitude definition used here does NOT match that of Dehnen (1993)

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

alpha - inner power, restricted to [0, 3)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2019-10-07 - Started - Starkman (UofT)

Spherical Dehnen potential

class galpy.potential.**DehnenSphericalPotential**(*amp=1.0*, *a=1.0*, *alpha=1.5*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Dehnen Spherical Potential from Dehnen (1993)

$$\rho(r) = \frac{\mathrm{amp}(3-\alpha)}{4\,\pi\,a^3}\,\frac{1}{(r/a)^\alpha\,(1+r/a)^{4-\alpha}}$$

__init__(*amp=1.0*, *a=1.0*, *alpha=1.5*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Dehnen Spherical Potential; note that the amplitude definitio used here does NOT match that of Dehnen (1993)

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

alpha - inner power, restricted to [0, 3)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2019-10-07 - Started - Starkman (UofT)

Hernquist potential

class galpy.potential.**HernquistPotential**(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Hernquist potential

$$\rho(r) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(r/a)\,(1+r/a)^3}$$

__init__(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a Hernquist potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass (note that amp is 2 x [total mass] for the chosen definition of the Hernquist potential)

a - scale radius (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Written - Bovy (NYU)

Homogeneous sphere potential

class galpy.potential.**HomogeneousSpherePotential**(*amp=1.0*, *R=1.1*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the homogeneous sphere potential for $\rho(r) = \rho_0 = \text{constant}$ for all $r < R$ and zero otherwise. The potential is given by

$$\Phi(r) = \text{amp} \times \begin{cases} (r^2 - 3R^2), & \text{for } r < R \\ -\frac{2R^3}{r} & \text{for } r \geq R \end{cases}$$

We have that $\rho_0 = 3\,\text{amp}/[2\pi G]$.

__init__(*amp=1.0*, *R=1.1*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a homogeneous sphere potential

INPUT:

amp= amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass density or Gxmass density

R= size of the sphere (can be Quantity)

normalize= if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2019-12-20 - Written - Bovy (UofT)

Interpolated spherical potential

The `interpSphericalPotential` class provides a general interface to generate interpolated instances of spherical potentials or lists of such potentials. This interpolated potential can be used in any function where other three-dimensional galpy potentials can be used. This includes functions that use `C` to speed up calculations.

The `interpSphericalPotential` interpolates the radial force of a spherical potential and determines the potential and its second derivative from the base radial-force interpolation object. To set up an `interpSphericalPotential` instance, either provide it with a function that returns the radial force and the grid to interpolate it on, as for example,

```
>>> from galpy import potential
>>> ip= potential.interpSphericalPotential(rforce=lambda r: -1./r,
                          rgrid=numpy.geomspace(0.01,20,101),Phi0=0.)
```

which sets up an `interpSphericalPotential` instance that has the same radial force as the spherical `LogarithmicHaloPotential`. If you have a function that gives the enclosed mass within a given radius, simply pass it divided by $-r^2$ to set up a `interpSphericalPotential` instance for this enclosed-mass profile.

Alternatively, you can specify a `galpy` potential or list of potentials and (again) the radial interpolation grid, as for example,

```
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> ip= potential.interpSphericalPotential(rforce=lp,
                       rgrid=numpy.geomspace(0.01,20,101))
```

Note that, because the potential is defined through integration of the (negative) radial force, we need to specify the potential at the smallest grid point, which is done through the `Phi0=` keyword in the first example. When using a `galpy` potential (or list), this value is automatically determined.

Also note that the density of the potential is assumed to be zero outside of the final radial grid point. That is, the potential outside of the final grid point is $-GM/r$ where M is the mass within the final grid point. If during an orbit integration, the orbit strays outside of the interpolation grid, a warning is issued.

Warning: The density of a `interpSphericalPotential` instance is assumed to be zero outside of the largest radial grid point.

class galpy.potential.**interpSphericalPotential**(*self*, *rforce=None*, *rgrid=numpy.geomspace(0.01, 20, 101)*, *Phi0=None*, *ro=None*, *vo=None*)

Class that interpolates a spherical potential on a grid

__init__(*self*, *rforce=None*, *rgrid=numpy.geomspace(0.01, 20, 101)*, *Phi0=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize an interpolated, spherical potential

INPUT:

rforce= (None) Either a) function that gives the radial force as a function of r or b) a galpy Potential instance or list thereof

rgrid= (numpy.geomspace(0.01,20,101)) radial grid on which to evaluate the potential for interpolation (note that beyond rgrid[-1], the potential is extrapolated as -GM(<rgrid[-1])/r)

Phi0= (0.) value of the potential at rgrid[0] (only necessary when rforce is a function, for galpy potentials automatically determined)

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2020-07-13 - Written - Bovy (UofT)

Isochrone potential

class galpy.potential.**IsochronePotential**(*amp=1.0*, *b=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Isochrone potential

$$\Phi(r) = -\frac{\mathrm{amp}}{b + \sqrt{b^2 + r^2}}$$

with $\mathrm{amp} = GM$ the total mass.

__init__(*amp=1.0*, *b=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize an isochrone potential

INPUT:

amp= amplitude to be applied to the potential, the total mass (default: 1); can be a Quantity with units of mass or Gxmass

b= scale radius of the isochrone potential (can be Quantity)

normalize= if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2013-09-08 - Written - Bovy (IAS)

Jaffe potential

class galpy.potential.**JaffePotential**(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Jaffe potential

$$\rho(r) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(r/a)^2\,(1+r/a)^2}$$

__init__(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a Jaffe potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Written - Bovy (NYU)

Kepler potential

class galpy.potential.**KeplerPotential**(*amp=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Kepler (point mass) potential

$$\Phi(r) = -\frac{\mathrm{amp}}{r}$$

with $\mathrm{amp} = GM$ the mass.

__init__(*amp=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Kepler, point-mass potential

INPUT:

amp - amplitude to be applied to the potential, the mass of the point mass (default: 1); can be a Quantity with units of mass density or Gxmass density

alpha - inner power

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-10 - Written - Bovy (NYU)

NFW potential

class galpy.potential.**NFWPotential**(*amp=1.0*, *a=1.0*, *normalize=False*, *rmax=None*, *vmax=None*, *conc=None*, *mvir=None*, *vo=None*, *ro=None*, *H=70.0*, *Om=0.3*, *overdens=200.0*, *wrtcrit=False*)

Class that implements the NFW potential

$$\rho(r) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(r/a)\,(1+r/a)^2}$$

__init__(*amp=1.0*, *a=1.0*, *normalize=False*, *rmax=None*, *vmax=None*, *conc=None*, *mvir=None*, *vo=None*, *ro=None*, *H=70.0*, *Om=0.3*, *overdens=200.0*, *wrtcrit=False*)

NAME:

__init__

PURPOSE:

Initialize a NFW potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

Alternatively, NFW potentials can be initialized in the following two manners:

a) rmax= radius where the rotation curve peaks (can be a Quantity, otherwise assumed to be in internal units)

   vmax= maximum circular velocity (can be a Quantity, otherwise assumed to be in internal units)

b) conc= concentration

   mvir= virial mass in 10^12 Msolar

in which case you also need to supply the following keywords

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

overdens= (200) overdensity which defines the virial radius

wrtcrit= (False) if True, the overdensity is wrt the critical density rather than the mean matter density

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Written - Bovy (NYU)

2014-04-03 - Initialization w/ concentration and mass - Bovy (IAS)

2020-04-29 - Initialization w/ rmax and vmax - Bovy (UofT)

Plummer potential

class galpy.potential.**PlummerPotential**(*amp=1.0*, *b=0.8*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Plummer potential

$$\Phi(R,z) = -\frac{\mathrm{amp}}{\sqrt{R^2+z^2+b^2}}$$

with $\mathrm{amp} = GM$ the total mass.

__init__(*amp=1.0*, *b=0.8*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Plummer potential

INPUT:

amp - amplitude to be applied to the potential, the total mass (default: 1); can be a Quantity with units of mass or Gxmass

b - scale parameter (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2015-06-15 - Written - Bovy (IAS)

Power-law density spherical potential

class galpy.potential.**PowerSphericalPotential**(*amp=1.0*, *alpha=1.0*, *normalize=False*, *r1=1.0*, *ro=None*, *vo=None*)

Class that implements spherical potentials that are derived from power-law density models

$$\rho(r) = \frac{\mathrm{amp}}{r_1^3}\,\left(\frac{r_1}{r}\right)^\alpha$$

__init__(*amp=1.0*, *alpha=1.0*, *normalize=False*, *r1=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a power-law-density potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

alpha - power-law exponent

r1= (1.) reference radius for amplitude (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-10 - Written - Bovy (NYU)

Power-law density spherical potential with an exponential cut-off

class galpy.potential.**PowerSphericalPotentialwCutoff**(*amp=1.0*, *alpha=1.0*, *rc=1.0*, *normalize=False*, *r1=1.0*, *ro=None*, *vo=None*)

Class that implements spherical potentials that are derived from power-law density models

$$\rho(r) = \mathrm{amp}\,\left(\frac{r_1}{r}\right)^{\alpha}\,\exp\left(-(r/rc)^2\right)$$

__init__(*amp=1.0*, *alpha=1.0*, *rc=1.0*, *normalize=False*, *r1=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a power-law-density potential

INPUT:

amp= amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass density or Gxmass density

alpha= inner power

rc= cut-off radius (can be Quantity)

r1= (1.) reference radius for amplitude (can be Quantity)

normalize= if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2013-06-28 - Written - Bovy (IAS)

Pseudo-isothermal potential

class galpy.potential.**PseudoIsothermalPotential**(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the pseudo-isothermal potential

$$\rho(r) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{1+(r/a)^2}$$

__init__(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a pseudo-isothermal potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - core radius (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2015-12-04 - Started - Bovy (UofT)

Spherical Shell Potential

class galpy.potential.**SphericalShellPotential**(*amp=1.0*, *a=0.75*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the potential of an infinitesimally-thin, spherical shell

$$\rho(r) = \frac{\mathrm{amp}}{4\pi\,a^2}\,\delta(r-a)$$

with $\mathrm{amp} = GM$ the mass of the shell.

__init__(*amp=1.0*, *a=0.75*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a spherical shell potential

INPUT:

amp - mass of the shell (default: 1); can be a Quantity with units of mass or Gxmass

a= (0.75) radius of the shell (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.; note that because the force is always zero at r < a, this does not work if a > 1

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2018-08-04 - Written - Bovy (UofT)

2020-03-30 - Re-implemented using SphericalPotential - Bovy (UofT)

Axisymmetric potentials

Additional axisymmetric potentials can be obtained by setting the x/y axis ratio equal to 1 for the triaxial potentials listed in the section on ellipsoidal triaxial potentials below.

Arbitrary razor-thin, axisymmetric potential

class galpy.potential.**AnyAxisymmetricRazorThinDiskPotential**(*surfdens=<function AnyAxisymmetricRazorThinDiskPotential.<lambda>>*, *amp=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the potential of an arbitrary axisymmetric, razor-thin disk with surface density $\Sigma(R)$

__init__(*surfdens=<function AnyAxisymmetricRazorThinDiskPotential.<lambda>>*, *amp=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize the potential of an arbitrary axisymmetric disk

INPUT:

surfdens= (1.5 e^[-R/0.3]) function of a single variable that gives the surface density as a function of radius (can return a Quantity)

amp= (1.) amplitude to be applied to the potential

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

AnyAxisymmetricRazorThinDiskPotential object

HISTORY:

2021-01-04 - Written - Bovy (UofT)

Double exponential disk potential

class galpy.potential.**DoubleExponentialDiskPotential**(*amp=1.0*, *hr=0.3333333333333333*, *hz=0.0625*, *normalize=False*, *ro=None*, *vo=None*, *de_h=0.001*, *de_n=10000*)

Class that implements the double exponential disk potential

$$\rho(R,z) = \mathrm{amp}\,\exp\left(-R/h_R - |z|/h_z\right)$$

__init__(*amp=1.0*, *hr=0.3333333333333333*, *hz=0.0625*, *normalize=False*, *ro=None*, *vo=None*, *de_h=0.001*, *de_n=10000*)

NAME:

__init__

PURPOSE:

initialize a double-exponential disk potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass density or Gxmass density

hr - disk scale-length (can be Quantity)

hz - scale-height (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

de_h= (1e-3) step used in numerical integration (use 1000 for a lower accuracy version that is typically still high accuracy enough, but faster)

de_b= (10000) number of points used in numerical integration

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

DoubleExponentialDiskPotential object

HISTORY:

2010-04-16 - Written - Bovy (NYU)

2013-01-01 - Re-implemented using faster integration techniques - Bovy (IAS)

2020-12-24 - Re-implemented again using more accurate integration techniques for Bessel integrals - Bovy (UofT)

Flattened Power-law potential

Flattening is in the potential as in Evans (1994) rather than in the density

class galpy.potential.**FlattenedPowerPotential**(*amp=1.0*, *alpha=0.5*, *q=0.9*, *core=1e-08*, *normalize=False*, *r1=1.0*, *ro=None*, *vo=None*)

Class that implements a power-law potential that is flattened in the potential (NOT the density)

$$\Phi(R,z) = -\frac{\mathrm{amp}\, r_1^\alpha}{\alpha\ (R^2 + (z/q)^2 + \mathrm{core}^2)^{\alpha/2}}$$

and the same as LogarithmicHaloPotential for $\alpha = 0$

See Figure 1 in Evans (1994) for combinations of alpha and q that correspond to positive densities

__init__(*amp=1.0*, *alpha=0.5*, *q=0.9*, *core=1e-08*, *normalize=False*, *r1=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a flattened power-law potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of velocity-squared

alpha - power

q - flattening

core - core radius (can be Quantity)

r1= (1.) reference radius for amplitude (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2013-01-09 - Written - Bovy (IAS)

Interpolated axisymmetric potential

The `interpRZPotential` class provides a general interface to generate interpolated instances of general three-dimensional, axisymmetric potentials or lists of such potentials. This interpolated potential can be used in any function where other three-dimensional galpy potentials can be used. This includes functions that use `C` to speed up calculations, if the `interpRZPotential` instance was set up with `enable_c=True`. Initialize as

```
>>> from galpy import potential
>>> ip= potential.interpRZPotential(potential.MWPotential,interpPot=True)
```

which sets up an interpolation of the potential itself only. The potential and all different forces and functions (`dens`, `vcirc`, `epifreq`, `verticalfreq`, `dvcircdR`) are interpolated separately and one needs to specify that these need to be interpolated separately (so, for example, one needs to set `interpRforce=True` to interpolate the radial force, or `interpvcirc=True` to interpolate the circular velocity).

When points outside the grid are requested within the python code, the instance will fall back on the original (non-interpolated) potential. However, when the potential is used purely in `C`, like during orbit integration in `C` or during action–angle evaluations in `C`, there is no way for the potential to fall back onto the original potential and nonsense or NaNs will be returned. Therefore, when using `interpRZPotential` in `C`, one must make sure that the whole relevant part of the `(R,z)` plane is covered. One more time:

Warning: When an interpolated potential is used purely in `C`, like during orbit integration in `C` or during action–angle evaluations in `C`, there is no way for the potential to fall back onto the original potential and nonsense or NaNs will be returned. Therefore, when using `interpRZPotential` in `C`, one must make sure that the whole relevant part of the `(R,z)` plane is covered.

class galpy.potential.**interpRZPotential**(*RZPot=None*, *rgrid=(-4.605170185988091, 2.995732273553991, 101)*, *zgrid=(0.0, 1.0, 101)*, *logR=True*, *interpPot=False*, *interpRforce=False*, *interpzforce=False*, *interpDens=False*, *interpvcirc=False*, *interpdvcircdr=False*, *interpepifreq=False*, *interpverticalfreq=False*, *ro=None*, *vo=None*, *use_c=False*, *enable_c=False*, *zsym=True*, *numcores=None*)

Class that interpolates a given potential on a grid for fast orbit integration

__init__(*RZPot=None*, *rgrid=(-4.605170185988091, 2.995732273553991, 101)*, *zgrid=(0.0, 1.0, 101)*, *logR=True*, *interpPot=False*, *interpRforce=False*, *interpzforce=False*, *interpDens=False*, *interpvcirc=False*, *interpdvcircdr=False*, *interpepifreq=False*, *interpverticalfreq=False*, *ro=None*, *vo=None*, *use_c=False*, *enable_c=False*, *zsym=True*, *numcores=None*)

NAME:

__init__

PURPOSE:

Initialize an interpRZPotential instance

INPUT:

RZPot - RZPotential to be interpolated

rgrid - R grid to be given to linspace as in rs= linspace(*rgrid)

zgrid - z grid to be given to linspace as in zs= linspace(*zgrid)

logR - if True, rgrid is in the log of R so logrs= linspace(*rgrid)

interpPot, interpRforce, interpzforce, interpDens,interpvcirc, interpepifreq, interpverticalfreq, interpdvcircdr= if True, interpolate these functions

use_c= use C to speed up the calculation of the grid

enable_c= enable use of C for interpolations

zsym= if True (default), the potential is assumed to be symmetric around z=0 (so you can use, e.g., zgrid=(0.,1.,101)).

numcores= if set to an integer, use this many cores (only used for vcirc, dvcircdR, epifreq, and verticalfreq; NOT NECESSARILY FASTER, TIME TO MAKE SURE)

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

instance

HISTORY:

2010-07-21 - Written - Bovy (NYU)

2013-01-24 - Started with new implementation - Bovy (IAS)

Interpolated SnapshotRZ potential

This class is built on the `interpRZPotential` class; see the documentation of that class *here* for additional information on how to setup objects of the `InterpSnapshotRZPotential` class.

class galpy.potential.**InterpSnapshotRZPotential**(*s*, *ro=None*, *vo=None*, *rgrid=(-4.605170185988091, 2.995732273553991, 101)*, *zgrid=(0.0, 1.0, 101)*, *interpepifreq=False*, *interpverticalfreq=False*, *interpPot=True*, *enable_c=True*, *logR=True*, *zsym=True*, *numcores=None*, *nazimuths=4*, *use_pkdgrav=False*)

Interpolated axisymmetrized potential extracted from a simulation output (see `interpRZPotential` and `SnapshotRZPotential`)

__init__(*s*, *ro=None*, *vo=None*, *rgrid=(-4.605170185988091, 2.995732273553991, 101)*, *zgrid=(0.0, 1.0, 101)*, *interpepifreq=False*, *interpverticalfreq=False*, *interpPot=True*, *enable_c=True*, *logR=True*, *zsym=True*, *numcores=None*, *nazimuths=4*, *use_pkdgrav=False*)

NAME:

__init__

PURPOSE:

Initialize an InterpSnapshotRZPotential instance

INPUT:

s - a simulation snapshot loaded with pynbody

rgrid - R grid to be given to linspace as in rs= linspace(*rgrid)

zgrid - z grid to be given to linspace as in zs= linspace(*zgrid)

logR - if True, rgrid is in the log of R so logrs= linspace(*rgrid)

interpPot, interpepifreq, interpverticalfreq= if True, interpolate these functions (interpPot=True also interpolates the R and zforce)

enable_c= enable use of C for interpolations

zsym= if True (default), the potential is assumed to be symmetric around z=0 (so you can use, e.g., zgrid=(0.,1.,101)).

numcores= if set to an integer, use this many cores

nazimuths= (4) number of azimuths to average over

use_pkdgrav= (False) use PKDGRAV to calculate the snapshot's potential and forces (CURRENTLY NOT IMPLEMENTED)

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

instance

HISTORY:

2013 - Written - Rok Roskar (ETH)

2014-11-24 - Edited for merging into main galpy - Bovy (IAS)

Kuzmin disk potential

class galpy.potential.**KuzminDiskPotential**(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Kuzmin Disk potential

$$\Phi(R,z) = -\frac{\mathrm{amp}}{\sqrt{R^2 + (a+|z|)^2}}$$

with $\mathrm{amp} = GM$ the total mass.

__init__(*amp=1.0*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Kuzmin disk Potential

INPUT:

amp - amplitude to be applied to the potential, the total mass (default: 1); can be a Quantity with units of mass density or Gxmass density

a - scale length (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

KuzminDiskPotential object

HISTORY:

2016-05-09 - Written - Aladdin

Kuzmin-Kutuzov Staeckel potential

class galpy.potential.**KuzminKutuzovStaeckelPotential**(*amp=1.0*, *ac=5.0*, *Delta=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Kuzmin-Kutuzov Staeckel potential

$$\Phi(R,z) = -\frac{\mathrm{amp}}{\sqrt{\lambda}+\sqrt{\nu}}$$

(see, e.g., Batsleer & Dejonghe 1994)

__init__(*amp=1.0*, *ac=5.0*, *Delta=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Kuzmin-Kutuzov Staeckel potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass density or Gxmass density

ac - axis ratio of the coordinate surfaces; (a/c) = sqrt(-alpha) / sqrt(-gamma) (default: 5.)

Delta - focal distance that defines the spheroidal coordinate system (default: 1.); Delta=sqrt(gamma-alpha) (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2015-02-15 - Written - Trick (MPIA)

Logarithmic halo potential

class galpy.potential.**LogarithmicHaloPotential**(*amp=1.0*, *core=1e-08*, *q=1.0*, *b=None*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the logarithmic potential

$$\Phi(R,z) = \frac{\mathrm{amp}}{2}\,\ln\left[R^2+\left(\frac{z}{q}\right)^2+\mathrm{core}^2\right]$$

Alternatively, the potential can be made triaxial by adding a parameter b

$$\Phi(x,y,z) = \frac{\mathrm{amp}}{2}\,\ln\left[x^2+\left(\frac{y}{b}\right)^2+\left(\frac{z}{q}\right)^2+\mathrm{core}^2\right]$$

With these definitions, $\sqrt{\text{amp}}$ is the circular velocity at $r \gg \text{core}$ at $(y, z) = (0, 0)$.

__init__(*amp=1.0*, *core=1e-08*, *q=1.0*, *b=None*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a logarithmic potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of velocity-squared

core - core radius at which the logarithm is cut (can be Quantity)

q - potential flattening (z/q)**2.

b= (None) if set, shape parameter in y-direction (y –> y/b; see definition)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-04-02 - Started - Bovy (NYU)

Miyamoto-Nagai potential

class galpy.potential.**MiyamotoNagaiPotential**(*amp=1.0*, *a=1.0*, *b=0.1*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Miyamoto-Nagai potential

$$\Phi(R, z) = -\frac{\text{amp}}{\sqrt{R^2 + (a + \sqrt{z^2 + b^2})^2}}$$

with $\text{amp} = GM$ the total mass.

__init__(*amp=1.0*, *a=1.0*, *b=0.1*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Miyamoto-Nagai potential

INPUT:

amp - amplitude to be applied to the potential, the total mass (default: 1); can be a Quantity with units of mass or Gxmass

a - scale length (can be Quantity)

b - scale height (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Started - Bovy (NYU)

Three Miyamoto-Nagai disk approximation to an exponential disk

class galpy.potential.**MN3ExponentialDiskPotential**(*amp=1.0*, *hr=0.3333333333333333*, *hz=0.0625*, *sech=False*, *posdens=False*, *normalize=False*, *ro=None*, *vo=None*)

class that implements the three Miyamoto-Nagai approximation to a radially-exponential disk potential of Smith et al. 2015

$$\rho(R,z) = \text{amp}\,\exp\left(-R/h_R - |z|/h_z\right)$$

or

$$\rho(R,z) = \text{amp}\,\exp\left(-R/h_R\right)\text{sech}^2\left(-|z|/h_z\right)$$

depending on whether sech=True or not. This density is approximated using three Miyamoto-Nagai disks

__init__(*amp=1.0*, *hr=0.3333333333333333*, *hz=0.0625*, *sech=False*, *posdens=False*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a 3MN approximation to an exponential disk potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass density or Gxmass density

hr - disk scale-length (can be Quantity)

hz - scale-height (can be Quantity)

sech= (False) if True, hz is the scale height of a sech vertical profile (default is exponential vertical profile)

posdens= (False) if True, allow only positive density solutions (Table 2 in Smith et al. rather than Table 1)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

MN3ExponentialDiskPotential object

HISTORY:

2015-02-07 - Written - Bovy (IAS)

Razor-thin exponential disk potential

class galpy.potential.**RazorThinExponentialDiskPotential**(*amp=1.0*, *hr=0.3333333333333333*, *normalize=False*, *ro=None*, *vo=None*, *new=True*, *glorder=100*)

Class that implements the razor-thin exponential disk potential

$$\rho(R,z) = \mathrm{amp}\,\exp\left(-R/h_R\right)\,\delta(z)$$

__init__(*amp=1.0*, *hr=0.3333333333333333*, *normalize=False*, *ro=None*, *vo=None*, *new=True*, *glorder=100*)

NAME:

__init__

PURPOSE:

initialize a razor-thin-exponential disk potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of surface-mass or Gxsurface-mass

hr - disk scale-length (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

RazorThinExponentialDiskPotential object

HISTORY:

2012-12-27 - Written - Bovy (IAS)

Ring potential

class galpy.potential.**RingPotential**(*amp=1.0*, *a=0.75*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the potential of an infinitesimally-thin, circular ring

$$\rho(R,z) = \frac{\mathrm{amp}}{2\pi\,R_0}\,\delta(R-R_0)\,\delta(z)$$

with $\mathrm{amp} = GM$ the mass of the ring.

__init__(*amp=1.0*, *a=0.75*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a circular ring potential

INPUT:

amp - mass of the ring (default: 1); can be a Quantity with units of mass or Gxmass

a= (0.75) radius of the ring (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.; note that because the force is always positive at r < a, this does not work if a > 1

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2018-08-04 - Written - Bovy (UofT)

Axisymmetrized N-body snapshot potential

class galpy.potential.**SnapshotRZPotential**(*s*, *num_threads=None*, *nazimuths=4*, *ro=None*, *vo=None*)

Class that implements an axisymmetrized version of the potential of an N-body snapshot (requires pynbody)

_evaluate, *_Rforce*, and *_zforce* calculate a hash for the array of points that is passed in by the user. The hash and corresponding potential/force arrays are stored – if a subsequent request matches a previously computed hash, the previous results are returned and not recalculated.

__init__(*s*, *num_threads=None*, *nazimuths=4*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a SnapshotRZ potential object

INPUT:

s - a simulation snapshot loaded with pynbody

num_threads= (4) number of threads to use for calculation

nazimuths= (4) number of azimuths to average over

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

instance

HISTORY:

2013 - Written - Rok Roskar (ETH)

2014-11-24 - Edited for merging into main galpy - Bovy (IAS)

Ellipsoidal triaxial potentials

`galpy` has very general support for implementing triaxial (or the oblate and prolate special cases) of ellipsoidal potentials through the general EllipsoidalPotential class. These potentials have densities that are uniform on ellipsoids, thus only functions of $m^2 = x^2 + \frac{y^2}{b^2} + \frac{z^2}{c^2}$. New potentials of this type can be implemented by inheriting from this class and implementing the `_mdens(self,m)`, `_psi(self,m)`, and `_mdens_deriv` functions for the density, its integral with respect to m^2, and its derivative with respect to m, respectively. For adding a C implementation, follow similar steps (use `PerfectEllipsoidPotential` as an example to follow).

Double power-law density triaxial potential

class galpy.potential.**TwoPowerTriaxialPotential**(*amp=1.0*, *a=5.0*, *alpha=1.5*, *beta=3.5*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements triaxial potentials that are derived from two-power density models

$$\rho(x,y,z) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(m/a)^\alpha\,(1+m/a)^{\beta-\alpha}}$$

with

$$m^2 = x'^2 + \frac{y'^2}{b^2} + \frac{z'^2}{c^2}$$

and (x', y', z') is a rotated frame wrt (x, y, z) specified by parameters `zvec` and `pa` which specify (a) `zvec`: the location of the z' axis in the (x, y, z) frame and (b) `pa`: the position angle of the x' axis wrt the $\tilde{x}$ axis, that is, the x axis after rotating to `zvec`.

Note that this general class of potentials does *not* automatically revert to the special TriaxialNFWPotential, TriaxialHernquistPotential, or TriaxialJaffePotential when using their (alpha,beta) values (like TwoPowerSphericalPotential).

__init__(*amp=1.0*, *a=5.0*, *alpha=1.5*, *beta=3.5*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a triaxial two-power-density potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

alpha - inner power (0 <= alpha < 3)

beta - outer power (beta > 2)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis (rad or Quantity)

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2016-05-30 - Started - Bovy (UofT)

2018-08-07 - Re-written using the general EllipsoidalPotential class - Bovy (UofT)

Perfect Ellipsoid potential

class galpy.potential.**PerfectEllipsoidPotential**(*amp=1.0*, *a=5.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

Potential of the perfect ellipsoid (de Zeeuw 1985):

$$\rho(x,y,z) = \frac{\mathrm{amp}\,\mathrm{a}}{\pi^2\,bc}\,\frac{1}{(m^2+a^2)^2}$$

where $\mathrm{amp} = GM$ is the total mass and $m^2 = x^2 + y^2/b^2 + z^2/c^2$.

__init__(*amp=1.0*, *a=5.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a perfect ellipsoid potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or G x mass

a - scale radius (can be Quantity)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis (rad or Quantity)

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2018-08-06 - Started - Bovy (UofT)

Triaxial power-law-density potential

class galpy.potential.**PowerTriaxialPotential**(*amp=1.0*, *alpha=1.0*, *r1=1.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements triaxial potentials that are derived from power-law density models (including an elliptical power law)

$$\rho(r) = \frac{\mathrm{amp}}{r_1^3}\,\left(\frac{r_1}{m}\right)^\alpha$$

where $m^2 = x^2 + y^2/b^2 + z^2/c^2$.

__init__(*amp=1.0*, *alpha=1.0*, *r1=1.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a triaxial power-law potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

alpha - power-law exponent

r1= (1.) reference radius for amplitude (can be Quantity)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis (rad or Quantity)

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2021-05-07 - Started - Bovy (UofT)

Triaxial Gaussian potential

class galpy.potential.**TriaxialGaussianPotential**(*amp=1.0*, *sigma=5.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

Potential of a triaxial Gaussian (Emsellem et al. 1994):

$$\rho(x,y,z) = \frac{\mathrm{amp}}{(2\pi\,\sigma)^{3/2}\,b\,c}\,e^{-\frac{m^2}{2\sigma^2}}$$

where $\mathrm{amp} = GM$ is the total mass and $m^2 = x^2 + y^2/b^2 + z^2/c^2$.

__init__(*amp=1.0*, *sigma=5.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Gaussian potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

sigma - Gaussian dispersion scale (can be Quantity)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis (rad or Quantity)

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2020-08-18 - Started - Bovy (UofT)

Triaxial Jaffe potential

class galpy.potential.**TriaxialJaffePotential**(*amp=1.0*, *a=2.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *normalize=False*, *glorder=50*, *ro=None*, *vo=None*)

Class that implements the Jaffe potential

$$\rho(x,y,z) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(m/a)^2\,(1+m/a)^2}$$

with

$$m^2 = x'^2 + \frac{y'^2}{b^2} + \frac{z'^2}{c^2}$$

and (x', y', z') is a rotated frame wrt (x, y, z) specified by parameters `zvec` and `pa` which specify (a) `zvec`: the location of the z' axis in the (x, y, z) frame and (b) `pa`: the position angle of the x' axis wrt the $\tilde{x}$ axis, that is, the x axis after rotating to `zvec`.

__init__(*amp=1.0*, *a=2.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *normalize=False*, *glorder=50*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a Jaffe potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Written - Bovy (UofT)

2018-08-07 - Re-written using the general EllipsoidalPotential class - Bovy (UofT)

Triaxial Hernquist potential

class galpy.potential.**TriaxialHernquistPotential**(*amp=1.0*, *a=2.0*, *normalize=False*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *ro=None*, *vo=None*)

Class that implements the triaxial Hernquist potential

$$\rho(x, y, z) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(m/a)\,(1+m/a)^3}$$

with

$$m^2 = x'^2 + \frac{y'^2}{b^2} + \frac{z'^2}{c^2}$$

and (x', y', z') is a rotated frame wrt (x, y, z) specified by parameters `zvec` and `pa` which specify (a) `zvec`: the location of the z' axis in the (x, y, z) frame and (b) `pa`: the position angle of the x' axis wrt the $\tilde{x}$ axis, that is, the x axis after rotating to `zvec`.

__init__(*amp=1.0*, *a=2.0*, *normalize=False*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *glorder=50*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a triaxial Hernquist potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2010-07-09 - Written - Bovy (UofT)

2018-08-07 - Re-written using the general EllipsoidalPotential class - Bovy (UofT)

Triaxial NFW potential

class galpy.potential.**TriaxialNFWPotential**(*amp=1.0*, *a=2.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *normalize=False*, *conc=None*, *mvir=None*, *glorder=50*, *vo=None*, *ro=None*, *H=70.0*, *Om=0.3*, *overdens=200.0*, *wrtcrit=False*)

Class that implements the triaxial NFW potential

$$\rho(x, y, z) = \frac{\mathrm{amp}}{4\,\pi\,a^3}\,\frac{1}{(m/a)\,(1+m/a)^2}$$

with

$$m^2 = x'^2 + \frac{y'^2}{b^2} + \frac{z'^2}{c^2}$$

and (x', y', z') is a rotated frame wrt (x, y, z) specified by parameters `zvec` and `pa` which specify (a) `zvec`: the location of the z' axis in the (x, y, z) frame and (b) `pa`: the position angle of the x' axis wrt the $\tilde{x}$ axis, that is, the x axis after rotating to `zvec`.

__init__(*amp=1.0*, *a=2.0*, *b=1.0*, *c=1.0*, *zvec=None*, *pa=None*, *normalize=False*, *conc=None*, *mvir=None*, *glorder=50*, *vo=None*, *ro=None*, *H=70.0*, *Om=0.3*, *overdens=200.0*, *wrtcrit=False*)

NAME:

__init__

PURPOSE:

Initialize a triaxial NFW potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

zvec= (None) If set, a unit vector that corresponds to the z axis

pa= (None) If set, the position angle of the x axis

glorder= (50) if set, compute the relevant force and potential integrals with Gaussian quadrature of this order

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

Alternatively, NFW potentials can be initialized using

conc= concentration

mvir= virial mass in 10^12 Msolar

in which case you also need to supply the following keywords

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

overdens= (200) overdensity which defines the virial radius

wrtcrit= (False) if True, the overdensity is wrt the critical density rather than the mean matter density

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2016-05-30 - Written - Bovy (UofT)

2018-08-06 - Re-written using the general EllipsoidalPotential class - Bovy (UofT)

Note that the Ferrers potential listed below is a potential of this type, but it is currently not implemented using the `EllipsoidalPotential` class. Further note that these potentials can all be rotated in 3D using the `zvec` and `pa` keywords; however, more general support for the same behavior is available through the `RotateAndTiltWrapperPotential` discussed below and the internal `zvec`/`pa` keywords will likely be deprecated in a future version.

Spiral, bar, other triaxial, and miscellaneous potentials

Dehnen bar potential

class galpy.potential.**DehnenBarPotential**(*amp=1.0*, *omegab=None*, *rb=None*, *chi=0.8*, *rolr=0.9*, *barphi=0.4363323129985824*, *tform=-4.0*, *tsteady=None*, *beta=0.0*, *alpha=0.01*, *Af=None*, *ro=None*, *vo=None*)

Class that implements the Dehnen bar potential (Dehnen 2000), generalized to 3D following Monari et al. (2016)

$$\Phi(R,z,\phi) = A_b(t)\,\cos\left(2\left(\phi-\Omega_b\,t\right)\right)\left(\frac{R}{r}\right)^2 \times \begin{cases} -(R_b/r)^3\,, & \text{for } r \geq R_b \\ (r/R_b)^3 - 2\,, & \text{for } r \leq R_b. \end{cases}$$

where $r^2 = R^2 + z^2$ is the spherical radius and

$$A_b(t) = A_f\left(\frac{3}{16}\xi^5 - \frac{5}{8}\xi^3 + \frac{15}{16}\xi + \frac{1}{2}\right), \xi = 2\frac{t/T_b - t_{\rm form}}{T_{\rm steady}} - 1\,, \text{ if } t_{\rm form} \leq \frac{t}{T_b} \leq t_{\rm form} + T_{\rm steady}$$

and

$$A_b(t) = \begin{cases} 0\,, & \frac{t}{T_b} < t_{\rm form} \\ A_f\,, & \frac{t}{T_b} > t_{\rm form} + T_{\rm steady} \end{cases}$$

where

$$T_b = \frac{2\pi}{\Omega_b}$$

is the bar period and the strength can also be specified using α

$$\alpha = 3\,\frac{A_f}{v_0^2}\left(\frac{R_b}{r_0}\right)^3.$$

__init__(*amp=1.0*, *omegab=None*, *rb=None*, *chi=0.8*, *rolr=0.9*, *barphi=0.4363323129985824*, *tform=-4.0*, *tsteady=None*, *beta=0.0*, *alpha=0.01*, *Af=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Dehnen bar potential

INPUT:

amp - amplitude to be applied to the potential (default: 1., see alpha or Ab below)

barphi - angle between sun-GC line and the bar's major axis (in rad; default=25 degree; or can be Quantity))

tform - start of bar growth / bar period (default: -4)

tsteady - time from tform at which the bar is fully grown / bar period (default: -tform/2, st the perturbation is fully grown at tform/2)

Either provide:

a) rolr - radius of the Outer Lindblad Resonance for a circular orbit (can be Quantity)

chi - fraction R_bar / R_CR (corotation radius of bar)

alpha - relative bar strength (default: 0.01)

beta - power law index of rotation curve (to calculate OLR, etc.)

b) omegab - rotation speed of the bar (can be Quantity)

rb - bar radius (can be Quantity)

Af - bar strength (can be Quantity)

OUTPUT:

(none)

HISTORY:

2010-11-24 - Started - Bovy (NYU)

2017-06-23 - Converted to 3D following Monari et al. (2016) - Bovy (UofT/CCA)

Ferrers potential

class galpy.potential.**FerrersPotential**(*amp=1.0*, *a=1.0*, *n=2*, *b=0.35*, *c=0.2375*, *omegab=0.0*, *pa=0.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements triaxial Ferrers potential for the ellipsoidal density profile with the short axis along the z-direction

$$\rho(x,y,z) = \frac{\mathrm{amp}}{\pi^{1.5}a^3bc}\frac{\Gamma(n+\frac{5}{2})}{\Gamma(n+1)}\left(1-(m/a)^2\right)^n$$

with

$$m^2 = x'^2 + \frac{y'^2}{b^2} + \frac{z'^2}{c^2}$$

and (x', y', z') is a rotated frame wrt (x, y, z) so that the major axis is aligned with x'.

Note that this potential has not yet been optimized for speed and has no C implementation, so orbit integration is currently slow.

__init__(*amp=1.0*, *a=1.0*, *n=2*, *b=0.35*, *c=0.2375*, *omegab=0.0*, *pa=0.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Ferrers potential

INPUT:

amp - total mass of the ellipsoid determines the amplitude of the potential; can be a Quantity with units of mass or Gxmass

a - scale radius (can be Quantity)

n - power of Ferrers density (n > 0)

b - y-to-x axis ratio of the density

c - z-to-x axis ratio of the density

omegab - rotation speed of the ellipsoid (can be Quantity)

pa= (None) If set, the position angle of the x axis (rad or Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

Moving object potential

class galpy.potential.**MovingObjectPotential**(*orbit*, *pot=None*, *amp=1.0*, *ro=None*, *vo=None*)

Class that implements the potential coming from a moving object by combining any galpy potential with an integrated galpy orbit.

__init__(*orbit*, *pot=None*, *amp=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a MovingObjectPotential

INPUT:

orbit - the Orbit of the object (Orbit object)

pot - A potential object or list of potential objects representing the potential of the moving object; should be spherical, but this is not checked [default= PlummerPotential(amp=0.06,b=0.01)]

amp (=1.) another amplitude to apply to the potential

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2011-04-10 - Started - Bovy (NYU)

2018-10-18 - Re-implemented to represent general object potentials using galpy potential models - James Lane (UofT)

Constant (null) potential

class galpy.potential.**NullPotential**(*amp=1.0*, *ro=None*, *vo=None*)

Class that implements a constant potential with, thus, zero forces. Can be used, for example, for integrating orbits in the absence of forces or for adjusting the value of the total gravitational potential, say, at infinity

__init__(*amp=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a null potential: a constant potential with, thus, zero forces

INPUT:

amp - constant value of the potential (default: 1); can be a Quantity with units of velocity-squared

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2022-03-18 - Written - Bovy (UofT)

Softened-needle bar potential

class galpy.potential.**SoftenedNeedleBarPotential**(*amp=1.0*, *a=4.0*, *b=0.0*, *c=1.0*, *normalize=False*, *pa=0.4*, *omegab=1.8*, *ro=None*, *vo=None*)

Class that implements the softened needle bar potential from Long & Murali (1992)

$$\Phi(x,y,z) = \frac{\mathrm{amp}}{2a}\,\ln\left(\frac{x-a+T_-}{x+a+T_+}\right)$$

where

$$T_\pm = \sqrt{(a\pm x)^2+y^2+(b+\sqrt{z^2+c^2})^2}$$

For a prolate bar, set b to zero.

__init__(*amp=1.0*, *a=4.0*, *b=0.0*, *c=1.0*, *normalize=False*, *pa=0.4*, *omegab=1.8*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a softened-needle bar potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass

a= (4.) Bar half-length (can be Quantity)

b= (1.) Triaxial softening length (can be Quantity)

c= (1.) Prolate softening length (can be Quantity)

pa= (0.4) The position angle of the x axis (rad or Quantity)

omegab= (1.8) Pattern speed (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

(none)

HISTORY:

2016-11-02 - Started - Bovy (UofT)

Spiral arms potential

class galpy.potential.**SpiralArmsPotential**(*amp=1, ro=None, vo=None, amp_units='density', N=2, alpha=0.2, r_ref=1, phi_ref=0, Rs=0.3, H=0.125, omega=0, Cs=[1]*)

Class that implements the spiral arms potential from (Cox and Gomez 2002). Should be used to modulate an existing potential (density is positive in the arms, negative outside; note that because of this, a contour plot of this potential will appear to have twice as many arms, where half are the underdense regions).

$$\Phi(R,\phi,z) = -4\pi GH\,\rho_0 exp\left(-\frac{R-r_{ref}}{R_s}\right)\sum\frac{C_n}{K_nD_n}\,\cos(n\gamma)\,\mathrm{sech}^{B_n}\left(\frac{K_nz}{B_n}\right)$$

where

$$K_n = \frac{nN}{R\sin(\alpha)}$$
$$B_n = K_nH(1+0.4K_nH)$$
$$D_n = \frac{1+K_nH+0.3(K_nH)^2}{1+0.3K_nH}$$

and

$$\gamma = N\left[\phi-\phi_{ref}-\frac{\ln(R/r_{ref})}{\tan(\alpha)}\right]$$

The default of $C_n = [1]$ gives a sinusoidal profile for the potential. An alternative from Cox and Gomez (2002) creates a density that behaves approximately as a cosine squared in the arms but is separated by a flat interarm region by setting

$$C_n = \left[\frac{8}{3\pi},\frac{1}{2},\frac{8}{15\pi}\right]$$

__init__(*amp=1, ro=None, vo=None, amp_units='density', N=2, alpha=0.2, r_ref=1, phi_ref=0, Rs=0.3, H=0.125, omega=0, Cs=[1]*)

NAME: __init__

PURPOSE: initialize a spiral arms potential

INPUT:

amp amplitude to be applied to the potential (default: 1); can be a Quantity with units of density. ($amp = 4\pi G\rho_0$)

ro distance scales for translation into internal units (default from configuration file)

vo velocity scales for translation into internal units (default from configuration file)

N number of spiral arms

alpha pitch angle of the logarithmic spiral arms in radians (can be Quantity)

r_ref fiducial radius where $\rho = \rho_0$ (r_0 in the paper by Cox and Gomez) (can be Quantity)

phi_ref reference angle ($\phi_p(r_0)$ in the paper by Cox and Gomez) (can be Quantity)

Rs radial scale length of the drop-off in density amplitude of the arms (can be Quantity)

H scale height of the stellar arm perturbation (can be Quantity)

Cs list of constants multiplying the $\cos(n\gamma)$ terms

omega rotational pattern speed of the spiral arms (can be Quantity)

OUTPUT: (none)

HISTORY: Started - 2017-05-12 Jack Hong (UBC)

Completed - 2017-07-04 Jack Hong (UBC)

All `galpy` potentials can also be made to rotate using the `SolidBodyRotationWrapperPotential` listed in the section on wrapper potentials *below*.

General Poisson solvers for disks and halos

Disk potential using SCF basis-function-expansion

class galpy.potential.**DiskSCFPotential**(*amp=1.0*, *normalize=False*, *dens=<function DiskSCFPotential.<lambda>>*, *Sigma={'amp': 1.0, 'h': 0.3333333333333333, 'type': 'exp'}*, *hz={'h': 0.037037037037037035, 'type': 'exp'}*, *Sigma_amp=None*, *dSigmadR=None*, *d2SigmadR2=None*, *Hz=None*, *dHzdz=None*, *N=10*, *L=10*, *a=1.0*, *radial_order=None*, *costheta_order=None*, *phi_order=None*, *ro=None*, *vo=None*)

Class that implements a basis-function-expansion technique for solving the Poisson equation for disk (+halo) systems. We solve the Poisson equation for a given density $\rho(R, \phi, z)$ by introducing *K* helper function pairs $[\Sigma_i(R), h_i(z)]$, with $h_i(z) = \mathrm{d}^2H(z)/\mathrm{d}z^2$ and search for solutions of the form

$$\Phi(R, \phi, z = \Phi_{\rm ME}(R, \phi, z) + 4\pi G \sum_i \Sigma_i(r)\, H_i(z)\,,$$

where r is the spherical radius $r^2 = R^2 + z^2$. We can solve for $\Phi_{\rm ME}(R, \phi, z)$ by solving

$$\frac{\Delta\Phi_{\rm ME}(R, \phi, z)}{4\pi G} = \rho(R, \phi, z) - \sum_i \left\{ \Sigma_i(r)\, h_i(z) + \frac{\mathrm{d}^2\Sigma_i(r)}{\mathrm{d}r^2}\, H_i(z) + \frac{2}{r}\,\frac{\mathrm{d}\Sigma_i(r)}{\mathrm{d}r}\left[H_i(z) + z\,\frac{\mathrm{d}H_i(z)}{\mathrm{d}z}\right]\right\}.$$

We solve this equation by using the *SCFPotential* class and methods (*scf_compute_coeffs_axi* or *scf_compute_coeffs* depending on whether $\rho(R,\phi,z)$ is axisymmetric or not). This technique works very well if the disk portion of the potential can be exactly written as $\rho_{\rm disk} = \sum_i \Sigma_i(R)\,h_i(z)$, because the effective density on the right-hand side of this new Poisson equation is then not 'disky' and can be well represented using spherical harmonics. But the technique is general and can be used to compute the potential of any disk+halo potential; the closer the disk is to $\rho_{\rm disk} \approx \sum_i \Sigma_i(R)\,h_i(z)$, the better the technique works.

This technique was introduced by Kuijken & Dubinski (1995) and was popularized by Dehnen & Binney (1998). The current implementation is a slight generalization of the technique in those papers and uses the SCF approach of Hernquist & Ostriker (1992) to solve the Poisson equation for $\Phi_{\rm ME}(R,\phi,z)$ rather than solving it on a grid using spherical harmonics and interpolating the solution (as done in Dehnen & Binney 1998).

__init__(*amp=1.0*, *normalize=False*, *dens=<function DiskSCFPotential.<lambda>>*, *Sigma={'amp': 1.0, 'h': 0.3333333333333333, 'type': 'exp'}*, *hz={'h': 0.037037037037037035, 'type': 'exp'}*, *Sigma_amp=None*, *dSigmadR=None*, *d2SigmadR2=None*, *Hz=None*, *dHzdz=None*, *N=10*, *L=10*, *a=1.0*, *radial_order=None*, *costheta_order=None*, *phi_order=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a DiskSCF Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); cannot have units currently

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

dens= function of R,z[,phi optional] that gives the density [in natural units, cannot return a Quantity currently]

N=, L=, a=, radial_order=, costheta_order=, phi_order= keywords setting parameters for SCF solution for Phi_ME (see *scf_compute_coeffs_axi* or *scf_compute_coeffs* depending on whether $\rho(R,\phi,z)$ is axisymmetric or not)

Either:

(a) Sigma= Dictionary of surface density (example: {'type':'exp','h':1./3.,'amp':1.,'Rhole':0.} for amp x exp(-Rhole/R-R/h))

hz= Dictionary of vertical profile, either 'exp' or 'sech2' (example {'type':'exp','h':1./27.} for exp(-|z|/h)/[2h], sech2 is sech^2(z/[2h])/[4h])

(b) Sigma= function of R that gives the surface density

dSigmadR= function that gives d Sigma / d R

d2SigmadR2= function that gives d^2 Sigma / d R^2

Sigma_amp= amplitude to apply to all Sigma functions

hz= function of z that gives the vertical profile

Hz= function of z such that d^2 Hz(z) / d z^2 = hz

dHzdz= function of z that gives d Hz(z) / d z

In both of these cases lists of arguments can be given for multiple disk components; can't mix (a) and (b) in these lists; if hz is a single item the same vertical profile is assumed for all Sigma

OUTPUT:

DiskSCFPotential object

HISTORY:

2016-12-26 - Written - Bovy (UofT)

Hernquist & Ostriker Self-Consistent-Field-type potential

class galpy.potential.**SCFPotential**(*amp=1.0*, *Acos=array([[[1]]])*, *Asin=None*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

Class that implements the Hernquist & Ostriker (1992) Self-Consistent-Field-type potential. Note that we divide the amplitude by 2 such that $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$ and $Asin = 0$ corresponds to *Galpy's Hernquist Potential*.

$$\rho(r,\theta,\phi) = \frac{amp}{2}\sum_{n=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{l} N_{lm}P_{lm}(\cos(\theta))\tilde{\rho}_{nl}(r)\left(A_{cos,nlm}\cos(m\phi) + A_{sin,nlm}\sin(m\phi)\right)$$

where

$$\tilde{\rho}_{nl}(r) = \frac{K_{nl}}{\sqrt{\pi}}\frac{(ar)^l}{(r/a)(a+r)^{2l+3}}C_n^{2l+3/2}(\xi)$$

$$\Phi(r,\theta,\phi) = \sum_{n=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=0}^{l} N_{lm}P_{lm}(\cos(\theta))\tilde{\Phi}_{nl}(r)\left(A_{cos,nlm}\cos(m\phi) + A_{sin,nlm}\sin(m\phi)\right)$$

where

$$\tilde{\Phi}_{nl}(r) = -\sqrt{4\pi}K_{nl}\frac{(ar)^l}{(a+r)^{2l+1}}C_n^{2l+3/2}(\xi)$$

where

$$\xi = \frac{r-a}{r+a} \qquad N_{lm} = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}(2-\delta_{m0}) \qquad K_{nl} = \frac{1}{2}n(n+4l+3) + (l+1)(2l+1)$$

and P_{lm} is the Associated Legendre Polynomials whereas C_n^{α} is the Gegenbauer polynomial.

__init__(*amp=1.0*, *Acos=array([[[1]]])*, *Asin=None*, *a=1.0*, *normalize=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a SCF Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1); can be a Quantity with units of mass or Gxmass

Acos - The real part of the expansion coefficent (NxLxL matrix, or optionally NxLx1 if Asin=None)

Asin - The imaginary part of the expansion coefficient (NxLxL matrix or None)

a - scale length (can be Quantity)

normalize - if True, normalize such that vc(1.,0.)=1., or, if given as a number, such that the force is this fraction of the force necessary to make vc(1.,0.)=1.

ro=, vo= distance and velocity scales for translation into internal units (default from configuration file)

OUTPUT:

SCFPotential object

HISTORY:

2016-05-13 - Written - Aladdin Seaifan (UofT)

Dissipative forces

Chandrasekhar dynamical friction

class galpy.potential.**ChandrasekharDynamicalFrictionForce**(*amp=1.0*, *GMs=0.1*, *gamma=1.0*, *rhm=0.0*, *dens=None*, *sigmar=None*, *const_lnLambda=False*, *minr=0.0001*, *maxr=25.0*, *nr=501*, *ro=None*, *vo=None*)

Class that implements the Chandrasekhar dynamical friction force

$$\mathbf{F}(\mathbf{x},\mathbf{v}) = -2\pi\left[G\,M\right]\left[G\,\rho(\mathbf{x})\right]\,\ln[1+\Lambda^2]\,\left[\mathrm{erf}(X) - \frac{2X}{\sqrt{\pi}}\exp\left(-X^2\right)\right]\,\frac{\mathbf{v}}{|\mathbf{v}|^3}$$

on a mass (e.g., a satellite galaxy or a black hole) M at position $\mathbf{x}$ moving at velocity $\mathbf{v}$ through a background density ρ. The quantity X is the usual $X = |\mathbf{v}|/[\sqrt{2}\sigma_r(r)$. The factor Λ that goes into the Coulomb logarithm is taken to be

$$\Lambda = \frac{r/\gamma}{\max\left(r_{\mathrm{hm}}, GM/|\mathbf{v}|^2\right)}\,,$$

where γ is a constant. This γ should be the absolute value of the logarithmic slope of the density $\gamma = |\mathrm{d}\ln\rho/\mathrm{d}\ln r|$, although for $\gamma < 1$ it is advisable to set $\gamma = 1$. Implementation here roughly follows 2016MNRAS.463..858P and earlier work.

__init__(*amp=1.0*, *GMs=0.1*, *gamma=1.0*, *rhm=0.0*, *dens=None*, *sigmar=None*, *const_lnLambda=False*, *minr=0.0001*, *maxr=25.0*, *nr=501*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Chandrasekhar Dynamical Friction force

INPUT:

amp - amplitude to be applied to the potential (default: 1)

GMs - satellite mass; can be a Quantity with units of mass or Gxmass; can be adjusted after initialization by setting obj.GMs= where obj is your ChandrasekharDynamicalFrictionForce instance (note that the mass of the satellite can *not* be changed simply by multiplying the instance by a number, because he mass is not only used as an amplitude)

rhm - half-mass radius of the satellite (set to zero for a black hole; can be a Quantity); can be adjusted after initialization by setting obj.rhm= where obj is your ChandrasekharDynamicalFrictionForce instance

gamma - Free-parameter in Λ

dens - Potential instance or list thereof that represents the density [default: LogarithmicHaloPotential(normalize=1.,q=1.)]

sigmar= (None) function that gives the velocity dispersion as a function of r (has to be in natural units!); if None, computed from the dens potential using the spherical Jeans equation (in galpy.df.jeans) assuming zero anisotropy; if set to a lambda function, *the object cannot be pickled* (so set it to a real function)

cont_lnLambda= (False) if set to a number, use a constant ln(Lambda) instead with this value

minr= (0.0001) minimum r at which to apply dynamical friction: at r < minr, friction is set to zero (can be a Quantity)

Interpolation:

maxr= (25) maximum r for which sigmar gets interpolated; for best performance set this to the maximum r you will consider (can be a Quantity)

nr= (501) number of radii to use in the interpolation of sigmar

You can check that sigmar is interpolated correctly by comparing the methods sigmar [the interpolated version] and sigmar_orig [the original or directly computed version]

OUTPUT:

(none)

HISTORY:

2011-12-26 - Started - Bovy (NYU)

2018-03-18 - Re-started: updated to r dependent Lambda form and integrated into galpy framework - Bovy (UofT)

2018-07-23 - Calculate sigmar from the Jeans equation and interpolate it; allow GMs and rhm to be set on the fly - Bovy (UofT)

Fictitious forces in non-inertial frames

Fictitious forces in non-intertial frames

class galpy.potential.**NonInertialFrameForce**(*amp=1.0*, *Omega=None*, *Omegadot=None*, *x0=None*, *v0=None*, *a0=None*, *ro=None*, *vo=None*)

Class that implements the fictitious forces present when integrating orbits in a non-intertial frame. Coordinates in the inertial frame $\mathbf{x}$ and in the non-inertial frame $\mathbf{r}$ are related through rotation and linear motion as

$$\mathbf{x} = \mathbf{R}\,(\mathbf{r} + \mathbf{x}_0)$$

where $\mathbf{R}$ is a rotation matrix and $\mathbf{x}_0$ is the motion of the origin. The rotation matrix has angular frequencies $\mathbf{\Omega}$ with time derivative $\dot{\mathbf{\Omega}}$; $\mathbf{\Omega}$ can be any function of time. The motion of the origin can also be any function of time. This leads to the fictitious force

$$\mathbf{F} = -\mathbf{a}_0 - \mathbf{\Omega} \times (\mathbf{\Omega} \times [\mathbf{r} + \mathbf{x}_0]) - \dot{\mathbf{\Omega}} \times [\mathbf{r} + \mathbf{x}_0] - 2\mathbf{\Omega} \times [\dot{\mathbf{r}} + \mathbf{v}_0]$$

where $\mathbf{a}_0$, $\mathbf{v}_0$, and $\mathbf{x}_0$ are the acceleration, velocity, and position of the origin of the non-inertial frame, respectively, as a function of time. Note that if the non-inertial frame is not rotating, it is not necessary to specify $\mathbf{v}_0$ and $\mathbf{x}_0$. In that case, the fictitious force is simply

$$\mathbf{F} = -\mathbf{a}_0 \quad (\mathbf{\Omega} = 0)$$

If the non-inertial frame only rotates without any motion of the origin, the fictitious force is the familiar combination of the centrifugal force and the Coriolis force (plus an additional term if $\dot{\mathbf{\Omega}}$ is not constant)

$$\mathbf{F} = -\mathbf{\Omega} \times (\mathbf{\Omega} \times \mathbf{r}) - \dot{\mathbf{\Omega}} \times \mathbf{r} - 2\mathbf{\Omega} \times \dot{\mathbf{r}} \quad (\mathbf{a}_0 = \mathbf{v}_0 = \mathbf{x}_0 = 0)$$

The functions of time are passed to the C code for fast orbit integration by attempting to build fast `numba` versions of them. Significant speed-ups can therefore be obtained by making sure that the provided functions can be turned into `nopython=True` `numba` functions (try running `numba.njit` on them and then evaluate them to check).

__init__(*amp=1.0*, *Omega=None*, *Omegadot=None*, *x0=None*, *v0=None*, *a0=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a NonInertialFrameForce

INPUT:

amp= (1.) amplitude to be applied to the potential (default: 1)

Omega= (1.) Angular frequency of the rotation of the non-inertial frame in an inertial one; can either be a function of time or a number (when the frequency is assumed to be Omega + Omegadot x t) and in each case can be a list [Omega_x,Omega_y,Omega_z] or a single value Omega_z (when not a function, can be a Quantity; when a function, need to take input time in internal units and output the frequency in internal units; see galpy.util.conversion.time_in_Gyr and galpy.util.conversion.freq_in_XXX conversion functions)

Omegadot= (None) Time derivative of the angular frequency of the non-intertial frame's rotation. format should match Omega input ([list of] function[s] when Omega is one, number/list if Omega is a number/list; when a function, need to take input time in internal units and output the frequency derivative in internal units; see galpy.util.conversion.time_in_Gyr and galpy.util.conversion.freq_in_XXX conversion functions)

x0= (None) Position vector x_0 (cartesian) of the center of mass of the non-intertial frame (see definition in the class documentation); list of functions [x_0x,x_0y,x_0z]; only necessary when considering both rotation and center-of-mass acceleration of the inertial frame (functions need to take input time in internal units and output the position in internal units; see galpy.util.conversion.time_in_Gyr and divided physical positions by the *ro* parameter in kpc)

v0= (None) Velocity vector v_0 (cartesian) of the center of mass of the non-intertial frame (see definition in the class documentation); list of functions [v_0x,v_0y,v_0z]; only necessary when considering both rotation and center-of-mass acceleration of the inertial frame (functions need to take input time in internal units and output the velocity in internal units; see galpy.util.conversion.time_in_Gyr and divided physical positions by the *vo* parameter in km/s)

a0= (None) Acceleration vector a_0 (cartesian) of the center of mass of the non-intertial frame (see definition in the class documentation); constant or a list of functions [a_0x,a_0y, a_0z] (functions need to take input time in internal units and output the acceleration in internal units; see galpy.util.conversion.time_in_Gyr and galpy.util.conversion.force_in_XXX conversion functions [force is actually acceleration in galpy])

OUTPUT:

(none)

HISTORY:

2022-03-02 - Started - Bovy (UofT)

2022-03-26 - Generalized Omega to any function of time - Bovy (UofT)

Helper classes

Mixin to compute forces and second potential derivatives numerically

class galpy.potential.**NumericalPotentialDerivativesMixin**(*kwargs*)

Mixin to add numerical derivatives to a Potential class, use as, e.g.,

```
class PotWithNumericalDerivs(Potential,NumericalPotentialDerivativesMixin):
    def __init__(self,*args,**kwargs):
        NumericalPotentialDerivativesMixin.__init__(self,kwargs) # *not* **kwargs!
        # Remainder of initialization
        ...

    def _evaluate(self,R,z,phi=0.,t=0.):
        # Evaluate the potential

    # All forces and second derivatives then computed by␣
↪NumericalPotentialDerivativesMixin
```

to add numerical derivatives to a new potential class `PotWithNumericalDerivs` that only implements the potential itself, but not the forces. The class may implement any of the forces or second derivatives, all non-implemented forces/second-derivatives will be computed numerically by adding this Mixin

The step used to compute the first (force) and second derivatives can be controlled at object instantiation by the keyword arguments `dR`, `dz`, `dphi` (for the forces; 1e-8 default) and `dR2`, `dz2`, and `dphi2` (for the second derivaives; 1e-4 default)

__init__(*kwargs*)

Initialize self. See help(type(self)) for accurate signature.

2.2.2 Milky-Way-like potentials

`galpy` contains various simple models for the Milky Way's gravitational potential. The recommended model, described in Bovy (2015), is included as `galpy.potential.MWPotential2014`. This potential was fit to a large variety of data on the Milky Way and thus serves as both a simple and accurate model for the Milky Way's potential (see Bovy 2015 for full information on how this potential was fit). Note that this potential assumes a circular velocity of 220 km/s at the solar radius at 8 kpc. This potential is defined as

```
>>> bp= PowerSphericalPotentialwCutoff(alpha=1.8,rc=1.9/8.,normalize=0.05)
>>> mp= MiyamotoNagaiPotential(a=3./8.,b=0.28/8.,normalize=.6)
>>> np= NFWPotential(a=16/8.,normalize=.35)
>>> MWPotential2014= bp+mp+np
```

and can thus be used like any list of `Potentials`. The mass of the dark-matter halo in `MWPotential2014` is on the low side of estimates of the Milky Way's halo mass; if you want to adjust it, for example making it 50% larger, you can simply multiply the halo part of `MWPotential2014` by 1.5 as (this type of multiplication works for *any* potential in galpy)

```
>>> MWPotential2014[2]*= 1.5
```

If one wants to add the supermassive black hole at the Galactic center, this can be done by

```
>>> from galpy.potential import KeplerPotential
>>> from galpy.util import conversion
>>> MWPotential2014wBH= MWPotential2014+KeplerPotential(amp=4*10**6./conversion.mass_
↪in_msol(220.,8.))
```

for a black hole with a mass of $4 \times 10^6\ M_\odot$. If you want to take into account dynamical friction for, say, an object of mass $5 \times 10^{10}\ M_\odot$ and a half-mass radius of 5 kpc, do

```
>>> from galpy.potential import ChandrasekharDynamicalFrictionForce
>>> from astropy import units
>>> cdf= ChandrasekharDynamicalFrictionForce(GMs=5.*10.**10.*units.Msun,
                                             rhm=5.*units.kpc,
                                             dens=MWPotential2014)
>>> MWPotential2014wDF= MWPotential2014+cdf
```

where we have specified the parameters of the dynamical friction with units; alternatively, convert them directly to `galpy` natural units as

```
>>> cdf= ChandrasekharDynamicalFrictionForce(GMs=5.*10.**10./conversion.mass_in_
↪msol(220.,8.),
                                             rhm=5./8.,
                                             dens=MWPotential2014)
>>> MWPotential2014wDF= MWPotential2014+cdf
```

As explained in *this section*, *without* this black hole or dynamical friction, `MWPotential2014` can be used with Dehnen's gyrfalcON code using `accname=PowSphwCut+MiyamotoNagai+NFW` and `accpars=0,1001.79126907,1.8,1.9#0,306770.418682,3.0,0.28#0,16.0,162.958241887`.

`galpy` also contains other models for the Milky Way's potential from the literature in the `galpy.potential.mwpotentials` module (which also contains `MWPotential2014`). Currently, these are:

- `McMillan17`: the potential model from McMillan (2017)
- `Irrgang13I`: model I from Irrgang et al. (2013), which is an updated version of the classic Allen & Santillan (1991)
- `Irrgang13II` and `Irrgang13III`: model II and III from Irrgang et al. (2013)
- `Cautun20`: the potential model from Cautun et al. (2020)
- `DehnenBinney98I`, `DehnenBinney98II`, `DehnenBinney98III`, and `DehnenBinney98IV` for models 1 through 4 from Dehnen & Binney (1998).

Unlike `MWPotential2014`, these potentials have physical units turned on, using as the unit scaling parameters `ro` and `vo` the distance to the Galactic center and the circular velocity at the Sun's radius of each potential. These can be obtained using the `galpy.util.conversion.get_physical` function, e.g.,

```
>>> from galpy.potential.mwpotentials import McMillan17
>>> from galpy.util.conversion import get_physical
>>> get_physical(McMillan17)
# {'ro': 8.21, 'vo': 233.1}
```

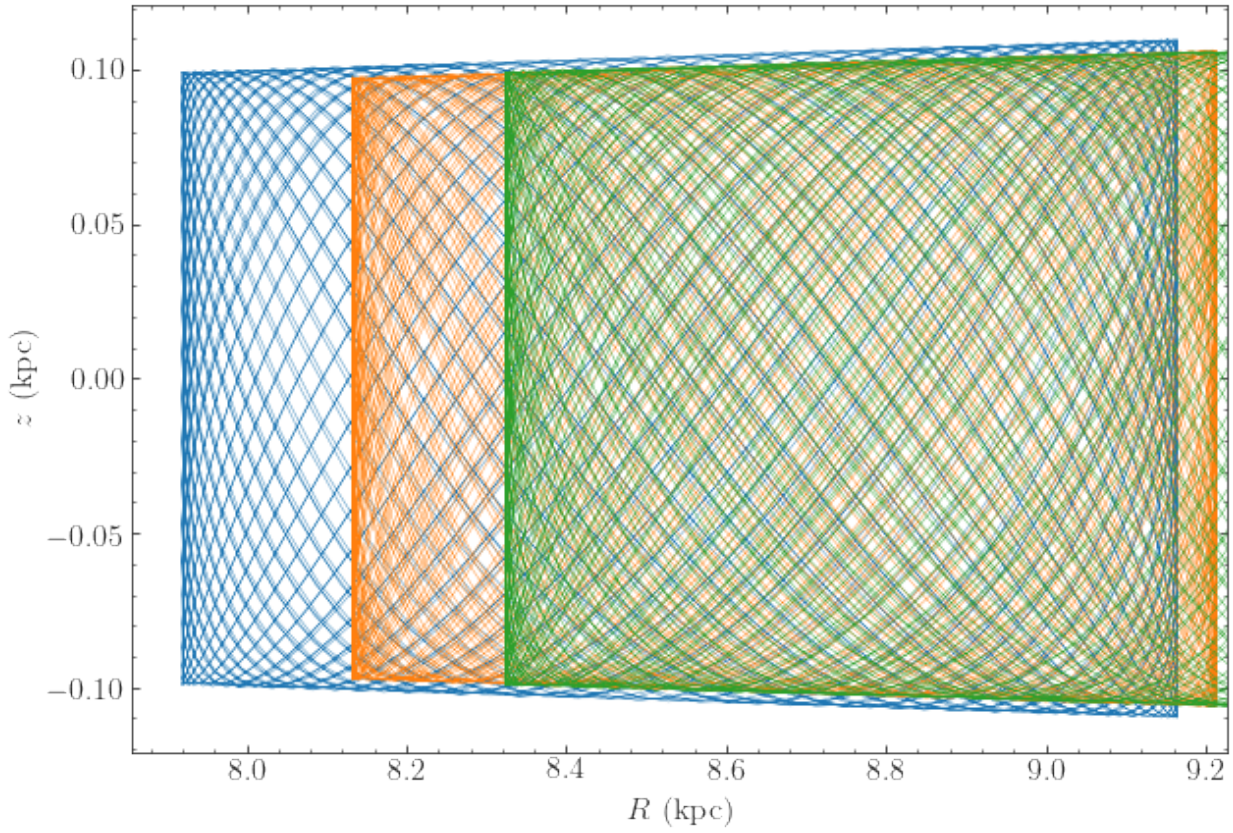
This function returns the unit-conversion parameters as a dictionary, so they can be easily passed to other functions. For example, when integrating an orbit in these potentials and either initializing the orbit using observed coordinates or converting the integrated orbit to observed coordinates, it is important to use the same unit-conversion parameters (otherwise an error will be raised). For example, to obtain the orbit of the Sun in the `McMillan17` potential, we do

```
>>> from galpy.orbit import Orbit
>>> o= Orbit(**get_physical(McMillan17))
```

As an example, we integrate the Sun's orbit for 10 Gyr in `MWPotential2014`, `McMillan17` and `Irrgang13I`

```
>>> from galpy.potential.mwpotentials import MWPotential2014, McMillan17, Irrgang13I
>>> from galpy.orbit import Orbit
>>> from galpy.util.conversion import get_physical
>>> from astropy import units
>>> times= numpy.linspace(0.,10.,3001)*units.Gyr
>>> o_mwp14= Orbit(ro=8.,vo=220.) # Need to set these by hand
>>> o_mcm17= Orbit(**get_physical(McMillan17))
>>> o_irrI= Orbit(**get_physical(Irrgang13I))
>>> o_mwp14.integrate(times,MWPotential2014)
>>> o_mcm17.integrate(times,McMillan17)
>>> o_irrI.integrate(times,Irrgang13I)
>>> o_mwp14.plot(lw=0.6)
>>> o_mcm17.plot(overplot=True,lw=0.6)
>>> o_irrI.plot(overplot=True,lw=0.6)
```

which gives



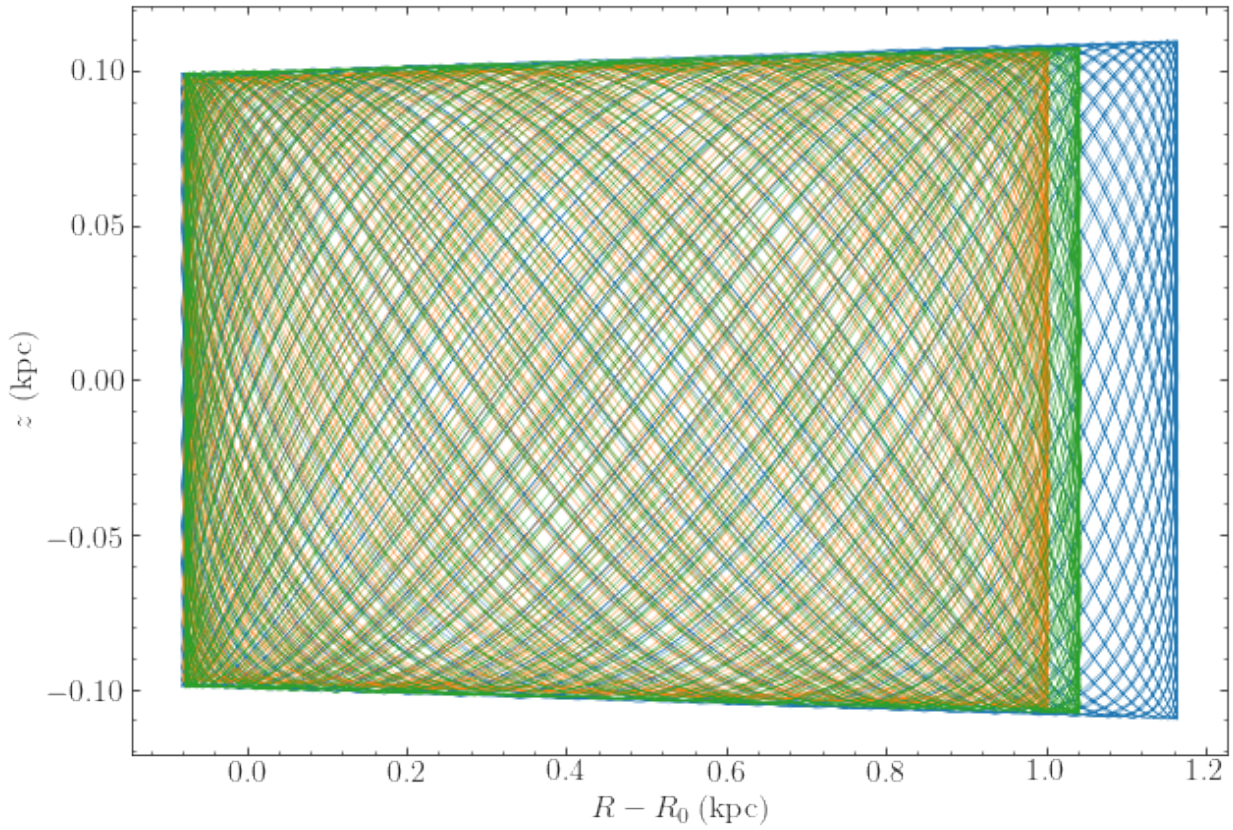
Much of the difference between these orbits is due to the different present Galactocentric radius of the Sun, if we simply plot the difference with respect to the present Galactocentric radius, they agree better

```
>>> o_mwp14.plot(d1='R-8.',d2='z',lw=0.6,xlabel=r'$R-R_0\,(\mathrm{kpc})$')
>>> o_mcm17.plot(d1='R-{}'.format(get_physical(McMillan17)['ro']),d2='z',
↪overplot=True,lw=0.6)
```

(continues on next page)

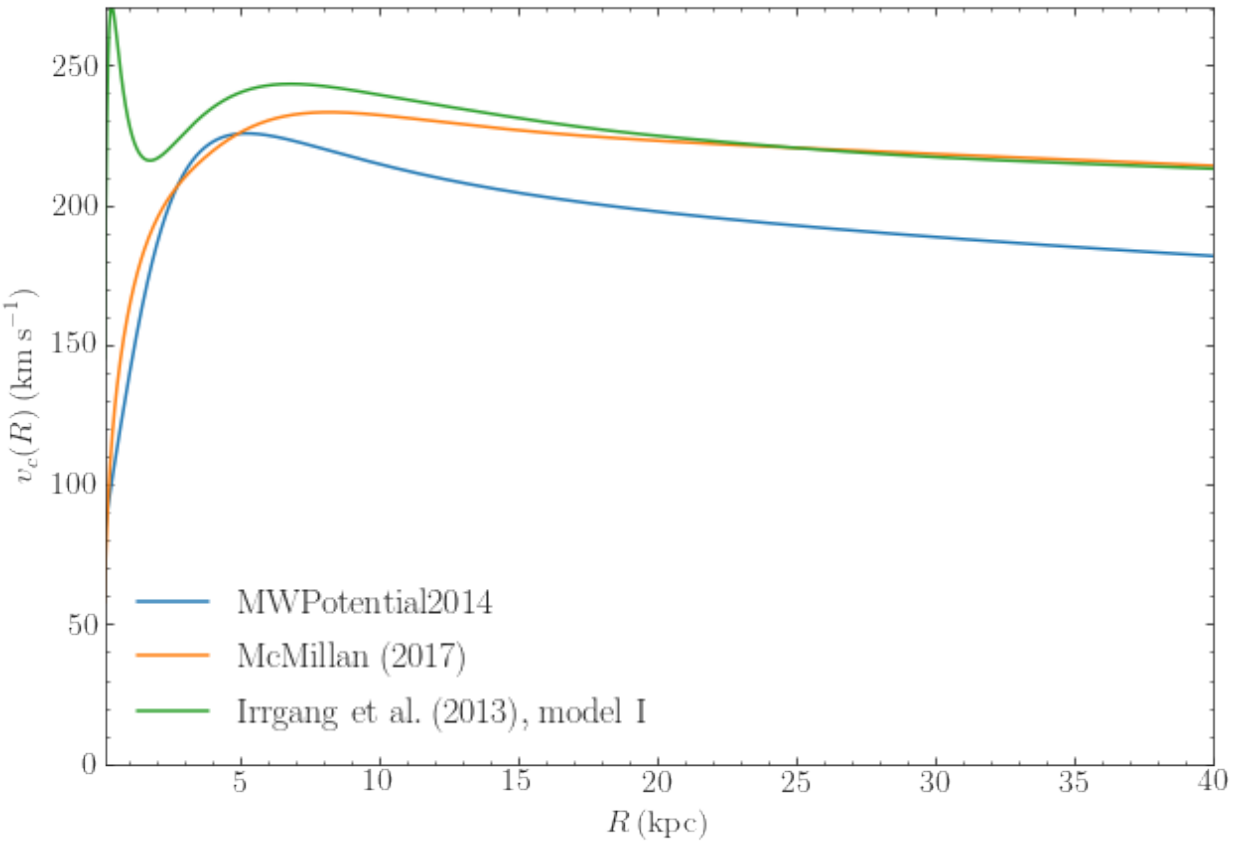
(continued from previous page)

```
>>> o_irrI.plot(d1='R-{}'.format(get_physical(Irrgang13I)['ro']),d2='z',overplot=True,
↪lw=0.6)
```



We can also compare the rotation curves of these different models

```
>>> from galpy.potential import plotRotcurve
>>> plotRotcurve(MWPotential2014,label=r'$\mathrm{MWPotential2014}$',ro=8.,vo=220.) #␣
↪need to set ro and vo explicitly, because MWPotential2014 has units turned off
>>> plotRotcurve(McMillan17,overplot=True,label=r'$\mathrm{McMillan\, (2017)}$')
>>> plotRotcurve(Irrgang13I,overplot=True,label=r'$\mathrm{Irrgang\ et\ al.\, (2017),␣
↪model\ I}$')
>>> legend()
```



An older version `galpy.potential.MWPotential` of `MWPotential2014` that was *not* fit to data on the Milky Way is defined as

```
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,normalize=.6)
>>> np= NFWPotential(a=4.5,normalize=.35)
>>> hp= HernquistPotential(a=0.6/8,normalize=0.05)
>>> MWPotential= mp+np+hp
```

but `galpy.potential.MWPotential2014` supersedes `galpy.potential.MWPotential` and its use is no longer recommended.

2.2.3 2D potentials

General instance routines

Use as `Potential-instance.method(...)`

galpy.potential.planarPotential.__add__

`planarPotential.__add__(b)`

NAME:

__add__

PURPOSE:

Add planarPotential instances together to create a multi-component potential (e.g., pot= pot1+pot2+pot3)

INPUT:

b - planarPotential instance or a list thereof

OUTPUT:

List of planarPotential instances that represents the combined potential

HISTORY:

2019-01-27 - Written - Bovy (UofT)

galpy.potential.planarPotential.__mul__

planarPotential.**__mul__**(*b*)

NAME:

__mul__

PURPOSE:

Multiply a planarPotential's amplitude by a number

INPUT:

b - number

OUTPUT:

New instance with amplitude = (old amplitude) x b

HISTORY:

2019-01-27 - Written - Bovy (UofT)

galpy.potential.planarPotential.__call__

planarPotential.**__call__**(*R*, *phi=0.0*, *t=0.0*, *dR=0*, *dphi=0*)

NAME:

__call__

PURPOSE:

evaluate the potential

INPUT:

R - Cylindrica radius (can be Quantity)

phi= azimuth (optional; can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

Phi(R(,phi,t)))

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.planarPotential.phiforce

planarPotential.**phiforce**(*R*, *phi=0.0*, *t=0.0*)

NAME:

phiforce

PURPOSE:

evaluate the phi force = - d Phi / d phi (note that this is a torque, not a force!)

INPUT:

R - Cylindrical radius (can be Quantity)

phi= azimuth (optional; can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

F_phi(R,(phi,t)))

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.planarPotential.Rforce

planarPotential.**Rforce**(*R*, *phi=0.0*, *t=0.0*)

NAME:

Rforce

PURPOSE:

evaluate the radial force

INPUT:

R - Cylindrical radius (can be Quantity)

phi= azimuth (optional; can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

F_R(R,(phi,t)))

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.planarPotential.turn_physical_off

planarPotential.**turn_physical_off**()

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

galpy.potential.planarPotential.turn_physical_on

planarPotential.**turn_physical_on**(*ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

2020-04-22 - Don't turn on a parameter when it is False - Bovy (UofT)

General axisymmetric potential instance routines

Use as `Potential-instance.method(...)`

galpy.potential.planarAxiPotential.epifreq

Potential.**epifreq**(*R*, *t=0.0*)

NAME:

epifreq

PURPOSE:

calculate the epicycle frequency at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

epicycle frequency

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.planarAxiPotential.lindbladR

Potential.**lindbladR**(*OmegaP*, *m=2*, *t=0.0*, ***kwargs*)

NAME:

lindbladR

PURPOSE:

calculate the radius of a Lindblad resonance

INPUT:

OmegaP - pattern speed (can be Quantity)

m= order of the resonance (as in m(O-Op)=kappa (negative m for outer) use m='corotation' for corotation +scipy.optimize.brentq xtol,rtol,maxiter kwargs

t - time (optional; can be Quantity)

OUTPUT:

radius of Linblad resonance, None if there is no resonance

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.planarAxiPotential.omegac

Potential.**omegac**(*R*, *t=0.0*)

NAME:

omegac

PURPOSE:

calculate the circular angular speed at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

circular angular speed

HISTORY:

2011-10-09 - Written - Bovy (IAS)

galpy.potential.planarAxiPotential.plot

planarAxiPotential.**plot**(**args*, ***kwargs*)

NAME: plot

PURPOSE: plot the potential

INPUT: Rrange - range (can be Quantity) grid - number of points to plot savefilename - save to or restore from this savefile (pickle) +galpy.util.plot.plot(*args,**kwargs)

OUTPUT: plot to output device

HISTORY: 2010-07-13 - Written - Bovy (NYU)

galpy.potential.planarAxiPotential.plotEscapecurve

planarAxiPotential.**plotEscapecurve**(**args*, ***kwargs*)

NAME:

plotEscapecurve

PURPOSE:

plot the escape velocity curve for this potential

INPUT:

Rrange - range (can be Quantity)

grid - number of points to plot

savefilename - save to or restore from this savefile (pickle)

+galpy.util.plot.plot(*args,**kwargs)

OUTPUT:

plot to output device

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.planarAxiPotential.plotRotcurve

planarAxiPotential.**plotRotcurve**(**args*, ***kwargs*)

NAME:

plotRotcurve

PURPOSE:

plot the rotation curve for this potential

INPUT:

Rrange - range (can be Quantity)

grid - number of points to plot

savefilename - save to or restore from this savefile (pickle)

+galpy.util.plot.plot(*args,**kwargs)

OUTPUT:

plot to output device

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.planarAxiPotential.vcirc

Potential.**vcirc**(*R*, *phi=None*, *t=0.0*)

NAME:

vcirc

PURPOSE:

calculate the circular velocity at R in this potential

INPUT:

R - Galactocentric radius (can be Quantity)

phi= (None) azimuth to use for non-axisymmetric potentials

t - time (optional; can be Quantity)

OUTPUT:

circular rotation velocity

HISTORY:

2011-10-09 - Written - Bovy (IAS)

2016-06-15 - Added phi= keyword for non-axisymmetric potential - Bovy (UofT)

galpy.potential.planarAxiPotential.vesc

Potential.**vesc**(*R*, *t=0.0*)

NAME:

vesc

PURPOSE:

calculate the escape velocity at R for this potential

INPUT:

R - Galactocentric radius (can be Quantity)

t - time (optional; can be Quantity)

OUTPUT:

escape velocity

HISTORY:

2011-10-09 - Written - Bovy (IAS)

General 2D potential routines

Use as `method(...)`

galpy.potential.evaluateplanarphiforces

galpy.potential.**evaluateplanarphiforces**(*Pot*, *R*, *phi=None*, *t=0.0*)

NAME:

evaluateplanarphiforces

PURPOSE:

evaluate the phiforce of a (list of) planarPotential instance(s)

INPUT:

Pot - (list of) planarPotential instance(s)

R - Cylindrical radius (can be Quantity)

phi= azimuth (optional; can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

F_phi(R(,phi,t))

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.evaluateplanarPotentials

galpy.potential.**evaluateplanarPotentials**(*Pot*, *R*, *phi=None*, *t=0.0*, *dR=0*, *dphi=0*)

NAME:

evaluateplanarPotentials

PURPOSE:

evaluate a (list of) planarPotential instance(s)

INPUT:

Pot - (list of) planarPotential instance(s)

R - Cylindrical radius (can be Quantity)

phi= azimuth (optional; can be Quantity)

t= time (optional; can be Quantity)

dR=, dphi= if set to non-zero integers, return the dR,dphi't derivative instead

OUTPUT:

Phi(R(,phi,t))

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.evaluateplanarRforces

galpy.potential.**evaluateplanarRforces**(*Pot*, *R*, *phi=None*, *t=0.0*)

NAME:

evaluateplanarRforces

PURPOSE:

evaluate the Rforce of a (list of) planarPotential instance(s)

INPUT:

Pot - (list of) planarPotential instance(s)

R - Cylindrical radius (can be Quantity)

phi= azimuth (optional can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

F_R(R(,phi,t))

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.evaluateplanarR2derivs

galpy.potential.**evaluateplanarR2derivs**(*Pot*, *R*, *phi=None*, *t=0.0*)

NAME:

evaluateplanarR2derivs

PURPOSE:

evaluate the second radial derivative of a (list of) planarPotential instance(s)

INPUT:

Pot - (list of) planarPotential instance(s)

R - Cylindrical radius (can be Quantity)

phi= azimuth (optional; can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

F_R(R(,phi,t))

HISTORY:

2010-10-09 - Written - Bovy (IAS)

galpy.potential.LinShuReductionFactor

galpy.potential.**LinShuReductionFactor**(*axiPot*, *R*, *sigmar*, *nonaxiPot=None*, *k=None*, *m=None*, *OmegaP=None*)

NAME:

LinShuReductionFactor

PURPOSE:

Calculate the Lin & Shu (1966) reduction factor: the reduced linear response of a kinematically-warm stellar disk to a perturbation

INPUT:

axiPot - The background, axisymmetric potential

R - Cylindrical radius (can be Quantity)

sigmar - radial velocity dispersion of the population (can be Quantity)

Then either provide:

1) m= m in the perturbation's m x phi (number of arms for a spiral)

   k= wavenumber (see Binney & Tremaine 2008)

   OmegaP= pattern speed (can be Quantity)

2) nonaxiPot= a non-axisymmetric Potential instance (such as SteadyLogSpiralPotential) that has functions that return OmegaP, m, and wavenumber

OUTPUT:

reduction factor

HISTORY:

2014-08-23 - Written - Bovy (IAS)

galpy.potential.plotplanarPotentials

galpy.potential.**plotplanarPotentials**(*Pot*, **args*, ***kwargs*)

NAME:

plotplanarPotentials

PURPOSE:

plot a planar potential

INPUT:

Rrange - range (can be Quantity)

xrange, yrange - if relevant (can be Quantity)

grid, gridx, gridy - number of points to plot

savefilename - save to or restore from this savefile (pickle)

ncontours - number of contours to plot (if applicable)

+galpy.util.plot.plot(*args,**kwargs) or galpy.util.plot.dens2d(**kwargs)

OUTPUT:

plot to output device

HISTORY:

2010-07-13 - Written - Bovy (NYU)

Specific potentials

All of the 3D potentials above can be used as two-dimensional potentials in the mid-plane.

galpy.potential.toPlanarPotential

galpy.potential.**toPlanarPotential**(*Pot*)

NAME:

toPlanarPotential

PURPOSE:

convert an Potential to a planarPotential in the mid-plane (z=0)

INPUT:

Pot - Potential instance or list of such instances (existing planarPotential instances are just copied to the output)

OUTPUT:

planarPotential instance(s)

HISTORY:

2016-06-11 - Written - Bovy (UofT)

galpy.potential.RZToplanarPotential

galpy.potential.**RZToplanarPotential**(*RZPot*)

NAME:

RZToplanarPotential

PURPOSE:

convert an RZPotential to a planarPotential in the mid-plane (z=0)

INPUT:

RZPot - RZPotential instance or list of such instances (existing planarPotential instances are just copied to the output)

OUTPUT:

planarPotential instance(s)

HISTORY:

2010-07-13 - Written - Bovy (NYU)

In addition, a two-dimensional bar potential, two spiral potentials, the Henon & Heiles (1964) potential, and some static non-axisymmetric perturbations are included

Cos(m phi) disk potential

Generalization of the *lopsided* and *elliptical* disk potentials to any m and to allow for a break radius within which the radial dependence of the potential changes from R^p to R^-p.

class galpy.potential.**CosmphiDiskPotential**(*amp=1.0*, *phib=0.4363323129985824*, *p=1.0*, *phio=0.01*, *m=4*, *r1=1.0*, *rb=None*, *cp=None*, *sp=None*, *ro=None*, *vo=None*)

Class that implements the disk potential

$$\Phi(R,\phi) = \mathrm{amp}\,\phi_0\ \cos\left[m\left(\phi-\phi_b\right)\right] \times \begin{cases} \left(\frac{R}{R_1}\right)^p, & \text{for } R \geq R_b \\ \left[2-\left(\frac{R_b}{R}\right)^p\right] \times \left(\frac{R_b}{R_1}\right)^p, & \text{for } R \leq R_b. \end{cases}$$

This potential can be grown between t_{form} and $t_{\mathrm{form}} + T_{\mathrm{steady}}$ in a similar way as DehnenBarPotential by wrapping it with a DehnenSmoothWrapperPotential

__init__(*amp=1.0*, *phib=0.4363323129985824*, *p=1.0*, *phio=0.01*, *m=4*, *r1=1.0*, *rb=None*, *cp=None*, *sp=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize an cosmphi disk potential

INPUT:

amp= amplitude to be applied to the potential (default: 1.), degenerate with phio below, but kept for overall consistency with potentials

m= cos(m * (phi - phib)), integer

p= power-law index of the phi(R) = (R/Ro)^p part

r1= (1.) normalization radius for the amplitude (can be Quantity); amp x phio is only the potential at (R,phi) = (r1,pib) when r1 > rb; otherwise more complicated

rb= (None) if set, break radius for power-law: potential R^p at R > Rb, R^-p at R < Rb, potential and force continuous at Rb

Either:

a) phib= angle (in rad; default=25 degree; or can be Quantity)

   phio= potential perturbation (in terms of phio/vo^2 if vo=1 at Ro=1; or can be Quantity with units of velocity-squared)

b) cp, sp= m * phio * cos(m * phib), m * phio * sin(m * phib); can be Quantity with units of velocity-squared)

OUTPUT:

(none)

HISTORY:

2011-10-27 - Started - Bovy (IAS)

2017-09-16 - Added break radius rb - Bovy (UofT)

Elliptical disk potential

Like in Kuijken & Tremaine. See *galpy.potential.CosmphiDiskPotential* for a more general version that allows for a break radius within which the radial dependence of the potential changes from R^p to R^-p (elliptical disk corresponds to `m=2`).

class galpy.potential.**EllipticalDiskPotential**(*amp=1.0*, *phib=0.4363323129985824*, *p=1.0*, *twophio=0.01*, *r1=1.0*, *tform=None*, *tsteady=None*, *cp=None*, *sp=None*, *ro=None*, *vo=None*)

Class that implements the Elliptical disk potential of Kuijken & Tremaine (1994)

$$\Phi(R,\phi) = \mathrm{amp}\,\phi_0 \left(\frac{R}{R_1}\right)^p \cos\left(2\left(\phi-\phi_b\right)\right)$$

This potential can be grown between t_{form} and $t_{\mathrm{form}} + T_{\mathrm{steady}}$ in a similar way as DehnenBarPotential, but times are given directly in galpy time units

__init__(*amp=1.0*, *phib=0.4363323129985824*, *p=1.0*, *twophio=0.01*, *r1=1.0*, *tform=None*, *tsteady=None*, *cp=None*, *sp=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize an Elliptical disk potential

phi(R,phi) = phio (R/Ro)^p cos[2(phi-phib)]

INPUT:

amp= amplitude to be applied to the potential (default: 1.), see twophio below

tform= start of growth (to smoothly grow this potential (can be Quantity)

tsteady= time delay at which the perturbation is fully grown (default: 2.; can be Quantity)

p= power-law index of the phi(R) = (R/Ro)^p part

r1= (1.) normalization radius for the amplitude (can be Quantity)

Either:

a) phib= angle (in rad; default=25 degree; or can be Quantity)

   twophio= potential perturbation (in terms of 2phio/vo^2 if vo=1 at Ro=1; can be Quantity with units of velocity-squared)

b) cp, sp= twophio * cos(2phib), twophio * sin(2phib) (can be Quantity with units of velocity-squared)

OUTPUT:

(none)

HISTORY:

2011-10-19 - Started - Bovy (IAS)

Henon-Heiles potential

class galpy.potential.**HenonHeilesPotential**(*amp=1.0*, *ro=None*, *vo=None*)

Class that implements a the Henon & Heiles (1964) potential

$$\Phi(R,\phi) = \frac{\mathrm{amp}}{2}\left[R^2 + \frac{2\,R^3}{3}\,\sin\left(3\,\phi\right)\right]$$

__init__(*amp=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a Henon-Heiles potential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

OUTPUT:

(none)

HISTORY:

2017-10-16 - Written - Bovy (UofT)

Lopsided disk potential

Like in Kuijken & Tremaine, but for m=1. See *galpy.potential.CosmphiDiskPotential* for a more general version that allows for a break radius within which the radial dependence of the potential changes from R^p to R^-p (lopsided disk corresponds to `m=1`).

class galpy.potential.**LopsidedDiskPotential**(*amp=1.0*, *phib=0.4363323129985824*, *p=1.0*, *phio=0.01*, *r1=1.0*, *cp=None*, *sp=None*, *ro=None*, *vo=None*)

Class that implements the disk potential

$$\Phi(R,\phi) = \mathrm{amp}\,\phi_0 \left(\frac{R}{R_1}\right)^p \cos\left(\phi - \phi_b\right)$$

Special case of CosmphiDiskPotential with m=1; see documentation for CosmphiDiskPotential

__init__(*amp=1.0*, *phib=0.4363323129985824*, *p=1.0*, *phio=0.01*, *r1=1.0*, *cp=None*, *sp=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize an cosmphi disk potential

INPUT:

amp= amplitude to be applied to the potential (default: 1.), degenerate with phio below, but kept for overall consistency with potentials

m= cos(m * (phi - phib)), integer

p= power-law index of the phi(R) = (R/Ro)^p part

r1= (1.) normalization radius for the amplitude (can be Quantity); amp x phio is only the potential at (R,phi) = (r1,pib) when r1 > rb; otherwise more complicated

rb= (None) if set, break radius for power-law: potential R^p at R > Rb, R^-p at R < Rb, potential and force continuous at Rb

Either:

a) phib= angle (in rad; default=25 degree; or can be Quantity)

phio= potential perturbation (in terms of phio/vo^2 if vo=1 at Ro=1; or can be Quantity with units of velocity-squared)

b) cp, sp= m * phio * cos(m * phib), m * phio * sin(m * phib); can be Quantity with units of velocity-squared)

OUTPUT:

(none)

HISTORY:

2011-10-27 - Started - Bovy (IAS)

2017-09-16 - Added break radius rb - Bovy (UofT)

Steady-state logarithmic spiral potential

class galpy.potential.**SteadyLogSpiralPotential**(*amp=1.0*, *omegas=0.65*, *A=-0.035*, *alpha=-7.0*, *m=2*, *gamma=0.7853981633974483*, *p=None*, *tform=None*, *tsteady=None*, *ro=None*, *vo=None*)

Class that implements a steady-state spiral potential

$$\Phi(R,\phi) = \frac{\mathrm{amp} \times A}{\alpha}\cos\left(\alpha\,\ln R - m\,(\phi - \Omega_s\,t - \gamma)\right)$$

Can be grown in a similar way as the DehnenBarPotential, but using $T_s = 2\pi/\Omega_s$ to normalize t_{form} and T_{steady}.

__init__(*amp=1.0*, *omegas=0.65*, *A=-0.035*, *alpha=-7.0*, *m=2*, *gamma=0.7853981633974483*, *p=None*, *tform=None*, *tsteady=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a logarithmic spiral potential

INPUT:

amp - amplitude to be applied to the potential (default: 1., A below)

gamma - angle between sun-GC line and the line connecting the peak of the spiral pattern at the Solar radius (in rad; default=45 degree; or can be Quantity)

A - amplitude (alpha*potential-amplitude; default=0.035; can be Quantity

omegas= - pattern speed (default=0.65; can be Quantity)

m= number of arms

Either provide:

a) alpha=

b) p= pitch angle (rad; can be Quantity)

tform - start of spiral growth / spiral period (default: -Infinity)

tsteady - time from tform at which the spiral is fully grown / spiral period (default: 2 periods)

OUTPUT:

(none)

HISTORY:

2011-03-27 - Started - Bovy (NYU)

Transient logarithmic spiral potential

class galpy.potential.**TransientLogSpiralPotential**(*amp=1.0*, *omegas=0.65*, *A=-0.035*, *alpha=-7.0*, *m=2*, *gamma=0.7853981633974483*, *p=None*, *sigma=1.0*, *to=0.0*, *ro=None*, *vo=None*)

Class that implements a steady-state spiral potential

$$\Phi(R,\phi) = \frac{\mathrm{amp}(t)}{\alpha}\,\cos\left(\alpha\,\ln R - m\,(\phi - \Omega_s\,t - \gamma)\right)$$

where

$$\mathrm{amp}(t) = \mathrm{amp}\,\times A\,\exp\left(-\frac{[t-t_0]^2}{2\,\sigma^2}\right)$$

__init__(*amp=1.0*, *omegas=0.65*, *A=-0.035*, *alpha=-7.0*, *m=2*, *gamma=0.7853981633974483*, *p=None*, *sigma=1.0*, *to=0.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a transient logarithmic spiral potential localized around to

INPUT:

amp - amplitude to be applied to the potential (default: 1., A below)

gamma - angle between sun-GC line and the line connecting the peak of the spiral pattern at the Solar radius (in rad; default=45 degree; can be Quantity)

A - amplitude (alpha*potential-amplitude; default=0.035; can be Quantity)

omegas= - pattern speed (default=0.65; can be Quantity)

m= number of arms

to= time at which the spiral peaks (can be Quantity)

sigma= "spiral duration" (sigma in Gaussian amplitude; can be Quantity)

Either provide:

a) alpha=

b) p= pitch angle (rad; can be Quantity)

OUTPUT:

(none)

HISTORY:

2011-03-27 - Started - Bovy (NYU)

2.2.4 1D potentials

General instance routines

Use as `Potential-instance.method(...)`

galpy.potential.linearPotential.__add__

`linearPotential.__add__(b)`

NAME:

__add__

PURPOSE:

Add linearPotential instances together to create a multi-component potential (e.g., pot= pot1+pot2+pot3)

INPUT:

b - linearPotential instance or a list thereof

OUTPUT:

List of linearPotential instances that represents the combined potential

HISTORY:

2019-01-27 - Written - Bovy (UofT)

galpy.potential.linearPotential.__mul__

`linearPotential.__mul__(b)`

NAME:

__mul__

PURPOSE:

Multiply a linearPotential's amplitude by a number

INPUT:

b - number

OUTPUT:

New instance with amplitude = (old amplitude) x b

HISTORY:

2019-01-27 - Written - Bovy (UofT)

galpy.potential.linearPotential.__call__

`linearPotential.__call__(x, t=0.0)`

NAME: __call__

PURPOSE:

evaluate the potential

INPUT:

x - position (can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

Phi(x,t)

HISTORY:

2010-07-12 - Written - Bovy (NYU)

galpy.potential.linearPotential.force

linearPotential.**force**(*x*, *t=0.0*)

NAME:

force

PURPOSE:

evaluate the force

INPUT:

x - position (can be Quantity)

t= time (optional; can be Quantity)

OUTPUT:

F(x,t)

HISTORY:

2010-07-12 - Written - Bovy (NYU)

galpy.potential.linearPotential.plot

linearPotential.**plot**(*t=0.0*, *min=-15.0*, *max=15*, *ns=21*, *savefilename=None*)

NAME:

plot

PURPOSE:

plot the potential

INPUT:

t - time to evaluate the potential at

min - minimum x

max - maximum x

ns - grid in x

savefilename - save to or restore from this savefile (pickle)

OUTPUT:

plot to output device

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.linearPotential.turn_physical_off

linearPotential.**turn_physical_off**()

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

galpy.potential.linearPotential.turn_physical_on

linearPotential.**turn_physical_on**(*ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2016-01-30 - Written - Bovy (UofT)

2020-04-22 - Don't turn on a parameter when it is False - Bovy (UofT)

General 1D potential routines

Use as `method(...)`

galpy.potential.evaluatelinearForces

galpy.potential.**evaluatelinearForces**(*Pot*, *x*, *t=0.0*)

NAME:

evaluatelinearForces

PURPOSE:

evaluate the forces due to a list of potentials

INPUT:

Pot - (list of) linearPotential instance(s)

x - evaluate forces at this position (can be Quantity)

t - time to evaluate at (can be Quantity)

OUTPUT:

force(x,t)

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.evaluatelinearPotentials

galpy.potential.**evaluatelinearPotentials**(*Pot*, *x*, *t=0.0*)

NAME:

evaluatelinearPotentials

PURPOSE:

evaluate the sum of a list of potentials

INPUT:

Pot - (list of) linearPotential instance(s)

x - evaluate potentials at this position (can be Quantity)

t - time to evaluate at (can be Quantity)

OUTPUT:

pot(x,t)

HISTORY:

2010-07-13 - Written - Bovy (NYU)

galpy.potential.plotlinearPotentials

galpy.potential.**plotlinearPotentials**(*Pot*, *t=0.0*, *min=-15.0*, *max=15*, *ns=21*, *savefilename=None*)

NAME:

plotlinearPotentials

PURPOSE:

plot a combination of potentials

INPUT:

t - time to evaluate potential at

min - minimum x

max - maximum x

ns - grid in x

savefilename - save to or restore from this savefile (pickle)

OUTPUT:

plot to output device

HISTORY:

2010-07-13 - Written - Bovy (NYU)

Specific potentials

Isothermal disk potential

class galpy.potential.**IsothermalDiskPotential**(*amp=1.0*, *sigma=0.1*, *ro=None*, *vo=None*)

Class representing the one-dimensional self-gravitating isothermal disk

$$\rho(x) = \mathrm{amp}\,\mathrm{sech}^2\left(\frac{x}{2H}\right)$$

where the scale height $H^2 = \sigma^2/[8\pi G\,\mathrm{amp}]$. The parameter to setup the disk is the velocity dispersion σ.

__init__(*amp=1.0*, *sigma=0.1*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize an IsothermalDiskPotential

INPUT:

amp - an overall amplitude

sigma - velocity dispersion (can be a Quantity)

OUTPUT:

instance

HISTORY:

2018-04-11 - Written - Bovy (UofT)

Vertical Kuijken & Gilmore potential

class galpy.potential.**KGPotential**(*K=1.15*, *F=0.03*, *D=1.8*, *amp=1.0*, *ro=None*, *vo=None*)

Class representing the Kuijken & Gilmore (1989) potential

$$\Phi(x) = \mathrm{amp}\,\left(K\,\left(\sqrt{x^2+D^2}-D\right)+F\,x^2\right)$$

__init__(*K=1.15*, *F=0.03*, *D=1.8*, *amp=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a KGPotential

INPUT:

K= K parameter (= $2\pi\Sigma_{\rm disk}$; specify either as surface density or directly as force [i.e., including $2\pi G$]; can be Quantity)

F= F parameter (= $4\pi\rho_{\rm halo}$; specify either as density or directly as second potential derivative [i.e., including $4\pi G$]; can be Quantity)

D= D parameter (natural units or Quantity length units)

amp - an overall amplitude

OUTPUT:

instance

HISTORY:

2010-07-12 - Written - Bovy (NYU)

One-dimensional potentials can also be derived from 3D axisymmetric potentials as the vertical potential at a certain Galactocentric radius

galpy.potential.toVerticalPotential

galpy.potential.**toVerticalPotential**(*Pot*, *R*, *phi=None*, *t0=0.0*)

NAME:

toVerticalPotential

PURPOSE:

convert a Potential to a vertical potential at a given R: Phi(z,phi,t) = Phi(R,z,phi,t)-Phi(R,0.,phi0,t0) where phi0 and t0 are the phi and t inputs

INPUT:

Pot - Potential instance or list of such instances

R - Galactocentric radius at which to evaluate the vertical potential (can be Quantity)

phi= (None) Galactocentric azimuth at which to evaluate the vertical potential (can be Quantity); required if Pot is non-axisymmetric

t0= (0.) time at which to evaluate the vertical potential (can be Quantity)

OUTPUT:

(list of) linearPotential instance(s)

HISTORY:

2018-10-07 - Written - Bovy (UofT)

galpy.potential.RZToverticalPotential

galpy.potential.**RZToverticalPotential**(*RZPot*, *R*)

NAME:

RZToverticalPotential

PURPOSE:

convert a RZPotential to a vertical potential at a given R

INPUT:

RZPot - RZPotential instance or list of such instances

R - Galactocentric radius at which to evaluate the vertical potential (can be Quantity)

OUTPUT:

(list of) linearPotential instance(s)

HISTORY:

2010-07-21 - Written - Bovy (NYU)

2.2.5 Potential wrappers

Gravitational potentials in `galpy` can also be modified using wrappers, for example, to change their amplitude as a function of time. These wrappers can be applied to *any* `galpy` potential (although whether they can be used in C depends on whether the wrapper *and* all of the potentials that it wraps are implemented in C). Multiple wrappers can be applied to the same potential.

Specific wrappers

Adiabatic contraction wrapper potential

class galpy.potential.**AdiabaticContractionWrapperPotential**(*amp=1.0*, *pot=None*, *baryonpot=None*, *method='cautun'*, *f_bar=0.157*, *rmin=None*, *rmax=50.0*, *ro=None*, *vo=None*)

AdiabaticContractionWrapperPotential: Wrapper to adiabatically contract a DM halo in response to the growth of a baryonic component. Use for example as:

```
dm= AdiabaticContractionWrapperPotential(pot=MWPotential2014[2],
↪baryonpot=MWPotential2014[:2])
```

to contract the dark-matter halo in MWPotential2014 according to the baryon distribution within it. The basic physics of the adiabatic contraction is that a fraction *f_bar* of the mass in the original potential *pot* cools adiabatically to form a baryonic component *baryonpot*; this wrapper computes the resulting dark-matter potential using different approximations in the literature.

__init__(*amp=1.0*, *pot=None*, *baryonpot=None*, *method='cautun'*, *f_bar=0.157*, *rmin=None*, *rmax=50.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a AdiabaticContractionWrapper Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

pot - Potential instance or list thereof representing the density that is adiabatically contracted

baryonpot - Potential instance or list thereof representing the density of baryons whose growth causes the contraction

method= ('cautun') Type of adiabatic-contraction formula:

- 'cautun' for that from Cautun et al. 2020 (2020MNRAS.494.4291C),
- 'blumenthal' for that from Blumenthal et al. 1986 (1986ApJ...301...27B 1986ApJ...301...27B)
- 'gnedin' for that from Gnedin et al. 2004 (2004ApJ...616...16G)

f_bar= (0.157) universal baryon fraction; if None, calculated from pot and baryonpot assuming that at rmax the halo contains the universal baryon fraction; leave this at the default value unless you know what you are doing

rmin= (None) minimum radius to consider (default: rmax/2500; don't set this to zero)

rmax= (50.) maximum radius to consider (can be Quantity)

ro, vo= standard unit-conversion parameters

OUTPUT:

(none)

HISTORY:

2021-03-21 - Started based on Marius Cautun's code - Bovy (UofT)

Any time-dependent amplitude wrapper potential

class galpy.potential.**TimeDependentAmplitudeWrapperPotential**(*amp=1.0*, *A=None*, *pot=None*, *ro=None*, *vo=None*)

Potential wrapper class that allows the amplitude of any potential to be any function of time. That is, the amplitude of a potential gets modulated to

$$\text{amp} \to \text{amp} \times A(t)$$

where $A(t)$ is an arbitrary function of time. Note that *amp* itself can already be a function of time.

__init__(*amp=1.0*, *A=None*, *pot=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a TimeDependentAmplitudeWrapperPotential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

A - function of time giving the time-dependence of the amplitude; should be able to be called with a single time and return a numbers.Number (that is, a number); input time is in internal units (see galpy.util.conversion.time_in_Gyr to convert) and output is a dimensionless amplitude modulation

pot - Potential instance or list thereof; the amplitude of this will modified by this wrapper

OUTPUT:

(none)

HISTORY:

2022-03-29 - Started - Bovy (UofT)

Corotating rotation wrapper potential

class galpy.potential.**CorotatingRotationWrapperPotential**(*amp=1.0*, *pot=None*, *vpo=1.0*, *beta=0.0*, *to=0.0*, *pa=0.0*, *ro=None*, *vo=None*)

Potential wrapper class that implements rotation with fixed R x pattern-speed around the z-axis. Can be used to make spiral structure that is everywhere co-rotating. The potential is rotated by replacing

$$\phi \to \phi + \frac{V_p(R)}{R} \times (t - t_0) + \mathrm{pa}$$

with $V_p(R)$ the circular velocity curve, t_0 a reference time—time at which the potential is unchanged by the wrapper—and pa the position angle at $t = 0$. The circular velocity is parameterized as

$$V_p(R) = V_{p,0} \left(\frac{R}{R_0}\right)^{\beta} .$$

__init__(*amp=1.0*, *pot=None*, *vpo=1.0*, *beta=0.0*, *to=0.0*, *pa=0.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a CorotatingRotationWrapper Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

pot - Potential instance or list thereof; this potential is made to rotate around the z axis by the wrapper

vpo= (1.) amplitude of the circular-velocity curve (can be a Quantity)

beta= (0.) power-law amplitude of the circular-velocity curve

to= (0.) reference time at which the potential == pot

pa= (0.) the position angle (can be a Quantity)

OUTPUT:

(none)

HISTORY:

2018-02-21 - Started - Bovy (UofT)

Dehnen-like smoothing wrapper potential

class galpy.potential.**DehnenSmoothWrapperPotential**(*amp=1.0*, *pot=None*, *tform=-4.0*, *tsteady=None*, *decay=False*, *ro=None*, *vo=None*)

Potential wrapper class that implements the growth of a gravitational potential following Dehnen (2000). The amplitude A applied to a potential wrapped by an instance of this class is changed as

$$A(t) = amp\left(\frac{3}{16}\xi^5 - \frac{5}{8}\xi^3 + \frac{15}{16}\xi + \frac{1}{2}\right)$$

where

$$\xi = \begin{cases} -1 & t < t_{\rm form} \\ 2\left(\frac{t-t_{\rm form}}{t_{\rm steady}}\right) - 1\,, & t_{\rm form} \leq t \leq t_{\rm form} + t_{\rm steady} \\ 1 & t > t_{\rm form} + t_{\rm steady} \end{cases}$$

if `decay=True`, the amplitude decays rather than grows as decay = 1 - grow.

__init__(*amp=1.0*, *pot=None*, *tform=-4.0*, *tsteady=None*, *decay=False*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a DehnenSmoothWrapper Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

pot - Potential instance or list thereof; the amplitude of this will be grown by this wrapper

tform - start of growth (can be a Quantity)

tsteady - time from tform at which the potential is fully grown (default: -tform/2, st the perturbation is fully grown at tform/2; can be a Quantity)

decay= (False) if True, decay the amplitude instead of growing it (as 1-grow)

OUTPUT:

(none)

HISTORY:

2017-06-26 - Started - Bovy (UofT)

2018-10-07 - Added 'decay' option - Bovy (UofT)

Gaussian-modulated amplitude wrapper potential

class galpy.potential.**GaussianAmplitudeWrapperPotential**(*amp=1.0*, *pot=None*, *to=0.0*, *sigma=1.0*, *ro=None*, *vo=None*)

Potential wrapper class that allows the amplitude of a Potential object to be modulated as a Gaussian. The

amplitude A applied to a potential wrapped by an instance of this class is changed as

$$A(t) = amp \exp\left(-\frac{[t-t_0]^2}{2\,\sigma^2}\right)$$

__init__(*amp=1.0*, *pot=None*, *to=0.0*, *sigma=1.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a GaussianAmplitudeWrapper Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

pot - Potential instance or list thereof; this potential is made to rotate around the z axis by the wrapper

to= (0.) time at which the Gaussian peaks

sigma= (1.) standard deviation of the Gaussian (can be a Quantity)

OUTPUT:

(none)

HISTORY:

2018-02-21 - Started - Bovy (UofT)

Solid-body rotation wrapper potential

class galpy.potential.**SolidBodyRotationWrapperPotential**(*amp=1.0*, *pot=None*, *omega=1.0*, *pa=0.0*, *ro=None*, *vo=None*)

Potential wrapper class that implements solid-body rotation around the z-axis. Can be used to make a bar or other perturbation rotate. The potential is rotated by replacing

$$\phi \to \phi + \Omega \times t + \mathrm{pa}$$

with Ω the fixed pattern speed and pa the position angle at $t = 0$.

__init__(*amp=1.0*, *pot=None*, *omega=1.0*, *pa=0.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a SolidBodyRotationWrapper Potential

INPUT:

amp - amplitude to be applied to the potential (default: 1.)

pot - Potential instance or list thereof; this potential is made to rotate around the z axis by the wrapper

omega= (1.) the pattern speed (can be a Quantity)

pa= (0.) the position angle (can be a Quantity)

OUTPUT:

(none)

HISTORY:

2017-08-22 - Started - Bovy (UofT)

Rotate-and-tilt wrapper potential

class galpy.potential.**RotateAndTiltWrapperPotential**(*amp=1.0*, *inclination=None*, *galaxy_pa=None*, *sky_pa=None*, *zvec=None*, *offset=None*, *pot=None*, *ro=None*, *vo=None*)

Potential wrapper that allows a potential to be rotated in 3D according to three orientation angles. These angles can either be specified using:

- A rotation around the original z-axis (*galaxy_pa*) and the new direction of the z-axis (*zvec*) or
- A rotation around the original z-axis (*galaxy_pa*), the *inclination*, and a rotation around the new z axis (*sky_pa*).

The second option allows one to specify the inclination and sky position angle (measured from North) in the usual manner in extragalactic observations. A final *offset* option allows one to apply a static offset in Cartesian coordinate space to be applied to the potential following the rotation and tilt.

__init__(*amp=1.0*, *inclination=None*, *galaxy_pa=None*, *sky_pa=None*, *zvec=None*, *offset=None*, *pot=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

initialize a RotateAndTiltWrapper Potential

INPUT:

amp= (1.) overall amplitude to apply to the potential

pot= Potential instance or list thereof for the potential to rotate and tilt

Orientation angles as

galaxy_pa= rotation angle of the original potential around the original z axis (can be a Quantity)

and either

1) zvec= 3D vector specifying the direction of the rotated z axis

2) inclination= usual inclination angle (with the line-of-sight being the z axis)

sky_pa= rotation angle around the inclined z axis (usual sky position angle measured from North)

offset= optional static offset in Cartesian coordinates (can be a Quantity)

OUTPUT:

(none)

HISTORY:

2021-03-29 - Started - Mackereth (UofT)

2021-04-18 - Added inclination, sky_pa, galaxy_pa setup - Bovy (UofT)

2022-03-14 - added offset kwarg - Mackereth (UofT)

2.3 actionAngle (`galpy.actionAngle`)

2.3.1 (x, v) –> (J, O, a)

General instance routines

Not necessarily supported for all different types of actionAngle calculations. These have extra arguments for different `actionAngle` modules, so check the documentation of the module-specific functions for more info (e.g., `?actionAngleIsochrone.__call__`)

galpy.actionAngle.actionAngle.__call__

`actionAngle.__call__`(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

evaluate the actions (jr,lz,jz)

INPUT:

Either:

a) R,vR,vT,z,vz[,phi]:

1) floats: phase-space value for single object (phi is optional) (each can be a Quantity)

2) numpy.ndarray: [N] phase-space values for N objects (each can be a Quantity)

b) Orbit instance: initial condition used if that's it, orbit(t) if there is a time given as well as the second argument

OUTPUT:

(jr,lz,jz)

HISTORY:

2014-01-03 - Written for top level - Bovy (IAS)

galpy.actionAngle.actionAngle.actionsFreqs

`actionAngle.actionsFreqs`(**args*, ***kwargs*)

NAME:

actionsFreqs

PURPOSE:

evaluate the actions and frequencies (jr,lz,jz,Omegar,Omegaphi,Omegaz)

INPUT:

Either:

a) R,vR,vT,z,vz[,phi]:

1) floats: phase-space value for single object (phi is optional) (each can be a Quantity)

2) numpy.ndarray: [N] phase-space values for N objects (each can be a Quantity)

b) Orbit instance: initial condition used if that's it, orbit(t) if there is a time given as well as the second argument

OUTPUT:

(jr,lz,jz,Omegar,Omegaphi,Omegaz)

HISTORY:

2014-01-03 - Written for top level - Bovy (IAS)

galpy.actionAngle.actionAngle.actionsFreqsAngles

actionAngle.**actionsFreqsAngles**(**args*, ***kwargs*)

NAME:

actionsFreqsAngles

PURPOSE:

evaluate the actions, frequencies, and angles (jr,lz,jz,Omegar,Omegaphi,Omegaz,angler,anglephi,anglez)

INPUT:

Either:

a) R,vR,vT,z,vz,phi:

1) floats: phase-space value for single object (phi is optional) (each can be a Quantity)

2) numpy.ndarray: [N] phase-space values for N objects (each can be a Quantity)

b) Orbit instance: initial condition used if that's it, orbit(t) if there is a time given as well as the second argument

OUTPUT:

(jr,lz,jz,Omegar,Omegaphi,Omegaz,angler,anglephi,anglez)

HISTORY:

2014-01-03 - Written for top level - Bovy (IAS)

galpy.actionAngle.actionAngle.EccZmaxRperiRap

actionAngle.**EccZmaxRperiRap**(**args*, ***kwargs*)

NAME:

EccZmaxRperiRap

PURPOSE:

evaluate the eccentricity, maximum height above the plane, peri- and apocenter

INPUT:

Either:

a) R,vR,vT,z,vz[,phi]:

1) floats: phase-space value for single object (phi is optional) (each can be a Quantity)

2) numpy.ndarray: [N] phase-space values for N objects (each can be a Quantity)

b) Orbit instance: initial condition used if that's it, orbit(t) if there is a time given as well as the second argument

OUTPUT:

(e,zmax,rperi,rap)

HISTORY:

2017-12-12 - Written - Bovy (UofT)

galpy.actionAngle.actionAngle.turn_physical_off

actionAngle.**turn_physical_off**()

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2017-06-05 - Written - Bovy (UofT)

galpy.actionAngle.actionAngle.turn_physical_on

actionAngle.**turn_physical_on**(*ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2016-06-05 - Written - Bovy (UofT)

2020-04-22 - Don't turn on a parameter when it is False - Bovy (UofT)

Specific actionAngle modules

actionAngleHarmonic

class galpy.actionAngle.**actionAngleHarmonic**(**args*, ***kwargs*)

Action-angle formalism for the one-dimensional harmonic oscillator

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

initialize an actionAngleHarmonic object

INPUT:

omega= frequencies (can be Quantity)

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2018-04-08 - Written - Bovy (Uoft)

actionAngleIsochrone

class galpy.actionAngle.**actionAngleIsochrone**(**args*, ***kwargs*)

Action-angle formalism for the isochrone potential, on the Jphi, Jtheta system of Binney & Tremaine (2008)

__init__(**args*, ***kwargs*)

NAME: __init__

PURPOSE: initialize an actionAngleIsochrone object

INPUT: Either:

b= scale parameter of the isochrone parameter (can be Quantity)

ip= instance of a IsochronePotential

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY: 2013-09-08 - Written - Bovy (IAS)

actionAngleSpherical

class galpy.actionAngle.**actionAngleSpherical**(**args*, ***kwargs*)

Action-angle formalism for spherical potentials

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

initialize an actionAngleSpherical object

INPUT:

pot= a Spherical potential

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2013-12-28 - Written - Bovy (IAS)

actionAngleAdiabatic

class galpy.actionAngle.**actionAngleAdiabatic**(**args*, ***kwargs*)

Action-angle formalism for axisymmetric potentials using the adiabatic approximation

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

initialize an actionAngleAdiabatic object

INPUT:

pot= potential or list of potentials (planarPotentials)

gamma= (default=1.) replace Lz by Lz+gamma Jz in effective potential

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2012-07-26 - Written - Bovy (IAS@MPIA)

actionAngleAdiabaticGrid

class galpy.actionAngle.**actionAngleAdiabaticGrid**(*pot=None*, *zmax=1.0*, *gamma=1.0*, *Rmax=5.0*, *nR=16*, *nEz=16*, *nEr=31*, *nLz=31*, *numcores=1*, ***kwargs*)

Action-angle formalism for axisymmetric potentials using the adiabatic approximation, grid-based interpolation

__init__(*pot=None*, *zmax=1.0*, *gamma=1.0*, *Rmax=5.0*, *nR=16*, *nEz=16*, *nEr=31*, *nLz=31*, *numcores=1*, ***kwargs*)

NAME: __init__

PURPOSE: initialize an actionAngleAdiabaticGrid object

INPUT:

pot= potential or list of potentials

zmax= zmax for building Ez grid

Rmax = Rmax for building grids

gamma= (default=1.) replace Lz by Lz+gamma Jz in effective potential

nEz=, nEr=, nLz, nR= grid size

numcores= number of cpus to use to parallellize

c= if True, use C to calculate actions

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

+scipy.integrate.quad keywords

OUTPUT:

instance

HISTORY:

2012-07-27 - Written - Bovy (IAS@MPIA)

actionAngleStaeckel

class galpy.actionAngle.**actionAngleStaeckel**(**args*, ***kwargs*)

Action-angle formalism for axisymmetric potentials using Binney (2012)'s Staeckel approximation

__init__(**args*, ***kwargs*)

NAME: __init__

PURPOSE: initialize an actionAngleStaeckel object

INPUT: pot= potential or list of potentials (3D)

delta= focus (can be Quantity)

useu0 - use u0 to calculate dV (NOT recommended)

c= if True, always use C for calculations

order= (10) number of points to use in the Gauss-Legendre numerical integration of the relevant action, frequency, and angle integrals

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2012-11-27 - Written - Bovy (IAS)

actionAngleStaeckelGrid

class galpy.actionAngle.**actionAngleStaeckelGrid**(*pot=None*, *delta=None*, *Rmax=5.0*, *nE=25*, *npsi=25*, *nLz=30*, *numcores=1*, *interpecc=False*, ***kwargs*)

Action-angle formalism for axisymmetric potentials using Binney (2012)'s Staeckel approximation, grid-based interpolation

__init__(*pot=None*, *delta=None*, *Rmax=5.0*, *nE=25*, *npsi=25*, *nLz=30*, *numcores=1*, *interpecc=False*, ***kwargs*)

NAME: __init__

PURPOSE: initialize an actionAngleStaeckelGrid object

INPUT: pot= potential or list of potentials

delta= focus of prolate confocal coordinate system (can be Quantity)

Rmax = Rmax for building grids (natural units)

nE=, npsi=, nLz= grid size

interpecc= (False) if True, also interpolate the approximate eccentricity, zmax, rperi, and rapo

numcores= number of cpus to use to parallellize

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2012-11-29 - Written - Bovy (IAS)

2017-12-15 - Written - Bovy (UofT)

actionAngleIsochroneApprox

class galpy.actionAngle.**actionAngleIsochroneApprox**(**args*, ***kwargs*)

Action-angle formalism using an isochrone potential as an approximate potential and using a Fox & Binney (2014?) like algorithm to calculate the actions using orbit integrations and a torus-machinery-like angle-fit to get the angles and frequencies (Bovy 2014)

__init__(**args*, ***kwargs*)

NAME: __init__

PURPOSE: initialize an actionAngleIsochroneApprox object

INPUT:

Either:

b= scale parameter of the isochrone parameter (can be Quantity)

ip= instance of a IsochronePotential

aAI= instance of an actionAngleIsochrone

pot= potential to calculate action-angle variables for

tintJ= (default: 100) time to integrate orbits for to estimate actions (can be Quantity)

ntintJ= (default: 10000) number of time-integration points

integrate_method= (default: 'dopr54_c') integration method to use

dt= (None) orbit.integrate dt keyword (for fixed stepsize integration)

maxn= (default: 3) Default value for all methods when using a grid in vec(n) up to this n (zero-based)

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY: 2013-09-10 - Written - Bovy (IAS)

2.3.2 (J, a) –> (x, v, O)

General instance routines

Warning: While the `actionAngleTorus` code below can compute the Jacobian and Hessian of the (**J**, **a**) –> (**x**, **v**, **O**) transformation, the accuracy of these does not appear to be very good using the current interface to the TorusMapper code, so care should be taken when using these.

Currently, only the interface to the TorusMapper code supports going from (**J**, **a**) –> (**x**, **v**, **O**). Instance methods are

galpy.actionAngle.actionAngleInverse.__call__

`actionAngleInverse.__call__`(**args*, ***kwargs*)

NAME:

evaluate the phase-space coordinates (x,v) for a number of angles on a single torus

INPUT:

jr - radial action (scalar)

jphi - azimuthal action (scalar)

jz - vertical action (scalar)

angler - radial angle (array [N])

anglephi - azimuthal angle (array [N])

anglez - vertical angle (array [N])

Some sub-classes (like actionAngleTorus) have additional parameters:

actionAngleTorus:

tol= (object-wide value) goal for |dJ|/|J| along the torus

OUTPUT:

[R,vR,vT,z,vz,phi]

HISTORY:

2017-11-14 - Written - Bovy (UofT)

galpy.actionAngle.actionAngleInverse.Freqs

actionAngleInverse.**Freqs**(**args*, ***kwargs*)

NAME:

Freqs

PURPOSE:

return the frequencies corresponding to a torus

INPUT:

jr - radial action (scalar)

jphi - azimuthal action (scalar)

jz - vertical action (scalar)

OUTPUT:

(OmegaR,Omegaphi,Omegaz)

HISTORY:

2017-11-15 - Written - Bovy (UofT)

galpy.actionAngle.actionAngleInverse.xvFreqs

actionAngleInverse.**xvFreqs**(**args*, ***kwargs*)

NAME:

xvFreqs

PURPOSE:

evaluate the phase-space coordinates (x,v) for a number of angles on a single torus as well as the frequencies

INPUT:

jr - radial action (scalar)

jphi - azimuthal action (scalar)

jz - vertical action (scalar)

angler - radial angle (array [N])

anglephi - azimuthal angle (array [N])

anglez - vertical angle (array [N])

OUTPUT:

([R,vR,vT,z,vz,phi],OmegaR,Omegaphi,Omegaz)

HISTORY:

2017-11-15 - Written - Bovy (UofT)

Specific actionAngle modules

actionAngleHarmonicInverse

class galpy.actionAngle.**actionAngleHarmonicInverse**(**args*, ***kwargs*)

Inverse action-angle formalism for the one-dimensional harmonic oscillator

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

initialize an actionAngleHarmonicInverse object

INPUT:

omega= frequency (can be Quantity)

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2018-04-08 - Started - Bovy (UofT)

actionAngleIsochroneInverse

class galpy.actionAngle.**actionAngleIsochroneInverse**(**args*, ***kwargs*)

Inverse action-angle formalism for the isochrone potential, on the Jphi, Jtheta system of Binney & Tremaine (2008); following McGill & Binney (1990) for transformations

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

initialize an actionAngleIsochroneInverse object

INPUT:

Either:

b= scale parameter of the isochrone parameter (can be Quantity)

ip= instance of a IsochronePotential

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OUTPUT:

instance

HISTORY:

2017-11-14 - Started - Bovy (UofT)

actionAngleTorus

class galpy.actionAngle.**actionAngleTorus**(**args*, ***kwargs*)

Action-angle formalism using the Torus machinery

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

initialize an actionAngleTorus object

INPUT:

pot= potential or list of potentials (3D)

tol= default tolerance to use when fitting tori (|dJ|/J)

dJ= default action difference when computing derivatives (Hessian or Jacobian)

OUTPUT:

instance

HISTORY:

2015-08-07 - Written - Bovy (UofT)

In addition to the methods listed above, `actionAngleTorus` also has the following methods:

galpy.actionAngle.actionAngleTorus.hessianFreqs

actionAngleTorus.**hessianFreqs**(*jr*, *jphi*, *jz*, ***kwargs*)

NAME:

hessianFreqs

PURPOSE:

return the Hessian d Omega / d J and frequencies Omega corresponding to a torus

INPUT:

jr - radial action (scalar)

jphi - azimuthal action (scalar)

jz - vertical action (scalar)

tol= (object-wide value) goal for |dJ|/|J| along the torus

dJ= (object-wide value) action difference when computing derivatives (Hessian or Jacobian)

nosym= (False) if True, don't explicitly symmetrize the Hessian (good to check errors)

OUTPUT:

(dO/dJ,Omegar,Omegaphi,Omegaz,Autofit error message)

HISTORY:

2016-07-15 - Written - Bovy (UofT)

galpy.actionAngle.actionAngleTorus.xvJacobianFreqs

actionAngleTorus.**xvJacobianFreqs**(*jr*, *jphi*, *jz*, *angler*, *anglephi*, *anglez*, ***kwargs*)

NAME:

xvJacobianFreqs

PURPOSE:

return [R,vR,vT,z,vz,phi], the Jacobian d [R,vR,vT,z,vz,phi] / d (J,angle), the Hessian dO/dJ, and frequencies Omega corresponding to a torus at multiple sets of angles

INPUT:

jr - radial action (scalar)

jphi - azimuthal action (scalar)

jz - vertical action (scalar)

angler - radial angle (array [N])

anglephi - azimuthal angle (array [N])

anglez - vertical angle (array [N])

tol= (object-wide value) goal for |dJ|/|J| along the torus

dJ= (object-wide value) action difference when computing derivatives (Hessian or Jacobian)

nosym= (False) if True, don't explicitly symmetrize the Hessian (good to check errors)

OUTPUT:

([R,vR,vT,z,vz,phi], [N,6] array

d[R,vR,vT,z,vz,phi]/d[J,angle], –> (N,6,6) array

dO/dJ, –> (3,3) array

Omegar,Omegaphi,Omegaz, [N] arrays

Autofit error message)

HISTORY:

2016-07-19 - Written - Bovy (UofT)

2.4 DF (galpy.df)

galpy.df contains tools for dealing with distribution functions of stars in galaxies. It mainly contains a number of classes that define different types of distribution function, but galpy.df.jeans also has some tools for solving the Jeans equations for equilibrium systems.

2.4.1 Jeans modeling tools (galpy.df.jeans)

galpy.df.jeans.sigmar

galpy.df.jeans.**sigmar**(*Pot*, *r*, *dens=None*, *beta=0.0*)

NAME:

sigmar

PURPOSE:

Compute the radial velocity dispersion using the spherical Jeans equation

INPUT:

Pot - potential or list of potentials (evaluated at R=r/sqrt(2),z=r/sqrt(2), sphericity not checked)

r - Galactocentric radius (can be Quantity)

dens= (None) tracer density profile (function of r); if None, the density is assumed to be that corresponding to the potential

beta= (0.) anisotropy; can be a constant or a function of r

OUTPUT:

sigma_r(r)

HISTORY:

2018-07-05 - Written - Bovy (UofT)

galpy.df.jeans.sigmalos

galpy.df.jeans.**sigmalos**(*Pot*, *R*, *dens=None*, *surfdens=None*, *beta=0.0*, *sigma_r=None*)

NAME:

sigmalos

PURPOSE:

Compute the line-of-sight velocity dispersion using the spherical Jeans equation

INPUT:

Pot - potential or list of potentials (evaluated at R=r/sqrt(2),z=r/sqrt(2), sphericity not checked)

R - Galactocentric projected radius (can be Quantity)

dens= (None) tracer density profile (function of r); if None, the density is assumed to be that corresponding to the potential

surfdens= (None) tracer surface density profile (value at R or function of R); if None, the surface density is assumed to be that corresponding to the density

beta= (0.) anisotropy; can be a constant or a function of r

sigma_r= (None) if given, the solution of the spherical Jeans equation sigma_r(r) (used instead of solving the Jeans equation as part of this routine)

OUTPUT:

sigma_los(R)

HISTORY:

2018-08-27 - Written - Bovy (UofT)

2.4.2 General instance routines for all df classes

galpy.actionAngle.actionAngle.turn_physical_off

actionAngle.**turn_physical_off**()

NAME:

turn_physical_off

PURPOSE:

turn off automatic returning of outputs in physical units

INPUT:

(none)

OUTPUT:

(none)

HISTORY:

2017-06-05 - Written - Bovy (UofT)

galpy.actionAngle.actionAngle.turn_physical_on

actionAngle.**turn_physical_on**(*ro=None*, *vo=None*)

NAME:

turn_physical_on

PURPOSE:

turn on automatic returning of outputs in physical units

INPUT:

ro= reference distance (kpc; can be Quantity)

vo= reference velocity (km/s; can be Quantity)

OUTPUT:

(none)

HISTORY:

2016-06-05 - Written - Bovy (UofT)

2020-04-22 - Don't turn on a parameter when it is False - Bovy (UofT)

2.4.3 NEW in v1.7 Spherical distribution functions

Isotropic and anisotropic distribution functions for spherical systems. Documentation of these is limited at this point, but generally, one can use them as:

```
from galpy import potential
from galpy.df import isotropicNFWdf
np= potential.NFWPotential(amp=1.2,a=2.3)
ndf= isotropicNFWdf(pot=np)
# sample
sam= ndf.sample(n=int(1e6))
print(numpy.std(sam[numpy.fabs(sam.r()-1.2) < 0.1].vr()))
# 0.2156787374302913
# Compute vel. dispersion
print(ndf.sigmar(1.2))
# 0.21985277878647172
```

or:

```
from galpy.df import kingdf
kdf= kingdf(M=2.3,rt=1.4,W0=3.)
sam= kdf.sample(n=int(1e6))
print(numpy.amax(sam.r()))
# 1.3883460662897116
print(numpy.std(sam[numpy.fabs(sam.r()-0.2) < 0.01].vr()))
# 1.081298923132113
print(kdf.sigmar(0.2))
# 1.0939934290993467
```

Various spherical DFs are explicitly implemented (e.g., Hernquist, NFW using a new approximation, King, Plummer) in isotropic and various anisotropic forms. General methods for computing isotropic, constant-beta anisotropic, and Osipkov-Merritt anisotropic for any potential/density pair are also included.

General instance routines

galpy.df.sphericaldf.__call__

sphericaldf.**__call__**(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

return the DF

INPUT:

Either:

a) **(E,L,Lz): tuple of E and (optionally) L and (optionally) Lz.** Each may be Quantity

b) R,vR,vT,z,vz,phi:

c) **Orbit instance: orbit.Orbit instance and if specific time** then orbit.Orbit(t)

OUTPUT:

Value of DF

HISTORY:

2020-07-22 - Written - Lane (UofT)

galpy.df.sphericaldf.beta

sphericaldf.**beta**(*r*)

NAME:

sigmar

PURPOSE:

calculate the anisotropy at radius r

INPUT:

r - spherical radius at which to calculate the anisotropy

OUTPUT:

beta(r)

HISTORY:

2020-09-04 - Written - Bovy (UofT)

galpy.df.sphericaldf.sigmar

sphericaldf.**sigmar**(*r*)

NAME:

sigmar

PURPOSE:

calculate the radial velocity dispersion at radius r

INPUT:

r - spherical radius at which to calculate the radial velocity dispersion

OUTPUT:

sigma_r(r)

HISTORY:

2020-09-04 - Written - Bovy (UofT)

galpy.df.sphericaldf.sigmat

sphericaldf.**sigmat**(*r*)

NAME:

sigmar

PURPOSE:

calculate the tangential velocity dispersion at radius r

INPUT:

r - spherical radius at which to calculate the tangential velocity dispersion

OUTPUT:

sigma_t(r)

HISTORY:

2020-09-04 - Written - Bovy (UofT)

galpy.df.sphericaldf.vmomentdensity

sphericaldf.**vmomentdensity**(*r*, *n*, *m*, ***kwargs*)

NAME:

vmomentdensity

PURPOSE:

calculate an arbitrary moment of the velocity distribution at r times the density

INPUT:

r - spherical radius at which to calculate the moment

n - vr^n, where vr = v x cos eta

m - vt^m, where vt = v x sin eta

OUTPUT:

<vr^n vt^m x density> at r

HISTORY:

2020-09-04 - Written - Bovy (UofT)

Sampling routines

galpy.df.sphericaldf.sample

sphericaldf.**sample**(*R=None*, *z=None*, *phi=None*, *n=1*, *return_orbit=True*, *rmin=0.0*)

NAME:

sample

PURPOSE:

Return full 6D samples of the DF

INPUT:

R= cylindrical radius at which to generate samples (can be Quantity)

z= height at which to generate samples (can be Quantity)

phi= azimuth at which to generate samples (can be Quantity)

n= number of samples to generate

rmin= (0.) only sample r > rmin (can be Quantity)

OPTIONAL INPUT:

return_orbit= (True) If True output is orbit.Orbit object, if False output is (R,vR,vT,z,vz,phi)

OUTPUT:

List of samples. Either vector (R,vR,vT,z,vz,phi) or orbit.Orbit

NOTES:

If R,z,phi are None then sample positions with CMF. If R,z,phi are floats then sample n velocities at location. If array then sample velocities at radii, ignoring n. phi can be None if R,z are set by any above mechanism, will then sample phi for output.

HISTORY:

2020-07-22 - Written - Lane (UofT)

Specific distribution functions

The following are isotropic distribution functions

Arbitrary Eddington-inversion DF

class galpy.df.**eddingtondf**(*pot=None*, *denspot=None*, *rmax=10000.0*, *scale=None*, *ro=None*, *vo=None*)

Class that implements isotropic spherical DFs computed using the Eddington formula

$$f(\mathcal{E}) = \frac{1}{\sqrt{8}\,\pi^2}\left[\int_0^{\mathcal{E}} \mathrm{d}\Psi\,\frac{1}{\sqrt{\mathcal{E}-\Psi}}\,\frac{\mathrm{d}^2\rho}{\mathrm{d}\Psi^2} + \frac{1}{\sqrt{\mathcal{E}}}\,\frac{\mathrm{d}\rho}{\mathrm{d}\Psi}\bigg|_{\Psi=0}\right],$$

where $\Psi = -\Phi + \Phi(\infty)$ is the relative potential, $\mathcal{E} = \Psi - v^2/2$ is the relative (binding) energy, and ρ is the density of the tracer population (not necessarily the density corresponding to Ψ according to the Poisson equation). Note that the second term on the right-hand side is currently assumed to be zero in the code.

__init__(*pot=None*, *denspot=None*, *rmax=10000.0*, *scale=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize an isotropic distribution function computed using the Eddington inversion

INPUT:

pot= (None) Potential instance or list thereof that represents the gravitational potential (assumed to be spherical)

denspot= (None) Potential instance or list thereof that represent the density of the tracers (assumed to be spherical; if None, set equal to pot)

rmax= (None) maximum radius to consider (can be Quantity); DF is cut off at E = Phi(rmax)

scale= Characteristic scale radius to aid sampling calculations. Optionaland will also be overridden by value from pot if available.

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2021-02-04 - Written - Bovy (UofT)

Isotropic Hernquist DF

class galpy.df.**isotropicHernquistdf**(*pot=None*, *ro=None*, *vo=None*)
Class that implements isotropic spherical Hernquist DF computed using the Eddington formula

__init__(*pot=None*, *ro=None*, *vo=None*)
NAME:

__init__

PURPOSE:

Initialize an isotropic Hernquist distribution function

INPUT:

pot= (None) Hernquist Potential instance

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2020-08-09 - Written - Lane (UofT)

King DF

class galpy.df.**kingdf**(*W0*, *M=1.0*, *rt=1.0*, *npt=1001*, *ro=None*, *vo=None*)
Class that represents a King DF

__init__(*W0*, *M=1.0*, *rt=1.0*, *npt=1001*, *ro=None*, *vo=None*)
NAME:

__init__

PURPOSE:

Initialize a King DF

INPUT:

W0 - dimensionless central potential W0 = Psi(0)/sigma^2 (in practice, needs to be <~ 200, where the DF is essentially isothermal)

M= (1.) total mass (can be a Quantity)

rt= (1.) tidal radius (can be a Quantity)

npt= (1001) number of points to use to solve for Psi(r)

ro=, vo= standard galpy unit scaling parameters

OUTPUT:

(none; sets up instance)

HISTORY:

2020-07-09 - Written - Bovy (UofT)

Isotropic NFW DF

class galpy.df.**isotropicNFWdf**(*pot=None*, *widrow=False*, *rmax=10000.0*, *ro=None*, *vo=None*)

Class that implements the approximate isotropic spherical NFW DF (either Widrow 2000 or an improved fit by Lane et al. 2021).

__init__(*pot=None*, *widrow=False*, *rmax=10000.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize an isotropic NFW distribution function

INPUT:

pot= (None) NFW Potential instance

widrow= (False) if True, use the approximate form from Widrow (2000), otherwise use improved fit that has <~1e-5 relative density errors

rmax= (1e4) maximum radius to consider (can be Quantity); set to numpy.inf to evaluate NFW w/o cut-off

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2021-02-01 - Written - Bovy (UofT)

Isotropic Plummer DF

class galpy.df.**isotropicPlummerdf**(*pot=None*, *ro=None*, *vo=None*)

Class that implements isotropic spherical Plummer DF:

$$f(E) = \frac{24\sqrt{2}}{7\pi^3}\frac{b^2}{(GM)^5}(-E)^{7/2}$$

for $-GM/b \leq E \leq 0$ and zero otherwise. The parameter GM is the total mass and b the Plummer profile's scale parameter.

__init__(*pot=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize an isotropic Plummer distribution function

INPUT:

pot= (None) Plummer Potential instance

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2020-10-01 - Written - Bovy (UofT)

Anisotropic versions also exist:

Arbitrary constant-anisotropy DF

class galpy.df.**constantbetadf**(*pot=None*, *denspot=None*, *beta=0.0*, *twobeta=None*, *rmax=None*, *scale=None*, *ro=None*, *vo=None*)

Class that implements DFs of the form $f(E,L) = L^{-2\beta} f_1(E)$ with constant β anisotropy parameter for a given density profile

__init__(*pot=None*, *denspot=None*, *beta=0.0*, *twobeta=None*, *rmax=None*, *scale=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a spherical DF with constant anisotropy parameter

INPUT:

pot= (None) Potential instance or list thereof

denspot= (None) Potential instance or list thereof that represent the density of the tracers (assumed to be spherical; if None, set equal to pot)

beta= (0.) anisotropy parameter

twobeta= (None) twice the anisotropy parameter (useful for eta = half-integer, which is a special case); has priority over beta

rmax= (None) maximum radius to consider (can be Quantity); DF is cut off at E = Phi(rmax)

scale - Characteristic scale radius to aid sampling calculations. Optionaland will also be overridden by value from pot if available.

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2021-02-14 - Written - Bovy (UofT)

Arbitrary Osipkov-Merritt DF

class galpy.df.**osipkovmerrittdf**(*pot=None*, *denspot=None*, *ra=1.4*, *rmax=10000.0*, *scale=None*, *ro=None*, *vo=None*)

Class that implements spherical DFs with Osipkov-Merritt-type orbital anisotropy

$$\beta(r) = \frac{1}{1 + r_a^2/r^2}$$

with r_a the anistropy radius for arbitrary combinations of potential and density profile.

__init__(*pot=None*, *denspot=None*, *ra=1.4*, *rmax=10000.0*, *scale=None*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a DF with Osipkov-Merritt anisotropy

INPUT:

pot= (None) Potential instance or list thereof

denspot= (None) Potential instance or list thereof that represent the density of the tracers (assumed to be spherical; if None, set equal to pot)

ra - anisotropy radius (can be a Quantity)

rmax= (1e4) maximum radius to consider (can be Quantity); DF is cut off at E = Phi(rmax)

scale - Characteristic scale radius to aid sampling calculations. Optionaland will also be overridden by value from pot if available.

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2021-02-07 - Written - Bovy (UofT)

Anisotropic Hernquist DF with constant β

class galpy.df.**constantbetaHernquistdf**(*pot=None*, *beta=0*, *ro=None*, *vo=None*)

Class that implements the anisotropic spherical Hernquist DF with constant beta parameter

__init__(*pot=None*, *beta=0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a Hernquist DF with constant anisotropy

INPUT:

pot - Hernquist potential which determines the DF

beta - anisotropy parameter

OUTPUT:

None

HISTORY:

2020-07-22 - Written - Lane (UofT)

Anisotropic Hernquist DF of the Osipkov-Merritt type

class galpy.df.**osipkovmerrittHernquistdf**(*pot=None*, *ra=1.4*, *ro=None*, *vo=None*)

Class that implements the anisotropic spherical Hernquist DF with radially varying anisotropy of the Osipkov-

Merritt type

$$\beta(r) = \frac{1}{1 + r_a^2/r^2}$$

with r_a the anistropy radius.

__init__ (*pot=None*, *ra=1.4*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a Hernquist DF with Osipkov-Merritt anisotropy

INPUT:

pot - Hernquist potential which determines the DF

ra - anisotropy radius (can be a Quantity)

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2020-11-12 - Written - Bovy (UofT)

Anisotropic NFW DF of the Osipkov-Merritt type

class galpy.df.**osipkovmerrittNFWdf** (*pot=None*, *ra=1.4*, *rmax=10000.0*, *ro=None*, *vo=None*)

Class that implements the anisotropic spherical NFW DF with radially varying anisotropy of the Osipkov-Merritt type

$$\beta(r) = \frac{1}{1 + r_a^2/r^2}$$

with r_a the anistropy radius.

__init__ (*pot=None*, *ra=1.4*, *rmax=10000.0*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a NFW DF with Osipkov-Merritt anisotropy

INPUT:

pot - NFW potential which determines the DF

ra - anisotropy radius (can be a Quantity)

rmax= (1e4) maximum radius to consider (can be Quantity); set to numpy.inf to evaluate NFW w/o cut-off

ro=, vo= galpy unit parameters

OUTPUT:

None

HISTORY:

2020-11-12 - Written - Bovy (UofT)

2.4.4 Two-dimensional, axisymmetric disk distribution functions

Distribution function for orbits in the plane of a galactic disk.

General instance routines

galpy.df.diskdf.__call__

diskdf.**__call__**(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

evaluate the distribution function

INPUT:

either an orbit instance, a list of such instances, or E,Lz

1) Orbit instance or list: a) Orbit instance alone: use initial condition b) Orbit instance + t: call the Orbit instance (for list, each instance is called at t)
2) E - energy (/vo^2; or can be Quantity) L - angular momentun (/ro/vo; or can be Quantity)
3) array vxvv [3/4,nt] [must be in natural units /vo,/ro; use Orbit interface for physical-unit input)

KWARGS:

marginalizeVperp - marginalize over perpendicular velocity (only supported with 1a) for single orbits above)

marginalizeVlos - marginalize over line-of-sight velocity (only supported with 1a) for single orbits above)

nsigma= number of sigma to integrate over when marginalizing

+scipy.integrate.quad keywords

OUTPUT:

DF(orbit/E,L)

HISTORY:

2010-07-10 - Written - Bovy (NYU)

galpy.df.diskdf.asymmetricdrift

diskdf.**asymmetricdrift**(*R*)

NAME:

asymmetricdrift

PURPOSE:

estimate the asymmetric drift (vc-mean-vphi) from an approximation to the Jeans equation

INPUT:

R - radius at which to calculate the asymmetric drift (can be Quantity)

OUTPUT:

asymmetric drift at R

HISTORY:

2011-04-02 - Written - Bovy (NYU)

galpy.df.diskdf.kurtosisvR

diskdf.**kurtosisvR**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

kurtosisvR

PURPOSE:

calculate excess kurtosis in vR at R by marginalizing over velocity

INPUT:

R - radius at which to calculate <vR> (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

kurtosisvR

HISTORY:

2011-12-07 - Written - Bovy (NYU)

galpy.df.diskdf.kurtosisvT

diskdf.**kurtosisvT**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

kurtosisvT

PURPOSE:

calculate excess kurtosis in vT at R by marginalizing over velocity

INPUT:

R - radius at which to calculate <vR> (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

kurtosisvT

HISTORY:

2011-12-07 - Written - Bovy (NYU)

galpy.df.diskdf.meanvR

diskdf.**meanvR**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

meanvR

PURPOSE:

calculate <vR> at R by marginalizing over velocity

INPUT:

R - radius at which to calculate <vR> (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

<vR> at R

HISTORY:

2011-03-30 - Written - Bovy (NYU)

galpy.df.diskdf.meanvT

diskdf.**meanvT**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

meanvT

PURPOSE:

calculate <vT> at R by marginalizing over velocity

INPUT:

R - radius at which to calculate <vT> (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

<vT> at R

HISTORY:

2011-03-30 - Written - Bovy (NYU)

galpy.df.diskdf.oortA

diskdf.**oortA**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

oortA

PURPOSE:

calculate the Oort function A

INPUT:

R - radius at which to calculate A (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

Oort A at R

HISTORY:

2011-04-19 - Written - Bovy (NYU)

BUGS:

could be made more efficient, e.g., surfacemass is calculated multiple times

galpy.df.diskdf.oortB

diskdf.**oortB**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

oortB

PURPOSE:

calculate the Oort function B

INPUT:

R - radius at which to calculate B (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

Oort B at R

HISTORY:

2011-04-19 - Written - Bovy (NYU)

BUGS:

could be made more efficient, e.g., surfacemass is calculated multiple times

galpy.df.diskdf.oortC

diskdf.**oortC**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

oortC

PURPOSE:

calculate the Oort function C

INPUT:

R - radius at which to calculate C (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

Oort C at R

HISTORY:

2011-04-19 - Written - Bovy (NYU)

BUGS:

could be made more efficient, e.g., surfacemass is calculated multiple times we know this is zero, but it is calculated anyway (bug or feature?)

galpy.df.diskdf.oortK

diskdf.**oortK**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

oortK

PURPOSE:

calculate the Oort function K

INPUT:

R - radius at which to calculate K (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

Oort K at R

HISTORY:

2011-04-19 - Written - Bovy (NYU)

BUGS:

could be made more efficient, e.g., surfacemass is calculated multiple times we know this is zero, but it is calculated anyway (bug or feature?)

galpy.df.diskdf.sigma2surfacemass

diskdf.**sigma2surfacemass**(*R*, *romberg=False*, *nsigma=None*, *relative=False*)

NAME:

sigma2surfacemass

PURPOSE:

calculate the product sigma_R^2 x surface-mass at R by marginalizing over velocity

INPUT:

R - radius at which to calculate the sigma_R^2 x surfacemass density (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

sigma_R^2 x surface-mass at R

HISTORY:

2010-03-XX - Written - Bovy (NYU)

galpy.df.diskdf.sigma2

diskdf.**sigma2**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

sigma2

PURPOSE:

calculate sigma_R^2 at R by marginalizing over velocity

INPUT:

R - radius at which to calculate sigma_R^2 density (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

sigma_R^2 at R

HISTORY:

2010-03-XX - Written - Bovy (NYU)

galpy.df.diskdf.sigmaR2

diskdf.**sigmaR2**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

sigmaR2 (duplicate of sigma2 for consistency)

PURPOSE:

calculate sigma_R^2 at R by marginalizing over velocity

INPUT:

R - radius at which to calculate sigma_R^2 (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

sigma_R^2 at R

HISTORY:

2011-03-30 - Written - Bovy (NYU)

galpy.df.diskdf.sigmaT2

diskdf.**sigmaT2**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

sigmaT2

PURPOSE:

calculate sigma_T^2 at R by marginalizing over velocity

INPUT:

R - radius at which to calculate sigma_T^2 (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

sigma_T^2 at R

HISTORY:

2011-03-30 - Written - Bovy (NYU)

galpy.df.diskdf.skewvR

diskdf.**skewvR**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

skewvR

PURPOSE:

calculate skew in vR at R by marginalizing over velocity

INPUT:

R - radius at which to calculate <vR> (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

skewvR

HISTORY:

2011-12-07 - Written - Bovy (NYU)

galpy.df.diskdf.skewvT

diskdf.**skewvT**(*R*, *romberg=False*, *nsigma=None*, *phi=0.0*)

NAME:

skewvT

PURPOSE:

calculate skew in vT at R by marginalizing over velocity

INPUT:

R - radius at which to calculate <vR> (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

skewvT

HISTORY:

2011-12-07 - Written - Bovy (NYU)

galpy.df.diskdf.surfacemass

diskdf.**surfacemass**(*R*, *romberg=False*, *nsigma=None*, *relative=False*)

NAME:

surfacemass

PURPOSE:

calculate the surface-mass at R by marginalizing over velocity

INPUT:

R - radius at which to calculate the surfacemass density (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

romberg - if True, use a romberg integrator (default: False)

OUTPUT:

surface mass at R

HISTORY:

2010-03-XX - Written - Bovy (NYU)

galpy.df.diskdf.surfacemassLOS

diskdf.**surfacemassLOS**(*d*, *l*, *deg=True*, *target=True*, *romberg=False*, *nsigma=None*, *relative=None*)

NAME:

surfacemassLOS

PURPOSE:

evaluate the surface mass along the LOS given l and d

INPUT:

d - distance along the line of sight (can be Quantity)

l - Galactic longitude (in deg, unless deg=False; can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

KEYWORDS:

target= if True, use target surfacemass (default)

romberg - if True, use a romberg integrator (default: False)

deg= if False, l is in radians

OUTPUT:

Sigma(d,l) x d

HISTORY:

2011-03-24 - Written - Bovy (NYU)

galpy.df.diskdf.targetSigma2

diskdf.**targetSigma2**(*R*, *log=False*)

NAME:

targetSigma2

PURPOSE:

evaluate the target Sigma_R^2(R)

INPUT:

R - radius at which to evaluate (can be Quantity)

OUTPUT:

target Sigma_R^2(R)

log - if True, return the log (default: False)

HISTORY:

2010-03-28 - Written - Bovy (NYU)

galpy.df.diskdf.targetSurfacemass

diskdf.**targetSurfacemass**(*R*, *log=False*)

NAME:

targetSurfacemass

PURPOSE:

evaluate the target surface mass at R

INPUT:

R - radius at which to evaluate (can be Quantity)

log - if True, return the log (default: False)

OUTPUT:

Sigma(R)

HISTORY:

2010-03-28 - Written - Bovy (NYU)

galpy.df.diskdf.targetSurfacemassLOS

diskdf.**targetSurfacemassLOS**(*d*, *l*, *log=False*, *deg=True*)

NAME:

targetSurfacemassLOS

PURPOSE:

evaluate the target surface mass along the LOS given l and d

INPUT:

d - distance along the line of sight (can be Quantity)

l - Galactic longitude (in deg, unless deg=False; can be Quantity)

deg= if False, l is in radians

log - if True, return the log (default: False)

OUTPUT:

Sigma(d,l) x d

HISTORY:

2011-03-23 - Written - Bovy (NYU)

galpy.df.diskdf._vmomentsurfacemass

diskdf.**_vmomentsurfacemass**(*R*, *n*, *m*, *romberg=False*, *nsigma=None*, *relative=False*, *phi=0.0*, *deriv=None*)

Non-physical version of vmomentsurfacemass, otherwise the same

Sampling routines

galpy.df.diskdf.sample

diskdf.**sample**(*n=1*, *rrange=None*, *returnROrbit=True*, *returnOrbit=False*, *nphi=1.0*, *los=None*, *losdeg=True*, *nsigma=None*, *maxd=None*, *target=True*)

NAME:

sample

PURPOSE:

sample n*nphi points from this DF

INPUT:

n - number of desired sample (specifying this rather than calling this routine n times is more efficient)

rrange - if you only want samples in this rrange, set this keyword (only works when asking for an (RZ)Orbit) (can be Quantity)

returnROrbit - if True, return a planarROrbit instance: [R,vR,vT] (default)

returnOrbit - if True, return a planarOrbit instance (including phi)

nphi - number of azimuths to sample for each E,L

los= line of sight sampling along this line of sight (can be Quantity)

losdeg= los in degrees? (default=True)

target= if True, use target surface mass and sigma2 profiles (default=True)

nsigma= number of sigma to rejection-sample on

maxd= maximum distance to consider (for the rejection sampling)

OUTPUT:

n*nphi list of [[E,Lz],...] or list of planar(R)Orbits

CAUTION: lists of EL need to be post-processed to account for the kappa/omega_R discrepancy

HISTORY:

2010-07-10 - Started - Bovy (NYU)

galpy.df.diskdf.sampledSurfacemassLOS

diskdf.**sampledSurfacemassLOS**(*l*, *n=1*, *maxd=None*, *target=True*)

NAME:

sampledSurfacemassLOS

PURPOSE:

sample a distance along the line of sight

INPUT:

l - Galactic longitude (in rad; can be Quantity)

n= number of distances to sample

maxd= maximum distance to consider (for the rejection sampling) (can be Quantity)

target= if True, sample from the 'target' surface mass density, rather than the actual surface mass density (default=True)

OUTPUT:

list of samples

HISTORY:

2011-03-24 - Written - Bovy (NYU)

hhgalpy.df.diskdf.sampleLOS

diskdf.**sampleLOS**(*los*, *n=1*, *deg=True*, *maxd=None*, *nsigma=None*, *targetSurfmass=True*, *targetSigma2=True*)

NAME:

sampleLOS

PURPOSE:

sample along a given LOS

INPUT:

los - line of sight (in deg, unless deg=False; can be Quantity)

n= number of desired samples

deg= los in degrees? (default=True)

targetSurfmass, targetSigma2= if True, use target surface mass and sigma2 profiles, respectively (there is not much p (default=True)

OUTPUT:

returns list of Orbits

BUGS: target=False uses target distribution for derivatives (this is a detail)

HISTORY:

2011-03-24 - Started - Bovy (NYU)

galpy.df.diskdf.sampleVRVT

diskdf.**sampleVRVT**(*R*, *n=1*, *nsigma=None*, *target=True*)

NAME:

sampleVRVT

PURPOSE:

sample a radial and azimuthal velocity at R

INPUT:

R - Galactocentric distance (can be Quantity)

n= number of distances to sample

nsigma= number of sigma to rejection-sample on

target= if True, sample using the 'target' sigma_R rather than the actual sigma_R (default=True)

OUTPUT:

list of samples

BUGS:

should use the fact that vR and vT separate

HISTORY:

2011-03-24 - Written - Bovy (NYU)

Specific distribution functions

Dehnen DF

class galpy.df.**dehnendf**(*surfaceSigma=<class 'galpy.df.surfaceSigmaProfile.expSurfaceSigmaProfile'>*, *profileParams=(0.3333333333333333, 1.0, 0.2)*, *correct=False*, *beta=0.0*, ***kwargs*)

Dehnen's 'new' df

__init__(*surfaceSigma=<class 'galpy.df.surfaceSigmaProfile.expSurfaceSigmaProfile'>*, *profileParams=(0.3333333333333333, 1.0, 0.2)*, *correct=False*, *beta=0.0*, ***kwargs*)

NAME: __init__

PURPOSE: Initialize a Dehnen 'new' DF

INPUT:

surfaceSigma - instance or class name of the target surface density and sigma_R profile (default: both exponential)

profileParams - parameters of the surface and sigma_R profile: (xD,xS,Sro) where

xD - disk surface mass scalelength (can be Quantity)

xS - disk velocity dispersion scalelength (can be Quantity)

Sro - disk velocity dispersion at Ro (can be Quantity)

Directly given to the 'surfaceSigmaProfile class, so could be anything that class takes

beta - power-law index of the rotation curve

correct - if True, correct the DF

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

+DFcorrection kwargs (except for those already specified)

OUTPUT:

instance

HISTORY:

2010-03-10 - Written - Bovy (NYU)

Schwarzschild DF

class galpy.df.**schwarzschilddf**(*surfaceSigma=<class 'galpy.df.surfaceSigmaProfile.expSurfaceSigmaProfile'>*, *profileParams=(0.3333333333333333, 1.0, 0.2)*, *correct=False*, *beta=0.0*, ***kwargs*)

Schwarzschild's df

__init__(*surfaceSigma=<class 'galpy.df.surfaceSigmaProfile.expSurfaceSigmaProfile'>*, *profileParams=(0.3333333333333333, 1.0, 0.2)*, *correct=False*, *beta=0.0*, ***kwargs*)

NAME: __init__

PURPOSE: Initialize a Schwarzschild DF

INPUT:

surfaceSigma - instance or class name of the target surface density and sigma_R profile (default: both exponential)

profileParams - parameters of the surface and sigma_R profile: (xD,xS,Sro) where

xD - disk surface mass scalelength (can be Quantity)

xS - disk velocity dispersion scalelength (can be Quantity)

Sro - disk velocity dispersion at Ro (can be Quantity)

Directly given to the 'surfaceSigmaProfile class, so could be anything that class takes

beta - power-law index of the rotation curve

correct - if True, correct the DF

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

+DFcorrection kwargs (except for those already specified)

OUTPUT:

instance

HISTORY:

2017-09-17 - Written - Bovy (UofT)

Shu DF

class galpy.df.**shudf**(*surfaceSigma=<class 'galpy.df.surfaceSigmaProfile.expSurfaceSigmaProfile'>*, *profileParams=(0.3333333333333333, 1.0, 0.2)*, *correct=False*, *beta=0.0*, ***kwargs*)

Shu's df (1969)

__init__(*surfaceSigma=<class 'galpy.df.surfaceSigmaProfile.expSurfaceSigmaProfile'>*, *profileParams=(0.3333333333333333, 1.0, 0.2)*, *correct=False*, *beta=0.0*, ***kwargs*)

NAME: __init__

PURPOSE: Initialize a Shu DF

INPUT:

surfaceSigma - instance or class name of the target surface density and sigma_R profile (default: both exponential)

profileParams - parameters of the surface and sigma_R profile: (xD,xS,Sro) where

xD - disk surface mass scalelength (can be Quantity)

xS - disk velocity dispersion scalelength (can be Quantity)

Sro - disk velocity dispersion at Ro (can be Quantity)

Directly given to the 'surfaceSigmaProfile class, so could be anything that class takes

beta - power-law index of the rotation curve

correct - if True, correct the DF

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

+DFcorrection kwargs (except for those already specified)

OUTPUT:

instance

HISTORY:

2010-05-09 - Written - Bovy (NYU)

2.4.5 Two-dimensional, non-axisymmetric disk distribution functions

Distribution function for orbits in the plane of a galactic disk in non-axisymmetric potentials. These are calculated using the technique of Dehnen 2000, where the DF at the current time is obtained as the evolution of an initially-axisymmetric DF at time `to` in the non-axisymmetric potential until the current time.

General instance routines

galpy.df.evolveddiskdf.__call__

evolveddiskdf.**__call__**(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

evaluate the distribution function

INPUT:

Orbit instance:

a) Orbit instance alone: use initial state and t=0

b) Orbit instance + t: Orbit instance *NOT* called (i.e., Orbit's initial condition is used, call Orbit yourself), t can be Quantity

If t is a list of t, DF is returned for each t, times must be in descending order and equally spaced (does not work with marginalize...)

marginalizeVperp - marginalize over perpendicular velocity (only supported with 1a) above) + nsigma, +scipy.integrate.quad keywords

marginalizeVlos - marginalize over line-of-sight velocity (only supported with 1a) above) + nsigma, +scipy.integrate.quad keywords

log= if True, return the log (not for deriv, bc that can be negative)

integrate_method= method argument of orbit.integrate

deriv= None, 'R', or 'phi': calculates derivative of the moment wrt R or phi **not with the marginalize options**

OUTPUT:

DF(orbit,t)

HISTORY:

2011-03-30 - Written - Bovy (NYU)

2011-04-15 - Added list of times option - Bovy (NYU)

The DF of a two-dimensional, non-axisymmetric disk

class galpy.df.**evolveddiskdf**(*initdf*, *pot*, *to=0.0*)

Class that represents a diskdf as initial DF + subsequent secular evolution

__init__(*initdf*, *pot*, *to=0.0*)

NAME:

__init__

PURPOSE:

initialize

INPUT:

initdf - the df at the start of the evolution (at to) (units are transferred)

pot - potential to integrate orbits in

to= initial time (time at which initdf is evaluated; orbits are integrated from current t back to to) (can be Quantity)

OUTPUT:

instance

HISTORY:

2011-03-30 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.meanvR

evolveddiskdf.**meanvR**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *surfacemass=None*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

meanvR

PURPOSE:

calculate the mean vR of the velocity distribution at (R,phi)

INPUT:

R - radius at which to calculate the moment(/ro) (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

surfacemass= if set use this pre-calculated surfacemass

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ratio of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

mean vR

HISTORY:

2011-03-31 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.meanvT

evolveddiskdf.**meanvT**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *surfacemass=None*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

meanvT

PURPOSE:

calculate the mean vT of the velocity distribution at (R,phi)

INPUT:

R - radius at which to calculate the moment (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

surfacemass= if set use this pre-calculated surfacemass

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ratio of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

mean vT

HISTORY:

2011-03-31 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.oortA

evolveddiskdf.**oortA**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrids=False*, *derivRGrid=None*, *derivphiGrid=None*, *derivGridpoints=101*, *derivHierarchgrid=False*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

oortA

PURPOSE:

calculate the Oort function A at (R,phi,t)

INPUT:

R - radius at which to calculate A (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

derivRGrid, derivphiGrid= if set to True, build a grid and use that to evaluate integrals of the derivatives of the DF;if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

derivGridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid objects (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

derivHierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

Oort A at R,phi,t

HISTORY:

2011-10-16 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.oortB

evolveddiskdf.**oortB**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrids=False*, *derivRGrid=None*, *derivphiGrid=None*, *derivGridpoints=101*, *derivHierarchgrid=False*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

oortB

PURPOSE:

calculate the Oort function B at (R,phi,t)

INPUT:

R - radius at which to calculate B (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

derivRGrid, derivphiGrid= if set to True, build a grid and use that to evaluat integrals of the derivatives of the DF: if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

derivGridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid objects (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

derivHierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

Oort B at R,phi,t

HISTORY:

2011-10-16 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.oortC

evolveddiskdf.**oortC**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrids=False*, *derivRGrid=None*, *derivphiGrid=None*, *derivGridpoints=101*, *derivHierarchgrid=False*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

oortC

PURPOSE:

calculate the Oort function C at (R,phi,t)

INPUT:

R - radius at which to calculate C (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

derivRGrid, derivphiGrid= if set to True, build a grid and use that to evaluate integrals of the derivatives of the DF; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

derivGridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid objects (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

derivHierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

Oort C at R,phi,t

HISTORY:

2011-10-16 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.oortK

evolveddiskdf.**oortK**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrids=False*, *derivRGrid=None*, *derivphiGrid=None*, *derivGridpoints=101*, *derivHierarchgrid=False*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

oortK

PURPOSE:

calculate the Oort function K at (R,phi,t)

INPUT:

R - radius at which to calculate K (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

derivRGrid, derivphiGrid= if set to True, build a grid and use that to evaluate integrals of the derivatives of the DF; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

derivGridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid objects (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

derivHierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

Oort K at R,phi,t

HISTORY:

2011-10-16 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.sigmaR2

evolveddiskdf.**sigmaR2**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *surfacemass=None*, *meanvR=None*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

sigmaR2

PURPOSE:

calculate the radial variance of the velocity distribution at (R,phi)

INPUT:

R - radius at which to calculate the moment (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

surfacemass, meanvR= if set use this pre-calculated surfacemass and mean vR

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ratio of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

variance of vR

HISTORY:

2011-03-31 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.sigmaRT

evolveddiskdf.**sigmaRT**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *epsrel=0.01*, *epsabs=1e-05*, *phi=0.0*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *surfacemass=None*, *meanvR=None*, *meanvT=None*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

sigmaRT

PURPOSE:

calculate the radial-tangential co-variance of the velocity distribution at (R,phi)

INPUT:

R - radius at which to calculate the moment (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

surfacemass, meanvR, meavT= if set use this pre-calculated surfacemass and mean vR and vT

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ration of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

covariance of vR and vT

HISTORY:

2011-03-31 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.sigmaT2

evolveddiskdf.**sigmaT2**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *phi=0.0*, *epsrel=0.01*, *epsabs=1e-05*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *surfacemass=None*, *meanvT=None*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

sigmaT2

PURPOSE:

calculate the tangential variance of the velocity distribution at (R,phi)

INPUT:

R - radius at which to calculate the moment (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

surfacemass, meanvT= if set use this pre-calculated surfacemass and mean tangential velocity

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ratio of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

variance of vT

HISTORY:

2011-03-31 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.vertexdev

evolveddiskdf.**vertexdev**(*R*, *t=0.0*, *nsigma=None*, *deg=False*, *epsrel=0.01*, *epsabs=1e-05*, *phi=0.0*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *sigmaR2=None*, *sigmaT2=None*, *sigmaRT=None*, *surfacemass=None*, *hierarchgrid=False*, *nlevels=2*, *integrate_method='dopr54_c'*)

NAME:

vertexdev

PURPOSE:

calculate the vertex deviation of the velocity distribution at (R,phi)

INPUT:

R - radius at which to calculate the moment (can be Quantity)

phi= azimuth (rad unless deg=True; can be Quantity)

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced) (can be Quantity)

sigmaR2, sigmaT2, sigmaRT= if set the vertex deviation is simply calculated using these

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous)

deg= azimuth is in degree (default=False); do not set this when giving phi as a Quantity

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ratio of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

integrate_method= orbit.integrate method argument

OUTPUT:

vertex deviation in rad

HISTORY:

2011-03-31 - Written - Bovy (NYU)

galpy.df.evolveddiskdf.vmomentsurfacemass

evolveddiskdf.**vmomentsurfacemass**(*R*, *n*, *m*, *t=0.0*, *nsigma=None*, *deg=False*, *epsrel=0.01*, *epsabs=1e-05*, *phi=0.0*, *grid=None*, *gridpoints=101*, *returnGrid=False*, *hierarchgrid=False*, *nlevels=2*, *print_progress=False*, *integrate_method='dopr54_c'*, *deriv=None*)

NAME:

vmomentsurfacemass

PURPOSE:

calculate the an arbitrary moment of the velocity distribution at (R,phi) times the surfacmass

INPUT:

R - radius at which to calculate the moment (in natural units)

phi= azimuth (rad unless deg=True)

n - vR^n

m - vT^m

t= time at which to evaluate the DF (can be a list or ndarray; if this is the case, list needs to be in descending order and equally spaced)

nsigma - number of sigma to integrate the velocities over (based on an estimate, so be generous, but not too generous)

deg= azimuth is in degree (default=False)

epsrel, epsabs - scipy.integrate keywords (the integration calculates the ratio of this vmoment to that of the initial DF)

grid= if set to True, build a grid and use that to evaluate integrals; if set to a grid-objects (such as returned by this procedure), use this grid; if this was created for a list of times, moments are calculated for each time

gridpoints= number of points to use for the grid in 1D (default=101)

returnGrid= if True, return the grid object (default=False)

hierarchgrid= if True, use a hierarchical grid (default=False)

nlevels= number of hierarchical levels for the hierarchical grid

print_progress= if True, print progress updates

integrate_method= orbit.integrate method argument

deriv= None, 'R', or 'phi': calculates derivative of the moment wrt R or phi **onnly with grid options**

OUTPUT:

<vR^n vT^m x surface-mass> at R,phi (no support for units)

COMMENT:

grid-based calculation is the only one that is heavily tested (although the test suite also tests the direct calculation)

HISTORY:

2011-03-30 - Written - Bovy (NYU)

2.4.6 Three-dimensional disk distribution functions

Distribution functions for orbits in galactic disks, including the vertical motion for stars reaching large heights above the plane. Currently only the *quasi-isothermal DF*.

General instance routines

galpy.df.quasiisothermaldf.__call__

quasiisothermaldf.**__call__**(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

return the DF

INPUT:

Either:

a)(jr,lz,jz) tuple; each can be a Quantity

where: jr - radial action lz - z-component of angular momentum jz - vertical action

b) R,vR,vT,z,vz

c) Orbit instance: initial condition used if that's it, orbit(t) if there is a time given as well

log= if True, return the natural log

+scipy.integrate.quadrature kwargs

func= function of (jr,lz,jz) to multiply f with (useful for moments)

OUTPUT:

value of DF

HISTORY:

2012-07-25 - Written - Bovy (IAS@MPIA)

NOTE:

For Miyamoto-Nagai/adiabatic approximation this seems to take about 30 ms / evaluation in the extended Solar neighborhood For a MWPotential/adiabatic approximation this takes about 50 ms / evaluation in the extended Solar neighborhood

For adiabatic-approximation grid this seems to take about 0.67 to 0.75 ms / evaluation in the extended Solar neighborhood (includes some out of the grid)

up to 200x faster when called with vector R,vR,vT,z,vz

galpy.df.quasiisothermaldf.density

quasiisothermaldf.**density**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

density

PURPOSE:

calculate the density at R,z by marginalizing over velocity

INPUT:

R - radius at which to calculate the density (can be Quantity)

z - height at which to calculate the density (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

density at (R,z)

HISTORY:

2012-07-26 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.estimate_hr

quasiisothermaldf.**estimate_hr**(*R*, *z=0.0*, *dR=1e-08*, ***kwargs*)

NAME:

estimate_hr

PURPOSE:

estimate the exponential scale length at R

INPUT:

R - Galactocentric radius (can be Quantity)

z= height (default: 0 pc) (can be Quantity)

dR- range in R to use (can be Quantity)

density kwargs

OUTPUT:

estimated hR

HISTORY:

2012-09-11 - Written - Bovy (IAS)

2013-01-28 - Re-written - Bovy

galpy.df.quasiisothermaldf.estimate_hsr

quasiisothermaldf.**estimate_hsr**(*R*, *z=0.0*, *dR=1e-08*, ***kwargs*)

NAME:

estimate_hsr

PURPOSE:

estimate the exponential scale length of the radial dispersion at R

INPUT:

R - Galactocentric radius (can be Quantity)

z= height (default: 0 pc) (can be Quantity)

dR- range in R to use (can be Quantity)

density kwargs

OUTPUT:

estimated hsR

HISTORY:

2013-03-08 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.estimate_hsz

quasiisothermaldf.**estimate_hsz**(*R*, *z=0.0*, *dR=1e-08*, ***kwargs*)

NAME:

estimate_hsz

PURPOSE:

estimate the exponential scale length of the vertical dispersion at R

INPUT:

R - Galactocentric radius (can be Quantity)

z= height (default: 0 pc) (can be Quantity)

dR- range in R to use (can be Quantity)

density kwargs

OUTPUT:

estimated hsz

HISTORY:

2013-03-08 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.estimate_hz

quasiisothermaldf.**estimate_hz**(*R*, *z*, *dz=1e-08*, ***kwargs*)

NAME:

estimate_hz

PURPOSE:

estimate the exponential scale height at R

INPUT:

R - Galactocentric radius (can be Quantity)

dz - z range to use (can be Quantity)

density kwargs

OUTPUT:

estimated hz

HISTORY:

2012-08-30 - Written - Bovy (IAS)

2013-01-28 - Re-written - Bovy

galpy.df.quasiisothermaldf._jmomentdensity

quasiisothermaldf.**_jmomentdensity**(*R*, *z*, *n*, *m*, *o*, *nsigma=None*, *mc=True*, *nmc=10000*, *_returnmc=False*, *_vrs=None*, *_vts=None*, *_vzs=None*, ***kwargs*)

Non-physical version of jmomentdensity, otherwise the same

galpy.df.quasiisothermaldf.meanjr

quasiisothermaldf.**meanjr**(*R*, *z*, *nsigma=None*, *mc=True*, *nmc=10000*, ***kwargs*)

NAME:

meanjr

PURPOSE:

calculate the mean radial action by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

OUTPUT:

meanjr

HISTORY:

2012-08-09 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.meanjz

quasiisothermaldf.**meanjz**(*R*, *z*, *nsigma=None*, *mc=True*, *nmc=10000*, ***kwargs*)

NAME:

meanjz

PURPOSE:

calculate the mean vertical action by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

OUTPUT:

meanjz

HISTORY:

2012-08-09 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.meanlz

quasiisothermaldf.**meanlz**(*R*, *z*, *nsigma=None*, *mc=True*, *nmc=10000*, ***kwargs*)

NAME:

meanlz

PURPOSE:

calculate the mean angular momemtum by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

OUTPUT:

meanlz

HISTORY:

2012-08-09 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.meanvR

quasiisothermaldf.**meanvR**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

meanvR

PURPOSE:

calculate the mean radial velocity by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

meanvR

HISTORY:

2012-12-23 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.meanvT

quasiisothermaldf.**meanvT**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

meanvT

PURPOSE:

calculate the mean rotational velocity by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

meanvT

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.meanvz

quasiisothermaldf.**meanvz**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

meanvz

PURPOSE:

calculate the mean vertical velocity by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

meanvz

HISTORY:

2012-12-23 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.pvR

quasiisothermaldf.**pvR**(*vR*, *R*, *z*, *gl=True*, *ngl=20*, *nsigma=4.0*, *vTmax=1.5*)

NAME:

pvR

PURPOSE:

calculate the marginalized vR probability at this location (NOT normalized by the density)

INPUT:

vR - radial velocity (can be Quantity)

R - radius (can be Quantity)

z - height (can be Quantity)

gl - use Gauss-Legendre integration (True, currently the only option)

ngl - order of Gauss-Legendre integration

nsigma - sets integration limits to [-1,+1]*nsigma*sigma_z(R) for integration over vz (default: 4)

vTmax - sets integration limits to [0,vTmax] for integration over vT (default: 1.5)

OUTPUT:

p(vR,R,z)

HISTORY:

2012-12-22 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.pvRvT

quasiisothermaldf.**pvRvT**(*vR*, *vT*, *R*, *z*, *gl=True*, *ngl=20*, *nsigma=4.0*)

NAME:

pvRvT

PURPOSE:

calculate the marginalized (vR,vT) probability at this location (NOT normalized by the density)

INPUT:

vR - radial velocity (can be Quantity)

vT - tangential velocity (can be Quantity)

R - radius (can be Quantity)

z - height (can be Quantity)

gl - use Gauss-Legendre integration (True, currently the only option)

ngl - order of Gauss-Legendre integration

nsigma - sets integration limits to [-1,+1]*nsigma*sigma_z(R) for integration over vz (default: 4)

OUTPUT:

p(vR,vT,R,z)

HISTORY:

2013-01-02 - Written - Bovy (IAS) 2018-01-12 - Added Gauss-Legendre integration prefactor nsigma/2 - Trick (MPA)

galpy.df.quasiisothermaldf.pvRvz

quasiisothermaldf.**pvRvz**(*vR*, *vz*, *R*, *z*, *gl=True*, *ngl=20*, *vTmax=1.5*)

NAME:

pvR

PURPOSE:

calculate the marginalized (vR,vz) probability at this location (NOT normalized by the density)

INPUT:

vR - radial velocity (can be Quantity)

vz - vertical velocity (can be Quantity)

R - radius (can be Quantity)

z - height (can be Quantity)

gl - use Gauss-Legendre integration (True, currently the only option)

ngl - order of Gauss-Legendre integration

vTmax - sets integration limits to [0,vTmax] for integration over vT (default: 1.5)

OUTPUT:

p(vR,vz,R,z)

HISTORY:

2013-01-02 - Written - Bovy (IAS) 2018-01-12 - Added Gauss-Legendre integration prefactor vTmax/2 - Trick (MPA)

galpy.df.quasiisothermaldf.pvT

quasiisothermaldf.**pvT**(*vT*, *R*, *z*, *gl=True*, *ngl=20*, *nsigma=4.0*)

NAME:

pvT

PURPOSE:

calculate the marginalized vT probability at this location (NOT normalized by the density)

INPUT:

vT - tangential velocity (can be Quantity)

R - radius (can be Quantity)

z - height (can be Quantity)

gl - use Gauss-Legendre integration (True, currently the only option)

ngl - order of Gauss-Legendre integration

nsigma - sets integration limits to [-1,+1]*nsigma*sigma(R) for integration over vz and vR (default: 4)

OUTPUT:

p(vT,R,z)

HISTORY:

2012-12-22 - Written - Bovy (IAS) 2018-01-12 - Added Gauss-Legendre integration prefactor nsigma^2/4 - Trick (MPA)

galpy.df.quasiisothermaldf.pvTvz

quasiisothermaldf.**pvTvz**(*vT*, *vz*, *R*, *z*, *gl=True*, *ngl=20*, *nsigma=4.0*)

NAME:

pvTvz

PURPOSE:

calculate the marginalized (vT,vz) probability at this location (NOT normalized by the density)

INPUT:

vT - tangential velocity (can be Quantity)

vz - vertical velocity (can be Quantity)

R - radius (can be Quantity)

z - height (can be Quantity)

gl - use Gauss-Legendre integration (True, currently the only option)

ngl - order of Gauss-Legendre integration

nsigma - sets integration limits to [-1,+1]*nsigma*sigma_R(R) for integration over vR (default: 4)

OUTPUT:

p(vT,vz,R,z)

HISTORY:

2012-12-22 - Written - Bovy (IAS) 2018-01-12 - Added Gauss-Legendre integration prefactor nsigma/2 - Trick (MPA)

galpy.df.quasiisothermaldf.pvz

quasiisothermaldf.**pvz**(*vz*, *R*, *z*, *gl=True*, *ngl=20*, *nsigma=4.0*, *vTmax=1.5*, *_return_actions=False*, *_jr=None*, *_lz=None*, *_jz=None*, *_return_freqs=False*, *_rg=None*, *_kappa=None*, *_nu=None*, *_Omega=None*, *_sigmaR1=None*)

NAME:

pvz

PURPOSE:

calculate the marginalized vz probability at this location (NOT normalized by the density)

INPUT:

vz - vertical velocity (can be Quantity)

R - radius (can be Quantity)

z - height (can be Quantity)

gl - use Gauss-Legendre integration (True, currently the only option)

ngl - order of Gauss-Legendre integration

nsigma - sets integration limits to [-1,+1]*nsigma*sigma_R(R) for integration over vR (default: 4)

vTmax - sets integration limits to [0,vTmax] for integration over vT (default: 1.5)

OUTPUT:

p(vz,R,z)

HISTORY:

2012-12-22 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.sampleV

quasiisothermaldf.**sampleV**(*R*, *z*, *n=1*, ***kwargs*)

NAME:

sampleV

PURPOSE:

sample a radial, azimuthal, and vertical velocity at R,z

INPUT:

R - Galactocentric distance (can be Quantity)

z - height (can be Quantity)

n= number of distances to sample

OUTPUT:

list of samples

HISTORY:

2012-12-17 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.sampleV_interpolate

quasiisothermaldf.**sampleV_interpolate**(*R*, *z*, *R_pixel*, *z_pixel*, *num_std=3*, *R_min=None*, *R_max=None*, *z_max=None*, ***kwargs*)

NAME:

sampleV_interpolate

PURPOSE:

Given an array of R and z coordinates of stars, return the positions and their radial, azimuthal, and vertical velocity.

INPUT:

R - array of Galactocentric distance (can be Quantity)

z - array of height (can be Quantity)

R_pixel, z_pixel= the pixel size for creating the grid for interpolation (in natural unit)

num_std= number of standard deviation to be considered outliers sampled separately from interpolation

R_min, R_max, z_max= optional edges of the grid

OUTPUT:

coord_v= a numpy array containing the sampled velocity, (vR, vT, vz), where each row correspond to the row of (R,z)

HISTORY:

2018-08-10 - Written - Samuel Wong (University of Toronto)

galpy.df.quasiisothermaldf.sigmaR2

quasiisothermaldf.**sigmaR2**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

sigmaR2

PURPOSE:

calculate sigma_R^2 by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

sigma_R^2

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.sigmaRz

quasiisothermaldf.**sigmaRz**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

sigmaRz

PURPOSE:

calculate sigma_RZ^2 by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

sigma_Rz^2

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.sigmaT2

quasiisothermaldf.**sigmaT2**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

sigmaT2

PURPOSE:

calculate sigma_T^2 by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

sigma_T^2

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.sigmaz2

quasiisothermaldf.**sigmaz2**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

sigmaz2

PURPOSE:

calculate sigma_z^2 by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

sigma_z^2

HISTORY:

2012-07-30 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.surfacemass_z

quasiisothermaldf.**surfacemass_z**(*R*, *nz=7*, *zmax=1.0*, *fixed_quad=True*, *fixed_order=8*, ***kwargs*)

NAME:

surfacemass_z

PURPOSE:

calculate the vertically-integrated surface density

INPUT:

R - Galactocentric radius (can be Quantity)

fixed_quad= if True (default), use Gauss-Legendre integration

fixed_order= (20), order of GL integration to use

nz= number of zs to use to estimate

zmax= maximum z to use (can be Quantity)

density kwargs

OUTPUT:

Sigma(R)

HISTORY:

2012-08-30 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.tilt

quasiisothermaldf.**tilt**(*R*, *z*, *nsigma=None*, *mc=False*, *nmc=10000*, *gl=True*, *ngl=10*, ***kwargs*)

NAME:

tilt

PURPOSE:

calculate the tilt of the velocity ellipsoid by marginalizing over velocity

INPUT:

R - radius at which to calculate this (can be Quantity)

z - height at which to calculate this (can be Quantity)

OPTIONAL INPUT:

nsigma - number of sigma to integrate the velocities over

scipy.integrate.tplquad kwargs epsabs and epsrel

mc= if True, calculate using Monte Carlo integration

nmc= if mc, use nmc samples

gl= if True, calculate using Gauss-Legendre integration

ngl= if gl, use ngl-th order Gauss-Legendre integration for each dimension

OUTPUT:

tilt in rad

HISTORY:

2012-12-23 - Written - Bovy (IAS)

2017-10-28 - Changed return unit to rad - Bovy (UofT)

galpy.df.quasiisothermaldf._vmomentdensity

quasiisothermaldf.**_vmomentdensity**(*R*, *z*, *n*, *m*, *o*, *nsigma=None*, *mc=False*, *nmc=10000*, *_returnmc=False*, *_vrs=None*, *_vts=None*, *_vzs=None*, *_rawgausssamples=False*, *gl=False*, *ngl=10*, *_returngl=False*, *_glqeval=None*, *_return_actions=False*, *_jr=None*, *_lz=None*, *_jz=None*, *_return_freqs=False*, *_rg=None*, *_kappa=None*, *_nu=None*, *_Omega=None*, *_sigmaR1=None*, *_sigmaz1=None*, ***kwargs*)

Non-physical version of vmomentdensity, otherwise the same

Specific distribution functions

Quasi-isothermal DF

class galpy.df.**quasiisothermaldf**(*hr*, *sr*, *sz*, *hsr*, *hsz*, *pot=None*, *aA=None*, *cutcounter=False*, *_precomputerg=True*, *_precomputergrmax=None*, *_precomputergnLz=51*, *refr=1.0*, *lo=0.005681818181818182*, *ro=None*, *vo=None*)

Class that represents a 'Binney' quasi-isothermal DF

__init__(*hr*, *sr*, *sz*, *hsr*, *hsz*, *pot=None*, *aA=None*, *cutcounter=False*, *_precomputerg=True*, *_precomputergrmax=None*, *_precomputergnLz=51*, *refr=1.0*, *lo=0.005681818181818182*, *ro=None*, *vo=None*)

NAME:

__init__

PURPOSE:

Initialize a quasi-isothermal DF

INPUT:

hr - radial scale length (can be Quantity)

sr - radial velocity dispersion at the solar radius (can be Quantity)

sz - vertical velocity dispersion at the solar radius (can be Quantity)

hsr - radial-velocity-dispersion scale length (can be Quantity)

hsz - vertial-velocity-dispersion scale length (can be Quantity)

pot= Potential instance or list thereof

aA= actionAngle instance used to convert (x,v) to actions [must be an instance of an actionAngle class that computes (J,Omega,angle) for a given (x,v)]

cutcounter= if True, set counter-rotating stars' DF to zero

refr= reference radius for dispersions (can be different from ro) (can be Quantity)

lo= reference angular momentum below where there are significant numbers of retrograde stars (can be Quantity)

ro= distance from vantage point to GC (kpc; can be Quantity)

vo= circular velocity at ro (km/s; can be Quantity)

OTHER INPUTS:

_precomputerg= if True (default), pre-compute the rL(L)

_precomputergrmax= if set, this is the maximum R for which to pre-compute rg (default: 5*hr)

_precomputergnLz if set, number of Lz to pre-compute rg for (default: 51)

OUTPUT:

object

HISTORY:

2012-07-25 - Started - Bovy (IAS@MPIA)

2.4.7 The distribution function of a tidal stream

From Bovy 2014; see *Dynamical modeling of tidal streams*.

General instance routines

galpy.df.streamdf.__call__

streamdf.**__call__**(**args*, ***kwargs*)

NAME:

__call__

PURPOSE:

evaluate the DF

INPUT:

Either:

a) R,vR,vT,z,vz,phi ndarray [nobjects]

b) (Omegar,Omegaphi,Omegaz,angler,anglephi,anglez) tuple if aAInput

where:

Omegar - radial frequency

Omegaphi - azimuthal frequency

Omegaz - vertical frequency

angler - radial angle

anglephi - azimuthal angle

anglez - vertical angle

c) Orbit instance or list thereof

log= if True, return the natural log

aaInput= (False) if True, option b above

OUTPUT:

value of DF

HISTORY:

2013-12-03 - Written - Bovy (IAS)

The stream DF

class galpy.df.**streamdf**(*sigv, progenitor=None, pot=None, aA=None, useTM=False, tdisrupt=None, sigMeanOffset=6.0, leading=True, sigangle=None, deltaAngleTrack=None, nTrackChunks=None, nTrackIterations=None, progIsTrack=False, ro=None, vo=None, Vnorm=None, Rnorm=None, R0=8.0, Zsun=0.0208, vsun=[-11.1, 241.92, 7.25], multi=None, interpTrack=True, useInterp=True, nosetup=False, nospreadsetup=False, approxConstTrackFreq=False, useTMHessian=False, custom_transform=None*)

The DF of a tidal stream

__init__(*sigv, progenitor=None, pot=None, aA=None, useTM=False, tdisrupt=None, sigMeanOffset=6.0, leading=True, sigangle=None, deltaAngleTrack=None, nTrackChunks=None, nTrackIterations=None, progIsTrack=False, ro=None, vo=None, Vnorm=None, Rnorm=None, R0=8.0, Zsun=0.0208, vsun=[-11.1, 241.92, 7.25], multi=None, interpTrack=True, useInterp=True, nosetup=False, nospreadsetup=False, approxConstTrackFreq=False, useTMHessian=False, custom_transform=None*)

NAME:

__init__

PURPOSE:

Initialize a quasi-isothermal DF

INPUT:

sigv - radial velocity dispersion of the progenitor (can be Quantity)

tdisrupt= (5 Gyr) time since start of disruption (can be Quantity)

leading= (True) if True, model the leading part of the stream if False, model the trailing part

progenitor= progenitor orbit as Orbit instance (will be re-integrated, so don't bother integrating the orbit before)

progIsTrack= (False) if True, then the progenitor (x,v) is actually the (x,v) of the stream track at zero angle separation; useful when initializing with an orbit fit; the progenitor's position will be calculated

pot= Potential instance or list thereof

aA= actionAngle instance used to convert (x,v) to actions

useTM= (False) if set to an actionAngleTorus instance, use this to speed up calculations

sigMeanOffset= (6.) offset between the mean of the frequencies and the progenitor, in units of the largest eigenvalue of the frequency covariance matrix (along the largest eigenvector), should be positive; to model the trailing part, set leading=False

sigangle= (sigv/122/[1km/s]=1.8sigv in natural coordinates) estimate of the angle spread of the debris initially (can be Quantity)

deltaAngleTrack= (None) angle to estimate the stream track over (rad; or can be Quantity)

nTrackChunks= (floor(deltaAngleTrack/0.15)+1) number of chunks to divide the progenitor track in

nTrackIterations= Number of iterations to perform when establishing the track; each iteration starts from a previous approximation to the track in (x,v) and calculates a new track based on the deviation between the previous track and the desired track in action-angle coordinates; if not set, an appropriate value is determined based on the magnitude of the misalignment between stream and orbit, with larger numbers of iterations for larger misalignments

interpTrack= (might change), interpolate the stream track while setting up the instance (can be done by hand by calling self._interpolate_stream_track() and self._interpolate_stream_track_aA())

useInterp= (might change), use interpolation by default when calculating approximated frequencies and angles

nosetup= (False) if True, don't setup the stream track and anything else that is expensive

nospreadsetup= (False) if True, don't setup the spread around the stream track (only for nosetup is False)

multi= (None) if set, use multi-processing

Coordinate transformation inputs:

vo= (220) circular velocity to normalize velocities with [used to be Vnorm; can be Quantity]

ro= (8) Galactocentric radius to normalize positions with [used to be Rnorm; can be Quantity]

R0= (8) Galactocentric radius of the Sun (kpc) [can be different from ro; can be Quantity]

Zsun= (0.0208) Sun's height above the plane (kpc; can be Quantity)

vsun= ([-11.1,241.92,7.25]) Sun's motion in cylindrical coordinates (vR positive away from center) (can be Quantity)

custom_transform= (None) matrix implementing the rotation from (ra,dec) to a custom set of sky coordinates

approxConstTrackFreq= (False) if True, approximate the stream assuming that the frequency is constant along the stream (only works with useTM, for which this leads to a significant speed-up)

useTMHessian= (False) if True, compute the basic Hessian dO/dJ_prog using TM; otherwise use aA

OUTPUT:

object

HISTORY:

2013-09-16 - Started - Bovy (IAS)

2013-11-25 - Started over - Bovy (IAS)

galpy.df.streamdf.calc_stream_lb

streamdf.**calc_stream_lb**(*vo=None*, *ro=None*, *R0=None*, *Zsun=None*, *vsun=None*)

NAME:

calc_stream_lb

PURPOSE:

convert the stream track to observational coordinates and store

INPUT:

Coordinate transformation inputs (all default to the instance-wide values):

vo= circular velocity to normalize velocities with

ro= Galactocentric radius to normalize positions with

R0= Galactocentric radius of the Sun (kpc)

Zsun= Sun's height above the plane (kpc)

vsun= Sun's motion in cylindrical coordinates (vR positive away from center)

OUTPUT:

(none)

HISTORY:

2013-12-02 - Written - Bovy (IAS)

galpy.df.streamdf.callMarg

streamdf.**callMarg**(*xy*, ***kwargs*)

NAME:

callMarg

PURPOSE: evaluate the DF, marginalizing over some directions, in Galactocentric rectangular coordinates (or in observed l,b,D,vlos,pmll,pmbb) coordinates)

INPUT:

xy - phase-space point [X,Y,Z,vX,vY,vZ]; the distribution of the dimensions set to None is returned

interp= (object-wide interp default) if True, use the interpolated stream track

cindx= index of the closest point on the (interpolated) stream track if not given, determined from the dimensions given

nsigma= (3) number of sigma to marginalize the DF over (approximate sigma)

ngl= (5) order of Gauss-Legendre integration

lb= (False) if True, xy contains [l,b,D,vlos,pmll,pmbb] in [deg,deg,kpc,km/s,mas/yr,mas/yr] and the marginalized PDF in these coordinates is returned

vo= (220) circular velocity to normalize with when lb=True

ro= (8) Galactocentric radius to normalize with when lb=True

R0= (8) Galactocentric radius of the Sun (kpc)

Zsun= (0.0208) Sun's height above the plane (kpc)

vsun= ([-11.1,241.92,7.25]) Sun's motion in cylindrical coordinates (vR positive away from center)

OUTPUT:

p(xy) marginalized over missing directions in xy

HISTORY:

2013-12-16 - Written - Bovy (IAS)

galpy.df.streamdf.density_par

streamdf.**density_par**(*dangle*, *coord='apar'*, *tdisrupt=None*, ***kwargs*)

NAME:

density_par

PURPOSE:

calculate the density as a function of a parallel coordinate

INPUT:

dangle - parallel angle offset for this coordinate value

coord - coordinate to return the density in ('apar' [default], 'll','ra','customra','phi')

OUTPUT:

density(angle)

HISTORY:

2015-11-17 - Written - Bovy (UofT)

galpy.df.streamdf.estimateTdisrupt

streamdf.**estimateTdisrupt**(*deltaAngle*)

NAME:

estimateTdisrupt

PURPOSE:

estimate the time of disruption

INPUT:

deltaAngle- spread in angle since disruption

OUTPUT:

time in natural units

HISTORY:

2013-11-27 - Written - Bovy (IAS)

galpy.df.streamdf.find_closest_trackpoint

streamdf.**find_closest_trackpoint**(*R*, *vR*, *vT*, *z*, *vz*, *phi*, *interp=True*, *xy=False*, *usev=False*)

NAME:

find_closest_trackpoint

PURPOSE:

find the closest point on the stream track to a given point

INPUT:

R,vR,vT,z,vz,phi - phase-space coordinates of the given point

interp= (True), if True, return the index of the interpolated track

xy= (False) if True, input is X,Y,Z,vX,vY,vZ in Galactocentric rectangular coordinates; if xy, some coordinates may be missing (given as None) and they will not be used

usev= (False) if True, also use velocities to find the closest point

OUTPUT:

index into the track of the closest track point

HISTORY:

2013-12-04 - Written - Bovy (IAS)

galpy.df.streamdf.find_closest_trackpointLB

streamdf.**find_closest_trackpointLB**(*l*, *b*, *D*, *vlos*, *pmll*, *pmbb*, *interp=True*, *usev=False*)

NAME:

find_closest_trackpointLB

PURPOSE: find the closest point on the stream track to a given point in (l,b,...) coordinates

INPUT:

l,b,D,vlos,pmll,pmbb- coordinates in (deg,deg,kpc,km/s,mas/yr,mas/yr)

interp= (True) if True, return the closest index on the interpolated track

usev= (False) if True, also use the velocity components (default is to only use the positions)

OUTPUT:

index of closest track point on the interpolated or not-interpolated track

HISTORY:

2013-12-17- Written - Bovy (IAS)

galpy.df.streamdf.freqEigvalRatio

streamdf.**freqEigvalRatio**(*isotropic=False*)

NAME:

freqEigvalRatio

PURPOSE:

calculate the ratio between the largest and 2nd-to-largest (in abs) eigenvalue of sqrt(dO/dJ^T V_J dO/dJ) (if this is big, a 1D stream will form)

INPUT:

isotropic= (False), if True, return the ratio assuming an isotropic action distribution (i.e., just of dO/dJ)

OUTPUT:

ratio between eigenvalues of fabs(dO / dJ)

HISTORY:

2013-12-05 - Written - Bovy (IAS)

galpy.df.streamdf.gaussApprox

streamdf.**gaussApprox**(*xy*, ***kwargs*)

NAME:

gaussApprox

PURPOSE:

return the mean and variance of a Gaussian approximation to the stream DF at a given phase-space point in Galactocentric rectangular coordinates (distribution is over missing directions)

INPUT:

xy - phase-space point [X,Y,Z,vX,vY,vZ]; the distribution of the dimensions set to None is returned

interp= (object-wide interp default) if True, use the interpolated stream track

cindx= index of the closest point on the (interpolated) stream track if not given, determined from the dimensions given

lb= (False) if True, xy contains [l,b,D,vlos,pmll,pmbb] in [deg,deg,kpc,km/s,mas/yr,mas/yr] and the Gaussian approximation in these coordinates is returned

OUTPUT:

(mean,variance) of the approximate Gaussian DF for the missing directions in xy

HISTORY:

2013-12-12 - Written - Bovy (IAS)

galpy.df.streamdf.length

streamdf.**length**(*threshold=0.2*, *phys=False*, *ang=False*, *tdisrupt=None*, ***kwargs*)

NAME:

length

PURPOSE:

calculate the length of the stream

INPUT:

threshold - threshold down from the density near the progenitor at which to define the 'end' of the stream

phys= (False) if True, return the length in physical kpc

ang= (False) if True, return the length in sky angular arc length in degree

coord - coordinate to return the density in ('apar' [default], 'll','ra','customra','phi')

OUTPUT:

length (rad for parallel angle; kpc for physical length; deg for sky arc length)

HISTORY:

2015-12-22 - Written - Bovy (UofT)

galpy.df.streamdf.meanangledAngle

streamdf.**meanangledAngle**(*dangle*, *smallest=False*)

NAME:

meanangledAngle

PURPOSE:

calculate the mean perpendicular angle at a given angle

INPUT:

dangle - angle offset along the stream

smallest= (False) calculate for smallest eigenvalue direction rather than for middle

OUTPUT:

mean perpendicular angle

HISTORY:

2013-12-06 - Written - Bovy (IAS)

galpy.df.streamdf.meanOmega

streamdf.**meanOmega**(*dangle*, *oned=False*, *offset_sign=None*, *tdisrupt=None*)

NAME:

meanOmega

PURPOSE:

calculate the mean frequency as a function of angle, assuming a uniform time distribution up to a maximum time

INPUT:

dangle - angle offset

oned= (False) if True, return the 1D offset from the progenitor (along the direction of disruption)

offset_sign= sign of the frequency offset (shouldn't be set)

OUTPUT:

mean Omega

HISTORY:

2013-12-01 - Written - Bovy (IAS)

galpy.df.streamdf.meantdAngle

streamdf.**meantdAngle**(*dangle*)

NAME:

meantdAngle

PURPOSE:

calculate the mean stripping time at a given angle

INPUT:

dangle - angle offset along the stream

OUTPUT:

mean stripping time at this dangle

HISTORY:

2013-12-05 - Written - Bovy (IAS)

galpy.df.streamdf.misalignment

streamdf.**misalignment**(*isotropic=False*, ***kwargs*)

NAME:

misalignment

PURPOSE:

calculate the misalignment between the progenitor's frequency and the direction along which the stream disrupts

INPUT:

isotropic= (False), if True, return the misalignment assuming an isotropic action distribution

OUTPUT:

misalignment in rad

HISTORY:

2013-12-05 - Written - Bovy (IAS)

2017-10-28 - Changed output unit to rad - Bovy (UofT)

galpy.df.streamdf.pangledAngle

streamdf.**pangledAngle**(*angleperp*, *dangle*, *smallest=False*)

NAME:

pangledAngle

PURPOSE: return the probability of a given perpendicular angle at a given angle along the stream

INPUT:

angleperp - perpendicular angle

dangle - angle offset along the stream

smallest= (False) calculate for smallest eigenvalue direction rather than for middle

OUTPUT:

p(angle_perp|dangle)

HISTORY:

2013-12-06 - Written - Bovy (IAS)

galpy.df.streamdf.plotCompareTrackAAModel

streamdf.**plotCompareTrackAAModel**(***kwargs*)

NAME:

plotCompareTrackAAModel

PURPOSE:

plot the comparison between the underlying model's dOmega_perp vs. dangle_r (line) and the track in (x,v)'s dOmega_perp vs. dangle_r (dots; explicitly calculating the track's action-angle coordinates)

INPUT:

galpy.util.plot.plot kwargs

OUTPUT:

plot

HISTORY:

2014-08-27 - Written - Bovy (IAS)

galpy.df.streamdf.plotProgenitor

streamdf.**plotProgenitor**(*d1='x'*, *d2='z'*, **args*, ***kwargs*)

NAME:

plotProgenitor

PURPOSE:

plot the progenitor orbit

INPUT:

d1= plot this on the X axis ('x','y','z','R','phi','vx','vy','vz','vR','vt','ll','bb','dist','pmll','pmbb','vlos')

d2= plot this on the Y axis (same list as for d1)

scaleToPhysical= (False), if True, plot positions in kpc and velocities in km/s

galpy.util.plot.plot args and kwargs

OUTPUT:

plot to output device

HISTORY:

2013-12-09 - Written - Bovy (IAS)

galpy.df.streamdf.plotTrack

streamdf.**plotTrack**(*d1='x'*, *d2='z'*, *interp=True*, *spread=0*, *simple=True*, **args*, ***kwargs*)

NAME:

plotTrack

PURPOSE:

plot the stream track

INPUT:

d1= plot this on the X axis ('x','y','z','R','phi','vx','vy','vz','vR','vt','ll','bb','dist','pmll','pmbb','vlos')

d2= plot this on the Y axis (same list as for d1)

interp= (True) if True, use the interpolated stream track

spread= (0) if int > 0, also plot the spread around the track as spread x sigma

scaleToPhysical= (False), if True, plot positions in kpc and velocities in km/s

simple= (False), if True, use a simple estimate for the spread in perpendicular angle

galpy.util.plot.plotplot args and kwargs

OUTPUT:

plot to output device

HISTORY:

2013-12-09 - Written - Bovy (IAS)

galpy.df.streamdf.pOparapar

streamdf.**pOparapar**(*Opar*, *apar*, *tdisrupt=None*)

NAME:

pOparapar

PURPOSE:

return the probability of a given parallel (frequency,angle) offset pair

INPUT:

Opar - parallel frequency offset (array) (can be Quantity)

apar - parallel angle offset along the stream (scalar) (can be Quantity)

OUTPUT:

p(Opar,apar)

HISTORY:

2015-12-07 - Written - Bovy (UofT)

galpy.df.streamdf.ptdAngle

streamdf.**ptdAngle**(*t*, *dangle*)

NAME:

ptdangle

PURPOSE:

return the probability of a given stripping time at a given angle along the stream

INPUT:

t - stripping time

dangle - angle offset along the stream

OUTPUT:

p(tdldangle)

HISTORY:

2013-12-05 - Written - Bovy (IAS)

galpy.df.streamdf.sample

streamdf.**sample**(*n*, *returnaAdt=False*, *returndt=False*, *interp=None*, *xy=False*, *lb=False*)

NAME:

sample

PURPOSE:

sample from the DF

INPUT:

n - number of points to return

returnaAdt= (False) if True, return (Omega,angle,dt)

returndT= (False) if True, also return the time since the star was stripped

interp= (object-wide default) use interpolation of the stream track

xy= (False) if True, return Galactocentric rectangular coordinates

lb= (False) if True, return Galactic l,b,d,vlos,pmll,pmbb coordinates

OUTPUT:

(R,vR,vT,z,vz,phi) of points on the stream in 6,N array

HISTORY:

2013-12-22 - Written - Bovy (IAS)

galpy.df.streamdf.sigangledAngle

streamdf.**sigangledAngle**(*dangle*, *assumeZeroMean=True*, *smallest=False*, *simple=False*)

NAME:

sigangledAngle

PURPOSE:

calculate the dispersion in the perpendicular angle at a given angle

INPUT:

dangle - angle offset along the stream

assumeZeroMean= (True) if True, assume that the mean is zero (should be)

smallest= (False) calculate for smallest eigenvalue direction rather than for middle

simple= (False), if True, return an even simpler estimate

OUTPUT:

dispersion in the perpendicular angle at this angle

HISTORY:

2013-12-06 - Written - Bovy (IAS)

galpy.df.streamdf.sigOmega

streamdf.**sigOmega**(*dangle*)

NAME:

sigmaOmega

PURPOSE:

calculate the 1D sigma in frequency as a function of angle, assuming a uniform time distribution up to a maximum time

INPUT:

dangle - angle offset

OUTPUT:

sigma Omega

HISTORY:

2013-12-05 - Written - Bovy (IAS)

galpy.df.streamdf.sigtdAngle

streamdf.**sigtdAngle**(*dangle*)

NAME:

sigtdAngle

PURPOSE:

calculate the dispersion in the stripping times at a given angle

INPUT:

dangle - angle offset along the stream

OUTPUT:

dispersion in the stripping times at this angle

HISTORY:

2013-12-05 - Written - Bovy (IAS)

galpy.df.streamdf.subhalo_encounters

streamdf.**subhalo_encounters**(*venc=inf*, *sigma=0.6818181818181818*, *nsubhalo=0.3*, *bmax=0.025*, *yoon=False*)

NAME:

subhalo_encounters

PURPOSE:

estimate the number of encounters with subhalos over the lifetime of this stream, using a formalism similar to that of Yoon et al. (2011)

INPUT:

venc= (numpy.inf) count encounters with (relative) speeds less than this (relative radial velocity in cylindrical stream frame, unless yoon is True) (can be Quantity)

sigma= (150/220) velocity dispersion of the DM subhalo population (can be Quantity)

nsubhalo= (0.3) spatial number density of subhalos (can be Quantity)

bmax= (0.025) maximum impact parameter (if larger than width of stream) (can be Quantity)

yoon= (False) if True, use erroneous Yoon et al. formula

OUTPUT:

number of encounters

HISTORY:

2016-01-19 - Written - Bovy (UofT)

2.4.8 The distribution function of a gap in a tidal stream

From Sanders, Bovy, & Erkal 2015; see *Modeling gaps in streams*. Implemented as a subclass of `streamdf`. No full implementation is available currently, but the model can be set up and sampled as in the above paper.

General instance routines

The stream gap DF

class galpy.df.**streamgapdf**(**args*, ***kwargs*)

The DF of a gap in a tidal stream

__init__(**args*, ***kwargs*)

NAME:

__init__

PURPOSE:

Initialize the DF of a gap in a stellar stream

INPUT:

streamdf args and kwargs

Subhalo and impact parameters:

impactb= impact parameter (can be Quantity)

subhalovel= velocity of the subhalo shape=(3) (can be Quantity)

timpact time since impact (can be Quantity)

impact_angle= angle offset from progenitor at which the impact occurred (rad) (can be Quantity)

Subhalo: specify either 1(mass and size of Plummer sphere or 2(general spherical-potential object (kick is numerically computed)

1(GM= mass of the subhalo (can be Quantity)

rs= size parameter of the subhalo (can be Quantity)

2(subhalopot= galpy potential object or list thereof (should be spherical)

3(hernquist= (False) if True, use Hernquist kicks for GM/rs

deltaAngleTrackImpact= (None) angle to estimate the stream track over to determine the effect of the impact [similar to deltaAngleTrack] (rad)

nTrackChunksImpact= (floor(deltaAngleTrack/0.15)+1) number of chunks to divide the progenitor track in near the impact [similar to nTrackChunks]

nKickPoints= (30xnTrackChunksImpact) number of points along the stream to compute the kicks at (kicks are then interpolated); '30' chosen such that higherorderTrack can be set to False and get calculations accurate to > 99%

nokicksetup= (False) if True, only run as far as setting up the coordinate transformation at the time of impact (useful when using this in streampepperdf)

spline_order= (3) order of the spline to interpolate the kicks with

higherorderTrack= (False) if True, calculate the track using higher-order terms

OUTPUT:

object

HISTORY:

2015-06-02 - Started - Bovy (IAS)

Helper routines to compute kicks

galpy.df.impulse_deltav_plummer

galpy.df.**impulse_deltav_plummer**(*v*, *y*, *b*, *w*, *GM*, *rs*)

NAME:

impulse_deltav_plummer

PURPOSE:

calculate the delta velocity to due an encounter with a Plummer sphere in the impulse approximation; allows for arbitrary velocity vectors, but y is input as the position along the stream

INPUT:

v - velocity of the stream (nstar,3)

y - position along the stream (nstar)

b - impact parameter

w - velocity of the Plummer sphere (3)

GM - mass of the Plummer sphere (in natural units)

rs - size of the Plummer sphere

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-04-30 - Written based on Erkal's expressions - Bovy (IAS)

galpy.df.impulse_deltav_plummer_curvedstream

galpy.df.**impulse_deltav_plummer_curvedstream**(*v*, *x*, *b*, *w*, *x0*, *v0*, *GM*, *rs*)

NAME:

impulse_deltav_plummer_curvedstream

PURPOSE:

calculate the delta velocity to due an encounter with a Plummer sphere in the impulse approximation; allows for arbitrary velocity vectors, and arbitrary position along the stream

INPUT:

v - velocity of the stream (nstar,3)

x - position along the stream (nstar,3)

b - impact parameter

w - velocity of the Plummer sphere (3)

x0 - point of closest approach

v0 - velocity of point of closest approach

GM - mass of the Plummer sphere (in natural units)

rs - size of the Plummer sphere

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-05-04 - Written based on above - SANDERS

galpy.df.impulse_deltav_hernquist

galpy.df.**impulse_deltav_hernquist**(*v*, *y*, *b*, *w*, *GM*, *rs*)

NAME:

impulse_deltav_hernquist

PURPOSE:

calculate the delta velocity to due an encounter with a Hernquist sphere in the impulse approximation; allows for arbitrary velocity vectors, but y is input as the position along the stream

INPUT:

v - velocity of the stream (nstar,3)

y - position along the stream (nstar)

b - impact parameter

w - velocity of the Hernquist sphere (3)

GM - mass of the Hernquist sphere (in natural units)

rs - size of the Hernquist sphere

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-08-13 SANDERS, using Wyn Evans calculation

galpy.df.impulse_deltav_hernquist_curvedstream

galpy.df.**impulse_deltav_hernquist_curvedstream**(*v*, *x*, *b*, *w*, *x0*, *v0*, *GM*, *rs*)

NAME:

impulse_deltav_plummer_hernquist

PURPOSE:

calculate the delta velocity to due an encounter with a Hernquist sphere in the impulse approximation; allows for arbitrary velocity vectors, and arbitrary position along the stream

INPUT:

v - velocity of the stream (nstar,3)

x - position along the stream (nstar,3)

b - impact parameter

w - velocity of the Hernquist sphere (3)

x0 - point of closest approach

v0 - velocity of point of closest approach

GM - mass of the Hernquist sphere (in natural units)

rs - size of the Hernquist sphere

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-08-13 - SANDERS, using Wyn Evans calculation

galpy.df.impulse_deltav_general

galpy.df.**impulse_deltav_general**(*v*, *y*, *b*, *w*, *pot*)

NAME:

impulse_deltav_general

PURPOSE:

calculate the delta velocity to due an encounter with a general spherical potential in the impulse approximation; allows for arbitrary velocity vectors, but y is input as the position along the stream

INPUT:

v - velocity of the stream (nstar,3)

y - position along the stream (nstar)

b - impact parameter

w - velocity of the subhalo (3)

pot - Potential object or list thereof (should be spherical)

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-05-04 - SANDERS

2015-06-15 - Tweak to use galpy' potential objects - Bovy (IAS)

galpy.df.impulse_deltav_general_curvedstream

galpy.df.**impulse_deltav_general_curvedstream**(*v*, *x*, *b*, *w*, *x0*, *v0*, *pot*)

NAME:

impulse_deltav_general_curvedstream

PURPOSE:

calculate the delta velocity to due an encounter with a general spherical potential in the impulse approximation; allows for arbitrary velocity vectors and arbitrary shaped streams

INPUT:

v - velocity of the stream (nstar,3)

x - position along the stream (nstar,3)

b - impact parameter

w - velocity of the subhalo (3)

x0 - position of closest approach (3)

v0 - velocity of stream at closest approach (3)

pot - Potential object or list thereof (should be spherical)

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-05-04 - SANDERS

2015-06-15 - Tweak to use galpy' potential objects - Bovy (IAS)

galpy.df.impulse_deltav_general_orbitintegration

galpy.df.**impulse_deltav_general_orbitintegration**(*v*, *x*, *b*, *w*, *x0*, *v0*, *pot*, *tmax*, *galpot*, *tmaxfac=10.0*, *nsamp=1000*, *integrate_method='symplec4_c'*)

NAME:

impulse_deltav_general_orbitintegration

PURPOSE:

calculate the delta velocity to due an encounter with a general spherical potential NOT in the impulse approximation by integrating each particle in the underlying galactic potential; allows for arbitrary velocity vectors and arbitrary shaped streams.

INPUT:

v - velocity of the stream (nstar,3)

x - position along the stream (nstar,3)

b - impact parameter

w - velocity of the subhalo (3)

x0 - position of closest approach (3)

v0 - velocity of stream at closest approach (3)

pot - Potential object or list thereof (should be spherical)

tmax - maximum integration time

galpot - galpy Potential object or list thereof

nsamp(1000) - number of forward integration points

integrate_method= ('symplec4_c') orbit integrator to use (see Orbit.integrate)

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-08-17 - SANDERS

galpy.df.impulse_deltav_general_fullplummerintegration

galpy.df.**impulse_deltav_general_fullplummerintegration**(*v*, *x*, *b*, *w*, *x0*, *v0*, *galpot*, *GM*, *rs*, *tmaxfac=10.0*, *N=1000*, *integrate_method='symplec4_c'*)

NAME:

impulse_deltav_general_fullplummerintegration

PURPOSE:

calculate the delta velocity to due an encounter with a moving Plummer sphere and galactic potential relative to just in galactic potential

INPUT:

v - velocity of the stream (nstar,3)

x - position along the stream (nstar,3)

b - impact parameter

w - velocity of the subhalo (3)

x0 - position of closest approach (3)

v0 - velocity of stream at closest approach (3)

galpot - Galaxy Potential object

GM - mass of Plummer

rs - scale of Plummer

tmaxfac(10) - multiple of rs/fabs(w - v0) to use for time integration interval

N(1000) - number of forward integration points

integrate_method('symplec4_c') - orbit integrator to use (see Orbit.integrate)

OUTPUT:

deltav (nstar,3)

HISTORY:

2015-08-18 - SANDERS

2.5 Utilities (`galpy.util`)

2.5.1 galpy.util.config

Configuration module

galpy.util.config.set_ro

`galpy.util.config.`**`set_ro`**(*ro*)

NAME: set_ro

PURPOSE: set the global configuration value of ro (distance scale)

INPUT: ro - scale in kpc or astropy Quantity

OUTPUT: (none)

HISTORY: 2016-01-05 - Written - Bovy (UofT)

galpy.util.config.set_vo

`galpy.util.config.`**`set_vo`**(*vo*)

NAME: set_vo

PURPOSE: set the global configuration value of vo (velocity scale)

INPUT: vo - scale in km/s or astropy Quantity

OUTPUT: (none)

HISTORY: 2016-01-05 - Written - Bovy (UofT)

2.5.2 galpy.util.plot

Warning: Importing `galpy.util.plot` (or having it be imported by other `galpy` routines) with `seaborn` installed may change the `seaborn` plot style. If you don't like this, set the configuration parameter `seaborn-plotting-defaults` to False in the *configuration file*

Various plotting routines:

galpy.util.plot.dens2d

galpy.util.plot.**dens2d**(*X*, ***kwargs*)

NAME:

dens2d

PURPOSE:

plot a 2d density with optional contours

INPUT:

first argument is the density

matplotlib.pyplot.imshow keywords (see http://matplotlib.sourceforge.net/api/axes_api.html#matplotlib.axes.Axes.imshow)

xlabel - (raw string!) x-axis label, LaTeX math mode, no $s needed

ylabel - (raw string!) y-axis label, LaTeX math mode, no $s needed

xrange

yrange

noaxes - don't plot any axes

overplot - if True, overplot

colorbar - if True, add colorbar

shrink= colorbar argument: shrink the colorbar by the factor (optional)

conditional - normalize each column separately (for probability densities, i.e., cntrmass=True)

gcf=True does not start a new figure (does change the ranges and labels)

Contours:

justcontours - if True, only draw contours

contours - if True, draw contours (10 by default)

levels - contour-levels

cntrmass - if True, the density is a probability and the levels are probability masses contained within the contour

cntrcolors - colors for contours (single color or array)

cntrlabel - label the contours

cntrlw, cntrls - linewidths and linestyles for contour

cntrlabelsize, cntrlabelcolors,cntrinline - contour arguments

cntrSmooth - use ndimage.gaussian_filter to smooth before contouring

onedhists - if True, make one-d histograms on the sides

onedhistcolor - histogram color

retAxes= return all Axes instances

retCont= return the contour instance

OUTPUT:

plot to output device, Axes instances depending on input

HISTORY:

2010-03-09 - Written - Bovy (NYU)

galpy.util.plot.end_print

galpy.util.plot.**end_print**(*filename*, ***kwargs*)

NAME:

end_print

PURPOSE:

saves the current figure(s) to filename

INPUT:

filename - filename for plot (with extension)

OPTIONAL INPUTS:

format - file-format

OUTPUT:

(none)

HISTORY:

2009-12-23 - Written - Bovy (NYU)

galpy.util.plot.hist

galpy.util.plot.**hist**(*x*, *xlabel=None*, *ylabel=None*, *overplot=False*, ***kwargs*)

NAME:

hist

PURPOSE:

wrapper around matplotlib's hist function

INPUT:

x - array to histogram

xlabel - (raw string!) x-axis label, LaTeX math mode, no $s needed

ylabel - (raw string!) y-axis label, LaTeX math mode, no $s needed

yrange - set the y-axis range

+all pyplot.hist keywords

OUTPUT: (from the matplotlib docs: http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.hist)

The return value is a tuple (n, bins, patches) or ([n0, n1, ...], bins, [patches0, patches1,...]) if the input contains multiple data

HISTORY:

2009-12-23 - Written - Bovy (NYU)

galpy.util.plot.plot

galpy.util.plot.**plot**(**args*, ***kwargs*)

NAME:

plot

PURPOSE:

wrapper around matplotlib's plot function

INPUT:

see http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot

xlabel - (raw string!) x-axis label, LaTeX math mode, no $s needed

ylabel - (raw string!) y-axis label, LaTeX math mode, no $s needed

xrange

yrange

scatter= if True, use pyplot.scatter and its options etc.

colorbar= if True, and scatter==True, add colorbar

crange - range for colorbar of scatter==True

clabel= label for colorbar

overplot=True does not start a new figure and does not change the ranges and labels

gcf=True does not start a new figure (does change the ranges and labels)

onedhists - if True, make one-d histograms on the sides

onedhistcolor, onedhistfc, onedhistec

onedhistxnormed, onedhistynormed - normed keyword for one-d histograms

onedhistxweights, onedhistyweights - weights keyword for one-d histograms

bins= number of bins for onedhists

semilogx=, semilogy=, loglog= if True, plot logs

OUTPUT:

plot to output device, returns what pyplot.plot returns, or 3 Axes instances if onedhists=True

HISTORY:

2009-12-28 - Written - Bovy (NYU)

galpy.util.plot.start_print

galpy.util.plot.**start_print**(*fig_width=5*, *fig_height=5*, *axes_labelsize=16*, *text_fontsize=11*, *legend_fontsize=12*, *xtick_labelsize=10*, *ytick_labelsize=10*, *xtick_minor_size=2*, *ytick_minor_size=2*, *xtick_major_size=4*, *ytick_major_size=4*)

NAME:

start_print

PURPOSE:

setup a figure for plotting

INPUT:

fig_width - width in inches

fig_height - height in inches

axes_labelsize - size of the axis-labels

text_fontsize - font-size of the text (if any)

legend_fontsize - font-size of the legend (if any)

xtick_labelsize - size of the x-axis labels

ytick_labelsize - size of the y-axis labels

xtick_minor_size - size of the minor x-ticks

ytick_minor_size - size of the minor y-ticks

OUTPUT:

(none)

HISTORY:

2009-12-23 - Written - Bovy (NYU)

galpy.util.plot.text

galpy.util.plot.**text**(**args*, ***kwargs*)

NAME:

text

PURPOSE:

thin wrapper around matplotlib's text and annotate

use keywords:

'bottom_left=True'

'bottom_right=True'

'top_left=True'

'top_right=True'

'title=True'

to place the text in one of the corners or use it as the title

INPUT:

see matplotlib's text (http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.text)

OUTPUT:

prints text on the current figure

HISTORY:

2010-01-26 - Written - Bovy (NYU)

galpy.util.plot.scatterplot

galpy.util.plot.**scatterplot**(*x*, *y*, **args*, ***kwargs*)

NAME:

scatterplot

PURPOSE:

make a 'smart' scatterplot that is a density plot in high-density regions and a regular scatterplot for outliers

INPUT:

x, y

xlabel - (raw string!) x-axis label, LaTeX math mode, no $s needed

ylabel - (raw string!) y-axis label, LaTeX math mode, no $s needed

xrange

yrange

bins - number of bins to use in each dimension

weights - data-weights

aspect - aspect ratio

conditional - normalize each column separately (for probability densities, i.e., cntrmass=True)

gcf=True does not start a new figure (does change the ranges and labels)

contours - if False, don't plot contours

justcontours - if True, only draw contours, no density

cntrcolors - color of contours (can be array as for dens2d)

cntrlw, cntrls - linewidths and linestyles for contour

cntrSmooth - use ndimage.gaussian_filter to smooth before contouring

levels - contour-levels; data points outside of the last level will be individually shown (so, e.g., if this list is descending, contours and data points will be overplotted)

onedhists - if True, make one-d histograms on the sides

onedhistx - if True, make one-d histograms on the side of the x distribution

onedhisty - if True, make one-d histograms on the side of the y distribution

onedhistcolor, onedhistfc, onedhistec

onedhistxnormed, onedhistynormed - normed keyword for one-d histograms

onedhistxweights, onedhistyweights - weights keyword for one-d histograms

cmap= cmap for density plot

hist= and edges= - you can supply the histogram of the data yourself, this can be useful if you want to censor the data, both need to be set and calculated using scipy.histogramdd with the given range

retAxes= return all Axes instances

OUTPUT:

plot to output device, Axes instance(s) or not, depending on input

HISTORY:

2010-04-15 - Written - Bovy (NYU)

`galpy` also contains a new matplotlib projection `'galpolar'` that can be used when working with older versions of matplotlib like `'polar'` to create a polar plot in which the azimuth increases clockwise (like when looking at the Milky Way from the north Galactic pole). In newer versions of matplotlib, this does not work, but the `'polar'` projection now supports clockwise azimuths by doing, e.g.,

```
>>> ax= pyplot.subplot(111,projection='polar')
>>> ax.set_theta_direction(-1)
```

2.5.3 galpy.util.conversion

Utility functions that provide conversions between galpy's *natural* units and *physical* units. These can be used to translate galpy outputs in natural coordinates to physical units by multiplying with the appropriate function.

These could also be used to figure out the conversion between different units. For example, if you want to know how many $\mathrm{GeV\,cm^{-3}}$ correspond to $1\,M_\odot\,\mathrm{pc^{-3}}$, you can calculate

```
>>> from galpy.util import conversion
>>> conversion.dens_in_gevcc(1.,1.)/conversion.dens_in_msolpc3(1.,1.)
# 37.978342941703616
```

or $1\,M_\odot\,\mathrm{pc^{-3}} \approx 40\,\mathrm{GeV\,cm^{-3}}$.

Also contains a few utility functions to deal with the `ro` and `vo` conversion parameters for galpy object or lists thereof.

Utility functions:

galpy.util.conversion.get_physical

`galpy.util.conversion.`**`get_physical`**(*obj*, *include_set=False*)

NAME:

get_physical

PURPOSE:

return the velocity and length units for converting between physical and internal units as a dictionary for any galpy object, so they can easily be fed to galpy routines

INPUT:

obj - a galpy object or list of such objects (e.g., a Potential, list of Potentials, Orbit, actionAngle instance, DF instance)

include_set= (False) if True, also include roSet and voSet, flags of whether the unit is explicitly set in the object

OUTPUT:

Dictionary {'ro':length unit in kpc,'vo':velocity unit in km/s}; note that this routine will *always* return these conversion units, even if the obj you provide does not have units turned on

HISTORY:

2019-08-03 - Written - Bovy (UofT)

galpy.util.conversion.physical_compatible

galpy.util.conversion.**physical_compatible**(*obj*, *other_obj*)

NAME:

physical_compatible

PURPOSE:

test whether the velocity and length units for converting between physical and internal units are compatible for two galpy objects

INPUT:

obj - a galpy object or list of such objects (e.g., a Potential, list of Potentials, Orbit, actionAngle instance, DF instance)

other_obj - another galpy object or list of such objects (e.g., a Potential, list of Potentials, Orbit, actionAngle instance, DF instance)

OUTPUT:

True if the units are compatible, False if not (compatible means that the units are the same when they are set for both objects)

HISTORY:

2020-04-22 - Written - Bovy (UofT)

Conversion functions:

galpy.util.conversion.dens_in_criticaldens

galpy.util.conversion.**dens_in_criticaldens**(*vo*, *ro*, *H=70.0*)

NAME:

dens_in_criticaldens

PURPOSE:

convert density to units of the critical density

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

H= (default: 70) Hubble constant in km/s/Mpc

OUTPUT:

conversion from units where vo=1. at ro=1. to units of the critical density

HISTORY:

2014-01-28 - Written - Bovy (IAS)

galpy.util.conversion.dens_in_gevcc

galpy.util.conversion.**dens_in_gevcc**(*vo*, *ro*)

NAME:

dens_in_gevcc

PURPOSE:

convert density to GeV / cm^3

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1. to GeV/cm^3

HISTORY:

2014-06-16 - Written - Bovy (IAS)

galpy.util.conversion.dens_in_meanmatterdens

galpy.util.conversion.**dens_in_meanmatterdens**(*vo*, *ro*, *H=70.0*, *Om=0.3*)

NAME:

dens_in_meanmatterdens

PURPOSE:

convert density to units of the mean matter density

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

OUTPUT:

conversion from units where vo=1. at ro=1. to units of the mean matter density

HISTORY:

2014-01-28 - Written - Bovy (IAS)

galpy.util.conversion.dens_in_msolpc3

galpy.util.conversion.**dens_in_msolpc3**(*vo*, *ro*)

NAME:

dens_in_msolpc3

PURPOSE:

convert density to Msolar / pc^3

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1. to Msolar/pc^3

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.force_in_2piGmsolpc2

galpy.util.conversion.**force_in_2piGmsolpc2**(*vo*, *ro*)

NAME:

force_in_2piGmsolpc2

PURPOSE:

convert a force or acceleration to 2piG x Msolar / pc^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.force_in_pcMyr2

galpy.util.conversion.**force_in_pcMyr2**(*vo*, *ro*)

NAME:

force_in_pcMyr2

PURPOSE:

convert a force or acceleration to pc/Myr^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.force_in_10m13kms2

galpy.util.conversion.**force_in_10m13kms2**(*vo*, *ro*)

NAME:

force_in_10m13kms2

PURPOSE:

convert a force or acceleration to 10^(-13) km/s^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2014-01-22 - Written - Bovy (IAS)

galpy.util.conversion.force_in_kmsMyr

galpy.util.conversion.**force_in_kmsMyr**(*vo*, *ro*)

NAME:

force_in_kmsMyr

PURPOSE:

convert a force or acceleration to km/s/Myr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.freq_in_Gyr

galpy.util.conversion.**freq_in_Gyr**(*vo*, *ro*)

NAME:

freq_in_Gyr

PURPOSE:

convert a frequency to 1/Gyr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.freq_in_kmskpc

galpy.util.conversion.**freq_in_kmskpc**(*vo*, *ro*)

NAME:

freq_in_kmskpc

PURPOSE:

convert a frequency to km/s/kpc

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.surfdens_in_msolpc2

galpy.util.conversion.**surfdens_in_msolpc2**(*vo*, *ro*)

NAME:

surfdens_in_msolpc2

PURPOSE:

convert a surface density to Msolar / pc^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.mass_in_msol

galpy.util.conversion.**mass_in_msol**(*vo*, *ro*)

NAME:

mass_in_msol

PURPOSE:

convert a mass to Msolar

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.mass_in_1010msol

galpy.util.conversion.**mass_in_1010msol**(*vo*, *ro*)

NAME:

mass_in_1010msol

PURPOSE:

convert a mass to 10^10 x Msolar

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.time_in_Gyr

galpy.util.conversion.**time_in_Gyr**(*vo*, *ro*)

NAME:

time_in_Gyr

PURPOSE:

convert a time to Gyr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2013-09-01 - Written - Bovy (IAS)

galpy.util.conversion.velocity_in_kpcGyr

galpy.util.conversion.**velocity_in_kpcGyr**(*vo*, *ro*)

NAME:

velocity_in_kpcGyr

PURPOSE:

convert a velocity to kpc/Gyr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2014-12-19 - Written - Bovy (IAS)

2.5.4 galpy.util.coords

Warning: galpy uses a left-handed Galactocentric coordinate frame, as is common in studies of the kinematics of the Milky Way. Care should be taken when using the coordinate transformation routines below for coordinates in a right-handed frame, the routines do not always apply and are only tested for the standard galpy left-handed frame.

Various coordinate transformation routines with fairly self-explanatory names:

galpy.util.coords.cov_dvrpmllbb_to_vxyz

galpy.util.coords.**cov_dvrpmllbb_to_vxyz**(*d*, *e_d*, *e_vr*, *pmll*, *pmbb*, *cov_pmllbb*, *l*, *b*, *plx=False*, *degree=False*)

NAME:

cov_dvrpmllbb_to_vxyz

PURPOSE:

propagate distance, radial velocity, and proper motion uncertainties to Galactic coordinates

INPUT:

d - distance [kpc, as/mas for plx]

e_d - distance uncertainty [kpc, [as/mas] for plx]

e_vr - low velocity uncertainty [km/s]

pmll - proper motion in l (*cos(b)) [[as/mas]/yr]

pmbb - proper motion in b [[as/mas]/yr]

cov_pmllbb - uncertainty covariance for proper motion [pmll is pmll x cos(b)]

l - Galactic longitude

b - Galactic lattitude

KEYWORDS:

plx - if True, d is a parallax, and e_d is a parallax uncertainty

degree - if True, l and b are given in degree

OUTPUT:

cov(vx,vy,vz) [3,3] or [:,3,3]

HISTORY:

2010-04-12 - Written - Bovy (NYU)

2020-09-21 - Adapted for array input - Mackereth (UofT)

galpy.util.coords.cov_galcenrect_to_galcencyl

galpy.util.coords.**cov_galcenrect_to_galcencyl**(*cov_galcenrect*, *phi*)

NAME:

cov_galcenrect_to_galcencyl

PURPOSE:

propagate uncertainties in galactocentric rectangular to galactocentric cylindrical coordinates

INPUT:

cov_galcenrect - uncertainty covariance in Galactocentric rectangular coords

OUTPUT:

cov(vR,vT,vz) [3,3]

HISTORY:

2018-03-22 - Written - Mackereth (LJMU)

2020-09-21- Moved to coords.py - Mackereth (UofT)

galpy.util.coords.cov_pmrapmdec_to_pmllpmbb

galpy.util.coords.**cov_pmrapmdec_to_pmllpmbb**(*cov_pmradec*, *ra*, *dec*, *degree=False*, *epoch=2000.0*)

NAME:

cov_pmrapmdec_to_pmllpmbb

PURPOSE:

propagate the proper motions errors through the rotation from (ra,dec) to (l,b)

INPUT:

covar_pmradec - uncertainty covariance matrix of the proper motion in ra (multplied with cos(dec)) and dec [2,2] or [:,2,2]

ra - right ascension

dec - declination

degree - if True, ra and dec are given in degrees (default=False)

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

covar_pmllbb [2,2] or [:,2,2] [pmll here is pmll x cos(b)]

HISTORY:

2010-04-12 - Written - Bovy (NYU)

2020-09-21 - Adapted for array input - Mackereth (UofT)

galpy.util.coords.cov_vxyz_to_galcencyl

galpy.util.coords.**cov_vxyz_to_galcencyl**(*cov_vxyz*, *phi*, *Xsun=1.0*, *Zsun=0.0*)

NAME:

cov_vxyz_to_galcencyl

PURPOSE:

propagate uncertainties in vxyz to galactocentric cylindrical coordinates

INPUT:

cov_vxyz - uncertainty covariance in U,V,W

phi - angular position of star in galactocentric cylindrical coords

OUTPUT:

cov(vR,vT,vz) [3,3]

HISTORY:

2018-03-22 - Written - Mackereth (LJMU)

2020-09-21- Moved to coords.py - Mackereth (UofT)

galpy.util.coords.cov_vxyz_to_galcenrect

galpy.util.coords.**cov_vxyz_to_galcenrect**(*cov_vxyz*, *Xsun=1.0*, *Zsun=0.0*)

NAME:

cov_vxyz_to_galcenrect

PURPOSE:

propagate uncertainties in vxyz to galactocentric rectangular coordinates

INPUT:

cov_vxyz - uncertainty covariance in U,V,W

OUTPUT:

cov(vx,vy,vz) [3,3]

HISTORY:

2018-03-22 - Written - Mackereth (LJMU)

2020-09-21- Moved to coords.py - Mackereth (UofT)

galpy.util.coords.custom_to_pmrapmdec

galpy.util.coords.**custom_to_pmrapmdec**(*pmphi1*, *pmphi2*, *phi1*, *phi2*, *T=None*, *degree=False*)

NAME:

custom_to_pmrapmdec

PURPOSE:

rotate proper motions in a custom set of sky coordinates (phi1,phi2) to ICRS (ra,dec)

INPUT:

pmphi1 - proper motion in custom (multplied with cos(phi2)) [mas/yr]

pmphi2 - proper motion in phi2 [mas/yr]

phi1 - custom longitude

phi2 - custom latitude

T= matrix defining the transformation in cartesian coordinates: new_rect = T dot old_rect where old_rect = [cos(dec)cos(ra), cos(dec)sin(ra), sin(dec)] and similar for new_rect

degree= (False) if True, phi1 and phi2 are given in degrees (default=False)

OUTPUT:

(pmra x cos(dec), dec) for vector inputs [:,2]

HISTORY:

2019-03-02 - Written - Nathaniel Starkman (UofT)

galpy.util.coords.custom_to_radec

galpy.util.coords.**custom_to_radec**(*phi1*, *phi2*, *T=None*, *degree=False*)

NAME:

custom_to_radec

PURPOSE:

rotate a custom set of sky coordinates (phi1, phi2) to (ra, dec) given the rotation matrix T for (ra, dec) -> (phi1, phi2)

INPUT:

phi1 - custom sky coord

phi2 - custom sky coord

T - matrix defining the transformation (ra, dec) -> (phi1, phi2)

degree - default: False. If True, phi1 and phi2 in degrees

OUTPUT:

(ra, dec) for vector inputs [:, 2]

HISTORY:

2018-10-23 - Written - Nathaniel (UofT)

galpy.util.coords.cyl_to_rect

galpy.util.coords.**cyl_to_rect**(*R*, *phi*, *Z*)

NAME:

cyl_to_rect

PURPOSE:

convert from cylindrical to rectangular coordinates

INPUT:

R, phi, Z - cylindrical coordinates

OUTPUT:

X,Y,Z

HISTORY:

2011-02-23 - Written - Bovy (NYU)

galpy.util.coords.cyl_to_rect_vec

galpy.util.coords.**cyl_to_rect_vec**(*vr*, *vt*, *vz*, *phi*)

NAME:

cyl_to_rect_vec

PURPOSE:

transform vectors from cylindrical to rectangular coordinate vectors

INPUT:

vr - radial velocity

vt - tangential velocity

vz - vertical velocity

phi - azimuth

OUTPUT:

vx,vy,vz

HISTORY:

2011-02-24 - Written - Bovy (NYU)

galpy.util.coords.cyl_to_spher

galpy.util.coords.**cyl_to_spher**(*R*, *Z*, *phi*)

NAME:

cyl_to_spher

PURPOSE:

convert from cylindrical to spherical coordinates

INPUT:

R, Z, phi- cylindrical coordinates

OUTPUT:

R, theta, phi - spherical coordinates

HISTORY:

2016-05-16 - Written - Aladdin

galpy.util.coords.cyl_to_spher_vec

galpy.util.coords.**cyl_to_spher_vec**(*vR*, *vT*, *vz*, *R*, *z*)

NAME:

cyl_to_spher_vec

PURPOSE:

transform vectors from cylindrical to spherical coordinates. vtheta is positive from pole towards equator.

INPUT:

vR - Galactocentric cylindrical radial velocity

vT - Galactocentric cylindrical tangential velocity

vz - Galactocentric cylindrical vertical velocity

R - Galactocentric cylindrical radius

z - Galactocentric cylindrical height

OUTPUT:

vr,vT,vtheta

HISTORY:

2020-07-01 - Written - James Lane (UofT)

galpy.util.coords.dl_to_rphi_2d

galpy.util.coords.**dl_to_rphi_2d**(*d*, *l*, *degree=False*, *ro=1.0*, *phio=0.0*)

NAME:

dl_to_rphi_2d

PURPOSE:

convert Galactic longitude and distance to Galactocentric radius and azimuth

INPUT:

d - distance

l - Galactic longitude [rad/deg if degree]

KEYWORDS:

degree= (False): l is in degrees rather than rad

ro= (1) Galactocentric radius of the observer

phio= (0) Galactocentric azimuth of the observer [rad/deg if degree]

OUTPUT:

(R,phi); phi in degree if degree

HISTORY:

2012-01-04 - Written - Bovy (IAS)

galpy.util.coords.galcencyl_to_XYZ

galpy.util.coords.**galcencyl_to_XYZ**(*R*, *phi*, *Z*, *Xsun=1.0*, *Zsun=0.0*, *_extra_rot=True*)

NAME:

galcencyl_to_XYZ

PURPOSE:

transform cylindrical Galactocentric coordinates to XYZ coordinates (wrt Sun)

INPUT:

R, phi, Z - Galactocentric cylindrical coordinates

Xsun - cylindrical distance to the GC (can be array of same length as R)

Zsun - Sun's height above the midplane (can be array of same length as R)

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

X,Y,Z

HISTORY:

2011-02-23 - Written - Bovy (NYU)

2017-10-24 - Allowed Xsun/Zsun to be arrays - Bovy (UofT)

galpy.util.coords.galcencyl_to_vxvyvz

galpy.util.coords.**galcencyl_to_vxvyvz**(*vR*, *vT*, *vZ*, *phi*, *vsun=[0.0, 1.0, 0.0]*, *Xsun=1.0*, *Zsun=0.0*, *_extra_rot=True*)

NAME:

galcencyl_to_vxvyvz

PURPOSE:

transform cylindrical Galactocentric coordinates to XYZ (wrt Sun) coordinates for velocities

INPUT:

vR - Galactocentric radial velocity

vT - Galactocentric tangential velocity

vZ - Galactocentric vertical velocity

phi - Galactocentric azimuth

vsun - velocity of the sun in the GC frame ndarray[3] (can be array of same length as vRg; shape [3,N])

Xsun - cylindrical distance to the GC (can be array of same length as vRg)

Zsun - Sun's height above the midplane (can be array of same length as vRg)

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

vx,vy,vz

HISTORY:

2011-02-24 - Written - Bovy (NYU)

2017-10-24 - Allowed vsun/Xsun/Zsun to be arrays - Bovy (NYU)

galpy.util.coords.galcenrect_to_XYZ

galpy.util.coords.**galcenrect_to_XYZ**(*X*, *Y*, *Z*, *Xsun=1.0*, *Zsun=0.0*, *_extra_rot=True*)

NAME:

galcenrect_to_XYZ

PURPOSE:

transform rectangular Galactocentric to XYZ coordinates (wrt Sun) coordinates

INPUT:

X, Y, Z - Galactocentric rectangular coordinates

Xsun - cylindrical distance to the GC (can be array of same length as X)

Zsun - Sun's height above the midplane (can be array of same length as X)

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

(X, Y, Z)

HISTORY:

2011-02-23 - Written - Bovy (NYU)

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

2017-10-24 - Allowed Xsun/Zsun to be arrays - Bovy (UofT)

2018-04-18 - Tweaked to be consistent with astropy's Galactocentric frame - Bovy (UofT)

galpy.util.coords.galcenrect_to_vxvyvz

galpy.util.coords.**galcenrect_to_vxvyvz**(*vXg, vYg, vZg, vsun=[0.0, 1.0, 0.0], Xsun=1.0, Zsun=0.0, _extra_rot=True*)

NAME:

galcenrect_to_vxvyvz

PURPOSE:

transform rectangular Galactocentric coordinates to XYZ coordinates (wrt Sun) for velocities

INPUT:

vXg - Galactocentric x-velocity

vYg - Galactocentric y-velocity

vZg - Galactocentric z-velocity

vsun - velocity of the sun in the GC frame ndarray[3] (can be array of same length as vXg; shape [3,N])

Xsun - cylindrical distance to the GC (can be array of same length as vXg)

Zsun - Sun's height above the midplane (can be array of same length as vXg)

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

[:,3]= vx, vy, vz

HISTORY:

2011-02-24 - Written - Bovy (NYU)

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

2017-10-24 - Allowed vsun/Xsun/Zsun to be arrays - Bovy (UofT)

2018-04-18 - Tweaked to be consistent with astropy's Galactocentric frame - Bovy (UofT)

galpy.util.coords.lb_to_radec

galpy.util.coords.**lb_to_radec**(*l*, *b*, *degree=False*, *epoch=2000.0*)

NAME:

lb_to_radec

PURPOSE:

transform from Galactic coordinates to equatorial coordinates

INPUT:

l - Galactic longitude

b - Galactic lattitude

degree - (Bool) if True, l and b are given in degree and ra and dec will be as well

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

ra,dec

For vector inputs [:,2]

HISTORY:

2010-04-07 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

2016-05-13 - Added support for using astropy's coordinate transformations and for non-standard epochs - Bovy (UofT)

galpy.util.coords.lb_to_radec

galpy.util.coords.**lbd_to_XYZ**(*l*, *b*, *d*, *degree=False*)

NAME:

lbd_to_XYZ

PURPOSE:

transform from spherical Galactic coordinates to rectangular Galactic coordinates (works with vector inputs)

INPUT:

l - Galactic longitude (rad)

b - Galactic lattitude (rad)

d - distance (arbitrary units)

degree - (bool) if True, l and b are in degrees

OUTPUT:

[X,Y,Z] in whatever units d was in

For vector inputs [:,3]

HISTORY:

2009-10-24- Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.coords.pmllpmbb_to_pmrapmdec

galpy.util.coords.**pmllpmbb_to_pmrapmdec**(*pmll*, *pmbb*, *l*, *b*, *degree=False*, *epoch=2000.0*)

NAME:

pmllpmbb_to_pmrapmdec

PURPOSE:

rotate proper motions in (l,b) into proper motions in (ra,dec)

INPUT:

pmll - proper motion in l (multplied with cos(b)) [mas/yr]

pmbb - proper motion in b [mas/yr]

l - Galactic longitude

b - Galactic lattitude

degree - if True, l and b are given in degrees (default=False)

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

(pmra x cos(dec),pmdec), for vector inputs [:,2]

HISTORY:

2010-04-07 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.coords.pmrapmdec_to_pmllpmbb

galpy.util.coords.**pmrapmdec_to_pmllpmbb**(*pmra*, *pmdec*, *ra*, *dec*, *degree=False*, *epoch=2000.0*)

NAME:

pmrapmdec_to_pmllpmbb

PURPOSE:

rotate proper motions in (ra,dec) into proper motions in (l,b)

INPUT:

pmra - proper motion in ra (multplied with cos(dec)) [mas/yr]

pmdec - proper motion in dec [mas/yr]

ra - right ascension

dec - declination

degree - if True, ra and dec are given in degrees (default=False)

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

(pmll x cos(b),pmbb) for vector inputs [:,2]

HISTORY:

2010-04-07 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.coords.pmrapmdec_to_custom

galpy.util.coords.**pmrapmdec_to_custom**(*pmra*, *pmdec*, *ra*, *dec*, *T=None*, *degree=False*)

NAME:

pmrapmdec_to_custom

PURPOSE:

rotate proper motions in (ra,dec) to proper motions in a custom set of sky coordinates (phi1,phi2)

INPUT:

pmra - proper motion in ra (multplied with cos(dec)) [mas/yr]

pmdec - proper motion in dec [mas/yr]

ra - right ascension

dec - declination

T= matrix defining the transformation: new_rect= T dot old_rect, where old_rect = [cos(dec)cos(ra),cos(dec)sin(ra),sin(dec)] and similar for new_rect

degree= (False) if True, ra and dec are given in degrees (default=False)

OUTPUT:

(pmphi1 x cos(phi2),pmph2) for vector inputs [:,2]

HISTORY:

2016-10-24 - Written - Bovy (UofT/CCA)

2019-03-09 - uses custom_to_radec - Nathaniel Starkman (UofT)

galpy.util.coords.pupv_to_vRvz

galpy.util.coords.**pupv_to_vRvz**(*pu*, *pv*, *u*, *v*, *delta=1.0*, *oblate=False*)

NAME:

pupv_to_vRvz

PURPOSE:

calculate cylindrical vR and vz from momenta in prolate or oblate confocal u and v coordinates for a given focal length delta

INPUT:

pu - u momentum

pv - v momentum

u - u coordinate

v - v coordinate

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(vR,vz)

HISTORY:

2017-12-04 - Written - Bovy (UofT)

galpy.util.coords.radec_to_lb

galpy.util.coords.**radec_to_lb**(*ra*, *dec*, *degree=False*, *epoch=2000.0*)

NAME:

radec_to_lb

PURPOSE:

transform from equatorial coordinates to Galactic coordinates

INPUT:

ra - right ascension

dec - declination

degree - (Bool) if True, ra and dec are given in degree and l and b will be as well

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

l,b

For vector inputs [:,2]

HISTORY:

2009-11-12 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

2016-05-13 - Added support for using astropy's coordinate transformations and for non-standard epochs - Bovy (UofT)

galpy.util.coords.radec_to_custom

galpy.util.coords.**radec_to_custom**(*ra*, *dec*, *T=None*, *degree=False*)

NAME:

radec_to_custom

PURPOSE:

transform from equatorial coordinates to a custom set of sky coordinates

INPUT:

ra - right ascension

dec - declination

T= matrix defining the transformation: new_rect= T dot old_rect, where old_rect = [cos(dec)cos(ra),cos(dec)sin(ra),sin(dec)] and similar for new_rect

degree - (Bool) if True, ra and dec are given in degree and l and b will be as well

OUTPUT:

custom longitude, custom latitude (with longitude -180 to 180)

For vector inputs [:,2]

HISTORY:

2009-11-12 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

2019-03-02 - adjusted angle ranges - Nathaniel (UofT)

galpy.util.coords.rectgal_to_sphergal

galpy.util.coords.**rectgal_to_sphergal**(*X*, *Y*, *Z*, *vx*, *vy*, *vz*, *degree=False*)

NAME:

rectgal_to_sphergal

PURPOSE:

transform phase-space coordinates in rectangular Galactic coordinates to spherical Galactic coordinates (can take vector inputs)

INPUT:

X - component towards the Galactic Center (kpc)

Y - component in the direction of Galactic rotation (kpc)

Z - component towards the North Galactic Pole (kpc)

vx - velocity towards the Galactic Center (km/s)

vy - velocity in the direction of Galactic rotation (km/s)

vz - velocity towards the North Galactic Pole (km/s)

degree - (Bool) if True, return l and b in degrees

OUTPUT:

(l,b,d,vr,pmll x cos(b),pmbb) in (rad,rad,kpc,km/s,mas/yr,mas/yr)

HISTORY:

2009-10-25 - Written - Bovy (NYU)

galpy.util.coords.rect_to_cyl

galpy.util.coords.**rect_to_cyl**(*X*, *Y*, *Z*)

NAME:

rect_to_cyl

PURPOSE:

convert from rectangular to cylindrical coordinates

INPUT:

X, Y, Z - rectangular coordinates

OUTPUT:

R,phi,z

HISTORY:

2010-09-24 - Written - Bovy (NYU)

2019-06-21 - Changed such that phi in [-pi,pi] - Bovy (UofT)

galpy.util.coords.rect_to_cyl_vec

galpy.util.coords.**rect_to_cyl_vec**(*vx*, *vy*, *vz*, *X*, *Y*, *Z*, *cyl=False*)

NAME:

rect_to_cyl_vec

PURPOSE:

transform vectors from rectangular to cylindrical coordinates vectors

INPUT:

vx -

vy -

vz -

X - X

Y - Y

Z - Z

cyl - if True, X,Y,Z are already cylindrical

OUTPUT:

vR,vT,vz

HISTORY:

2010-09-24 - Written - Bovy (NYU)

galpy.util.coords.rphi_to_dl_2d

galpy.util.coords.**rphi_to_dl_2d**(*R*, *phi*, *degree=False*, *ro=1.0*, *phio=0.0*)

NAME:

rphi_to_dl_2d

PURPOSE:

convert Galactocentric radius and azimuth to distance and Galactic longitude

INPUT:

R - Galactocentric radius

phi - Galactocentric azimuth [rad/deg if degree]

KEYWORDS:

degree= (False): phi is in degrees rather than rad

ro= (1) Galactocentric radius of the observer

phio= (0) Galactocentric azimuth of the observer [rad/deg if degree]

OUTPUT:

(d,l); phi in degree if degree

HISTORY:

2012-01-04 - Written - Bovy (IAS)

galpy.util.coords.Rz_to_coshucosv

galpy.util.coords.**Rz_to_coshucosv**(*R*, *z*, *delta=1.0*, *oblate=False*)

NAME:

Rz_to_coshucosv

PURPOSE:

calculate prolate confocal cosh(u) and cos(v) coordinates from R,z, and delta

INPUT:

R - radius

z - height

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(cosh(u),cos(v))

HISTORY:

2012-11-27 - Written - Bovy (IAS)

2017-10-11 - Added oblate coordinates - Bovy (UofT)

galpy.util.coords.Rz_to_uv

galpy.util.coords.**Rz_to_uv**(*R*, *z*, *delta=1.0*, *oblate=False*)

NAME:

Rz_to_uv

PURPOSE:

calculate prolate or oblate confocal u and v coordinates from R,z, and delta

INPUT:

R - radius

z - height

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(u,v)

HISTORY:

2012-11-27 - Written - Bovy (IAS)

2017-10-11 - Added oblate coordinates - Bovy (UofT)

galpy.util.coords.sphergal_to_rectgal

galpy.util.coords.**sphergal_to_rectgal**(*l*, *b*, *d*, *vr*, *pmll*, *pmbb*, *degree=False*)

NAME:

sphergal_to_rectgal

PURPOSE:

transform phase-space coordinates in spherical Galactic coordinates to rectangular Galactic coordinates (can take vector inputs)

INPUT:

l - Galactic longitude (rad)

b - Galactic lattitude (rad)

d - distance (kpc)

vr - line-of-sight velocity (km/s)

pmll - proper motion in the Galactic longitude direction (mu_l*cos(b)) (mas/yr)

pmbb - proper motion in the Galactic lattitude (mas/yr)

degree - (bool) if True, l and b are in degrees

OUTPUT:

(X,Y,Z,vx,vy,vz) in (kpc,kpc,kpc,km/s,km/s,km/s)

HISTORY:

2009-10-25 - Written - Bovy (NYU)

galpy.util.coords.spher_to_cyl

galpy.util.coords.**spher_to_cyl**(*r*, *theta*, *phi*)

NAME:

spher_to_cyl

PURPOSE:

convert from spherical to cylindrical coordinates

INPUT:

r, theta, phi - spherical coordinates

OUTPUT:

R, z, phi - spherical coordinates

HISTORY:

2016-05-20 - Written - Aladdin

galpy.util.coords.spher_to_cyl_vec

galpy.util.coords.**spher_to_cyl_vec**(*vr*, *vT*, *vtheta*, *theta*)

NAME:

spher_to_cyl_vec

PURPOSE:

transform vectors from spherical polar to cylindrical coordinates. vtheta is positive from pole towards equator, theta is 0 at pole

INPUT:

vr - Galactocentric spherical radial velocity

vT - Galactocentric spherical azimuthal velocity

vtheta - Galactocentric spherical polar velocity

theta - Galactocentric spherical polar angle

OUTPUT:

vR,vT,vz

HISTORY:

2020-07-01 - Written - James Lane (UofT)

galpy.util.coords.uv_to_Rz

galpy.util.coords.**uv_to_Rz**(*u*, *v*, *delta=1.0*, *oblate=False*)

NAME:

uv_to_Rz

PURPOSE:

calculate R and z from prolate confocal u and v coordinates

INPUT:

u - confocal u

v - confocal v

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(R,z)

HISTORY:

2012-11-27 - Written - Bovy (IAS)

2017-10-11 - Added oblate coordinates - Bovy (UofT)

galpy.util.coords.vrpmllpmbb_to_vxvyvz

galpy.util.coords.**vrpmllpmbb_to_vxvyvz**(*vr*, *pmll*, *pmbb*, *l*, *b*, *d*, *XYZ=False*, *degree=False*)

NAME:

vrpmllpmbb_to_vxvyvz

PURPOSE:

Transform velocities in the spherical Galactic coordinate frame to the rectangular Galactic coordinate frame (can take vector inputs)

INPUT:

vr - line-of-sight velocity (km/s)

pmll - proper motion in the Galactic longitude (mu_l * cos(b))(mas/yr)

pmbb - proper motion in the Galactic lattitude (mas/yr)

l - Galactic longitude

b - Galactic lattitude

d - distance (kpc)

XYZ - (bool) If True, then l,b,d is actually X,Y,Z (rectangular Galactic coordinates)

degree - (bool) if True, l and b are in degrees

OUTPUT:

(vx,vy,vz) in (km/s,km/s,km/s)

For vector inputs [:,3]

HISTORY:

2009-10-24 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.coords.vRvz_to_pupv

galpy.util.coords.**vRvz_to_pupv**(*vR*, *vz*, *R*, *z*, *delta=1.0*, *oblate=False*, *uv=False*)

NAME:

vRvz_to_pupv

PURPOSE:

calculate momenta in prolate or oblate confocal u and v coordinates from cylindrical velocities vR,vz for a given focal length delta

INPUT:

vR - radial velocity in cylindrical coordinates

vz - vertical velocity in cylindrical coordinates

R - radius

z - height

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

uv= (False) if True, the given R,z are actually u,v

OUTPUT:

(pu,pv)

HISTORY:

2017-11-28 - Written - Bovy (UofT)

galpy.util.coords.vxvyvz_to_galcencyl

galpy.util.coords.**vxvyvz_to_galcencyl**(*vx, vy, vz, X, Y, Z, vsun=[0.0, 1.0, 0.0], Xsun=1.0, Zsun=0.0, galcen=False, _extra_rot=True*)

NAME:

vxvyvz_to_galcencyl

PURPOSE:

transform velocities in XYZ coordinates (wrt Sun) to cylindrical Galactocentric coordinates for velocities

INPUT:

vx - U

vy - V

vz - W

X - X in Galactocentric rectangular coordinates

Y - Y in Galactocentric rectangular coordinates

Z - Z in Galactocentric rectangular coordinates

vsun - velocity of the sun in the GC frame ndarray[3]

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

galcen - if True, then X,Y,Z are in cylindrical Galactocentric coordinates rather than rectangular coordinates

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

vRg, vTg, vZg

HISTORY:

2010-09-24 - Written - Bovy (NYU)

galpy.util.coords.vxvyvz_to_galcenrect

galpy.util.coords.**vxvyvz_to_galcenrect**(*vx, vy, vz, vsun=[0.0, 1.0, 0.0], Xsun=1.0, Zsun=0.0, _extra_rot=True*)

NAME:

vxvyvz_to_galcenrect

PURPOSE:

transform velocities in XYZ coordinates (wrt Sun) to rectangular Galactocentric coordinates for velocities

INPUT:

vx - U

vy - V

vz - W

vsun - velocity of the sun in the GC frame ndarray[3]

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

[:,3]= vXg, vYg, vZg

HISTORY:

2010-09-24 - Written - Bovy (NYU)

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

2018-04-18 - Tweaked to be consistent with astropy's Galactocentric frame - Bovy (UofT)

galpy.util.coords.vxvyvz_to_vrpmllpmbb

galpy.util.coords.**vxvyvz_to_vrpmllpmbb**(*vx*, *vy*, *vz*, *l*, *b*, *d*, *XYZ=False*, *degree=False*)

NAME:

vxvyvz_to_vrpmllpmbb

PURPOSE:

Transform velocities in the rectangular Galactic coordinate frame to the spherical Galactic coordinate frame (can take vector inputs)

INPUT:

vx - velocity towards the Galactic Center (km/s)

vy - velocity in the direction of Galactic rotation (km/s)

vz - velocity towards the North Galactic Pole (km/s)

l - Galactic longitude

b - Galactic lattitude

d - distance (kpc)

XYZ - (bool) If True, then l,b,d is actually X,Y,Z (rectangular Galactic coordinates)

degree - (bool) if True, l and b are in degrees

OUTPUT:

(vr,pmll x cos(b),pmbb) in (km/s,mas/yr,mas/yr); pmll = mu_l * cos(b)

For vector inputs [:,3]

HISTORY:

2009-10-24 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.coords.XYZ_to_galcencyl

galpy.util.coords.**XYZ_to_galcencyl**(*X*, *Y*, *Z*, *Xsun=1.0*, *Zsun=0.0*, *_extra_rot=True*)

NAME:

XYZ_to_galcencyl

PURPOSE:

transform XYZ coordinates (wrt Sun) to cylindrical Galactocentric coordinates

INPUT:

X - X

Y - Y

Z - Z

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

R,phi,z

HISTORY:

2010-09-24 - Written - Bovy (NYU)

galpy.util.coords.XYZ_to_galcenrect

galpy.util.coords.**XYZ_to_galcenrect**(*X*, *Y*, *Z*, *Xsun=1.0*, *Zsun=0.0*, *_extra_rot=True*)

NAME:

XYZ_to_galcenrect

PURPOSE:

transform XYZ coordinates (wrt Sun) to rectangular Galactocentric coordinates

INPUT:

X - X

Y - Y

Z - Z

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

_extra_rot= (True) if True, perform an extra tiny rotation to align the Galactocentric coordinate frame with astropy's definition

OUTPUT:

(Xg, Yg, Zg)

HISTORY:

2010-09-24 - Written - Bovy (NYU)

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

2018-04-18 - Tweaked to be consistent with astropy's Galactocentric frame - Bovy (UofT)

galpy.util.coords.XYZ_to_lbd

galpy.util.coords.**XYZ_to_lbd**(*X*, *Y*, *Z*, *degree=False*)

NAME:

XYZ_to_lbd

PURPOSE:

transform from rectangular Galactic coordinates to spherical Galactic coordinates (works with vector inputs)

INPUT:

X - component towards the Galactic Center (in kpc; though this obviously does not matter))

Y - component in the direction of Galactic rotation (in kpc)

Z - component towards the North Galactic Pole (kpc)

degree - (Bool) if True, return l and b in degrees

OUTPUT:

[l,b,d] in (rad or degree,rad or degree,kpc)

For vector inputs [:,3]

HISTORY:

2009-10-24 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

2.5.5 galpy.util.ars.ars

galpy.util.ars.**ars**(*domain*, *isDomainFinite*, *abcissae*, *hx*, *hpx*, *nsamples=1*, *hxparams=()*, *maxn=100*)

ars: Implementation of the Adaptive-Rejection Sampling algorithm by Gilks & Wild (1992): Adaptive Rejection Sampling for Gibbs Sampling, Applied Statistics, 41, 337 Based on Wild & Gilks (1993), Algorithm AS 287: Adaptive Rejection Sampling from Log-concave Density Functions, Applied Statistics, 42, 701

Input:

domain - [.,.] upper and lower limit to the domain

isDomainFinite - [.,.] is there a lower/upper limit to the domain?

abcissae - initial list of abcissae (must lie on either side of the peak in hx if the domain is unbounded

hx - function that evaluates h(x) = ln g(x)

hpx - function that evaluates hp(x) = d h(x) / d x

nsamples - (optional) number of desired samples (default=1)

hxparams - (optional) a tuple of parameters for h(x) and h'(x)

maxn - (optional) maximum number of updates to the hull (default=100)

Output:

list with nsamples of samples from exp(h(x))

External dependencies:

math scipy scipy.stats

History: 2009-05-21 - Written - Bovy (NYU)

CHAPTER 3

Acknowledging galpy

If you use galpy in a publication, please cite the following paper

- *galpy: A Python Library for Galactic Dynamics*, Jo Bovy (2015), *Astrophys. J. Supp.*, **216**, 29 (arXiv/1412.3451).

and link to `http://github.com/jobovy/galpy`. Some of the code's functionality is introduced in separate papers:

- `galpy.actionAngle.EccZmaxRperiRap` and `galpy.orbit.Orbit` methods with `analytic=True`: Fast method for computing orbital parameters from *this section*: please cite Mackereth & Bovy (2018).
- `galpy.actionAngle.actionAngleAdiabatic`: please cite Binney (2010).
- `galpy.actionAngle.actionAngleStaeckel`: please cite Bovy & Rix (2013) and Binney (2012).
- `galpy.actionAngle.actionAngleIsochroneApprox`: please cite Bovy (2014).
- `galpy.df.streamdf`: please cite Bovy (2014).
- `galpy.df.streamgapdf`: please cite Sanders, Bovy, & Erkal (2016).
- `galpy.potential.ttensor` and `galpy.potential.rtide`: please cite Webb et al. (2019a).
- `galpy.potential.to_amuse`: please cite Webb et al. (2019b).

Please also send me a reference to the paper or send a pull request including your paper in the list of galpy papers on this page (this page is at doc/source/index.rst). Thanks!

CHAPTER 4

Papers using galpy

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z