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.
1.1 Installation

galpy can be installed using pip as:

> pip install galpy

or to upgrade without upgrading the dependencies:

> pip install -U --no-deps galpy

Some advanced features require the GNU Scientific Library (GSL; see below). If you want to use these, install the GSL first (or install it later and re-install using the upgrade command above).

galpy 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. A basic installation works with just the numpy/scipy/matplotlib stack. Some basic tests can be performed by executing:

> nosetests -v -w nose/

1.1.1 Advanced installation

certain advanced features require the GNU Scientific Library (GSL), with action calculations requiring version 1.14 or higher. On a Mac you can make sure that the correct architecture is installed using Homebrew as:

> brew install gsl --universal

You should be able to check your version using:

> gsl-config --version

Other advanced features, including calculating the normalization of certain distribution functions using Gauss-Legendre integration require numpy version 1.7.0 or higher.
1.2 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`.

1.2.1 Rotation curves

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

```python
>>> 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.

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

```python
>>> 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. The resulting rotation curve is approximately flat. To show the rotation curves of the three components do

```python
>>> 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

\[
\text{>>> from galpy.potential import MWPotential}
\]

1.2.2 Units in galpy

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 \(v= V/[220 \text{ km/s}]\) and all of the positions should be scaled as \(x = X/[8 \text{ kpc}]\) when using galpy’s natural units.

For convenience, a utility module `bovy_conversion` is included in galpy that helps in converting between physical units and natural units for various quantities. For example, in natural units the orbital time of a circular orbit at R=1 is \(2\pi\); in physical units this corresponds to

\[
\text{>>> from galpy.util import bovy_conversion}
\]

\[
\text{>>> print 2.*numpy.pi*bovy_conversion.time_in_Gyr(220.,8.)}
\]

\[0.223405444283\]
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

```python
>>> from galpy.potential import MWPotential, evaluatezforces
>>> -evaluatezforces(1.,1.1/8.,MWPotential)*bovy_conversion.force_in_pcMyr2(220.,8.)
2.3941221528330314
```

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

```python
>>> -evaluatezforces(1.,1.1/8.,MWPotential)*bovy_conversion.force_in_2piGmsolpc2(220.,8.)
84.681625645335686
```

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

bovy_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

```python
>>> MWPotential[1].dens(1.,0.)*bovy_conversion.dens_in_msolpc3(220.,8.)
0.0085853601686596628
```

or about $0.0085 M_\odot pc^{-3}$, in line with current measurements (e.g., 2012ApJ...756...89B).

### 1.2.3 Orbit integration

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

```python
>>> from galpy.orbit import Orbit
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,amp=1.,normalize=1.)
>>> o= Orbit(vxvv=[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 $ts$

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

Now we plot the resulting orbit as

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

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

```python
>>> o.plotE(xlabel=r'$t$', ylabel=r'$E(t)/E(0)$')
```
When we use a symplectic leapfrog integrator, we see that the energy error remains constant

```python
>>> 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)

```python
>>> ts = numpy.linspace(0,1000,10000)
>>> o.integrate(ts,mp)
>>> o.plot()
```
As before, we can also integrate orbits in combinations of potentials. Assuming $mp$, $np$, and $hp$ were defined as above, we can

```python
>>> ts = numpy.linspace(0,100,10000)
>>> o.integrate(ts,[mp,hp,np])
>>> o.plot()
```
Energy is again approximately conserved

```python
>>> o.plotE(xlabel=r'$t$',ylabel=r'$E(t)/E(0)$')
```
1.2.4 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

```python
>>> mp.plotEscpecur{Rrange=[0.01,10.],grid=1001)
```
or of the combination of potentials defined above

```python
>>> from galpy.potential import plotEscapercue
>>> plotEscapercue([mp, hp, np], Rrange=[0.01, 10], grid=1001)
```
1.3 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.

1.3.1 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

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

```python
>>> 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. galpy.potential.MWPotential is such a list of three Potential instances

```python
>>> from galpy.potential import MWPotential
>>> print MWPotential
[<galpy.potential_src.MiyamotoNagaiPotential.MiyamotoNagaiPotential instance at 0x1078d5c20>, ... instance at 0x1078d5c68>, <galpy.potential_src.TwoPowerSphericalPotential.HernquistPotential instance at 0x1078d5cb0>]
```

and we can evaluate the potential by using the evaluatePotentials function

```python
>>> from galpy.potential import evaluatePotentials
>>> evaluatePotentials(1.,0.,MWPotential)
-4.5525780402192924
```

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

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

which produces the following plot
Similarly, we can plot combinations of Potentials using `plotPotentials`, e.g.,

```python
>>> plotPotentials(MWPotential)
```
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

```python
>>> 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

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

but not further from the mid-plane

```python
>>> 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.bovy_conversion` module. For example, assuming a physical circular velocity of 220 km/s at $R=8$ kpc

```python
>>> from galpy.util import bovy_conversion
>>> mp.zforce(1.,0.125)*bovy_conversion.force_in_kmsMyr(220.,8.)
-3.3095671288657584 #km/s/Myr
>>> mp.zforce(1.,0.125)*bovy_conversion.force_in_2piGmsolpc2(220.,8.)
```

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

```python
>>> from galpy.potential import evaluateRforces
>>> evaluateRforces(1.,0.,MWPotential)
-1.0
>>> from galpy.potential import evaluatezforces
>>> evaluatezforces(1.,0.125,MWPotential)*bovy_conversion.force_in_2piGmsolpc2(220.,8.)
>>> -82.898379883714099 #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

```python
>>> mp.flattening(1.,0.125)
0.4549542914935209
>>> from galpy.potential import flattening
>>> flattening(MWPotential,1.,0.125)
0.5593251065691105
```

### 1.3.2 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

```python
>>> mp.dens(1.,0.)
1.114544383277576
and we can also calculate this using the Poisson equation

```python
>>> mp.dens(1.,0.,forcepoisson=True)
1.114544383277574
```

which are the same to machine precision

```python
>>> mp.dens(1.,0.,forcepoisson=True)-mp.dens(1.,0.)
-2.220446049250313e-16
```

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

```python
>>> from galpy.potential import evaluateDensities
>>> evaluateDensities(1.,0.,MWPotential)
0.71812049194200644
```

In physical coordinates, this becomes

```python
>>> evaluateDensities(1.,0.,MWPotential)*bovy_conversion.dens_in_msolpc3(220.,8.)
0.1262386383150029 #Msol / pc^3
```

We can also plot densities

```python
>>> from galpy.potential import plotDensities
>>> plotDensities(MWPotential,rmin=0.1,zmax=0.25,zmin=-0.25,nrs=101,nzs=101)
```
Another example of this is for an exponential disk potential

```python
>>> 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

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

The density is

```python
>>> dp.plotDensity(rmin=0.1,zmax=0.25,zmin=-0.25,nrs=101,nzs=101)
```
and the potential is

```python
dp.plot(rmin=0.1, zmin=-0.25, zmax=0.25)
```
Clearly, the potential is much less flattened than the density.

### 1.3.3 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

```python
>>> mp.vcirc(1.)
1.0
>>> mp.dvcircdR(1.)
-0.16377427566978
```

or, for lists of Potential instances

```python
>>> from galpy.potential import vcirc
>>> vcirc(MWPotential,1.)
1.0
>>> from galpy.potential import dvcircdR
>>> dvcircdR(MWPotential,1.)
0.012084123754590059
```

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

```python
>>> mp.omegac(0.8)
1.2784598203204887
>>> from galpy.potential import omegac
```
>>> omegac(MWPotential, 0.8)
1.2389547535552212

and the epicycle frequency

```python
>>> mp.epifreq(0.8)
1.7774973530267848
```

```python
>>> from galpy.potential import epifreq
>>> epifreq(MWPotential, 0.8)
1.8144833284444094
```
as well as the vertical frequency

```python
>>> mp.verticalfreq(1.0)
3.7859388972001828
```

```python
>>> from galpy.potential import verticalfreq
>>> verticalfreq(MWPotential, 1.)
3.0000000000000004
```

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

```python
>>> mp.lindbladR(5./3., m='corotation')  # args are pattern speed and m of pattern
0.6027911166042229 # ~ 5kpc
```

```python
>>> print mp.lindbladR(5./3., m=2)
None
```

```python
>>> 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).

### 1.3.4 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 (some of these are also given in the file README.dev in the galpy distribution):

1. Implement the new potential in a class that inherits from `galpy.potential.Potential`. The new class should have an `__init__` method that sets up the necessary parameters for the class. An amplitude parameter `amp` should be taken as an argument for this class and before performing any other setup, the `galpy.potential.Potential.__init__(self, amp=amp)` method should be called to setup the amplitude. 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(R, z, phi=0, t=0, dR=0, dphi=0)` which evaluates the potential itself (without the amp factor, which is added in the `__call__` method of the general `Potential` class). This function should also call the relevant derivatives if `dR` or `dphi` is not equal to zero (this is used only in some of the razor-thin disk distribution functions, so doing this properly is not that important).

   - `_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 (d² potential / d R²).

• \_z2deriv(self, R, z, phi=0., t=0.) which evaluates the second (cylindrical) vertical derivative of the potential (d² potential / d z²).

• \_Rzderiv(self, R, z, phi=0., t=0.) which evaluates the mixed (cylindrical) radial and vertical derivative of the potential (d² 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).

• \_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 (d² potential / d φ²; 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 (d² potential / d R d φ; assumed zero if not given).

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.

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.

2. To add a C implementation of the potential, implement it in a .c file under potential_src/potential_c_ext. Look at potential_src/potential_c_ext/LogarithmicHaloPotential.c for the right format for 3D, axisymmetric potentials, or at potential_src/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). 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_src/potential_c_ext/galpy_potentials.h.

3. Add the potential’s function declarations to potential_src/potential_c_ext/galpy_potentials.h

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

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

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

8. (for using the `actionAngleStaeckel` methods in C) Edit the code in `actionAngle_src/actionAngle_c_ext/actionAngle.c` to parse the new potential (in the `parse_actionAngleArgs` function).

9. 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).

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.

### 1.4 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.

#### 1.4.1 Types of disk distribution functions

galpy contains the following distribution functions for razor-thin disks: `galpy.df.dehnendf` and `galpy.df.shudf`. These are the distribution functions of Dehnen (1999AJ....118.1201D) and Shu (1969ApJ...158..505S). Everything shown below for `dehnendf` can also be done for `shudf`.

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

```python
>>> 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,

```python
>>> 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

```python
>>> 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 $R=1$.

```python
>>> 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)
>>> from galpy.util.bovy_plot import bovy_dens2d
>>> bovy_dens2d(dfro,origin='lower',cmap='gist_yarg',contours=True,xrange=[-0.7,0.7],yrange=[0.4,1.6])
```
1.4.2 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)

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

and the velocity dispersions

```python
>>> numpy.sqrt(dfc.sigmaR2(1.))
0.19321086259083936
>>> numpy.sqrt(dfc.sigmaT2(1.))
0.15084122011271159
```

and their ratio

```python
>>> 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

```python
>>> 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

```python
>>> 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.,

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

We can plot the surface-mass density as follows

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

```python
>>> 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

```python
>>> 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

```python
>>> 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

```python
>>> dfc.asymmetricdrift(1.)
0.09000000000000024
```

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

```python
>>> 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).

### 1.4.3 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)

```python
>>> 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

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

and the mean rotation velocity is

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

and (correct) asymmetric drift

```python
>>> 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.

### 1.4.4 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

```python
>>> dfc = dehnendf(beta=0.)
>>> dfc.oortA(1.)
0.43190780889218749
>>> dfc.oortB(1.)
-0.48524496090228575
>>> dfc.oortC(1.)
0.0
>>> dfc.oortK(1.)
0.0
```

The $K$ and $C$ Oort constants are zero for axisymmetric DFs

```python
>>> 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

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

For the cold DF from above the approximation is much better
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)

```python
>>> 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

```python
>>> 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.,

![Figure 8](image)

We can also plot the spatial distribution of the sampled disk

1.4.5 Sampling data from the DF
```python
>>> xs = [e.x() for e in o]
>>> ys = [e.y() for e in o]
>>> figure()
>>> plot(xs,ys,',')

E.g.,

![Figure 9](image)

We can also sample points in a specific radial range (might take a few minutes)
```
```python
>>> 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 $r_{\text{range}}=[0.95,1.05]$ 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

```python
>>> dfc = dehnendf(beta=0., correct=True)
>>> vrvt = dfc.sampleVRVT(1.)
>>> 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

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

which is very fast. We can histogram these

```python
>>> 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

```python
>>> 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
1.4.6 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([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)]

>>> int=[[o.integrate(ts, [lp, dp]) for o in j] for j in ins]
```
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 \( \text{dp.tform()} \) for each of the orbits

```python
>>> 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

```python
>>> from galpy.util.bovy_plot import bovy_dens2d
>>> bovy_dens2d(out, origin='lower', cmap=' gist_yarg', contours=True, xrange=[-0.7, 0.7], yrange=[0.4, 1.6])
```

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.

## 1.5 A closer look at orbit integration

### 1.5.1 Orbit 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 \( \text{vxvv}=[R,v_R,v_T,z,v_z,\phi] \) or one can choose not to specify the azimuth of the orbit and initialize with \( \text{vxvv}=[R,v_R,v_T,z,v_z] \). 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,

```python
>>> o = Orbit(vxvv=[1.0, 0.1, 1.1, 0., 0.1, 0.])
```
initializes a fully three-dimensional orbit, while

```python
>>> o = Orbit(vxvv=[1.0, 0.1, 1.1, 0., 0.])
```
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 \( \text{vxvv}=[R,v_R,v_T,\phi] \) or choose not to track the azimuth and initialize as \( \text{vxvv}=[R,v_R,v_T] \).

In one dimension we simply initialize as \( \text{vxvv}=[x,v_x] \).

For orbit integration and characterization of observed stars or clusters, initial conditions can also be specified directly as observed quantities when \( \text{radec=True} \) is set. In this case a full three-dimensional orbit is initialized as \( \text{vxvv}=[\text{RA}, \text{Dec}, \text{distance}, \text{pmRA}, \text{pmDec}, \text{Vlos}] \) where RA and Dec are expressed in degrees, the distance is expressed in kpc, proper motions are expressed in mas/yr (\( \text{pmra} = \text{pmra}' \cdot \cos[\text{Dec}] \)), and the line-of-sight velocity is given in km/s. These observed coordinates are translated to the Galactocentric cylindrical coordinate frame by assuming a Solar motion that can be specified as either \( \text{solarmotion=hogg} \) (default; 2005ApJ...629..268H), \( \text{solarmotion=dehnen} \) (1998MNRAS.298..387D) or \( \text{solarmotion=shoenrich} \) (2010MNRAS.403.1829S). A circular velocity can be specified as \( \text{vo=235} \) in km/s and a value for the distance between the Galactic center and the Sun can be given as \( \text{ro=8.5} \) in kpc. While the inputs are given in physical units, the orbit is initialized assuming a circular velocity of one at the distance of the Sun.

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

### 1.5.2 Orbit integration

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

```python
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> o= Orbit(vxvv=[1.0, 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.

### 1.5.3 Displaying the orbit

After integrating the orbit, it can be displayed by using the \texttt{plot()} function. The quantities that are plotted when \texttt{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,v_R) is shown otherwise; in 1D \( (x,v_x) \) is shown. E.g., for the example given above,
Other projections of the orbit can be displayed by specifying the quantities to plot. E.g.,

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

gives the projection onto the plane of the orbit:
while

```python
>>> 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

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

which shows
or RA and Dec

```python
>>> 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.

### 1.5.4 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:

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

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

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

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

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

```python
>>> o.plotE()
gives
```
We can specify another quantity to plot the energy against by specifying `dl=`. We can also show the vertical energy, for example, as a function of R

```python
>>> o.plotEz(dl='R')
```
Often, a better approximation to an integral of the motion is given by $E_z/\sqrt{\text{density}(R)}$. We refer to this quantity as $E_z J_z$ and we can plot its behavior.

```python
>>> o.plotEzJz(d1='R')
```
1.5.5 Accessing the raw orbit

The value of \( \text{R, vR, vT, z, vz, x, vx, y, vy, phi, and vphi} \) at any time can be obtained by calling the corresponding function with as argument the time (the same holds for other coordinates \( \text{ra, dec, pmra, pmdec, vra, vdec, ll, bb, pmll, pmbb, vll, vbb, vlos, dist, helioX, helioY, helioZ, U, V, and W} \)). 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

\[
\begin{align*}
&\text{>>> o.R(1.)} \\
&1.1545076874679474 \\
&\text{>>> o.phi(99.)} \\
&88.105603035901169 \\
&\text{>>> o.ra(2.,obs=[8.,0.,0.],ro=8.)} \\
&\text{array([ 285.76403985])} \\
&\text{>>> o.helioX(5.)} \\
&\text{array([ 1.24888927])} \\
&\text{>>> o.pmll(10.,obs=[8.,0.,0.,245.,0.],ro=8.,vo=230.)} \\
&\text{array([-6.45263888])}
\end{align*}
\]

We can also initialize an \texttt{Orbit} instance using the phase-space position of another \texttt{Orbit} instance evaluated at time \( t \). For example,

\[
\begin{align*}
&\text{>>> newOrbit = o(10.)}
\end{align*}
\]

will initialize a new \texttt{Orbit} instance with as initial condition the phase-space position of orbit \texttt{o} at \texttt{time=10.}. 

The whole orbit can also be obtained using the function `getOrbit`

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

which returns a matrix of phase-space points with dimensions [ntimes,ndim].

### 1.5.6 Fast orbit integration

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`

```python
>>> 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`

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).

For most applications I recommend `dopr54_c`. For example, compare

```python
>>> o= Orbit(vxvv=[1.,0.1,1.1,0.,0.1])
>>> timeit(o.integrate(ts,mp))
1 loops, best of 3: 553 ms per loop
>>> timeit(o.integrate(ts,mp,method='dopr54_c'))
galpyWarning: Using C implementation to integrate orbits
10 loops, best of 3: 25.6 ms per loop
```

As this example shows, galpy will issue a warning that C is being used. Speed-ups by a factor of 20 are typical.

### 1.5.7 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) at

http://www.mpia-hd.mpg.de/homes/rix/Data/Dierickx-etal-tab2.txt

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 each orbit and save its eccentricity (running this for all 30,000-ish stars will take about half an hour)
We then find the following eccentricity distribution

![Eccentricity Distribution](image)

The eccentricity calculated by galpy compare well with those calculated by Dierickx et al., except for a few objects.
The script that calculates and plots everything can be downloaded here.

## 1.6 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.

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, for extremely fast action-angle calculations (see below).

### 1.6.1 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

```python
>>> 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 \((J_R, L_Z, J_Z)\)

```python
>>> aAI(1.,0.1,1.1,0.1,0.)
(array([ 0.00713759]), array([ 1.1]), array([ 0.00553155]))
```

or for a more eccentric orbit

```python
>>> 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

```python
>>> 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

```python
>>> 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, \(\Omega_{\phi} = \text{sgn}(L_Z)\Omega_Z\), such that the last two frequencies are the same.

We obtain the angles as well by calling

```python
>>> 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

```python
>>> 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

```python
>>> 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

```python
>>> plot(ts,jfa[6],’b.’)
>>> plot(ts,jfa[7],’g.’)
>>> plot(ts,jfa[8],’r.’)
```
1.6.2 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

```python
>>> 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

```python
>>> 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([ 0.2574507]),
 array([ 0.87630459]),
 array([ 0.60872881]),
 array([ 0.60872881]))
>>> aAS.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.)
(array([ 0.22022112]),
 array([ 1.3]),
 array([ 0.2574507]),
 array([ 0.87630459]),
 array([ 0.60872881]),
 array([ 0.60872881]))
```
We can again check that the actions are conserved along the orbit and that the angles increase linearly with time:

```python
>>> 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

```python
>>> 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

![Action-angle coordinates](image)

showing that the actions are all conserved. The angles again increase linearly with time

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

```python
>>> 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

```python
>>> 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]))
```

```python
>>> aAS.actionsFreqsAngles(1., 0.5, 1.3, 0.2, 0.1, 0.)
(array([ 0.13769498]),
 array([ 1.3]),
 array([ 0.02574507]),
 array([ 1.29136096]),
```

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

```python
>>> 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

```python
>>> aAI.actionsFreqsAngles(1., 0.5, 1.3, 0.2, 0.1, 0.)
(array([ 0.13769498]),
 array([ 1.3]),
 array([ 0.02574507]),
 array([ 1.29136096]),
```

```python
>>> aAS.actionsFreqsAngles(1., 0.5, 1.3, 0.2, 0.1, 0.)
(array([ 0.13769498]),
 array([ 1.3]),
 array([ 0.02574507]),
 array([ 1.29136096]),
```
or more explicitly comparing the two

```python
>>> [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([ -4.53851845e-10]),
 array([-1.65965242e-10]),
 array([ 9.0479645e-08]),
 array([ 9.0479649e-08])]
```

### 1.6.3 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

```python
>>> from galpy.potential import MWPotential
>>> from galpy.actionAngle import actionAngleAdiabatic

```aAA= actionAngleAdiabatic(pot=MWPotential)`

and evaluation then proceeds similarly as before

```python
>>> aAA(1.,0.1,1.1,0.,0.05)
(0.011551694768963469, 1.1, 0.00042376727426256727)
```

We can again check that the actions are conserved along the orbit

```python
>>> from galpy.orbit import Orbit

```ts=numpy.linspace(0.,100.,1001)

```o= Orbit([1.,0.1,1.1,0.,0.05])
>>> o.integrate(ts,MWPotential)
>>> 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

```python
>>> timeit(aAA(1.,0.1,1.1,0.,0.05))
10 loops, best of 3: 48.7 ms per loop
```aAA= actionAngleAdiabatic(pot=MWPotential,c=True)`

```python
>>> timeit(aAA(1.,0.1,1.1,0.,0.05))
1000 loops, best of 3: 1.2 ms per loop
```
or about a 40 times speed-up. For arrays the speed-up is even more impressive

```python
>>> s= numpy.ones(100)
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
```

### 1.6. Action-angle coordinates 57
1000 loops, best of 3: 1.8 ms per loop
>>> aAA = actionAngleAdiabatic(pot=MWPotential)  #back to no C
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
1 loops, best of 3: 4.94 s per loop

or a speed-up of 2700! Back to the previous example, you can run it with c=True to speed up the computation

```python
>>> aAA = actionAngleAdiabatic(pot=MWPotential, 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

```python
>>> 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

![Graph](image)

The radial action is conserved to about half a percent, the vertical action to two percent.

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.

```python
>>> o.zmax()*8.
0.1561562486879895
```

For orbits that reach distances of a kpc and more from the plane, the adiabatic approximation does not work as well. For example,
```python
>>> o = Orbit([1., 0.1, 1.1, 0., 0.25])
>>> o.integrate(ts, MWPotential)
>>> o.zmax() * 8.
1.1288142099238863
```

and we can again calculate the actions along the orbit

```python
>>> 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]))))
```

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.

### 1.6.4 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 $\Delta$ 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 MWPotential 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)

```python
>>> from galpy.actionAngle import estimateDeltaStaeckel
>>> estimateDeltaStaeckel(o.R(ts), o.z(ts), pot=MWPotential)
0.54421090762027347
```

We will use $\Delta = 0.55$ in what follows. We set up the actionAngleStaeckel object

```python
>>> aAS= actionAngleStaeckel(pot=MWPotential, delta=0.55, c=False)  # c=True is the default
```

and calculate the actions

```python
>>> aAS(o.R(), o.vR(), o.vT(), o.z(), o.vz())
(0.015760720988339319, 1.1000000000000001, 0.013466290557851267)
```

The adiabatic approximation from above gives

```python
>>> aAA(o.R(), o.vR(), o.vT(), o.z(), o.vz())
(0.0138915441284973, 1.1000000000000001, 0.01383357354294852)
```

The actionAngleStaeckel calculations are sped up in two ways. First, the action integrals can be calculated using Gaussian quadrature by specifying `fixed_quad=True`

```python
>>> aAS(o.R(), o.vR(), o.vT(), o.z(), o.vz(), fixed_quad=True)
(0.015767954890517084, 1.1000000000000001, 0.013468235165983522)
```

which in itself leads to a ten times speed up

```python
>>> timeit(aAS(o.R(), o.vR(), o.vT(), o.z(), o.vz(), fixed_quad=False))
10 loops, best of 3: 43.9 ms per loop
```

```python
>>> timeit(aAS(o.R(), o.vR(), o.vT(), o.z(), o.vz(), fixed_quad=True))
100 loops, best of 3: 3.87 ms per loop
```

Second, the actionAngleStaeckel calculations have also been implemented in C, which leads to even greater speed-ups, especially for arrays

```python
>>> aAS= actionAngleStaeckel(pot=MWPotential, delta=0.55, c=True)
>>> s= numpy.ones(100)
>>> timeit(aAS(1.*s, 0.1*s, 1.1*s, 0.*s, 0.05*s))
100 loops, best of 3: 2.37 ms per loop
```

```python
>>> timeit(aAS(1.*s, 0.1*s, 1.1*s, 0.*s, 0.05*s, fixed_quad=True))
1 loops, best of 3: 410 ms per loop
```

or a two hundred times speed up.

We can now go back to checking that the actions are conserved along the orbit

```python
>>> 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.abs((js[0]-numpy.mean(js[0]))/numpy.mean(js[0]))))
>>> plot(ts, numpy.log10(numpy.abs((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.

```
>>> aAS = actionAngleStaeckel(pot=MWPotential,delta=0.55,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([ 0.01576795]),
 array([ 1.1]),
 array([ 0.01346824]),
 array([ 1.22171491]),
 array([ 0.85773142]),
 array([ 1.60476805]),
 array([ 0.41881231]),
 array([ 6.18908605]),
 array([ 4.57359281]))
```

and we can check that the angles increase linearly along the orbit

```
Warning: Frequencies and angles using the Staeckel approximation are only implemented in C. So use c=True in the setup of the actionAngleStaeckel object.
```
```python
>>> o.integrate(ts,MWPotential)
>>> 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

```python
>>> plot(jfa[6], jfa[8], 'b. ')
```
1.6.5 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

```python
>>> 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

```python
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]))
```

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

```python
>>> actionAngleIsochroneApprox(pot=lp,b=10.8)
```

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 \texttt{estimateDeltaStaeckel} above except that it also gives a minimum and maximum \( b \) if multiple \( R \) and \( z \) are given

```python
>>> from galpy.actionAngle import estimateBIsochrone
```

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 convergence 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

```python
>>> actionAngleIsochroneApprox(pot=lp,b=1.5)
```

which does not agree with the previous calculation. We can inspect how the angles increase and how the actions converge by using the \texttt{aAIA.plot} function. For example, we can plot the radial versus the vertical angle in the auxiliary potential

```python
>>> 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

```python
>>> 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:
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 convergence and good behavior of the angles

```python
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=0.8)
>>> aAIA.plot(*obs,type='jr')
```

gives

and

```python
>>> 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

```python
>>> aAIA.plot(*obs,type='araz',deperiod=True)
```

We can then calculate the frequencies and the angles for this orbit as

```python
>>> 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 $\text{maxn}=$ the maximum $n$ for which to remove sinusoidal wiggles. So we can raise this, for example to 4 from 3

```python
>>> aAIA.actionsFreqsAngles(*obs,maxn=4)
(array([ 0.16392384]),
 array([-1.80322155]),
 array([ 0.50999882]),
 array([ 0.55808776]),
 array([-0.38475733]),
 array([ 0.4219968]),
 array([ 0.18732009]),
 array([ 0.31318534]),
 array([ 2.18421296]))
```

Clearly, there is very little change, as most of the wiggles are of low $n$.

**Warning:** While the orbit-based actionAngle technique in principle works for triaxial potentials, angles and frequencies for non-axisymmetric potentials are not implemented yet.

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`:

```python
>>> aAIA(*obs,nonaxi=True)
(array([ 0.16605011]), array([-1.80322155]), array([ 0.5070439]))
```
galpy currently does not contain any triaxial potentials, so we cannot illustrate this here with any real triaxial potentials.

## 1.6.6 Accessing action-angle coordinates for Orbit instances

While the recommended 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. This is illustrated here briefly. We initialize an Orbit instance

```python
>>> from galpy.orbit import Orbit
>>> from galpy.potential import MWPotential
>>> o = Orbit([1.,0.1,1.1,0.,0.25,0.])
```

and we can then calculate the actions (default is to use the adiabatic approximation)

```python
>>> o.jr(MWPotential), o.jp(MWPotential), o.jz(MWPotential)
(0.0138915441284973, 1.1, 0.01383357354294852)
```

`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, but note that these require extra parameters related to the approximation to be specified (see above):

```python
>>> o.jr(MWPotential,type='staeckel',delta=0.55), o.jp(MWPotential,type='staeckel',delta=0.55), o.jz(MWPotential,type='staeckel',delta=0.55)
(array([ 0.01576795]), array([ 1.1]), array([ 0.01346824]))
>>> o.jr(MWPotential,type='isochroneApprox',b=0.8), o.jp(MWPotential,type='isochroneApprox',b=0.8), o.jz(MWPotential,type='isochroneApprox',b=0.8)
(array([ 0.0155484]), array([ 1.1]), array([ 0.01350128]))
```

These two methods give very precise actions for this orbit (both are converged to about 1%) and they agree very well

```python
>>> (o.jr(MWPotential,type='staeckel',delta=0.55)-o.jr(MWPotential,type='isochroneApprox',b=0.8))/o.jr(MWPotential,type='isochroneApprox',b=0.8)
array([-0.00244754])
>>> (o.jz(MWPotential,type='staeckel',delta=0.55)-o.jz(MWPotential,type='isochroneApprox',b=0.8))/o.jz(MWPotential,type='isochroneApprox',b=0.8)
array([ 0.01412076])
```

**Warning:** Once an action, frequency, or angle is calculated for a given type of calculation (e.g., staeckel), the parameters for that type are fixed in the Orbit instance. Call `o.resetA()` to reset the action-angle instance used when using different parameters (i.e., different `delta=` for staeckel or different `b=` for isochroneApprox.

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

```python
>>> o.Or(MWPotential,type='staeckel',delta=0.55)
1.221714911136343
>>> o.Or(MWPotential,type='isochroneApprox',b=0.8)
1.222457055706389
```

and the radial angle

```python
>>> o.wr(MWPotential,type='staeckel',delta=0.55)
0.4188123062144965
>>> o.wr(MWPotential,type='isochroneApprox',b=0.8)
0.42281897198881867
```

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`.
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:

```python
do = 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:

```python
lp = LogarithmicHaloPotential(normalize=1.)
myjr[ii] = o.jr(lp)
```

etc.

Plotting the radial action versus the angular momentum:

```python
plot.bovy_plot(myjp, myjr, 'k.', ms=2., xlabel=r'$J_{\phi}$', ylabel=r'$J_R$', xrange=[0.7, 1.3], yrange=[0., .05])
```

shows a feature in the distribution.

If instead we use a power-law rotation curve with power-law index 1:

```python
pp = PowerSphericalPotential(normalize=1., alpha=-2.)
myjr[ii] = o.jr(pp)
```
We find that the distribution is stretched, but the feature remains

![Graph showing the distribution function](image)

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 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`).

### 1.7.1 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.MWPotential` potential using the adiabatic approximation for the actions, we import and define the following

```python
>>> from galpy.potential import MWPotential
>>> from galpy.actionAngle import actionAngleAdiabatic
>>> from galpy.df import quasiisothermaldf

>>> aA = actionAngleAdiabatic(pot=MWPotential, c=True)
and then setup the `quasiisothermaldf` instance

```python
>>> qdf = quasiisothermaldf(1./3., 0.2, 0.1, 1., 1., pot=MWPotential, aA=aA, cutcounter=True)
```
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)

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.))
0.20918647082092351
>>> numpy.sqrt(qdf.sigmaz2(1., 0.))
0.09256422252783468
```

and they are pretty close. We can also calculate the mixed $R$ and $z$ moment, for example,
or expressed as an angle (the *tilt of the velocity ellipsoid*)

```python
>>> 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

```python
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=MWPotential,delta=0.55,c=True)
>>> qdfS= quasiisothermaldf(1./3.,0.2,0.1,1.,1.,pot=MWPotential,aA=aAS,cutcounter=True)
```

we can similarly calculate the tilt

```python
>>> qdfS.tilt(1.,0.125)
5.4669442080366721
```

or about 5 degrees. As a function of height, we find

```python
>>> tilts= numpy.array([qdfS.tilt(1.,z) for z in zs])
>>> plot(zs,tilts)
```

which gives
We can also calculate the density and surface density (the zero-th velocity moments). For example, the vertical density

```python
>>> densz = numpy.array([qdf.density(1., z) for z in zs])
```
and

```python
>>> 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

```python
>>> semilogy(zs, densz/densz[0])
>>> semilogy(zs, denszS/denszS[0])
```

which gives
Similarly, we can calculate the radial profile of the surface density

```python
>>> 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

```python
>>> 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`.

### 1.7.3 Evaluating and sampling the full probability distribution function

We can evaluate the distribution itself by calling the object, e.g.,

```python
>>> qdf(1., 0.1, 1.1, 0.1, 0.) # input: R, vR, vT, z, vz
array([ 10.16445158])
```

or as a function of rotational velocity, for example in the mid-plane

```python
>>> 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

```python
>>> qdfS.pvT(1., 1., 0.) # input vT,R,z
15.464330302557528
```

or as a function of rotational velocity

```python
>>> pvt = numpy.array([qdfS.pvT(vt, 1., 0.) for vt in vts])
```

overplotting this over the previous distribution gives
which is slightly different from the conditioned distribution. Similarly, we can calculate marginalized velocity probabilities \(pv_R, pv_z, pv_{RvT}, pv_{Rvz}, \text{ and } pv_{Tvz}\). 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):

```python
>>> 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 $v_R$ and $v_z$) curve (as it should).
2.1 Orbit

2.1.1 Class

galpy.orbit.Orbit

class galpy.orbit.Orbit(vxvv=None, uvw=False, lb=False, radec=False, vo=235.0, ro=8.5, zo=0.025, 
solarmotion='hogg')

General orbit class representing an orbit

__init__(vxvv=None, uvw=False, lb=False, radec=False, vo=235.0, ro=8.5, zo=0.025, 
solarmotion='hogg')

NAME:

__init__

PURPOSE:

Initialize an Orbit instance

INPUT:

vxvv - initial conditions

3D can be either

1. in Galactocentric cylindrical coordinates [R,vR,vT] in [deg,deg,km/s] (all J2000.0; mu_ra = 
   mu_ra * cos dec)

2. [ra,dec,d,ra, mu_dec, vlos] in [deg,deg,kpc,mas/yr,mas/yr,km/s] (all J2000.0; mu_ra = 
   mu_ra * cos dec)

3. [ra,dec,d,U,V,W] in [deg,deg,kpc/km/s]

4. (l,b,d,lmu, mu_b, vlos) in [deg,deg,kpc/yr,mas/yr,km/s] (all J2000.0; mu_l = mu_l * 
   cos b)

5. [l,b,d,U,V,W] in [deg,deg,kpc/km/s]

4. and 5) also work when leaving out b and mu_b/W

OPTIONAL INPUTS:
radec - if True, input is 2) (or 3) above
uvw - if True, velocities are UVW
lb - if True, input is 4) or 5) above
vo - circular velocity at ro
ro - distance from vantage point to GC (kpc)
zo - offset toward the NGP of the Sun wrt the plane (kpc)
solarmotion - ‘hogg’ or ‘dehnen’, or ‘schoenrich’, or value in [-U,V,W]

OUTPUT:

instance

HISTORY:

2010-07-20 - Written - Bovy (NYU)

2.1.2 Methods

galpy.orbit.Orbit.__add__

Orbit.__add__(linOrb)

NAME:

__add__

PURPOSE:

add a linear orbit and a planar orbit to make a 3D orbit

INPUT:

linear or plane orbit instance

OUTPUT:

3D orbit

HISTORY:

2010-07-21 - Written - Bovy (NYU)

galpy.orbit.Orbit.__call__

Orbit.__call__(*args, **kwargs)

NAME:

__call__

PURPOSE:

return the orbit at time t

INPUT:

t - desired time
rect - if true, return rectangular coordinates

OUTPUT:
an Orbit instance with initial condition set to the phase-space at time t or list of Orbit instances if multiple times are given

HISTORY:
2010-07-10 - Written - Bovy (NYU)

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=[8.5,0.,0.]) OR Orbit object that corresponds to the orbit of the observer
ro= distance in kpc corresponding to R=1. (default: 8.5)
OUTPUT:
b(t)
HISTORY:
2011-02-23 - Written - Bovy (NYU)

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=[8.5,0.,0.]) OR Orbit object that corresponds to the orbit of the observer
ro= distance in kpc corresponding to R=1. (default: 8.5)
OUTPUT:
dec(t)
HISTORY:
2011-02-23 - Written - Bovy (NYU)
**galpy.orbit.Orbit.dist**

`Orbit.dist(*args, **kwargs)`

**NAME:**

dist

**PURPOSE:**

return distance from the observer

**INPUT:**

- t - (optional) time at which to get dist
- obs=[X,Y,Z] - (optional) position of observer (in kpc) (default=[8.5,0.,0.]) OR Orbit object that corresponds to the orbit of the observer
- ro= distance in kpc corresponding to R=1. (default: 8.5)

**OUTPUT:**

- dist(t) in kpc

**HISTORY:**

2011-02-23 - Written - Bovy (NYU)

---

**galpy.orbit.Orbit.E**

`Orbit.E(*args, **kwargs)`

**NAME:**

E

**PURPOSE:**

calculate the energy

**INPUT:**

- t - (optional) time at which to get the energy
- pot= Potential instance or list of such instances

**OUTPUT:**

- energy

**HISTORY:**

2010-09-15 - Written - Bovy (NYU)

---

**galpy.orbit.Orbit.e**

`Orbit.e(analytic=False, pot=None)`

**NAME:**

e

**PURPOSE:**

calculate the eccentricity

**INPUT:**
analytic - compute this analytically
pot - potential to use for analytical calculation

OUTPUT:
eccentricity

HISTORY:
2010-09-15 - Written - Bovy (NYU)

**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
pot= Potential instance or list of such instances

OUTPUT:
radial energy

HISTORY:
2013-11-30 - Written - Bovy (IAS)

**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
pot= Potential instance or list of such instances

OUTPUT:
vertical energy

HISTORY:
2013-11-30 - Written - Bovy (IAS)
galpy.orbit.Orbit.integrate

Orbit.integrate(t, pot, method='leapfrog_c')

NAME: integrate

PURPOSE: integrate the orbit

INPUT:
- t - list of times at which to output (0 has to be in this!)
- pot - potential instance or list of instances
- method= 'odeint' for scipy's odeint or 'leapfrog' for a simple leapfrog implementation

OUTPUT:
- (none) (get the actual orbit using getOrbit())

HISTORY:
- 2010-07-10 - Written - Bovy (NYU)

galpy.orbit.Orbit.getOrbit

Orbit.getOrbit()

NAME: getOrbit

PURPOSE: return a previously calculated orbit

INPUT:
- (none)

OUTPUT:
- array orbit[nt,nd]

HISTORY:
- 2010-07-10 - Written - Bovy (NYU)

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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

ro= distance in kpc corresponding to R=1. (default: 8.5)

OUTPUT:

helioX(t)

HISTORY:
2011-02-24 - Written - Bovy (NYU)

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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

ro= distance in kpc corresponding to R=1. (default: 8.5)

OUTPUT:

helioY(t)

HISTORY:
2011-02-24 - Written - Bovy (NYU)

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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

ro= distance in kpc corresponding to R=1. (default: 8.5)

OUTPUT:

helioZ(t)

HISTORY:
galpy Documentation, Release v0.1

2011-02-24 - Written - Bovy (NYU)

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
OmegaP= pattern speed
pot= potential instance or list of such instances

OUTPUT:

Jacobi integral

HISTORY:

2011-04-18 - Written - Bovy (NYU)

galpy.orbit.Orbit.jp

Orbit.jp(pot=None, **kwargs)

NAME:

jp

PURPOSE:

calculate the azimuthal action

INPUT:

pot - potential

 type= ('adiabatic') type of actionAngle module to use
 1. 'adiabatic'
 2. 'staeckel'
 3. 'isochroneApprox'
 4. 'spherical'

 + actionAngle module setup kwargs

OUTPUT:

jp

HISTORY:

2010-11-30 - Written - Bovy (NYU)
2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)
galpy.orbit.Orbit.jr

Orbit.jr(pot=None, **kwargs)
NAME:
  jr
PURPOSE:
  calculate the radial action
INPUT:
  pot - potential
type= (‘adiabatic’) type of actionAngle module to use
  1.’adiabatic’
  2.’staeckel’
  3.’isochroneApprox’
  4.’spherical’
+actionAngle module setup kwargs
OUTPUT:
  jr
HISTORY:
  2010-11-30 - Written - Bovy (NYU)
  2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)

galpy.orbit.Orbit.jz

Orbit.jz(pot=None, **kwargs)
NAME:
  jz
PURPOSE:
  calculate the vertical action
INPUT:
  pot - potential
type= (‘adiabatic’) type of actionAngle module to use
  1.’adiabatic’
  2.’staeckel’
  3.’isochroneApprox’
  4.’spherical’
+actionAngle module setup kwargs
OUTPUT:
  jz
galpy Documentation, Release v0.1

HISTORY:
2012-06-01 - Written - Bovy (IAS)
2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)

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=[8.5,0.,0.]) OR Orbit object that
 corresponds to the orbit of the observer
ro= distance in kpc corresponding to R=1. (default: 8.5)
OUTPUT:
l(t)
HISTORY:
2011-02-23 - Written - Bovy (NYU)

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
OUTPUT:
angular momentum
HISTORY:
2010-09-15 - Written - Bovy (NYU)

galpy.orbit.Orbit.Op

Orbit.Op(pot=None,**kwargs)
NAME:
Op
PURPOSE:
calculate the azimuthal frequency

INPUT:
pot - potential
type= (‘adiabatic’) type of actionAngle module to use
  1. ‘adiabatic’
  2. ‘staeckel’
  3. ‘isochroneApprox’
  4. ‘spherical’
+actionAngle module setup kwargs

OUTPUT:
Op

HISTORY:
2013-11-27 - Written - Bovy (IAS)

galpy.orbit.Orbit.Or

Or (pot=None, **kwargs)

NAME:
Or

PURPOSE:
calculate the radial frequency

INPUT:
pot - potential
type=(‘adiabatic’) type of actionAngle module to use
  1. ‘adiabatic’
  2. ‘staeckel’
  3. ‘isochroneApprox’
  4. ‘spherical’
+actionAngle module setup kwargs

OUTPUT:
Or

HISTORY:
2013-11-27 - Written - Bovy (IAS)
galpy.orbit.Orbit.Oz

Orbit.Oz(pot=None, **kwargs)

NAME:
Oz

PURPOSE:
calculate the vertical frequency

INPUT:
  pot - potential
  type= (‘adiabatic’) type of actionAngle module to use
      1. ‘adiabatic’
      2. ‘staeckel’
      3. ‘isochroneApprox’
      4. ‘spherical’
  +actionAngle module setup kwargs

OUTPUT:
  Oz

HISTORY:
  2013-11-27 - Written - Bovy (IAS)

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)

HISTORY:
  2010-09-21 - Written - Bovy (NYU)

galpy.orbit.Orbit.plot

Orbit.plot(*args, **kwargs)

NAME:
plot
PURPOSE:
plot a previously calculated orbit (with reasonable defaults)

INPUT:
d2= second dimension to plot
matplotlib.plot inputs+bovy_plot.plot inputs

OUTPUT:
sends plot to output device

HISTORY:
2010-07-10 - Written - Bovy (NYU)

---

galpy.orbit.Orbit.plot3d

Orbit.plot3d(*args, **kwargs)

NAME:
plot3d

PURPOSE:
plot 3D aspects of an Orbit

INPUT:
bovy_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)

---

galpy.orbit.Orbit.plotE

Orbit.plotE(*args, **kwargs)

NAME:
plotE

PURPOSE:
plot E(.) along the orbit

INPUT:
pot= Potential instance or list of instances in which the orbit was integrated
d1= plot Ez vs d1: e.g., ‘t’, ‘z’, ‘R’, ‘vR’, ‘vT’, ‘vz’
+bovy_plot.bovy_plot inputs
OUTPUT:
figure to output device

HISTORY:
2010-07-10 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotEz

Orbit.plotEz(*args, **kwargs)
NAME:
plotEz
PURPOSE:
plot E_z(.) along the orbit
INPUT:
pot= Potential instance or list of instances in which the orbit was integrated
d1= plot Ez vs d1: e.g., ‘t’, ‘z’, ‘R’
+bovy_plot.bovy_plot inputs

OUTPUT:
figure to output device

HISTORY:
2010-07-10 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotEzJz

Orbit.plotEzJz(*args, **kwargs)
NAME:
plotEzJzt
PURPOSE:
plot E_z(t)/sqrt(dens(R)) along the orbit (an approximation to the vertical action)
INPUT:
pot - Potential instance or list of instances in which the orbit was integrated
d1= plot Ez vs d1: e.g., ‘t’, ‘z’, ‘R’
+bovy_plot.bovy_plot inputs

OUTPUT:
figure to output device

HISTORY:
2010-08-08 - Written - Bovy (NYU)
galpy.orbit.Orbit.plotphi

Orbit.plotphi(*args, **kwargs)
NAME:
    plotphi
PURPOSE:
    plot phi(.) along the orbit
INPUT:
    d1= plot vs d1: e.g., ‘t’, ‘z’, ‘R’
    bovy_plot.bovy_plot inputs
OUTPUT:
    figure to output device
HISTORY:
    2010-07-10 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotR

Orbit.plotR(*args, **kwargs)
NAME:
    plotR
PURPOSE:
    plot R(.) along the orbit
INPUT:
    d1= plot vs d1: e.g., ‘t’, ‘z’, ‘R’
    bovy_plot.bovy_plot inputs
OUTPUT:
    figure to output device
HISTORY:
    2010-07-10 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotvR

Orbit.plotvR(*args, **kwargs)
NAME:
    plotvR
PURPOSE:
    plot vR(.) along the orbit
    bovy_plot inputs
INPUT:
d1= plot vs d1: e.g., ‘t’, ‘z’, ‘R’
bovy_plot.bovy_plot inputs

**OUTPUT:**
figure to output device

**HISTORY:**
2010-07-10 - Written - Bovy (NYU)

### galpy.orbit.Orbit.plotvT

**Name:**
Orbit.plotvT(*args, **kwargs)

**Purpose:**
plot vT(.) along the orbit

**Input:**
d1= plot vs d1: e.g., ‘t’, ‘z’, ‘R’
bovy_plot.bovy_plot inputs

**Output:**
figure to output device

**History:**
2010-07-10 - Written - Bovy (NYU)

### galpy.orbit.Orbit.plotvx

**Name:**
Orbit.plotvx(*args, **kwargs)

**Purpose:**
plot vx(.) along the orbit

**Input:**
d1= plot vs d1: e.g., ‘t’, ‘z’, ‘R’
bovy_plot.bovy_plot inputs

**Output:**
figure to output device

**History:**
2010-07-21 - Written - Bovy (NYU)
galpy.orbit.Orbit.plotvy

Orbit.plotvy(*args, **kwargs)

NAME:

plotvy

PURPOSE:

plot vy(.) along the orbit

INPUT:

d1= plot vs d1: e.g., ’t’, ’z’, ’R’
bovy_plot.bovy_plot inputs

OUTPUT:

figure to output device

HISTORY:

2010-07-21 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotvz

Orbit.plotvz(*args, **kwargs)

NAME:

plotvz

PURPOSE:

plot vz(.) along the orbit

INPUT:

d1= plot vs d1: e.g., ’t’, ’z’, ’R’
bovy_plot.bovy_plot inputs

OUTPUT:

figure to output device

HISTORY:

2010-07-10 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotx

Orbit.plotx(*args, **kwargs)

NAME:

plotx

PURPOSE:

plot x(.) along the orbit

INPUT:

d1= plot vs d1: e.g., ’t’, ’z’, ’R’
bovy_plot.bovy_plot inputs
galpy Documentation, Release v0.1

OUTPUT:

    figure to output device

HISTORY:

    2010-07-21 - Written - Bovy (NYU)

galpy.orbit.Orbit.ploty

Orbit.ploty(*args, **kwargs)
NAME:
    ploty
PURPOSE:
    plot y(.) along the orbit
INPUT:
    d1= plot vs d1: e.g., 't', 'z', 'R'
    bovy_plot.bovy_plot inputs
OUTPUT:
    figure to output device
HISTORY:
    2010-07-21 - Written - Bovy (NYU)

galpy.orbit.Orbit.plotz

Orbit.plotz(*args, **kwargs)
NAME:
    plotz
PURPOSE:
    plot z(.) along the orbit
INPUT:
    d1= plot vs d1: e.g., 't', 'z', 'R'
    bovy_plot.bovy_plot inputs
OUTPUT:
    figure to output device
HISTORY:
    2010-07-10 - Written - Bovy (NYU)

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 pmmb

   \texttt{obs}=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default: [8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

   \texttt{ro}= distance in kpc corresponding to R=1. (default: 8.5)

   \texttt{vo}= velocity in km/s corresponding to v=1. (default: 235.)

OUTPUT:
   pm_b(t)

HISTORY:
   2011-02-24 - Written - Bovy (NYU)

\texttt{galpy.orbit.Orbit.pmdec}

\texttt{Orbit.pmdec(*args,**kwargs)}

NAME:
   pmdec

PURPOSE:
return proper motion in declination (in mas/yr)

INPUT:
   t - (optional) time at which to get pmdec

   \texttt{obs}=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default: [8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

   \texttt{ro}= distance in kpc corresponding to R=1. (default: 8.5)

   \texttt{vo}= velocity in km/s corresponding to v=1. (default: 235.)

OUTPUT:
   pm_dec(t)

HISTORY:
   2011-02-24 - Written - Bovy (NYU)

\texttt{galpy.orbit.Orbit.pml}

\texttt{Orbit.pml(*args,**kwargs)}

NAME:
   pml

PURPOSE:
return proper motion in Galactic longitude (in mas/yr)

INPUT:

2.1. Orbit
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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

**ro=** distance in kpc corresponding to R=1. (default: 8.5)

**vo=** velocity in km/s corresponding to v=1. (default: 235.)

**OUTPUT:**

pm_ra(t)

**HISTORY:**

2011-02-24 - Written - Bovy (NYU)

---

### 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

**OUTPUT:**

pm_ra(t)

**HISTORY:**

2011-02-24 - Written - Bovy (NYU)

---

### 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

**OUTPUT:**

R(t)
galpy Documentation, Release v0.1

HISTORY:
2010-09-21 - Written - Bovy (NYU)

**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=[8.5,0.,0.]) OR Orbit object that corresponds to the orbit of the observer
  ro= distance in kpc corresponding to R=1. (default: 8.5)
  OUTPUT:
  ra(t)
  HISTORY:
  2011-02-23 - Written - Bovy (NYU)

**galpy.orbit.Orbit.rap**

* Orbit.rap(analytic=False, pot=None)
  NAME:
  rap
  PURPOSE:
  calculate the apocenter radius
  INPUT:
  analytic - compute this analytically
  pot - potential to use for analytical calculation
  OUTPUT:
  R_ap
  HISTORY:
  2010-09-20 - Written - Bovy (NYU)

**galpy.orbit.Orbit.resetaA**

* Orbit.resetaA(pot=None, type=None)
  NAME:
  resetaA

2.1. Orbit
PURPOSE:
re-set up an actionAngle module for this Orbit

INPUT:

(output)

OUTPUT:
True if reset happened, False otherwise

HISTORY:
2014-01-06 - Written - Bovy (IAS)

**galpy.orbit.Orbit.rperi**

Orbit.rperi(analytic=False, pot=None)

NAME:
rperi

PURPOSE:
calculate the pericenter radius

INPUT:
analytic - compute this analytically
pot - potential to use for analytical calculation

OUTPUT:
R_peri

HISTORY:
2010-09-20 - Written - Bovy (NYU)

**galpy.orbit.Orbit.setphi**

Orbit.setphi(phi)

NAME:
setphi

PURPOSE:
set initial azimuth

INPUT:
phi - desired azimuth

OUTPUT:
(output)

HISTORY:
2010-08-01 - Written - Bovy (NYU)

BUGS:
Should perform check that this orbit has phi

**galpy.orbit.Orbit.toLinear**

Orbit.toLinear()

NAME:

toLinear

PURPOSE:

convert a 3D orbit into a 1D orbit (z)

INPUT:

(none)

OUTPUT:

linear Orbit

HISTORY:

2010-11-30 - Written - Bovy (NYU)

**galpy.orbit.Orbit.toPlanar**

Orbit.toPlanar()

NAME:

toPlanar

PURPOSE:

convert a 3D orbit into a 2D orbit

INPUT:

(none)

OUTPUT:

planar Orbit

HISTORY:

2010-11-30 - Written - Bovy (NYU)

**galpy.orbit.Orbit.Tp**

Orbit.Tp(pot=None, **kwargs)

NAME:

Tp

PURPOSE:

calculate the azimuthal period

INPUT:

pot - potential
type= (‘adiabatic’) type of actionAngle module to use
1. 'adiabatic'
2. 'staeckel'
3. 'isochroneApprox'
4. 'spherical'
+actionAngle module setup kwargs

OUTPUT:
Tp

HISTORY:
2010-11-30 - Written - Bovy (NYU)
2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)

galpy.orbit.Orbit.Tr

Orbit.Tr(pot=None, **kwargs)

NAME:
Tr

PURPOSE:
calculate the radial period

INPUT:
pot - potential
type= ('adiabatic') type of actionAngle module to use
1. 'adiabatic'
2. 'staeckel'
3. 'isochroneApprox'
4. 'spherical'
+actionAngle module setup kwargs

OUTPUT:
Tr

HISTORY:
2010-11-30 - Written - Bovy (NYU)
2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)

galpy.orbit.Orbit.TrTp

Orbit.TrTp(pot=None, **kwargs)

NAME:
TrTp

PURPOSE:
the ‘ratio’ between the radial and azimuthal period $T_r/T_{\phi}\pi$

**INPUT:**
- pot - potential
  - type= (‘adiabatic’) type of actionAngle module to use
    1. ‘adiabatic’
    2. ‘staeckel’
    3. ‘isochroneApprox’
    4. ‘spherical’
  + actionAngle module setup kwargs

**OUTPUT:**
- $T_r/T_{\phi}\pi$

**HISTORY:**
- 2010-11-30 - Written - Bovy (NYU)

---

**galpy.orbit.Orbit.Tz**

**Orbit.Tz** *(pot=None, **kwargs)*

**NAME:**
- Tz

**PURPOSE:**
- calculate the vertical period

**INPUT:**
- pot - potential
  - type= (‘adiabatic’) type of actionAngle module to use
    1. ‘adiabatic’
    2. ‘staeckel’
    3. ‘isochroneApprox’
    4. ‘spherical’
  + actionAngle module setup kwargs

**OUTPUT:**
- Tz

**HISTORY:**
- 2012-06-01 - Written - Bovy (IAS)
galpy Documentation, Release v0.1

**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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

   ro= distance in kpc corresponding to R=1. (default: 8.5)

   vo= velocity in km/s corresponding to v=1. (default: 235.)

OUTPUT:

   U(t)

HISTORY:

   2011-02-24 - Written - Bovy (NYU)

**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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer

   ro= distance in kpc corresponding to R=1. (default: 8.5)

   vo= velocity in km/s corresponding to v=1. (default: 235.)

OUTPUT:

   V(t)

HISTORY:

   2011-02-24 - Written - Bovy (NYU)
galpy.orbit.Orbit.vbb

**NAME:**

vbb

**PURPOSE:**

return velocity in Galactic latitude (km/s)

**INPUT:**

- t - (optional) time at which to get vbb
- obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer
- ro= distance in kpc corresponding to R=1. (default: 8.5)
- vo= velocity in km/s corresponding to v=1. (default: 235.)

**OUTPUT:**

v_b(t) in km/s

**HISTORY:**

2011-02-24 - Written - Bovy (NYU)

---

galpy.orbit.Orbit.vdec

**NAME:**

vdec

**PURPOSE:**

return velocity in declination (km/s)

**INPUT:**

- t - (optional) time at which to get vdec
- obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer
- ro= distance in kpc corresponding to R=1. (default: 8.5)
- vo= velocity in km/s corresponding to v=1. (default: 235.)

**OUTPUT:**

v_dec(t) in km/s

**HISTORY:**

2011-03-27 - Written - Bovy (NYU)
**galpy.orbit.Orbit.vll**

```python
Orbit.vll(*args, **kwargs)
```

**NAME:**

vll

**PURPOSE:**

return the velocity in Galactic longitude (km/s)

**INPUT:**

- `t` - (optional) time at which to get vll
- `obs=[X,Y,Z,vx,vy,vz]` - (optional) position and velocity of observer (in kpc and km/s) (default=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer
- `ro=` distance in kpc corresponding to R=1. (default: 8.5)
- `vo=` velocity in km/s corresponding to v=1. (default: 235.)

**OUTPUT:**

$v_l(t)$ in km/s

**HISTORY:**

2011-03-27 - Written - Bovy (NYU)

**galpy.orbit.Orbit.vlos**

```python
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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer
- `ro=` distance in kpc corresponding to R=1. (default: 8.5)
- `vo=` velocity in km/s corresponding to v=1. (default: 235.)

**OUTPUT:**

$v_{los}(t)$

**HISTORY:**

2011-02-24 - Written - Bovy (NYU)
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

OUTPUT:

   vphi(t)

HISTORY:
2010-09-21 - Written - Bovy (NYU)

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

OUTPUT:

   vR(t)

HISTORY:
2010-09-21 - Written - Bovy (NYU)

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

   obs=[X,Y,Z,vx,vy,vz] - (optional) position and velocity of observer (in kpc and km/s) (default=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer
\texttt{ro= distance in kpc corresponding to R=1. (default: 8.5)}
\texttt{vo= velocity in km/s corresponding to v=1. (default: 235.)}

OUTPUT:
\texttt{v_{ra(t)} in km/s}

HISTORY:
2011-03-27 - Written - Bovy (NYU)

\texttt{galpy.orbit.Orbit.vT}

\texttt{ Orbit.vT(*args, **kwargs)}
\texttt{ NAME:}
\texttt{ vT}
\texttt{ PURPOSE:}
\texttt{ return tangential velocity at time t}
\texttt{ INPUT:}
\texttt{ t - (optional) time at which to get the tangential velocity}
\texttt{ OUTPUT:}
\texttt{ vT(t)}

HISTORY:
2010-09-21 - Written - Bovy (NYU)

\texttt{galpy.orbit.Orbit.vx}

\texttt{ Orbit.vx(*args, **kwargs)}
\texttt{ NAME:}
\texttt{ vx}
\texttt{ PURPOSE:}
\texttt{ return x velocity at time t}
\texttt{ INPUT:}
\texttt{ t - (optional) time at which to get the velocity}
\texttt{ OUTPUT:}
\texttt{ vx(t)}

HISTORY:
2010-11-30 - Written - Bovy (NYU)

\texttt{galpy.orbit.Orbit.vy}

\texttt{ Orbit.vy(*args, **kwargs)}
\texttt{ NAME:}
\texttt{ vy}
PURPOSE:
return y velocity at time t

INPUT:
   t - (optional) time at which to get the velocity

OUTPUT:
   vy(t)

HISTORY:
   2010-11-30 - Written - Bovy (NYU)

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

OUTPUT:
   vz(t)

HISTORY:
   2010-09-21 - Written - Bovy (NYU)

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=[8.5,0.,0.,0.,235.,0.]) OR Orbit object that corresponds to the orbit of the observer
   
   ro= distance in kpc corresponding to R=1. (default: 8.5)
   
   vo= velocity in km/s corresponding to v=1. (default: 235.)

OUTPUT:
   W(t)

HISTORY:
galpy Documentation, Release v0.1

2011-02-24 - Written - Bovy (NYU)

galpy.orbit.Orbit.wp

Orbit.wp(pot=None, **kwargs)

NAME:

wp

PURPOSE:

calculate the azimuthal angle

INPUT:

pot - potential
type= (‘adiabatic’) type of actionAngle module to use
1. ‘adiabatic’
2. ‘staeckel’
3. ‘isochroneApprox’
4. ‘spherical’
+actionAngle module setup kwargs

OUTPUT:

wp

HISTORY:

2010-11-30 - Written - Bovy (NYU)
2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)

galpy.orbit.Orbit.wr

Orbit.wr(pot=None, **kwargs)

NAME:

wr

PURPOSE:

calculate the radial angle

INPUT:

pot - potential
type= (‘adiabatic’) type of actionAngle module to use
1. ‘adiabatic’
2. ‘staeckel’
3. ‘isochroneApprox’
4. ‘spherical’
+actionAngle module setup kwargs
galpy.orbit.Orbit.wz

Orbit.wz (pot=None, **kwargs)

NAME:
    wz

PURPOSE:
    calculate the vertical angle

INPUT:
    pot - potential
type=(‘adiabatic’) type of actionAngle module to use
    1.’adiabatic’
    2.’staeckel’
    3.’isochroneApprox’
    4.’spherical’
+actionAngle module setup kwargs

OUTPUT:
    wz

HISTORY:
    2012-06-01 - Written - Bovy (IAS)
    2013-11-27 - Re-written using new actionAngle modules - Bovy (IAS)

galpy.orbit.Orbit.x

Orbit.x(*args, **kwargs)

NAME:
    x

PURPOSE:
    return x

INPUT:
    t - (optional) time at which to get x

OUTPUT:
    x(t)
**galpy.orbit.Orbit.y**

\[ \text{Orbit.y}(\astargs, \**kwargs) \]

**NAME:**

\( y \)

**PURPOSE:**

return \( y \)

**INPUT:**

\( t \) - (optional) time at which to get \( y \)

**OUTPUT:**

\( y(t) \)

**HISTORY:**

2010-09-21 - Written - Bovy (NYU)

**galpy.orbit.Orbit.z**

\[ \text{Orbit.z}(\astargs, \**kwargs) \]

**NAME:**

\( z \)

**PURPOSE:**

return vertical height

**INPUT:**

\( t \) - (optional) time at which to get the vertical height

**OUTPUT:**

\( z(t) \)

**HISTORY:**

2010-09-21 - Written - Bovy (NYU)

**galpy.orbit.Orbit.zmax**

\[ \text{Orbit.zmax}(\text{analytic}=False, \text{pot}=None) \]

**NAME:**

\( \text{zmax} \)

**PURPOSE:**

calculate the maximum vertical height

**INPUT:**
analytic - compute this analytically
pot - potential to use for analytical calculation

OUTPUT:
Z_max

HISTORY:
2010-09-20 - Written - Bovy (NYU)

2.2 Potential

2.2.1 3D potentials

General instance routines

Use as Potential-instance.method(...)

galpy.potential.Potential.__call__

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
z - vertical height
phi - azimuth (optional)
t - time (optional)
dR= dphi=, if set to non-zero integers, return the dR, dphi’t derivative instead

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
z - vertical height
phi - azimuth (optional)
t - time (optional)

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)

NAME:

dvcircdR

PURPOSE:

calculate the derivative of the circular velocity at R wrt R in this potential

INPUT:

R - Galactocentric radius

OUTPUT:

derivative of the circular rotation velocity wrt R

HISTORY:

2013-01-08 - Written - Bovy (IAS)

galpy.potential.Potential.epifreq

Potential.epifreq(R)

NAME:

epifreq

PURPOSE:

calculate the epicycle frequency at R in this potential

INPUT:

R - Galactocentric radius

OUTPUT:

epicycle frequency

HISTORY:

2011-10-09 - Written - Bovy (IAS)
**galpy.potential.Potential.flattening**

\( \text{Potential}. \text{flattening}(R,z) \)

**NAME:**
flattening

**PURPOSE:**
calculate the potential flattening, defined as \( \sqrt{\frac{|z/R F_R/F_z|}{}} \)

**INPUT:**
- \( R \) - Galactocentric radius
- \( z \) - height

**OUTPUT:**
- flattening

**HISTORY:**
2012-09-13 - Written - Bovy (IAS)

**galpy.potential.Potential.lindbladR**

\( \text{Potential}. \text{lindbladR}(\Omega_p, m=2, \text{**kwargs}) \)

**NAME:**
lindbladR

**PURPOSE:**
calculate the radius of a Lindblad resonance

**INPUT:**
- \( \Omega_p \) - pattern speed
- \( m \) - order of the resonance (as in \( m(O-Op)=\kappa \) negative \( m \) for outer) use \( m='\text{corotation}' \)
  - for corotation +scipy.optimize.brentq xtol,rtol,maxiter kwargs

**OUTPUT:**
- radius of Linblad resonance, None if there is no resonance

**HISTORY:**
2011-10-09 - Written - Bovy (IAS)

**galpy.potential.Potential.omegac**

\( \text{Potential}. \text{omegac}(R) \)

**NAME:**
omegac

**PURPOSE:**
calculate the circular angular speed at \( R \) in this potential

**INPUT:**


\textbf{galpy Documentation, Release v0.1}

\begin{itemize}
  \item \textbf{R - Galactocentric radius}
  \end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item circular angular speed
  \end{itemize}

\textbf{HISTORY:}

\begin{itemize}
  \item 2011-10-09 - Written - Bovy (IAS)
  \end{itemize}

\textbf{galpy.potential.Potential.phiforce}

\begin{itemize}
  \item \textbf{Potential.phiforce}(R, z, phi=0.0, t=0.0)
  \end{itemize}

\textbf{NAME:}

\begin{itemize}
  \item phiforce
  \end{itemize}

\textbf{PURPOSE:}

\begin{itemize}
  \item evaluate the azimuthal force \textit{F}_\phi (R,z,\phi,t)
  \end{itemize}

\textbf{INPUT:}

\begin{itemize}
  \item R - Cylindrical Galactocentric radius
  \item z - vertical height
  \item \phi - azimuth (rad)
  \item t - time (optional)
  \end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item \textit{F}_\phi (R,z,\phi,t)
  \end{itemize}

\textbf{HISTORY:}

\begin{itemize}
  \item 2010-07-10 - Written - Bovy (NYU)
  \end{itemize}

\textbf{galpy.potential.Potential.phi2deriv}

\begin{itemize}
  \item \textbf{Potential.phi2deriv}(R, Z, phi=0.0, t=0.0)
  \end{itemize}

\textbf{NAME:}

\begin{itemize}
  \item phi2deriv
  \end{itemize}

\textbf{PURPOSE:}

\begin{itemize}
  \item evaluate the second azimuthal derivative
  \end{itemize}

\textbf{INPUT:}

\begin{itemize}
  \item R - Galactocentric radius
  \item Z - vertical height
  \item \phi - Galactocentric azimuth
  \item t - time
  \end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item \textit{d2Phi/dphi2}
  \end{itemize}

\textbf{HISTORY:}

\begin{itemize}
  \item 
  \end{itemize}
galpy.potential.Potential.plot

**Potential.plot** \(t=0.0, rmin=0.0, rmax=1.5, nrs=21, zmin=-0.5, zmax=0.5, nzs=21, ncontours=21, savefilename=None\)

**NAME:** plot

**PURPOSE:** plot the potential

**INPUT:**
- \(t\) - time to plot potential at
- \(rmin\) - minimum R
- \(rmax\) - maximum R
- \(nrs\) - grid in R
- \(zmin\) - minimum z
- \(zmax\) - maximum z
- \(nzs\) - grid in z
- \(ncontours\) - number of contours
- \(savefilename\) - save to or restore from this savefile (pickle)

**OUTPUT:**
plot to output device

**HISTORY:**
2010-07-09 - Written - Bovy (NYU)

galpy.potential.Potential.plotDensity

**Potential.plotDensity** \(rmin=0.0, rmax=1.5, nrs=21, zmin=-0.5, zmax=0.5, nzs=21, ncontours=21, savefilename=None, aspect=None, log=False\)

**NAME:** plotDensity

**PURPOSE:** plot the density of this potential

**INPUT:**
- \(rmin\) - minimum R
- \(rmax\) - maximum R
- \(nrs\) - grid in R
- \(zmin\) - minimum z
- \(zmax\) - maximum z
- \(nzs\) - grid in z
- \(ncontours\) - number of 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.plotEscapcecurve

```
Potential.plotEscapcecurve(*args, **kwargs)
```

**NAME:**

plotEscapcecurve

**PURPOSE:**

plot the escape velocity curve for this potential (in the z=0 plane for non-spherical potentials)

**INPUT:**

Rrange - range
grid - number of points to plot
savefilename - save to or restore from this savefile (pickle)
+bovy_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
grid - number of points to plot
savefilename - save to or restore from this savefile (pickle)
+bovy_plot(*args,**kwargs)

**OUTPUT:**

plot to output device

**HISTORY:**

2010-07-10 - Written - Bovy (NYU)
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
\(Z\) - vertical height
\(phi\) - Galactocentric azimuth
\(t\) - time

OUTPUT:

d2phi/dR2

HISTORY:

2011-10-09 - Written - Bovy (IAS)

---

galpy.potential.Potential.Rforce

Potential.Rforce \((R, z, phi=0.0, t=0.0)\)

NAME:

Rforce

---

2.2. Potential
PURPOSE:
evaluate radial force $F_R(R,z)$

INPUT:
- $R$ - Cylindrical Galactocentric radius
- $z$ - vertical height
- $\phi$ - azimuth (optional)
- $t$ - time (optional)

OUTPUT:
- $F_R(R,z,\phi,t)$

HISTORY:
- 2010-04-16 - Written - Bovy (NYU)

---

**galpy.potential.Potential.rl**

Potential.**rl**($lz$)

NAME:
- rl

PURPOSE:
calculate the radius of a circular orbit of $Lz$

INPUT:
- $lz$ - Angular momentum

OUTPUT:
- radius

HISTORY:
- 2012-07-30 - Written - Bovy (IAS@MPIA)

NOTE:
- seems to take about ~0.5 ms for a Miyamoto-Nagai potential; ~0.75 ms for a MWPotential

---

**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
**galpy.potential.Potential.toVertical**

\[ \text{Potential.toVertical} (R) \]

**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

**OUTPUT:** linear (vertical) potential

**HISTORY**

**galpy.potential.Potential.vcirc**

\[ \text{Potential.vcirc} (R) \]

**NAME:** vcirc

**PURPOSE:** calculate the circular velocity at \( R \) in this potential

**INPUT:**

\( R \) - Galactocentric radius

**OUTPUT:**

circular rotation velocity

**HISTORY:**

2011-10-09 - Written - Bovy (IAS)

**galpy.potential.Potential.verticalfreq**

\[ \text{Potential.verticalfreq} (R) \]

**NAME:** verticalfreq

**PURPOSE:** calculate the vertical frequency at \( R \) in this potential

**INPUT:**

\( R \) - Galactocentric radius

**OUTPUT:**

vertical frequency

**HISTORY:**

2012-07-25 - Written - Bovy (IAS@MPIA)
**galpy.potential.Potential.vesc**

Potential.\texttt{vesc}(R)

NAME: vesc

PURPOSE: calculate the escape velocity at R for this potential

INPUT: 
R - Galactocentric radius

OUTPUT: escape velocity

HISTORY: 2011-10-09 - Written - Bovy (IAS)

**galpy.potential.Potential.vterm**

Potential.\texttt{vterm}(l, \texttt{deg=True})

NAME: vterm

PURPOSE: calculate the terminal velocity at l in this potential

INPUT:
1 - Galactic longitude [deg/rad]
deg= if True (default), l in deg

OUTPUT:
terminal velocity

HISTORY: 2013-05-31 - Written - Bovy (IAS)

**galpy.potential.Potential.z2deriv**

Potential.\texttt{z2deriv}(R, Z, phi=0.0, t=0.0)

NAME: z2deriv

PURPOSE: evaluate the second vertical derivative

INPUT:
R - Galactocentric radius
Z - vertical height
phi - Galactocentric azimuth
t - time

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
z - vertical height
phi - azimuth (optional)
t - time (optional)

OUTPUT:
\( F_z (R,z,\phi,t) \)

HISTORY:
2010-04-16 - Written - Bovy (NYU)

---

General 3D potential routines

Use as method(…)

---

galpy.potential.
dvcircdR

galpy.potential.
dvcircdR(Pot, R)

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

OUTPUT:
derivative of the circular rotation velocity wrt R

HISTORY:
2013-01-08 - Written - Bovy (IAS)

galpy.potential.epifreq

galpy.potential.epifreq(Pot, R)

NAME:
epifreq

PURPOSE:
calculate the epicycle frequency at R in the potential Pot

INPUT:
Pot - Potential instance or list thereof
R - Galactocentric radius

OUTPUT:
epicycle frequency

HISTORY:
2012-07-25 - Written - Bovy (IAS)

galpy.potential.evaluateDensities

galpy.potential.evaluateDensities(R, z, Pot, phi=0.0, t=0.0, forcepoisson=False)

NAME:
evaluateDensities

PURPOSE:
convenience function to evaluate a possible sum of densities

INPUT:
R - cylindrical Galactocentric distance
z - distance above the plane
Pot - potential or list of potentials
phi - azimuth
t - time
forcepoisson= if True, calculate the density through the Poisson equation, even if an explicit expression for the density exists

OUTPUT:
rho(R,z)
2.2. Potential

galpy.potential.evaluatephiforces

galpy.potential.evaluatephiforces(R, z, Pot, phi=0.0, t=0.0)

NAME: evaluatephiforces

PURPOSE: convenience function to evaluate a possible sum of potentials

INPUT: R - cylindrical Galactocentric distance
z - distance above the plane
Pot - a potential or list of potentials
phi - azimuth (optional)
t - time (optional)

OUTPUT:
F_phi(R,z,phi,t)

HISTORY:
2010-04-16 - Written - Bovy (NYU)

galpy.potential.evaluatePotentials

galpy.potential.evaluatePotentials(R, z, Pot, phi=0.0, t=0.0)

NAME: evaluatePotentials

PURPOSE: convenience function to evaluate a possible sum of potentials

INPUT: R - cylindrical Galactocentric distance
z - distance above the plane
Pot - potential or list of potentials
phi - azimuth
t - time

OUTPUT: Phi(R,z)

HISTORY: 2010-04-16 - Written - Bovy (NYU)
galpy.potential.evaluateR2derivs

**galpy.potential.evaluateR2derivs** \((R, z, Pot, phi=0.0, t=0.0)\)

**NAME:** evaluateR2derivs

**PURPOSE:** convenience function to evaluate a possible sum of potentials

**INPUT:**
- \(R\) - cylindrical Galactocentric distance
- \(z\) - distance above the plane
- \(Pot\) - a potential or list of potentials
- \(phi\) - azimuth (optional)
- \(t\) - time (optional)

**OUTPUT:** \(d^2\Phi/d^2R(R,z,\phi,t)\)

**HISTORY:** 2012-07-25 - Written - Bovy (IAS)

---

galpy.potential.evaluateRzderivs

**galpy.potential.evaluateRzderivs** \((R, z, Pot, phi=0.0, t=0.0)\)

**NAME:** evaluateRzderivs

**PURPOSE:** convenience function to evaluate a possible sum of potentials

**INPUT:**
- \(R\) - cylindrical Galactocentric distance
- \(z\) - distance above the plane
- \(Pot\) - a potential or list of potentials
- \(phi\) - azimuth (optional)
- \(t\) - time (optional)

**OUTPUT:** \(d^2\Phi/dz/dR(R,z,\phi,t)\)

**HISTORY:** 2013-08-28 - Written - Bovy (IAS)

---

galpy.potential.evaluateRforces

**galpy.potential.evaluateRforces** \((R, z, Pot, phi=0.0, t=0.0)\)

**NAME:** evaluateRforce

**PURPOSE:** convenience function to evaluate a possible sum of potentials

**INPUT:**
- \(R\) - cylindrical Galactocentric distance
- \(z\) - distance above the plane
- \(Pot\) - a potential or list of potentials
- \(phi\) - azimuth (optional)
- \(t\) - time (optional)

**OUTPUT:** \(F_R(R,z,\phi,t)\)

**HISTORY:** 2010-04-16 - Written - Bovy (NYU)
**galpy.potential.evaluatez2derivs**

```python
galpy.potential.evaluatez2derivs(R, z, Pot, phi=0.0, t=0.0)
```

**NAME:** evaluatez2derivs

**PURPOSE:** convenience function to evaluate a possible sum of potentials

**INPUT:**
- `R` - cylindrical Galactocentric distance
- `z` - distance above the plane
- `Pot` - a potential or list of potentials
- `phi` - azimuth (optional)
- `t` - time (optional)

**OUTPUT:** \( \frac{d^2\Phi}{d^2z}(R,z,\phi,t) \)

**HISTORY:** 2012-07-25 - Written - Bovy (IAS)

**galpy.potential.evaluatezforces**

```python
galpy.potential.evaluatezforces(R, z, Pot, phi=0.0, t=0.0)
```

**NAME:** evaluatezforces

**PURPOSE:** convenience function to evaluate a possible sum of potentials

**INPUT:**
- `R` - cylindrical Galactocentric distance
- `z` - distance above the plane
- `Pot` - a potential or list of potentials
- `phi` - azimuth (optional)
- `t` - time (optional)

**OUTPUT:** \( F_z(R,z,\phi,t) \)

**HISTORY:** 2010-04-16 - Written - Bovy (NYU)

**galpy.potential.flattening**

```python
galpy.potential.flattening(Pot, R, z)
```

**NAME:** flattening

**PURPOSE:** calculate the potential flattening, defined as \( \sqrt{|z/R F_R/F_z|} \)

**INPUT:**
- `Pot` - a potential or list of potentials
- `R` - cylindrical Galactocentric distance
- `z` - distance above the plane
Pot - Potential instance or list thereof
R - Galactocentric radius
z - height

OUTPUT:
flattening

HISTORY:
2012-09-13 - Written - Bovy (IAS)

**galpy.potential.lindbladR**

galpy.potential.lindbladR(Pot, OmegaP, m=2, **kwargs)

NAME:
lindbladR

PURPOSE:
calculate the radius of a Lindblad resonance

INPUT:
Pot - Potential instance or list of such instances
OmegaP - pattern speed

m= order of the resonance (as in m(OO-Op)=kappa (negative m for outer) use m=’corotation’
for corotation
+scipy.optimize.brentq xtol,rtol,maxiter kwargs

OUTPUT:
radius of Lindblad resonance, None if there is no resonance

HISTORY:
2011-10-09 - Written - Bovy (IAS)

**galpy.potential.omegac**

galpy.potential.omegac(Pot, R)

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

OUTPUT:
circular angular speed

HISTORY:
galpy.potential.plotDensities

**NAME:** plotDensities  
**PURPOSE:** plot the density a set of potentials  
**INPUT:**  
- Pot - Potential or list of Potential instances  
  - rmin - minimum R  
  - rmax - maximum R  
  - nrs - grid in R  
  - zmin - minimum z  
  - zmax - maximum z  
  - nz - grid in z  
  - ncontours - number of 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

**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  
- grid - grid in R  
- savefilename - save to or restore from this savefile (pickle)  
**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, ncontours=21, savefilename=None, aspect=None)
```

**NAME:** plotPotentials

**PURPOSE:** plot a set of potentials

**INPUT:**
- Pot - Potential or list of Potential instances
  - rmin - minimum R
  - rmax - maximum R
  - nrs - grid in R
  - zmin - minimum z
  - zmax - maximum z
  - nzs - grid in z
  - ncontours - number of contours
  - 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
  - grid - grid in R
  - savefilename - save to or restore from this savefile (pickle)

**OUTPUT:**
- plot to output device

**HISTORY:** 2010-07-10 - Written - Bovy (NYU)
galpy.potential.rl

\[\text{galpy.potential.rl}(\text{Pot}, \text{lz})\]

**NAME:**

galpy.potential.rl

**PURPOSE:**

calculate the radius of a circular orbit of Lz

**INPUT:**

- Pot - Potential instance or list thereof
- lz - Angular momentum

**OUTPUT:**

radius

**HISTORY:**

2012-07-30 - Written - Bovy (IAS@MPIA)

**NOTE:**

seems to take about ~0.5 ms for a Miyamoto-Nagai potential; ~0.75 ms for a MWPotential

---

galpy.potential.vcirc

\[\text{galpy.potential.vcirc}(\text{Pot}, \text{R})\]

**NAME:**

vcirc

**PURPOSE:**

calculate the circular velocity at R in potential Pot

**INPUT:**

- Pot - Potential instance or list of such instances
- R - Galactocentric radius

**OUTPUT:**

circular rotation velocity

**HISTORY:**

2011-10-09 - Written - Bovy (IAS)

---

galpy.potential.verticalfreq

\[\text{galpy.potential.verticalfreq}(\text{Pot}, \text{R})\]

**NAME:**

verticalfreq

**PURPOSE:**

calculate the vertical frequency at R in the potential Pot
galpy Documentation, Release v0.1

INPUT:
   Pot - Potential instance or list thereof
   R - Galactocentric radius

OUTPUT:
   vertical frequency

HISTORY:
   2012-07-25 - Written - Bovy (IAS@MPIA)

**galpy.potential.vesc**

galpy.potential.vesc(Pot, R)

NAME:
   vesc

PURPOSE:
   calculate the escape velocity at R for potential Pot

INPUT:
   Pot - Potential instances or list thereof
   R - Galactocentric radius

OUTPUT:
   escape velocity

HISTORY:
   2011-10-09 - Written - Bovy (IAS)

**galpy.potential.vterm**

galpy.potential.vterm(Pot, l, deg=True)

NAME:
   vterm

PURPOSE:
   calculate the terminal velocity at l in this potential

INPUT:
   Pot - Potential instance
   l - Galactic longitude [deg/rad]
   deg= if True (default), l in deg

OUTPUT:
   terminal velocity

HISTORY:
   2013-05-31 - Written - Bovy (IAS)
Specific potentials

Double exponential disk potential

class galpy.potential.DoubleExponentialDiskPotential (amp=1.0, \( \rho_0 \), \( h_\rho \), \( h_z \), maxiter=20, tol=0.001, normalize=False, new=True, kmaxFac=2.0, glorder=10)

Class that implements the double exponential disk potential \( \rho(R,z) = \rho_0 e^{-R/h_\rho} e^{-|z|/h_z} \)

__init__ (amp=1.0, \( \rho_0 \), \( h_\rho \), \( h_z \), maxiter=20, tol=0.001, normalize=False, new=True, kmaxFac=2.0, glorder=10)

NAME: __init__

PURPOSE:

initialize a double-exponential disk potential

INPUT:

amp - amplitude to be applied to the potential (default: 1)
hr - disk scale-length in terms of ro
hz - scale-height
tol - relative accuracy of potential-evaluations
maxiter - scipy.integrate keyword
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.

OUTPUT:

DoubleExponentialDiskPotential object

HISTORY:

2010-04-16 - Written - Bovy (NYU)
2013-01-01 - Re-implemented using faster integration techniques - Bovy (IAS)

Double power-law density spherical potential

class galpy.potential.TwoPowerSphericalPotential (amp=1.0, \( a \), \( \alpha \), \( \beta \), normalize=False)

Class that implements spherical potentials that are derived from two-power density models

\[ \rho(r) = \frac{A}{(r/a)^{\alpha}(1+r/a)^{\beta}} \]

__init__ (amp=1.0, \( a \), \( \alpha \), \( \beta \), normalize=False)

NAME: __init__

PURPOSE:
initialize a two-power-density potential

INPUT:
amp - amplitude to be applied to the potential (default: 1)
a - “scale” (in terms of Ro)
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.

OUTPUT:
(none)

HISTORY:
2010-07-09 - Started - Bovy (NYU)

Jaffe potential

class galpy.potential.JaffePotential (amp=1.0, a=1.0, normalize=False)
Class that implements the Jaffe potential

__init__ (amp=1.0, a=1.0, normalize=False)
NAME:
__init__
PURPOSE:
Initialize a Jaffe potential

INPUT:
amp - amplitude to be applied to the potential
a - “scale” (in terms of Ro)
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.

OUTPUT:
(none)

HISTORY:
2010-07-09 - Written - Bovy (NYU)

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=1.0, q=1.0, core=1e-08, normalize=False)
Class that implements a power-law potential that is flattened in the potential (NOT the density) amp

\phi(R,z) = - \frac{m^2}{R^2 + z^2/q^2} \ \alpha \ m^{\alpha \phi a}
__init__(amp=1.0, alpha=1.0, q=1.0, core=1e-08, normalize=False)

NAME: __init__

PURPOSE: initialize a flattened power-law potential

INPUT:
amp - amplitude to be applied to the potential (default: 1)
alpha - power
q - flattening
core - core radius
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.

OUTPUT: (none)

HISTORY: 2013-01-09 - Written - Bovy (IAS)

Hernquist potential

class galpy.potential.HernquistPotential (amp=1.0, a=1.0, normalize=False)
Class that implements the Hernquist potential

__init__(amp=1.0, a=1.0, normalize=False)

NAME: __init__

PURPOSE: Initialize a Hernquist potential

INPUT:
amp - amplitude to be applied to the potential
a - “scale” (in terms of Ro)
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.

OUTPUT: (none)

HISTORY: 2010-07-09 - Written - Bovy (NYU)
**Kepler potential**

```python
class galpy.potential.KepelerPotential(amp=1.0, normalize=False)
    Class that implements the Kepler potential
    amp
    Phi(r) = \frac{1}{r}

__init__(amp=1.0, normalize=False)
NAME:
    __init__
PURPOSE:
    initialize a Kepler potential
INPUT:
    amp - amplitude to be applied to the potential (default: 1)
    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.
OUTPUT:
    (none)
HISTORY:
    2010-07-10 - Written - Bovy (NYU)
```

**Logarithmic halo potential**

```python
class galpy.potential.LogarithmicHaloPotential(amp=1.0, core=1e-08, q=1.0, normalize=False)
    Class that implements the logarithmic halo potential Phi(r)

__init__(amp=1.0, core=1e-08, q=1.0, normalize=False)
NAME:
    __init__
PURPOSE:
    initialize a Logarithmic Halo potential
INPUT:
    amp - amplitude to be applied to the potential (default: 1)
    core - core radius at which the logarithm is cut
    q - potential flattening (z/q)**2.
    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.
OUTPUT:
    (none)
```
Miyamoto-Nagai potential

class galpy.potential.MiyamotoNagaiPotential (amp=1.0, a=0.0, b=0.0, normalize=False)

Class that implements the Miyamoto-Nagai potential

\[ \phi(R,z) = - \frac{amp}{\sqrt{R^2+(a+\sqrt{z^2+b^2})^2}} \]

__init__(amp=1.0, a=0.0, b=0.0, normalize=False)

NAME: __init__

PURPOSE: initialize a Miyamoto-Nagai potential

INPUT:

amp - amplitude to be applied to the potential (default: 1)
a - "disk scale" (in terms of Ro)
b - "disk height" (in terms of Ro)

normalize - if True, normalize such that \(v_c(1.,0.)=1\), or, if given as a number, such that the force is this fraction of the force necessary to make \(v_c(1.,0.)=1\).

OUTPUT: (none)

HISTORY: 2010-07-09 - Started - Bovy (NYU)

NFW potential

class galpy.potential.NFWPotential (amp=1.0, a=1.0, normalize=False)

Class that implements the NFW potential

__init__(amp=1.0, a=1.0, normalize=False)

NAME: __init__

PURPOSE: Initialize a NFW potential

INPUT:

amp - amplitude to be applied to the potential
a - "scale" (in terms of Ro)

normalize - if True, normalize such that \(v_c(1.,0.)=1\), or, if given as a number, such that the force is this fraction of the force necessary to make \(v_c(1.,0.)=1\).

OUTPUT:
Power-law density spherical potential

class galpy.potential.PowerSphericalPotential (amp=1.0, alpha=1.0, normalize=False)
Class that implements spherical potentials that are derived from power-law density models

\[
\rho(r) = \frac{1}{r^{\alpha}}
\]

__init__ (amp=1.0, alpha=1.0, normalize=False)
NAME:
__init__
PURPOSE:
initialize a power-law-density potential
INPUT:
amp - amplitude to be applied to the potential (default: 1)
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 \(vc(1.,0.)=1\).
OUTPUT:
(none)
HISTORY:
2010-07-10 - Written - Bovy (NYU)

Razor-thin exponential disk potential

class galpy.potential.RazorThinExponentialDiskPotential (amp=1.0, ro=1.0, hr=0.3333333333333333, maxiter=20, tol=0.001, normalize=False, new=True, glorder=100)
Class that implements the razor-thin exponential disk potential \(\rho(R,z) = \rho_0 e^{-R/h_R} \delta(z)\)

__init__ (amp=1.0, ro=1.0, hr=0.3333333333333333, maxiter=20, tol=0.001, normalize=False, 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)
hr - disk scale-length in terms of ro
tol - relative accuracy of potential-evaluations
maxiter - scipy.integrate keyword
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.

OUTPUT:
RazorThinExponentialDiskPotential object

HISTORY:
2012-12-27 - Written - Bovy (IAS)

In addition to these classes, a Milky-Way-like potential is defined as `galpy.potential.MWPotential`. This potential is defined as

```python
>>> 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]
```

and can thus be used like any list of Potentials.

### 2.2.2 2D potentials

#### General instance routines

Use as `Potential-instance.method(...)`

`galpy.potential.planarPotential.__call__`

```python
galpy.potential.planarPotential.__call__(R, phi=0.0, t=0.0, dR=0, dphi=0)
```

**NAME:**
__call__

**PURPOSE:**
evaluate the potential

**INPUT:**
- R - Cylindrical radius
- phi= azimuth (optional)
- t= time (optional)
- dR=, dphi= if set to non-zero integers, return the dR,dphi’ derivative

**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
INPUT:
R - Cylindrical radius
phi= azimuth (optional)
t= time (optional)
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
phi= azimuth (optional)
t= time (optional)
OUTPUT:
F_R(R,(phi,t)))
HISTORY:
2010-07-13 - Written - Bovy (NYU)

General axisymmetric potential instance routines

Use as Potential-instance.method(...)
galpy.potential.planarAxiPotential.epifreq

\texttt{Potential.epifreq}(R)

\textbf{NAME:}

epifreq

\textbf{PURPOSE:}

calculate the epicycle frequency at R in this potential

\textbf{INPUT:}

\begin{itemize}
  \item R - Galactocentric radius
\end{itemize}

\textbf{OUTPUT:}

epicycle frequency

\textbf{HISTORY:}

2011-10-09 - Written - Bovy (IAS)

---

galpy.potential.planarAxiPotential.lindbladR

\texttt{Potential.lindbladR}(\textit{OmegaP}, \textit{m}=2, **kwargs)

\textbf{NAME:}

lindbladR

\textbf{PURPOSE:}

calculate the radius of a Lindblad resonance

\textbf{INPUT:}

\begin{itemize}
  \item \textit{OmegaP} - pattern speed
  \item \textit{m} = order of the resonance (as in \textit{m}(O-Op)=\kappaappa \textbf{negative m for outer}) use \texttt{m='corotation'} for corotation +scipy.optimize.brentq xtol,rtol,maxiter kwargs
\end{itemize}

\textbf{OUTPUT:}

radius of Lindblad resonance, None if there is no resonance

\textbf{HISTORY:}

2011-10-09 - Written - Bovy (IAS)

---

galpy.potential.planarAxiPotential.omegac

\texttt{Potential.omegac}(R)

\textbf{NAME:}

omegac

\textbf{PURPOSE:}

calculate the circular angular speed at R in this potential

\textbf{INPUT:}

\begin{itemize}
  \item R - Galactocentric radius
\end{itemize}
circular angular speed

2011-10-09 - Written - Bovy (IAS)

**galpy.potential.planarAxiPotential.plot**

```python
planarAxiPotential.plot(*args, **kwargs)
```

**NAME:** plot

**PURPOSE:** plot the potential

**INPUT:**
- Rrange - range grid
- number of points to plot
- savefilename - save to or restore from this savefile (pickle)

**OUTPUT:**
- plot to output device

2010-07-13 - Written - Bovy (NYU)

**galpy.potential.planarAxiPotential.plotEscapecurve**

```python
planarAxiPotential.plotEscapecurve(*args, **kwargs)
```

**NAME:** plotEscapecurve

**PURPOSE:**
- plot the escape velocity curve for this potential

**INPUT:**
- Rrange - range
- grid - number of points to plot
- savefilename - save to or restore from this savefile (pickle)

**OUTPUT:**
- plot to output device

2010-07-13 - Written - Bovy (NYU)

**galpy.potential.planarAxiPotential.plotRotcurve**

```python
planarAxiPotential.plotRotcurve(*args, **kwargs)
```

**NAME:** plotRotcurve

**PURPOSE:**
- plot the rotation curve for this potential
INPUT:
   Rrange - range
   grid - number of points to plot
   savefilename - save to or restore from this savefile (pickle)
   +bovy_plot(*args,**kwargs)

OUTPUT:
   plot to output device

HISTORY:
   2010-07-13 - Written - Bovy (NYU)

\texttt{galpy.potential.planarAxiPotential.vcirc}

\texttt{Potential.vcirc(R)}

NAME:
   vcirc

PURPOSE:
   calculate the circular velocity at \( R \) in this potential

INPUT:
   R - Galactocentric radius

OUTPUT:
   circular rotation velocity

HISTORY:
   2011-10-09 - Written - Bovy (IAS)

\texttt{galpy.potential.planarAxiPotential.vesc}

\texttt{Potential.vesc(R)}

NAME:
   vesc

PURPOSE:
   calculate the escape velocity at \( R \) for this potential

INPUT:
   R - Galactocentric radius

OUTPUT:
   escape velocity

HISTORY:
   2011-10-09 - Written - Bovy (IAS)
General 2D potential routines

Use as method(...)

galpy.potential.evaluateplanarphiforces

galpy.potential.evaluateplanarphiforces(R, Pot, phi=None, t=0.0)

NAME:
evaluateplanarphiforces

PURPOSE:
evaluate the phiforce of a (list of) planarPotential instance(s)

INPUT:
R - Cylindrical radius
Pot - (list of) planarPotential instance(s)
phi= azimuth (optional)
t= time (optional)

OUTPUT:
F_phi(R,phi,t))

HISTORY:
2010-07-13 - Written - Bovy (NYU)

galpy.potential.evaluateplanarPotentials

galpy.potential.evaluateplanarPotentials(R, Pot, phi=None, t=0.0, dR=0, dphi=0)

NAME:
evaluateplanarPotentials

PURPOSE:
evaluate a (list of) planarPotential instance(s)

INPUT:
R - Cylindrical radius
Pot - (list of) planarPotential instance(s)
phi= azimuth (optional)
t= time (optional)
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 (R, Pot, phi=\text{None}, t=0.0)**

**NAME:**

evaluateplanarRforces

**PURPOSE:**

evaluate the R-force of a (list of) planarPotential instance(s)

**INPUT:**

- R - Cylindrical radius
- Pot - (list of) planarPotential instance(s)
- phi= azimuth (optional)
- t= time (optional)

**OUTPUT:**

F_R(R(,phi,t))

**HISTORY:**

2010-07-13 - Written - Bovy (NYU)

**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
- grid - grid in R
- savefilename - save to or restore from this savefile (pickle)
  + bovy_plot.bovy_plot args and kwargs

**OUTPUT:**

plot to output device

**HISTORY:**

2010-08-08 - Written - Bovy (NYU)
**galpy.potential.plotplanarPotentials**

```python
galpy.potential.plotplanarPotentials(Pot, *args, **kwargs)
```

**NAME:**  
plotplanarPotentials

**PURPOSE:**  
plot a planar potential

**INPUT:**  
- Rrange - range  
- xrange, yrange - if relevant  
- 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)  
- `+bovy_plot(*args,**kwargs)` or `bovy_dens2d(**kwargs)`

**OUTPUT:**  
plot to output device

**HISTORY:**  
2010-07-13 - Written - Bovy (NYU)

**galpy.potential.plotRotcurve**

```python
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  
- grid - grid in R  
- savefilename - save to or restore from this savefile (pickle)  
- `+bovy_plot.bovy_plot` args and kwargs

**OUTPUT:**  
plot to output device

**HISTORY:**  
2010-07-10 - Written - Bovy (NYU)
Specific potentials

All of the 3D potentials above can be used as two-dimensional potentials in the mid-plane.

**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 and a two spiral potentials are included

**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)

Class that implements the Dehnen bar potential (Dehnen 2000)

__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)

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)
tform - start of bar growth / bar period (default: -4)
tsteady - time at which the bar is fully grown / bar period (default: tform/2)

Either provide:
1.rolr - radius of the Outer Lindblad Resonance for a circular orbit

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.)

2.omegab - rotation speed of the bar
rb - bar radius
Af - bar strength

OUTPUT:
(none)

HISTORY:
2010-11-24 - Started - Bovy (NYU)

Cos(m phi) disk potential

Generalization of the lopsided and elliptical disk potentials to any m.

class galpy.potential.CosmphiDiskPotential

Class that implements the disk potential phi(R,phi) = phio (R/Ro)^p cos[m(phi-phib)]

__init__

NAME:
__init__

PURPOSE:
initialize an cosmphi disk potential

phi(R,phi) = phio (R/Ro)^p cos[m(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
tsteady= time delay at which the perturbation is fully grown (default: 2.)
m= cos( m * (phi - phib) )
p= power-law index of the phi(R) = (R/Ro)^p part

Either:
1.phib= angle (in rad; default=25 degree)
phio= potential perturbation (in terms of phio/vo^2 if vo=1 at Ro=1)
2.cp, sp= m * phio * cos(m * phib), m * phio * sin(m * phib)

OUTPUT:
(none)

HISTORY:
2011-10-27 - Started - Bovy (IAS)
**Elliptical disk potential**

Like in Kuijken & Tremaine

class galpy.potential.EllipticalDiskPotential (amp=1.0, phib=0.436323129985824, p=0.0, twophio=0.01, tform=None, tsteady=None, cp=None, sp=None)

Class that implements the Elliptical disk potential of Kuijken & Tremaine (1994) \( \phi(R,\phi) = \phi_0 (R/R_0)^p \cos(2(\phi - \phi_b)) \)

__init__(amp=1.0, phib=0.436323129985824, p=0.0, twophio=0.01, tform=None, tsteady=None, cp=None, sp=None)

NAME: __init__

PURPOSE:

initialize an Elliptical disk potential

\( \phi(R,\phi) = \phi_0 (R/R_0)^p \cos(2(\phi - \phi_b)) \)

INPUT:

amp= amplitude to be applied to the potential (default: 1.), see twophio below
tform= start of growth (to smoothly grow this potential
tsteady= time delay at which the perturbation is fully grown (default: 2.)
p= power-law index of the \( \phi(R) = (R/R_0)^p \) part

Either:

1.phib= angle (in rad; default=25 degree)
twophio= potential perturbation (in terms of 2\( \phi_0/v_o^2 \) if \( v_o=1 \) at \( R_0=1 \))

2.cp, sp= twophio * cos(2phib), twophio * sin(2phib)

OUTPUT:

(none)

HISTORY:

2011-10-19 - Started - Bovy (IAS)

**Lopsided disk potential**

Like in Kuijken & Tremaine, but for \( m=1 \)

class galpy.potential.LopsidedDiskPotential (amp=1.0, phib=0.436323129985824, p=0.0, phio=0.01, tform=None, tsteady=None, cp=None, sp=None)

Class that implements the disk potential \( \phi(R,\phi) = \phi_0 (R/R_0)^p \cos(\phi-\phi_b) \) See documentation for CosmphiDiskPotential
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)

Class that implements a steady-state spiral potential

\[ V(r,\phi,t) = A/\alpha \cos(\alpha \ln(r) - m(\phi - \Omega \times t - \gamma)) \]

**NAME:**

**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)
- A - force amplitude (\alpha*potential-amplitude; default=0.035)
- omegas= - pattern speed (default=0.65)
- m= number of arms

Either provide:

1. alpha=
2. p= pitch angle (rad)

- tform - start of spiral growth / spiral period (default: -Infinity)
- tsteady - time from tform at which the spiral is fully grown / spiral period (default: tform+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)

Class that implements a steady-state spiral potential

\[ V(r,\phi,t) = A(t)/\alpha \cos(\alpha \ln(r) - m(\phi - \Omega \times t - \gamma)) \]

where

\[ A(t) = A_{\text{max}} \exp(- |t-\text{to}|^2/\sigma^2/2.) \]
__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)

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)
A - force amplitude (alpha*potential-amplitude; default=0.035)
omegas= - pattern speed (default=0.65)
m= number of arms
to= time at which the spiral peaks
sigma= “spiral duration” (sigma in Gaussian amplitude)

Either provide:
1. alpha=
2. p= pitch angle (rad)

OUTPUT: (none)

HISTORY:
2011-03-27 - Started - Bovy (NYU)

2.2.3 1D potentials

General instance routines

Use as Potential-instance.method(...)

galpy.potential.linearPotential.__call__

linearPotential.__call__(x, t=0.0)

NAME: __call__

PURPOSE: evaluate the potential

INPUT:
x - position
t= time (optional)

OUTPUT:
galpy Documentation, Release v0.1

\[ \Phi(x,t) \]

**HISTORY:**
2010-07-12 - Written - Bovy (NYU)

**galpy.potential.linearPotential.force**

```python
linearPotential.force(x, t=0.0)
```

**NAME:** force

**PURPOSE:** evaluate the force

**INPUT:**
- `x` - position
- `t` - time (optional)

**OUTPUT:**
- \( F(x,t) \)

**HISTORY:**
2010-07-12 - Written - Bovy (NYU)

**galpy.potential.linearPotential.plot**

```python
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)
General 1D potential routines

Use as `method(...)`

**galpy.potential.evaluatelinearForces**

```python
galpy.potential.evaluatelinearForces(x, Pot, t=0.0)
```

**NAME:**
evaluatelinearForces

**PURPOSE:**
evaluate the forces due to a list of potentials

**INPUT:**
- `x` - evaluate forces at this position
- `Pot` - (list of) linearPotential instance(s)
- `t` - time to evaluate at

**OUTPUT:**
force(x,t)

**HISTORY:**
2010-07-13 - Written - Bovy (NYU)

**galpy.potential.evaluatelinearPotentials**

```python
galpy.potential.evaluatelinearPotentials(x, Pot, t=0.0)
```

**NAME:**
evaluatelinearPotentials

**PURPOSE:**
evaluate the sum of a list of potentials

**INPUT:**
- `x` - evaluate potentials at this position
- `Pot` - (list of) linearPotential instance(s)
- `t` - time to evaluate at

**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

Vertical Kuijken & Gilmore potential

class galpy.potential.KGPotential(K=1.15, F=0.03, D=1.8, amp=1.0)
Class representing the Kuijken & Gilmore (1989) potential

__init__ (K=1.15, F=0.03, D=1.8, amp=1.0)

NAME: __init__

PURPOSE: Initialize a KGPotential

INPUT:
  K= K parameter
  F= F parameter
  D= D parameter
  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.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

**OUTPUT:**

(list of) linearPotential instance(s)

**HISTORY:**

2010-07-21 - Written - Bovy (NYU)

### 2.3 DF

#### 2.3.1 Two-dimensional 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 vxvv member b) Orbit instance + t: call the Orbit instance (for list, each instance is called at t)

2. E - energy (/vo^2) L - angular momentum (/ro/vo)

3. array vxvv [3/4,nt]

**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 (/ro)
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> (/ro)
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> (/ro)

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> (/ro)

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:
galpy.df.diskdf.meanvT

diskdf.meanvT(R, romberg=False, nsigma=None, phi=0.0)

NAME:
    meanvT

PURPOSE:
    calculate \langle v_T \rangle at R by marginalizing over velocity

INPUT:
    R - radius at which to calculate \langle v_T \rangle (/ro)

OPTIONAL INPUT:
    nsigma - number of sigma to integrate the velocities over

KEYWORDS:
    romberg - if True, use a romberg integrator (default: False)

OUTPUT:
    \langle v_T \rangle 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 (/ro)

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)
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 (/ro)

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 (/ro)

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:
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 (/ro)

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 surfacemass at R by marginalizing over velocity

INPUT:
R - radius at which to calculate the sigma_R^2 x surfacemass density (/ro)

OPTIONAL INPUT:
nsigma - number of sigma to integrate the velocities over

KEYWORDS:
romberg - if True, use a romberg integrator (default: False)
OUTPUT:

\( \sigma_R^2 \times \text{surface-mass at } R \)

HISTORY:

2010-03-XX - Written - Bovy (NYU)

galpy.df.diskdf.sigma2

diskdf.sigma2 \((R, \text{romberg}=\text{False, nsigma=}\text{None, phi}=0.0)\)

NAME:

\text{sigma2}

PURPOSE:

calculate \(\sigma_R^2\) at \(R\) by marginalizing over velocity

INPUT:

\(R\) - radius at which to calculate \(\sigma_R^2\) density \(/ro\)

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 \((R, \text{romberg}=\text{False, nsigma=}\text{None, phi}=0.0)\)

NAME:

\text{sigmaR2} (duplicate of \text{sigma2} for consistency)

PURPOSE:

calculate \(\sigma_R^2\) at \(R\) by marginalizing over velocity

INPUT:

\(R\) - radius at which to calculate \(\sigma_R^2\) \(/ro\)

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 \(R\)
galpy Documentation, Release v0.1

HISTORY:
2011-03-30 - Written - Bovy (NYU)

galpy.df.diskdf.sigmaT2

diskdf.\texttt{sigmaT2}(R, \texttt{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} (\textit{ro})

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.\texttt{skewvR}(R, \texttt{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 \langle vR \rangle (\textit{ro})

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> (\(\ell/ro\))

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 (\(\ell/ro\))

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**

```python
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
- l - Galactic longitude (in deg, unless deg=False)

**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)

**HISTORY:**
2011-03-24 - Written - Bovy (NYU)

**galpy.df.diskdf.targetSigma2**

```python
diskdf.targetSigma2(R, log=False)
```

**NAME:**
targetSigma2

**PURPOSE:**
evaluate the target Sigma_R^2(R)

**INPUT:**
- R - radius at which to evaluate (/ro)

**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
- 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
- $l$ - Galactic longitude (in deg, unless deg=False)
- deg= if False, $l$ is in radians
- log - if True, return the log (default: False)

OUTPUT:

- Sigma($d$, $l$)

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)

NAME:
vmomentSurfacemass

PURPOSE:
calculate the an arbitrary moment of the velocity distribution at $R$ times the surfacemass
INPUT:
   R - radius at which to calculate the moment(/ro)
   n - vR^n
   m - vT^m

OPTIONAL INPUT:
   nsigma - number of sigma to integrate the velocities over

KEYWORDS:
   romberg - if True, use a romberg integrator (default: False)
   deriv= None, ‘R’, or ‘phi’: calculates derivative of the moment wrt R or phi

OUTPUT:
   <vR^n vT^m x surface-mass> at R

HISTORY:
   2011-03-30 - Written - Bovy (NYU)

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
   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
   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 discrep-
galpy.df.diskdf.sampledSurfacemassLOS
diskdf.sampledSurfacemassLOS \((l, n=1, \text{maxd}=\text{None}, \text{target}=\text{True})\)

NAME:
sampledSurfacemassLOS

PURPOSE:
sample a distance along the line of sight

INPUT:
- \(l\) - Galactic longitude (in rad)
- \(n=\) number of distances to sample
- \(\text{maxd}=\) maximum distance to consider (for the rejection sampling)
- \(\text{target}=\) if True, sample from the ‘target’ surface mass density, rather than the actual surface mass density (default=\text{True})

OUTPUT:
list of samples

HISTORY:
2011-03-24 - Written - Bovy (NYU)

hhgalpy.df.diskdf.sampleLOS
diskdf.sampleLOS \((\text{los}, n=1, \text{deg}=\text{True}, \text{maxd}=\text{None}, \text{nsigma}=\text{None}, \text{target}=\text{True})\)

NAME:
sampleLOS

PURPOSE:
sample along a given LOS

INPUT:
- \(\text{los}=\) line of sight (in deg, unless \text{deg}=\text{False})
- \(n=\) number of desired samples
- \(\text{deg}=\) los in degrees? (default=\text{True})
- \(\text{target}=\) if \text{True}, use target surface mass and \text{sigma2} profiles (default=\text{True})

OUTPUT:
returns list of Orbits

BUGS: \(\text{target}=\text{False}\) uses target distribution for derivatives (this is a detail)

HISTORY:
2011-03-24 - Started - Bovy (NYU)
galpy Documentation, Release v0.1

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
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_src.surfaceSigmaProfile.expSurfaceSigmaProfile at 0x6970a78>, profileParams=(0.3333333333333333, 1.0, 0.2), correct=False, beta=0.0, **kwargs)

Dehnen’s ‘new’ df

__init__ (surfaceSigma=<class galpy.df_src.surfaceSigmaProfile.expSurfaceSigmaProfile at 0x6970a78>, 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 / Ro
xS - disk velocity dispersion scalelength / Ro
Sro - disk velocity dispersion at Ro (/vo)

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
+DFcorrection kwargs (except for those already specified)

OUTPUT:
instance

HISTORY:
2010-03-10 - Written - Bovy (NYU)

**Shu DF**

class galpy.df.shudf(surfaceSigma=<class galpy.df_src.surfaceSigmaProfile.expSurfaceSigmaProfile at 0x6970a78>, profileParams=(0.3333333333333333, 1.0, 0.2), correct=False, beta=0.0, **kwargs)

Shu’s df (1969)

__init__(surfaceSigma=<class galpy.df_src.surfaceSigmaProfile.expSurfaceSigmaProfile at 0x6970a78>, 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 / Ro
   xS - disk velocity dispersion scalelength / Ro
   Sro - disk velocity dispersion at Ro /vo
   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
+DFcorrection kwargs (except for those already specified)

OUTPUT:
instance

HISTORY:
2010-05-09 - Written - Bovy (NYU)

**2.3.2 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
      where:  jr - radial action  lz - z-component of angular momentum  jz - vertical action
   2. R,vR,vT,z,vz
   3. 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 ex-
tended 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
   z - height at which to calculate the density

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

\[ z = \text{height (default: 0 pc)} \]

\[ dR = \text{range in R to use} \]

\[ \text{density kwargs} \]

**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

\[ z = \text{height (default: 0 pc)} \]

\[ dR = \text{range in R to use} \]

\[ \text{density kwargs} \]

**OUTPUT:** estimated hsR

**HISTORY:** 2013-03-08 - Written - Bovy (IAS)

### galpy.df.quasiisothermaldf.estimate_hs_z

**quasiisothermaldf.estimate_hs_z** *(R, z=0.0, dR=1e-08, **kwargs)*

**NAME:** estimate_hs_z

**PURPOSE:** estimate the exponential scale length of the vertical dispersion at R

**INPUT:** R - Galactocentric radius

\[ z = \text{height (default: 0 pc)} \]

\[ dR = \text{range in R to use} \]

\[ \text{density kwargs} \]

2.3. DF
galpy Documentation, Release v0.1

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
dz - z range to use
density kwargs

OUTPUT: estimated hsz

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)

NAME: jmomentdensity

PURPOSE: calculate the an arbitrary moment of an action of the velocity distribution at R times the surfacmass

INPUT: R - radius at which to calculate the moment/(ro)
n - jr^n
m - lz^m
o - jz^o

OPTIONAL INPUT: nsigma - number of sigma to integrate the velocities over (when doing explicit numerical integral)
mc= if True, calculate using Monte Carlo integration
nmc= if mc, use nmc samples

OUTPUT: <jr^n lz^m jz^o x density> at R

HISTORY: 2012-08-09 - Written - Bovy (IAS@MPIA)

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
z - height at which to calculate this

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.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
z - height at which to calculate this

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 momentum by marginalizing over velocity

INPUT:
R - radius at which to calculate this
z - height at which to calculate this

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
z - height at which to calculate this

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
z - height at which to calculate this

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
z - height at which to calculate this

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)

NAME: pvR

PURPOSE: calculate the marginalized vR probability at this location (NOT normalized by the density)

INPUT:
vR - radial velocity (/vo)
R - radius (/ro)
z - height (/ro)
gl - use Gauss-Legendre integration (True, currently the only option)
ngl - order of Gauss-Legendre integration

**OUTPUT:**  \( p(v_R, R, z) \)

**HISTORY:**  2012-12-22 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.pvRvT

**quasiisothermaldf.pvRvT** \((v_R, v_T, R, z, gl=True, ngl=20)\)

**NAME:** pvRvT

**PURPOSE:** calculate the marginalized \((v_R,v_T)\) probability at this location (NOT normalized by the density)

**INPUT:**
- \( v_R \) - radial velocity (/vo)
- \( v_T \) - tangential velocity (/vo)
- \( R \) - radius (/ro)
- \( z \) - height (/ro)
- \( gl \) - use Gauss-Legendre integration (True, currently the only option)
- \( ngl \) - order of Gauss-Legendre integration

**OUTPUT:**  \( p(v_R,v_T,R,z) \)

**HISTORY:**  2013-01-02 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.pvRvz

**quasiisothermaldf.pvRvz** \((v_R, v_z, R, z, gl=True, ngl=20)\)

**NAME:** pvRvz

**PURPOSE:** calculate the marginalized \((v_R,v_z)\) probability at this location (NOT normalized by the density)

**INPUT:**
- \( v_R \) - radial velocity (/vo)
- \( v_z \) - vertical velocity (/vo)
- \( R \) - radius (/ro)
- \( z \) - height (/ro)
- \( gl \) - use Gauss-Legendre integration (True, currently the only option)
- \( ngl \) - order of Gauss-Legendre integration

**OUTPUT:**  \( p(v_R,v_z,R,z) \)

**HISTORY:**  2013-01-02 - Written - Bovy (IAS)
**galpy.df.quasiisothermaldf.pvT**

`quasiisothermaldf.pvT(vT, R, z, gl=True, ngl=20)`

**NAME:** pvT

**PURPOSE:** calculate the marginalized vT probability at this location (NOT normalized by the density)

**INPUT:**
- vT - tangential velocity (/vo)
- R - radius (/ro)
- z - height (/ro)
- gl - use Gauss-Legendre integration (True, currently the only option)
- ngl - order of Gauss-Legendre integration

**OUTPUT:** p(vT,R,z)

**HISTORY:** 2012-12-22 - Written - Bovy (IAS)

**galpy.df.quasiisothermaldf.pTvz**

`quasiisothermaldf.pTvz(vT, vz, R, z, gl=True, ngl=20)`

**NAME:** pvTvz

**PURPOSE:** calculate the marginalized (vT,vz) probability at this location (NOT normalized by the density)

**INPUT:**
- vT - tangential velocity (/vo)
- vz - vertical velocity (/vo)
- R - radius (/ro)
- z - height (/ro)
- gl - use Gauss-Legendre integration (True, currently the only option)
- ngl - order of Gauss-Legendre integration

**OUTPUT:** p(vT,vz,R,z)

**HISTORY:** 2012-12-22 - Written - Bovy (IAS)

**galpy.df.quasiisothermaldf.pvz**

`quasiisothermaldf.pvz(vz, R, z, gl=True, ngl=20, _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)
galpy Documentation, Release v0.1

INPUT: vz - vertical velocity (/vo)
       R - radius (/ro)
       z - height (/ro)
       gl - use Gauss-Legendre integration (True, currently the only option)
       ngl - order of Gauss-Legendre integration

OUTPUT: p(vz,R,z)

HISTORY: 2012-12-22 - Written - Bovy (IAS)

**galpy.df.quasiisothermaldf.sampleV**

quasiisothermaldf.sampleV(R, z, n=1)

NAME: sampleV

PURPOSE: sample a radial, azimuthal, and vertical velocity at R,z

INPUT:
       R - Galactocentric distance
       z - height
       n= number of distances to sample

OUTPUT: list of samples

HISTORY: 2012-12-17 - Written - Bovy (IAS)

**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
       z - height at which to calculate this

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
galpy.df.quasiisothermaldf.sigmaRz

\texttt{quasiisothermaldf.\texttt{sigmaRz}}(R, z, \texttt{nsigma=None}, \texttt{mc=False}, \texttt{nmc=10000}, \texttt{gl=True}, \texttt{ngl=10}, **\texttt{kwargs})

\textbf{NAME: sigmaRz}  
\textbf{PURPOSE:} calculate $\sigma_{RZ}^2$ by marginalizing over velocity  
\textbf{INPUT:}  
\begin{itemize}
  \item $R$ - radius at which to calculate this  
  \item $z$ - height at which to calculate this  
\end{itemize}
\textbf{OPTIONAL INPUT:}  
\begin{itemize}
  \item $\texttt{nsigma}$ - number of sigma to integrate the velocities over  
  \item scipy.integrate.tplquad \texttt{kwargs} \texttt{epsabs} and \texttt{epsrel}  
  \item \texttt{mc=} if True, calculate using Monte Carlo integration  
  \item \texttt{nmc=} if \texttt{mc}, use \texttt{nmc} samples  
  \item \texttt{gl=} if True, calculate using Gauss-Legendre integration  
  \item \texttt{ngl=} if \texttt{gl}, use \texttt{ngl}-th order Gauss-Legendre integration for each dimension  
\end{itemize}
\textbf{OUTPUT:} $\sigma_{Rz}^2$  
\textbf{HISTORY:} 2012-07-30 - Written - Bovy (IAS@MPIA)

galpy.df.quasiisothermaldf.sigmaT2

\texttt{quasiisothermaldf.\texttt{sigmaT2}}(R, z, \texttt{nsigma=None}, \texttt{mc=False}, \texttt{nmc=10000}, \texttt{gl=True}, \texttt{ngl=10}, **\texttt{kwargs})

\textbf{NAME: sigmaT2}  
\textbf{PURPOSE:} calculate $\sigma_T^2$ by marginalizing over velocity  
\textbf{INPUT:}  
\begin{itemize}
  \item $R$ - radius at which to calculate this  
  \item $z$ - height at which to calculate this  
\end{itemize}
\textbf{OPTIONAL INPUT:}  
\begin{itemize}
  \item $\texttt{nsigma}$ - number of sigma to integrate the velocities over  
  \item scipy.integrate.tplquad \texttt{kwargs} \texttt{epsabs} and \texttt{epsrel}  
  \item \texttt{mc=} if True, calculate using Monte Carlo integration  
  \item \texttt{nmc=} if \texttt{mc}, use \texttt{nmc} samples  
  \item \texttt{gl=} if True, calculate using Gauss-Legendre integration  
  \item \texttt{ngl=} if \texttt{gl}, use \texttt{ngl}-th order Gauss-Legendre integration for each dimension  
\end{itemize}
\textbf{OUTPUT:} $\sigma_T^2$  
\textbf{HISTORY:} 2012-07-30 - Written - Bovy (IAS@MPIA)
**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
- `z` - height at which to calculate this

**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
  - `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` - minimum z to use
  - `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
z - height at which to calculate this

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: tilt in degree

HISTORY: 2012-12-23 - Written - Bovy (IAS)

galpy.df.quasiisothermaldf.vmomentdensity

quasiisothermaldf.vmomentdensity(R, n, m, o, nsigma=None, mc=False, nmc=10000, _returnmc=False, _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)

NAME: vmomentdensity

PURPOSE: calculate the an arbitrary moment of the velocity distribution at R times the density

INPUT: R - radius at which to calculate the moment/(ro)

n - vR^n
m - vT^m
o - vz^o

OPTIONAL INPUT: nsigma - number of sigma to integrate the velocities over (when doing explicit numerical integral)
mc= if True, calculate using Monte Carlo integration
nmc= if mc, use nmc samples

OUTPUT: _returngl= if True, return the evaluated DF
Specific distribution functions

Quasi-isothermal DF

class galpy.df.quasiisothermaldf

\(hr\), \(sr\), \(sz\), \(hsr\), \(hsz\), \(pot=\)None, \(aA=\)None, \(cutcounter=\)False, 
\(_\text{precomputerg}=\)True, \(_\text{precomputergrmax}=\)None, \(_\text{precomputergnLz}=\)51, \(ro=\)1.0, \(lo=\)0.005681818181818182

Class that represents a ‘Binney’ quasi-isothermal DF

\texttt{\_\_init\_\_}(hr, sr, sz, \(hsr\), \(hsz\), \(pot=\)None, \(aA=\)None, \(cutcounter=\)False, \(_\text{precomputerg}=\)True, \(_\text{precomputergrmax}=\)None, \(_\text{precomputergnLz}=\)51, \(ro=\)1.0, \(lo=\)0.005681818181818182)

NAME:

\texttt{\_\_init\_\_}

PURPOSE:

Initialize a quasi-isothermal DF

INPUT:

\(hr\) - radial scale length
\(sr\) - radial velocity dispersion at the solar radius
\(sz\) - vertical velocity dispersion at the solar radius
\(hsr\) - radial-velocity-dispersion scale length
\(hsz\) - vertical-velocity-dispersion scale length
\(pot=\) Potential instance or list thereof
\(aA=\) actionAngle instance used to convert (\(x,v\)) to actions
\(cutcounter=\) if True, set counter-rotating stars’ DF to zero
\(ro=\) reference radius for surface mass and sigmas
\(lo=\) reference angular momentum below where there are significant numbers of retrograde stars

OTHER INPUTS:

\(_\text{precomputerg}=\) if True (default), pre-compute the \(rL(L)\)
\(_\text{precomputergrmax}=\) if set, this is the maximum \(R\) for which to pre-compute \(rg\) (default: \(5^*hr\))
\(_\text{precomputergnLz}\) if set, number of \(Lz\) to pre-compute \(rg\) for (default: \(51\))

OUTPUT:

\texttt{object}

HISTORY:

2012-07-25 - Started - Bovy (\texttt{IAS@MPIA})
2.4 actionAngle

2.4.1 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__)

```python
galpy.actionAngle.actionAngle.__call__
```

actionAngle.__call__(*args, **kwargs)

NAME: __call__
PURPOSE: evaluate the actions (jr,lz,jz)
INPUT:

Either:

1. R,vR,vT,z,vz:
   (a) floats: phase-space value for single object
   (b) numpy.ndarray: [N] phase-space values for N objects
   (c) numpy.ndarray: [N,M] phase-space values for N objects at M times

2. Orbit instance or list thereof; can be integrated already

OUTPUT: (jr,lz,jz)

HISTORY: 2014-01-03 - Written for top level - Bovy (IAS)

```python
galpy.actionAngle.actionAngle.actionsFreqs
```

actionAngle.actionsFreqs(*args, **kwargs)

NAME: actionsFreqs
PURPOSE: evaluate the actions and frequencies (jr,lz,jz,Omegar,Omegaphi,Omegaz)
INPUT:

Either:

1. R,vR,vT,z,vz:
   (a) floats: phase-space value for single object
   (b) numpy.ndarray: [N] phase-space values for N objects
   (c) numpy.ndarray: [N,M] phase-space values for N objects at M times

2. Orbit instance or list thereof; can be integrated already

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,Omegar,Omegaphi,Omegaz,angler,anglephi,anglez)
INPUT:
    Either:
    1.R,vR,vT,z,vz:
        (a) floats: phase-space value for single object
        (b) numpy.ndarray: [N] phase-space values for N objects
        (c) numpy.ndarray: [N,M] phase-space values for N objects at M times
    2.Orbit instance or list thereof; can be integrated already

OUTPUT: (jr,lz,Omegar,Omegaphi,Omegaz,angler,anglephi,anglez)
HISTORY: 2014-01-03 - Written for top level - Bovy (IAS)

2.4.2 Specific actionAngle modules

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
    ip= instance of a IsochronePotential

OUTPUT: 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
OUTPUT: HISTORY:
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

OUTPUT: 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
+scipy.integrate.quad keywords

OUTPUT: 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
               useu0 - use u0 to calculate dV (NOT recommended)
               c= if True, always use C for calculations
      OUTPUT: HISTORY:
               2012-11-27 - Written - Bovy (IAS)

actionAngleStaeckelGrid

class galpy.actionAngle.actionAngleStaeckelGrid(pot=None, delta=None, Rmax=5.0, nE=25, npsi=25, nLz=25, numcores=1, **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=25, numcores=1, **kwargs)
      NAME: __init__
      PURPOSE: initialize an actionAngleStaeckelGrid object
      INPUT: pot= potential or list of potentials
               delta= focus of prolate confocal coordinate system
               Rmax = Rmax for building grids
               nE, npsi, nLz = grid size
               numcores= number of cpus to use to parallelize
               +scipy.integrate.quad keywords
      OUTPUT: HISTORY:
               2012-11-29 - Written - Bovy (IAS)

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
- 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
- nttintJ= (default: 10000) number of time-integration points actions
- integrate_method= (default: ‘dopr54_c’) integration method to use

**OUTPUT:**

**HISTORY:**

2013-09-10 - Written - Bovy (IAS)

## 2.5 Utilities

### 2.5.1 galpy.util.bovy_plot

Various plotting routines:

**galpy.util.bovy_plot.bovy_dens2d**

**NAME:**

bovy_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)

**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.bovy_plot.bovy_end_print

galpy.util.bovy_plot.bovy_end_print(filename, **kwargs)
NAME:
bovy_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.bovy_plot.bovy_hist**

```python
galpy.util.bovy_plot.bovy_hist(x, xlabel=None, ylabel=None, overplot=False, **kwargs)
```

**NAME:**
bovy_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
- `**kwargs`: +all pyplot.hist keywords

**OUTPUT:**
(from the matplotlib docs: [http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.hist](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.bovy_plot.bovy_plot**

```python
galpy.util.bovy_plot.bovy_plot(*args, **kwargs)
```

**NAME:**
bovy_plot

**PURPOSE:**
wrapper around matplotlib’s plot function

**INPUT:**
- `*args`: see [http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot](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` and `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
- `onedhists`: if True, make one-d histograms on the sides

---

### 2.5. Utilities

189
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.bovy_plot.bovy_print

galpy.util.bovy_plot.bovy_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:
bovy_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.bovy_plot.bovy_text

galpy.util.bovy_plot.bovy_text(*args, **kwargs)

NAME:
bovy_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.bovy_plot.scatterplot**

**galpy.util.bovy_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
contours - if False, don’t plot contours
cntrcolors - color of contours (can be array as for bovy_dens2d)
cntrlw, cntrls - linewidths and linestyles for contour
cntrSmooth - use ndimage.gaussian_filter to smooth before contouring
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)

2.5.2 galpy.util.bovy_conversion

Conversion between galpy's natural units and physical units

galpy.util.bovy_conversion.dens_in_msolpc3

galpy.util.bovy_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.bovy_conversion.force_in_2piGmsolpc2

galpy.util.bovy_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.bovy_conversion.force_in_pcMyr2

galpy.util.bovy_conversion.force_in_pcMyr2(\text{vo}, \text{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.bovy_conversion.force_in_kmsMyr

galpy.util.bovy_conversion.force_in_kmsMyr(\text{vo}, \text{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.bovy_conversion.freq_in_Gyr

**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.bovy_conversion.freq_in_kmskpc

**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.bovy_conversion.surfdens_in_msolpc2

**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)

\texttt{galpy.util.bovy_conversion.mass\_in\_msol}

\texttt{galpy.util.bovy_conversion.mass\_in\_msol}(vo, ro)

NAME:
mass\_in\_msol

PURPOSE:
convert a mass to M\text{sol}

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)

\texttt{galpy.util.bovy_conversion.mass\_in\_1010msol}

\texttt{galpy.util.bovy_conversion.mass\_in\_1010msol}(vo, ro)

NAME:
mass\_in\_1010msol

PURPOSE:
convert a mass to 10^10 x M\text{sol}

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.bovy_conversion.time_in_Gyr

galpy.util.bovy_conversion.time_in_Gyr(\textit{vo}, \textit{ro})

\textbf{NAME:}

\textit{time\_in\_Gyr}

\textbf{PURPOSE:}

convert a time to Gyr

\textbf{INPUT:}

\textit{vo} - velocity unit in km/s
\textit{ro} - length unit in kpc

\textbf{OUTPUT:}

conversion from units where \textit{vo}=1. at \textit{ro}=1.

\textbf{HISTORY:}

2013-09-01 - Written - Bovy (IAS)

2.5.3 \textbf{galpy.util.bovy\_coords}

Various coordinate transformation routines with fairly self-explanatory names:

\textbf{galpy.util.bovy\_coords.cov\_dvrpmllbb\_to\_vxyz}

galpy.util.bovy_coords.cov_dvrpmllbb_to_vxyz\textit{(d, e\_d, e\_vr, pmll, pmbb, cov\_pmllbb, l, b, plx=False, degree=False)}

\textbf{NAME:}

cov\_dvrpmllbb\_to\_vxyz

\textbf{PURPOSE:}

propagate distance, radial velocity, and proper motion uncertainties to Galactic coordinates

\textbf{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

l - Galactic longitude
b - Galactic lattitude

\textbf{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
**galpy Documentation, Release v0.1**

### OUTPUT:

\[ \text{cov}(vx, vy, vz) \] [3,3] or [ :,3,3 ]

**HISTORY:**

2010-04-12 - Written - Bovy (NYU)

#### galpy.util.bovy_coords.cov_pmrapmdec_to_pmllpmbb

**NAME:**

cov_pmrapmdec_to_pmllpmbb

**PURPOSE:**

propagate the proper motions errors through the rotation from (ra,dec) to (l,b)

**INPUT:**

- \text{covar\_pmradec} - uncertainty covariance matrix of the proper motion in ra (multiplied with \text{cos}(dec)) and dec [2,2] or [ ,2,2 ]
- \text{ra} - right ascension
- \text{dec} - declination
- \text{degree} - if True, ra and dec are given in degrees (default=False)
- \text{epoch} - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported)

**OUTPUT:**

\text{covar\_pmlbb} [2,2] or [ ,2,2 ]

**HISTORY:**

2010-04-12 - Written - Bovy (NYU)

#### galpy.util.bovy_coords.cyl_to_rect

**NAME:**

cyl_to_rect

**PURPOSE:**

convert from cylindrical to rectangular coordinates

**INPUT:**

- \text{R, phi, Z} - cylindrical coordinates

**OUTPUT:**

[ :,3 ] X,Y,Z

**HISTORY:**

2011-02-23 - Written - Bovy (NYU)

### 2.5. Utilities
galpy.util.bovy_coords.cyl_to_rect_vec

```python
galpy.util.bovy_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.bovy_coords.dl_to_rphi_2d

```python
galpy.util.bovy_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.bovy_coords.galcencyl_to_XYZ

```python
galpy.util.bovy_coords.galcencyl_to_XYZ(R, phi, Z, Xsun=1.0, Ysun=0.0, Zsun=0.0)
```

**NAME:**
galcencyl_to_XYZ

**PURPOSE:**
transform cylindrical Galactocentric coordinates to XYZ coordinates (wrt Sun)

**INPUT:**
R, phi, Z - Galactocentric cylindrical coordinates

**OUTPUT:**
[:,3]= X,Y,Z

**HISTORY:**
2011-02-23 - Written - Bovy (NYU)

---

galpy.util.bovy_coords.galcencyl_to_vxvyvz

```python
galpy.util.bovy_coords.galcencyl_to_vxvyvz(vR, vT, vZ, phi, vsun=[0.0, 1.0, 0.0])
```

**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 ndarray[3]

**OUTPUT:**
vx,vy,vz

**HISTORY:**
2011-02-24 - Written - Bovy (NYU)

---

galpy.util.bovy_coords.galcenrect_to_vxvyvz

```python
galpy.util.bovy_coords.galcenrect_to_vxvyvz(vXg, vYg, vZg, vsun=[0.0, 1.0, 0.0])
```

**NAME:**
galcenrect_to_vxvyvz

**PURPOSE:**
transform rectangular Galactocentric coordinates to XYZ coordinates (wrt Sun) for velocities

---

2.5. Utilities
INPUT:

\( v_X^g \) - Galactocentric x-velocity
\( v_Y^g \) - Galactocentric y-velocity
\( v_Z^g \) - Galactocentric z-velocity
\( v_{\text{sun}} \) - velocity of the sun \( \text{ndarray}[3] \)

OUTPUT:

\( \[:,3\] = vx, vy, vz \)

HISTORY:

2011-02-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.lb_to_radec

galpy.util.bovy_coords.lb_to_radec(\( l, b, \text{degree}=False, \text{epoch}=2000.0 \))

NAME:

lb_to_radec

PURPOSE:

transform from Galactic coordinates to equatorial coordinates

INPUT:

\( l \) - Galactic longitude
\( b \) - Galactic latitude
degree - (Bool) if True, \( l \) and \( b \) are given in degree and ra and dec will be as well
epoch - epoch of target ra,dec (right now only 2000.0 and 1950.0 are supported)

OUTPUT:

ra,dec

For vector inputs \( \[:,2\] \)

HISTORY:

2010-04-07 - Written - Bovy (NYU)

galpy.util.bovy_coords.lbd_to_XYZ

galpy.util.bovy_coords.lbd_to_XYZ(\( l, b, d, \text{degree}=False \))

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 latitude (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)

galpy.util.bovy_coords.pmllpmbb_to_pmrapmdec

galpy.util.bovy_coords.pmllpmbb_to_pmrapmdec(pmll, pmmb, 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 (multiplied with cos(b)) [mas/yr]
pmmb - proper motion in b [mas/yr]
l - Galactic longitude
b - Galactic latitude
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)

OUTPUT:
(pmra,pmdec), for vector inputs [:,2]

HISTORY:
2010-04-07 - Written - Bovy (NYU)

galpy.util.bovy_coords.cov_pmrapmdec_to_pmllpmbb

galpy.util.bovy_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 (multiplied 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)

OUTPUT:
covar_pmllb [2,2] or [:,2,2]

HISTORY:
2010-04-12 - Written - Bovy (NYU)

---

galpy.util.bovy_coords.radec_to_lb

galpy.util.bovy_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)

OUTPUT:
l,b

For vector inputs [:,2]

HISTORY:
2009-11-12 - Written - Bovy (NYU)

---

galpy.util.bovy_coords.rectgal_to_sphergal

galpy.util.bovy_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,pmbb) in (rad,rad,kpc,km/s,mas/yr,mas/yr)

HISTORY:
2009-10-25 - Written - Bovy (NYU)

**galpy.util.bovy_coords.rect_to_cyl**

galpy.util.bovy_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:
[:,3] R,phi,z

HISTORY:
2010-09-24 - Written - Bovy (NYU)

**galpy.util.bovy_coords.rect_to_cyl_vec**

galpy.util.bovy_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

2.5. Utilities 203
\textbf{Y - Y}
\textbf{Z - Z}
\textbf{cyl - if True, X,Y,Z are already cylindrical}

\textbf{OUTPUT:}
\textbf{vR,vT,vz}

\textbf{HISTORY:}
2010-09-24 - Written - Bovy (NYU)

\textbf{galpy.util.bovy_coords.rphi_to_dl_2d}

\texttt{galpy.util.bovy_coords.rphi_to_dl_2d}(R, phi, degree=False, ro=1.0, phio=0.0)

\textbf{NAME:}
rphi_to_dl_2d

\textbf{PURPOSE:}
convert Galactocentric radius and azimuth to distance and Galactic longitude

\textbf{INPUT:}
R - Galactocentric radius
phi - Galactocentric azimuth [rad/deg if degree]

\textbf{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]

\textbf{OUTPUT:}
(d,l); phi in degree if degree

\textbf{HISTORY:}
2012-01-04 - Written - Bovy (IAS)

\textbf{galpy.util.bovy_coords.Rz_to_coshucosv}

\texttt{galpy.util.bovy_coords.Rz_to_coshucosv}(R, z, delta=1.0)

\textbf{NAME:}
Rz_to_coshucosv

\textbf{PURPOSE:}
calculate prolate confocal \texttt{cosh(u)} and \texttt{cos(v)} coordinates from \texttt{R,z, and delta}

\textbf{INPUT:}
R - radius
z - height
delta= focus
OUTPUT:
(cosh(u), cos(v))

HISTORY:
2012-11-27 - Written - Bovy (IAS)

galpy.util.bovy_coords.Rz_to_uv

galpy.util.bovy_coords.Rz_to_uv(R, z, delta=1.0)
NAME:
Rz_to_uv
PURPOSE:
calculate prolate confocal u and v coordinates from R,z, and delta
INPUT:
R - radius
z - height
delta= focus
OUTPUT:
(u,v)
HISTORY:
2012-11-27 - Written - Bovy (IAS)

galpy.util.bovy_coords.sphergal_to_rectgal

galpy.util.bovy_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)
galpy.util.bovy_coords.uv_to_Rz

**galpy.util.bovy_coords.uv_to_Rz** *(u, v, delta=1.0)*

**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

**OUTPUT:**

(R,z)

**HISTORY:**

2012-11-27 - Written - Bovy (IAS)

---

galpy.util.bovy_coords.vrpmllpmbb_to_vxvyvz

**galpy.util.bovy_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)
HISTORY:
2009-10-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.vxvyvz_to_galcencyl

galpy.util.bovy_coords.vxvyvz_to_galcencyl(vx, vy, vz, X, Y, Z, vsun=[0.0, 1.0, 0.0], galcen=False)

NAME:
vxvyvz_to_galcencyl

PURPOSE:
transform XYZ coordinates (wrt Sun) to cylindrical Galactocentric coordinates for velocities

INPUT:
vx - U
vy - V
vz - W
X - X
Y - Y
Z - Z
vsun - velocity of the sun ndarray[3]
galcen - if True, then X,Y,Z are in cylindrical Galactocentric coordinates

OUTPUT:
vRg, vTg, vZg

HISTORY:
2010-09-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.vxvyvz_to_galenrect

galpy.util.bovy_coords.vxvyvz_to_galenrect(vx, vy, vz, vsun=[0.0, 1.0, 0.0])

NAME:
vxvyvz_to_galenrect

PURPOSE:
transform XYZ coordinates (wrt Sun) to rectangular Galactocentric coordinates for velocities

INPUT:
vx - U
vy - V
vz - W
vsun - velocity of the sun ndarray[3]

OUTPUT:
[:,3]= vXg, vYg, vZg
galpy Documentation, Release v0.1

HISTORY:
2010-09-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.vxvyvz_to_vrpmllpmbb

galpy.util.bovy_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 latitude
  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, pmbb) in (km/s, mas/yr, mas/yr); pmll = mu_l * cos(b)
HISTORY:
2009-10-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.XYZ_to_galcencyl

galpy.util.bovy_coords.XYZ_to_galcencyl(X, Y, Z, Xsun=1.0, Ysun=0.0, Zsun=0.0)
NAME:
  XYZ_to_galcencyl
PURPOSE:
  transform XYZ coordinates (wrt Sun) to cylindrical Galactocentric coordinates
INPUT:
  X - X
  Y - Y
  Z - Z
OUTPUT:
  [:,3]= R,phi,z
HISTORY:
galpy.util.bovy_coords.XYZ_to_galcenrect

```python
galpy.util.bovy_coords.XYZ_to_galcenrect(X, Y, Z, Xsun=1.0, Ysun=0.0, Zsun=0.0)
```

**NAME:**

XYZ_to_galcenrect

**PURPOSE:**

transform XYZ coordinates (wrt Sun) to rectangular Galactocentric coordinates

**INPUT:**

- X - X
- Y - Y
- Z - Z

**OUTPUT:**

- (Xg, Yg, Zg)

**HISTORY:**

2010-09-24 - Written - Bovy (NYU)

---

galpy.util.bovy_coords.XYZ_to_lbd

```python
galpy.util.bovy_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,rad,kpc)

For vector inputs [:,3]

**HISTORY:**

2009-10-24 - Written - Bovy (NYU)
2.5.4 galpy.util.bovy_ars.bovy_ars

galpy.util.bovy_ars.bovy_ars(domain, isDomainFinite, abcissae, hx, hpx, nsamples=1, hx-
params=(), maxn=100)

Rejection Sampling for Gibbs Sampling, Applied Statistics, 41, 337 Based on Wild & Gilks (1993), Algorithm

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)
3.1 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; ). 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 \((J_R, J_\phi, J_Z)\), 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 \(J_\phi\) and the black dot shows the progenitor orbit. These actions were calculated using \texttt{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). 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 \(\theta_\phi\)).

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.

3.1.1 Modeling streams in galpy

In galpy we can model streams using the tools in \texttt{galpy.df.streamdf}. We setup a streamdf instance by specifying the host gravitational potential \texttt{pot=}, an actionAngle method (typically \texttt{galpy.actionAngle.actionAngleIsochroneApprox}), a \texttt{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

```python
>>> from galpy.df import streamdf
>>> from galpy.orbit import Orbit
>>> from galpy.potential import LogarithmicHaloPotential
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.util import bovy_conversion
```

setup the potential and actionAngle instances

```python
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAI= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

define a progenitor Orbit instance

```python
>>> obs= Orbit([1.56148083,0.35081535,-1.15481504,0.88719443,-0.47713334,0.12019596])
```

and instantiate the streamdf model

```python
>>> sigv= 0.365 #km/s
>>> sdf= streamdf(sigv/220.,progenitor=obs,pot=lp,aA=aAI,leading=True,nTrackChunks=11,tdisrupt=4.5/bovy_conversion.time_in_Gyr(220.,8.))
```

for a leading stream. This runs in about half a minute on a 2011 Macbook Air.

We can calculate some simple properties of the stream, such as the ratio of the largest and second-to-largest eigenvalue of the Hessian

```python
>>> 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

```python
>>> 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

```python
>>> sdf.misalignment()
-0.49526013844831596
```

which returns this angle in degrees. We can also calculate the angle between the frequency vector of the progenitor and the principal eigenvector of $\partial \Omega / \partial J$

```python
>>> sdf.misalignment(isotropic=True)
1.2825116841963993
```

(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).

### 3.1.2 Calculating the average stream location (track)

We can display the stream track in various coordinate systems as follows

```python
>>> sdf.plotTrack(d1='r',d2='z',interp=True,color='k',spread=2,overplot=False, lw=2., scaleToPhysical=True)
```
which gives

![Graph showing 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 \((x, v) \rightarrow (\Omega, \theta)\) transformation and the track position is explicitly calculated, by turning off the interpolation]

```python
>>> 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

```python
>>> 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)

```python
>>> 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.,

```python
>>> 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.

### 3.1.3 Mock stream data generation

We can also easily generate mock data from the stream model. This uses `streamdf.sample`. For example,

```python
>>> 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

```python
galpy Documentation, Release v0.1
galpy Documentation, Release v0.1
galpy Documentation, Release v0.1
```

```python
>>> sdf.plotTrack(d1='r',d2='z',interp=True,color='b',spread=2,overplot=False,lw=2.,scaleToPhysical=True)
```

```python
>>> plot(RvR[0]*8.,RvR[3]*8.,'k.',ms=2.)
```

```python
>>> xlim(12.,16.5); ylim(2.,7.6)
```

which gives
Similarly, we can generate mock data in observable coordinates

```python
galpy> lb = sdf.sample(n=1000, lb=True)
```

and plot it

```python
galpy> sdf.plotTrack(d1='ll', d2='dist', interp=True, color='b', spread=2, overplot=False, lw=2.)
galpy> plot(lb[0], lb[2], 'k.', ms=2.)
galpy> xlim(155., 235.); ylim(7.5, 10.8)
```

which displays
We can also just generate mock stream data in frequency-angle coordinates

```python
>>> mockA = 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 angle

```python
>>> plot(mockA[0][0], mockA[0][2], 'k.', ms=2.)
```
or we can plot the magnitude of the angle offset as a function of stripping time

```python
>>> plot(mockA[2], numpy.sqrt(numpy.sum((mockA[1] - numpy.tile(sdf._progenitor_angle, (1000, 1)).T)**2., axis=0)),'k.', ms=2.)
```
3.1.4 Evaluating and marginalizing the full PDF

We can also evaluate the stream PDF, the probability of a \((x,v)\) phase-space position in the stream. We can evaluate the PDF, for example, at the location of the progenitor

```python
>>> 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

```python
>>> 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)

```python
>>> 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.
We can now evaluate the PDF $p(X|Z)$ as a function of $X$ near the peak

```python
>>> 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

```python
>>> ps/= numpy.sum(ps)*(xs[1]-xs[0])*8.
```

and plot it together with the Gaussian approximation

```python
>>> 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

![PDF plot](image)

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).

```python
>>> 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:

```python
>>> 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

```python
>>> 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

```python
>>> 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, None, 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

```python
>>> 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.
Please let me (bovy -at- ias.edu) know if you make use of galpy in a publication.

  Uses what later became the orbit integration routines and Dehnen and Shu disk distribution functions.

  Employs galpy orbit integration in galpy.potential.MWPotential to characterize the orbits in the SEGUE G dwarf sample.

  Uses galpy.potential force and density routines to characterize the difference between the vertical force and the surface density at large heights above the MW midplane.

  Utilizes the Dehnen distribution function to inform a simple model of the velocity distribution of APOGEE stars in the Milky Way disk and to create mock data.

- **A direct dynamical measurement of the Milky Way’s disk surface density profile, disk scale length, and dark matter profile at 4 kpc**
  Makes use of potential models, the adiabatic and Staeckel actionAngle modules, and the quasiisothermal DF to model the dynamics of the SEGUE G dwarf sample in mono-abundance bins.

  Employs galpy to perform orbit integrations in galpy.potential.MWPotential to characterize the orbits of stars in the APOGEE sample.
ACKNOWLEDGING GALPY

Please link back to http://github.com/jobovy/galpy. When using the galpy.actionAngle modules, please cite 2013ApJ...779..115B in addition to the papers describing the algorithm used. When orbit integrations are used, you could cite 2010ApJ...725.1676B (first galpy paper).
Indices and tables

- genindex
- modindex
- search