galpy Documentation

Release v1.3.0

Jo Bovy

May 09, 2022

Contents

galpy is a Python 2 and 3 package for galactic dynamics. It supports orbit integration in a variety of potentials, evaluating and sampling various distribution functions, and the calculation of action-angle coordinates for all static potentials. galpy is an astropy affiliated package and provides full support for astropy's Quantity framework for variables with units.

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

As a preview of the kinds of things you can do with galpy, here's an *animation* of the orbit of the Sun in galpy's `MWPotential2014` potential over 7 Gyr:

CHAPTER 1

Quick-start guide

1.1 Installation

1.1.1 With conda

The easiest way to install the latest released version of galpy is using conda:

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

or:

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

1.1.2 With pip

galpy can also be installed using pip. 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). Then do:

```
pip install galpy
```

or to upgrade without upgrading the dependencies:

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

1.1.3 Latest version

The latest updates in galpy can be installed using:

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

or:

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

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

```
python setup.py install
```

or:

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

for a local installation.

1.1.4 Installing from a branch

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

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

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

1.1.5 Installing the TorusMapper code

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

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

```
git clone https://github.com/jobovy/Torus.git galpy/actionAngle_src/actionAngleTorus_
↪c_ext/torus
cd galpy/actionAngle_src/actionAngleTorus_c_ext/torus
git checkout galpy
cd -
```

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

1.1.6 Installation FAQ

What is the required `numpy` version?

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

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

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

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

This is typically because the TorusMapper code was not compiled, because it was unavailable during installation. This code is only necessary if you want to use `galpy.actionAngle.actionAngleTorus`. See *above* for instructions on how to install the TorusMapper code.

How do I install the GSL?

Certain advanced features require the GNU Scientific Library (GSL), with action calculations requiring version 1.14 or higher. On a Mac, the easiest way to install the GSL is using Homebrew as:

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

You should be able to check your version using:

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

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

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

The `galpy` installation fails because of C compilation errors

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

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

or:

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

or:

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

This is typically because the compiler cannot locate the GSL header files or the GSL library. You can tell the installation about where you've installed the GSL library by defining (for example, when the GSL was installed under `/usr`):

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

or:

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

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

I'm having issues with OpenMP

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

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

or when using pip as follows:

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

or:

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

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

```
ld: library not found for -lgomp

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

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

1.1.7 Configuration file

Since v1.2, `galpy` uses a configuration file to set a small number of configuration variables. This configuration file is parsed using ConfigParser/configparser. It is currently used to set a default set of distance and velocity scales (`ro` and `vo` throughout galpy) for conversion between physical and internal galpy units, to decide whether to use seaborn plotting with galpy's defaults (which affects *all* plotting after importing `galpy.util.bovy_plot`), to specify whether output from functions or methods should be given as an astropy Quantity with units as much as possible or not, and whether or not to use astropy's coordinate transformations (these are typically somewhat slower than galpy's own coordinate transformations, but they are more accurate and more general). The current configuration file therefore looks like this:

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

[plot]
seaborn-bovy-defaults = False
```

(continues on next page)

(continued from previous page)

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

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

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

1.2 What's new?

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

1.2.1 v1.3

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

  - Improved plotting functions: plotting of spherical radius and of arbitrary user-supplied functions of time in Orbit.plot, Orbit.plot3d, and Orbit.animate.

- `actionAngleStaeckel` upgrades:

  - `actionAngleStaeckel` methods now allow for different focal lengths delta for different phase-space points and for the order of the Gauss-Legendre integration to be specified (default: 10, which is good enough when using actionAngleStaeckel to compute approximate actions etc. for an axisymmetric potential).

  - Added an option to the estimateDeltaStaeckel function to facilitate the return of an estimated delta parameter at every phase space point passed, rather than returning a median of the estimate at each point.

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

1.2.2 v1.2

- Full support for providing inputs to all initializations, methods, and functions as astropy Quantity with units and for providing outputs as astropy Quantities.

- `galpy.potential.TwoPowerTriaxialPotential`, a set of triaxial potentials with iso-density contours that are arbitrary, similar, coaxial ellipsoids whose 'radial' density is a (different) power-law at small and large radii: 1/m^alpha/(1+m)^beta-alpha (the triaxial generalization of TwoPowerSphericalPotential, with flattening in the density rather than in the potential; includes triaxial Hernquist and NFW potentials.

- `galpy.potential.SCFPotential`, a class that implements general density/potential pairs through the basis expansion approach to solving the Poisson equation of Hernquist & Ostriker (1992). Also implemented functions to compute the coefficients for a given density function. See more explanation *here*.

- `galpy.actionAngle.actionAngleTorus`: an experimental interface to Binney & McMillan's TorusMapper code for computing positions and velocities for given actions and angles. See the installation instructions for how to properly install this. See *this section* and the `galpy.actionAngle` API page for documentation.

- `galpy.actionAngle.actionAngleIsochroneApprox` (Bovy 2014) now implemented for the general case of a time-independent potential.

- `galpy.df.streamgapdf`, a module for modeling the effect of a dark-matter subhalo on a tidal stream. See Sanders et al. (2016). Also includes the fast methods for computing the density along the stream and the stream track for a perturbed stream from Bovy et al. (2016).

- `Orbit.flip` can now flip the velocities of an orbit in-place by specifying `inplace=True`. This allows correct velocities to be easily obtained for backwards-integrated orbits.

- `galpy.potential.PseudoIsothermalPotential`, a standard pseudo-isothermal-sphere potential. `galpy.potential.KuzminDiskPotential`, a razor-thin disk potential.

- Internal transformations between equatorial and Galactic coordinates are now performed by default using astropy's coordinates module. Transformation of (ra,dec) to Galactic coordinates for general epochs.

1.2.3 v1.1

- Full support for Python 3.

- `galpy.potential.SnapshotRZPotential`, a potential class that can be used to get a frozen snapshot of the potential of an N-body simulation.

- Various other potentials: `PlummerPotential`, a standard Plummer potential; `MN3ExponentialDiskPotential`, an approximation to an exponential disk using three Miyamoto-Nagai potentials (Smith et al. 2015); `KuzminKutuzovStaeckelPotential`, a Staeckel potential that can be used to approximate the potential of a disk galaxy (Batsleer & Dejonghe 1994).
- Support for converting potential parameters to NEMO format and units.
- Orbit fitting in custom sky coordinates.

1.3 Introduction

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

1.3.1 Rotation curves

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

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

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

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

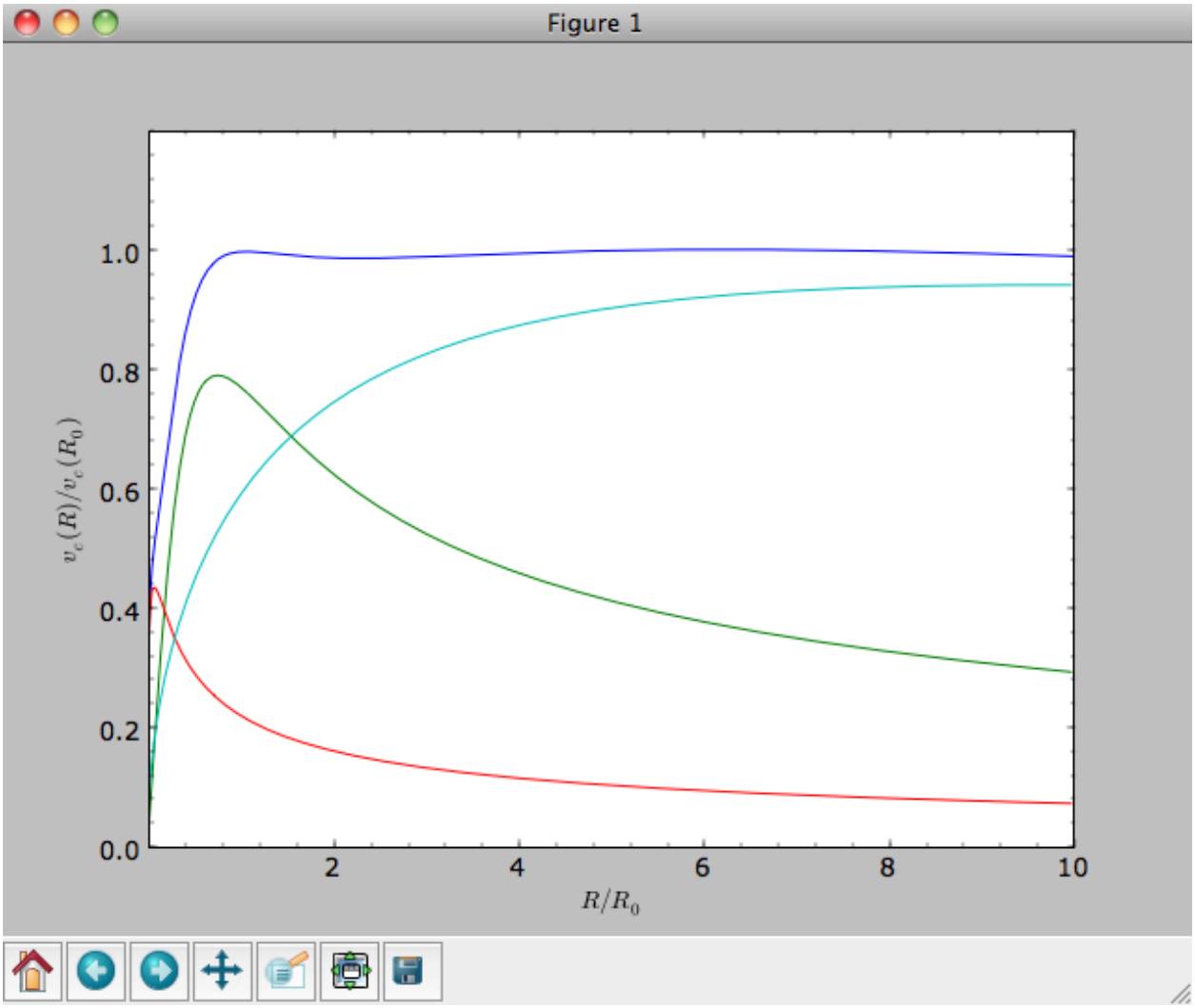
Similarly we can initialize other potentials and plot the combined rotation curve

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

Note that the `normalize` values add up to 1. such that the circular velocity will be 1 at R=1. The resulting rotation curve is approximately flat. To show the rotation curves of the three components do

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

You'll see the following



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

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

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

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

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

1.3.2 Units in galpy

Internal (natural) units

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

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

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

```
>>> from galpy.util import bovy_conversion
>>> 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

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

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

```
>>> -evaluatezforces(MWPotential2014, 1.,1.1/8.)*bovy_conversion.force_in_
↪2piGmsolpc2(220.,8.)
# 71.658016957792356
```

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

`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

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

or

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

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

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

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

Physical units

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

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

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

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

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

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

Warning: If you combine potentials in a list, galpy uses the `ro` and `vo` scales from the first potential in the list for physical <-> internal unit conversion. galpy does **not** always check whether the unit systems of various objects are consistent when they are combined (but does check this for many common cases, e.g., integrating an Orbit in a Potential).

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

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

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

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

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

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

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

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

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

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

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

outputs will be in internal units

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

If you setup a Potential, Orbit, etc. object without specifying the parameters as a Quantity, the default is to return output in natural units, except when `ro=` and `vo=` scales are specified. `ro=` and `vo=` can always be given as a Quantity themselves. `ro=` and `vo=` can always also be specified on a method-by-method basis, overwriting an object's default. For example

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

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

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

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

1.3.3 Orbit integration

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

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

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

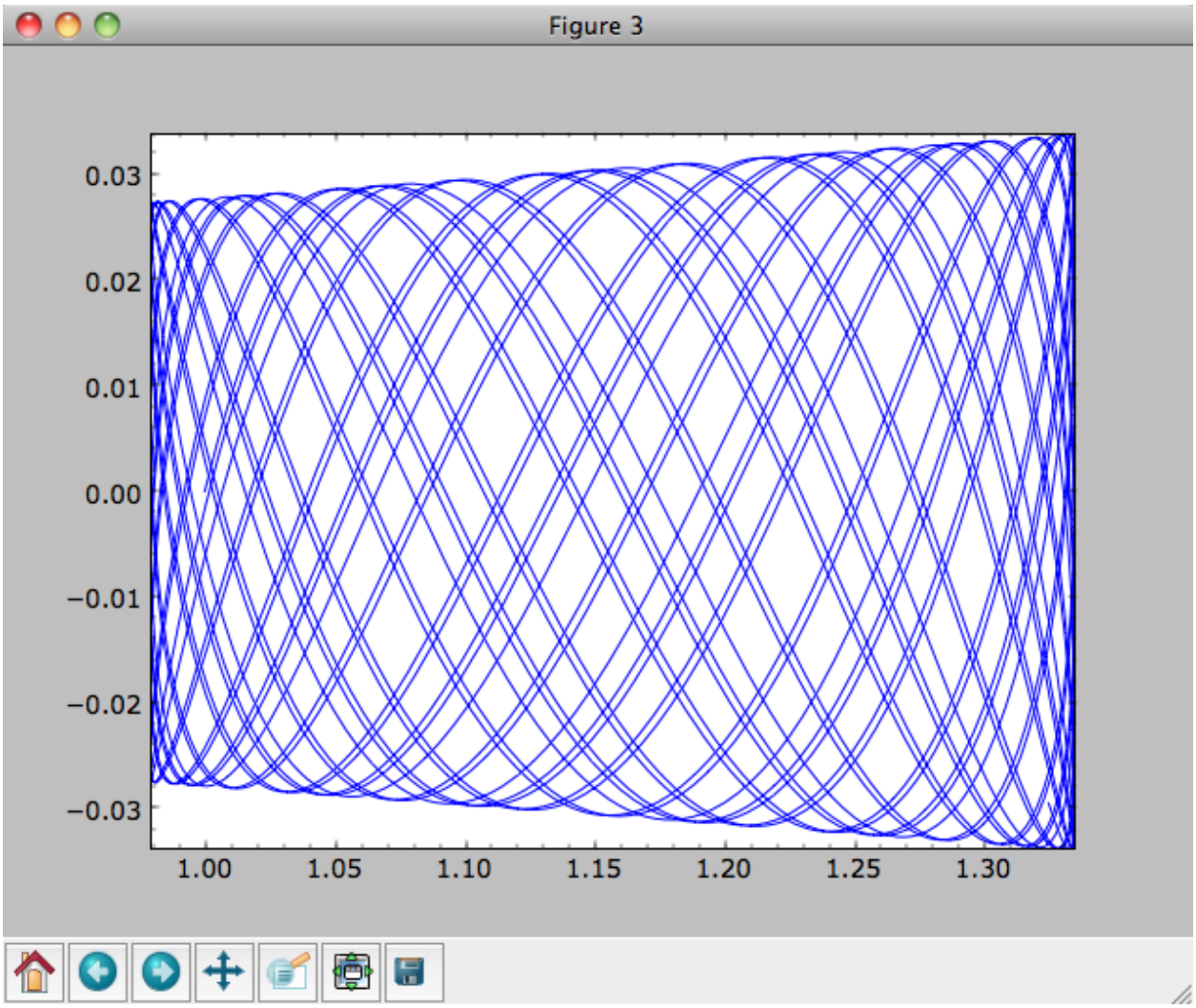
```
>>> import numpy
>>> ts= numpy.linspace(0,100,10000)
>>> o.integrate(ts,mp,method='odeint')
```

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

Now we plot the resulting orbit as

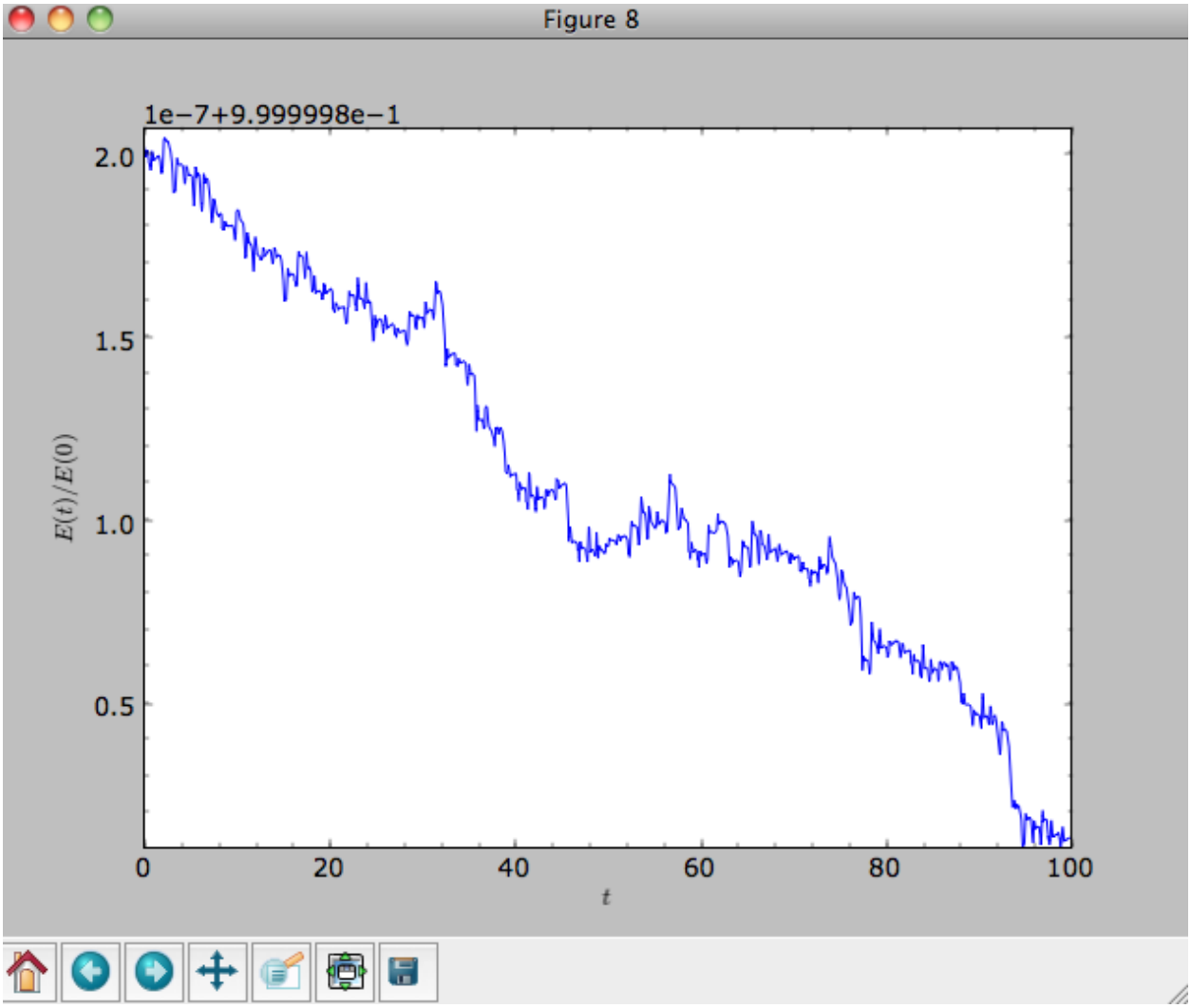
```
>>> o.plot()
```

Which gives



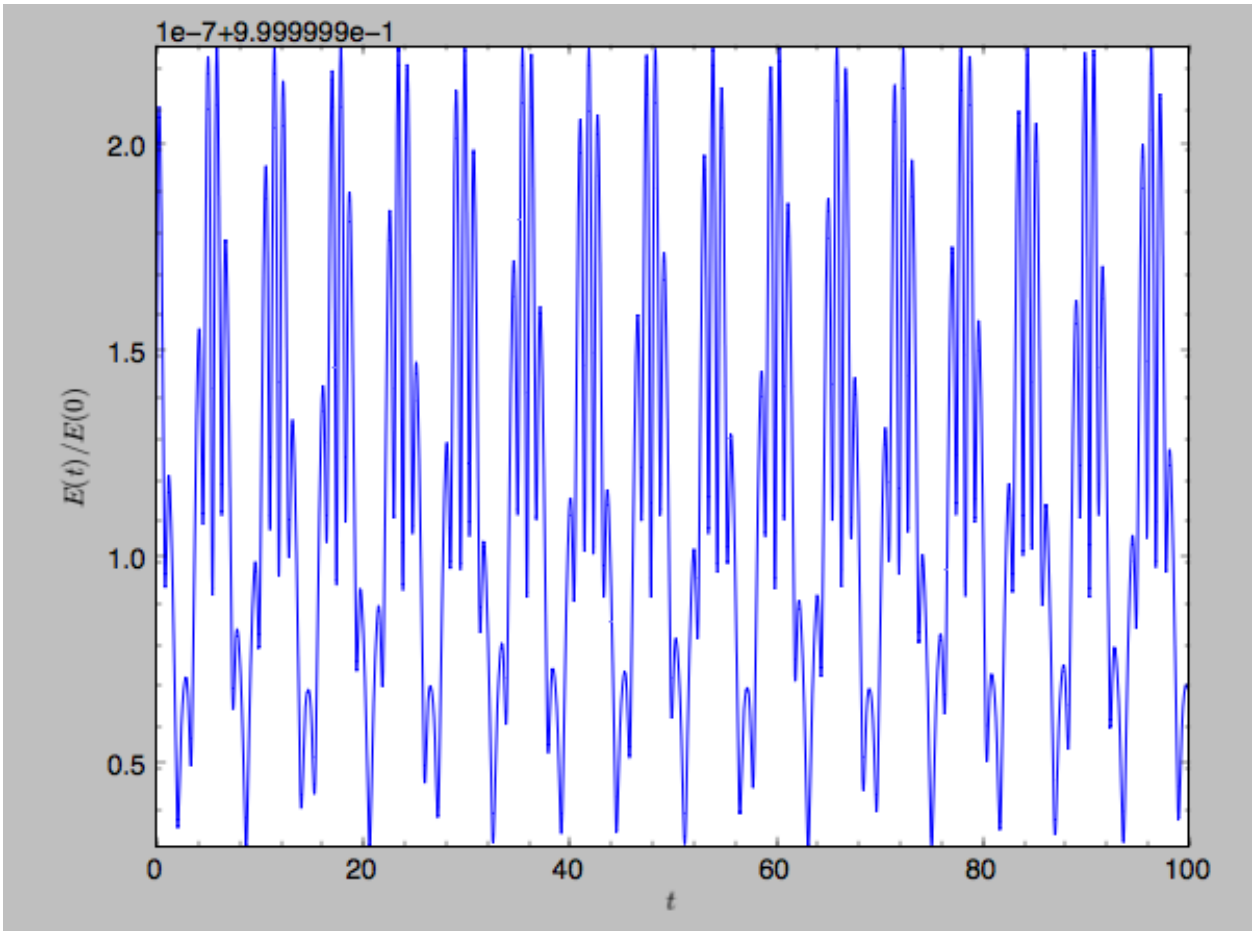
The integrator used is not symplectic, so the energy error grows with time, but is small nonetheless

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



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

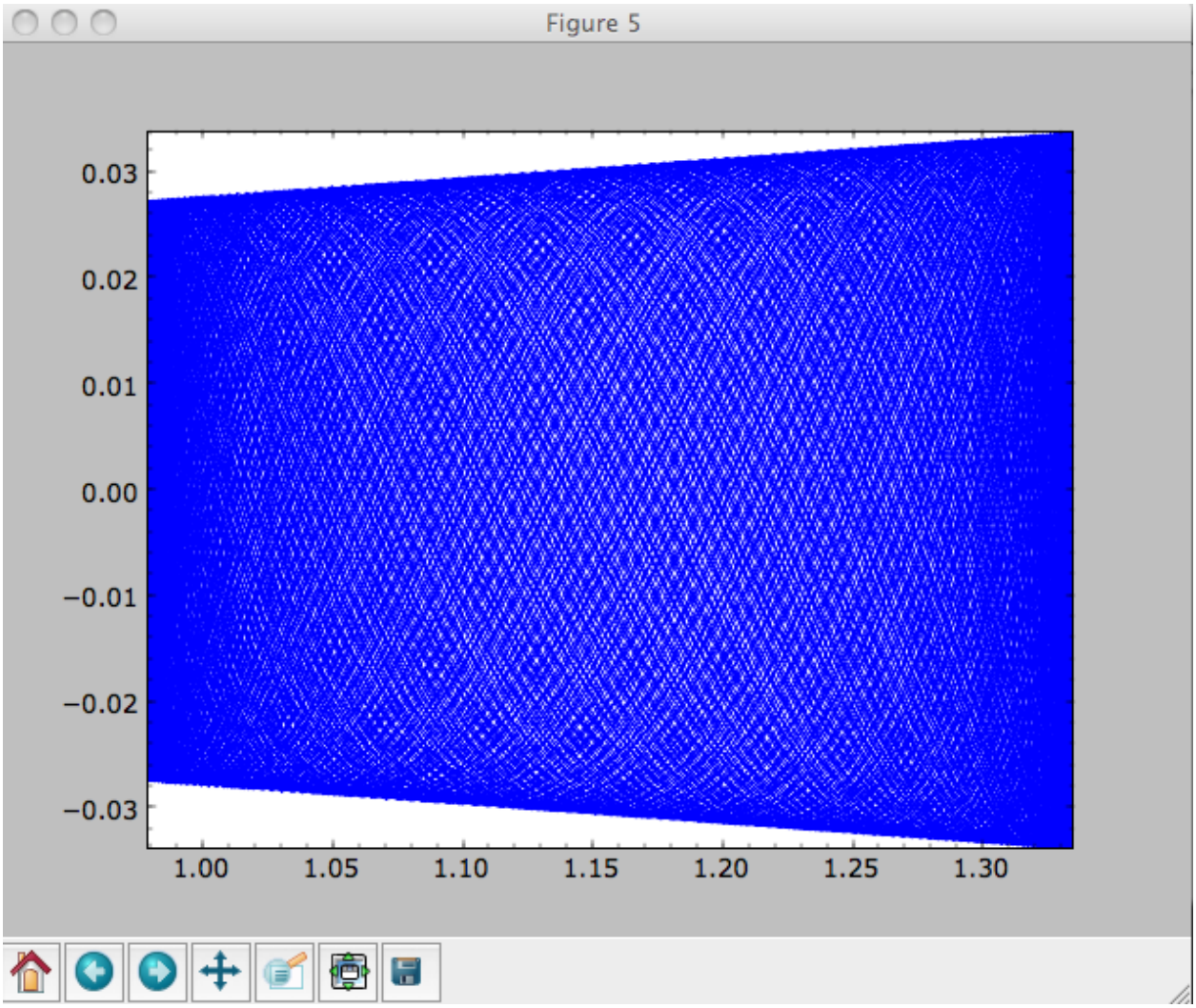
```
>>> o.integrate(ts,mp,method='leapfrog')
>>> o.plotE(xlabel=r'$t$',ylabel=r'$E(t)/E(0)$')
```



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

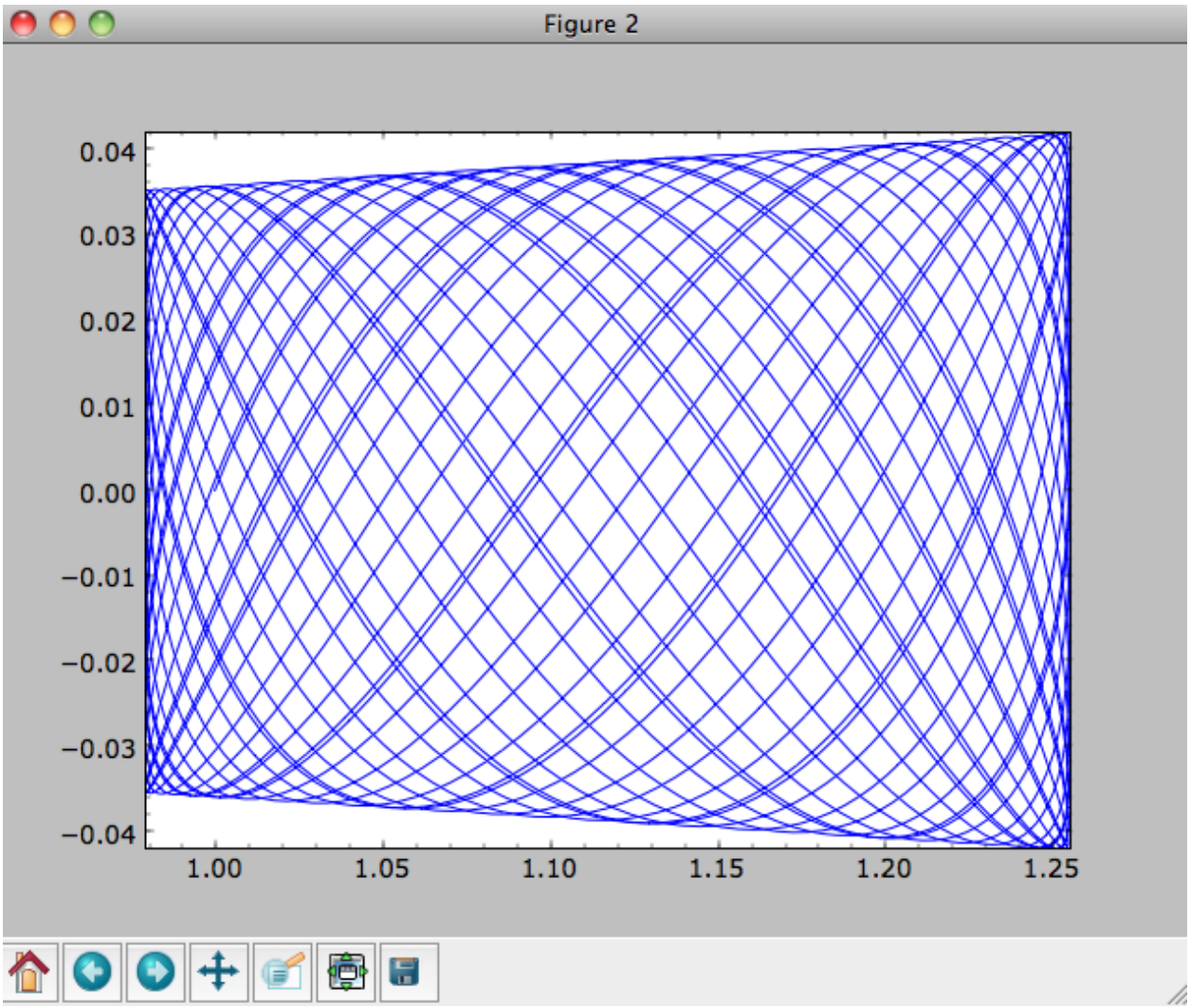
When we integrate for much longer we see how the orbit fills up a torus (this could take a minute)

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



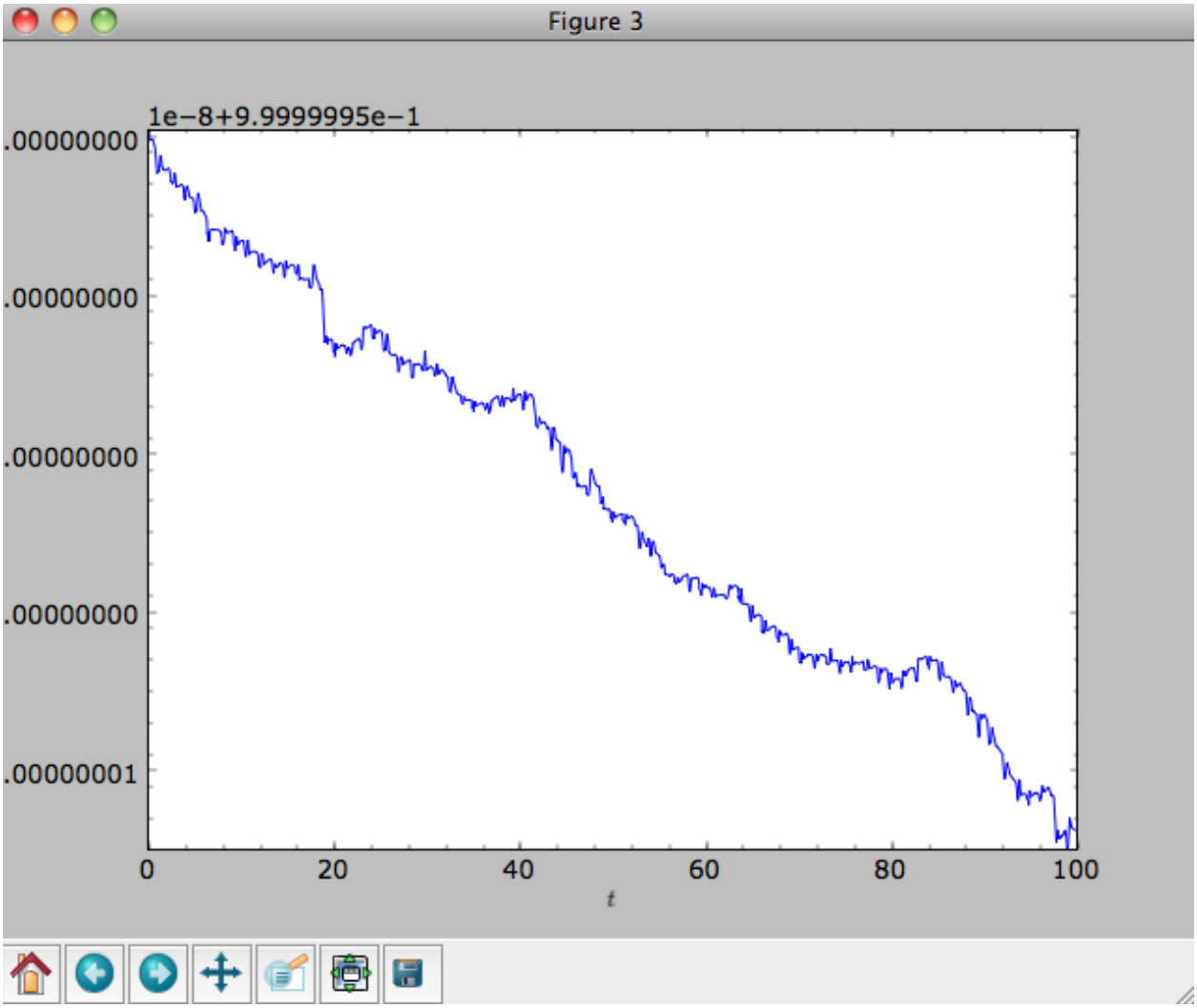
As before, we can also integrate orbits in combinations of potentials. Assuming `mp`, `np`, and `hp` were defined as above, we can

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



Energy is again approximately conserved

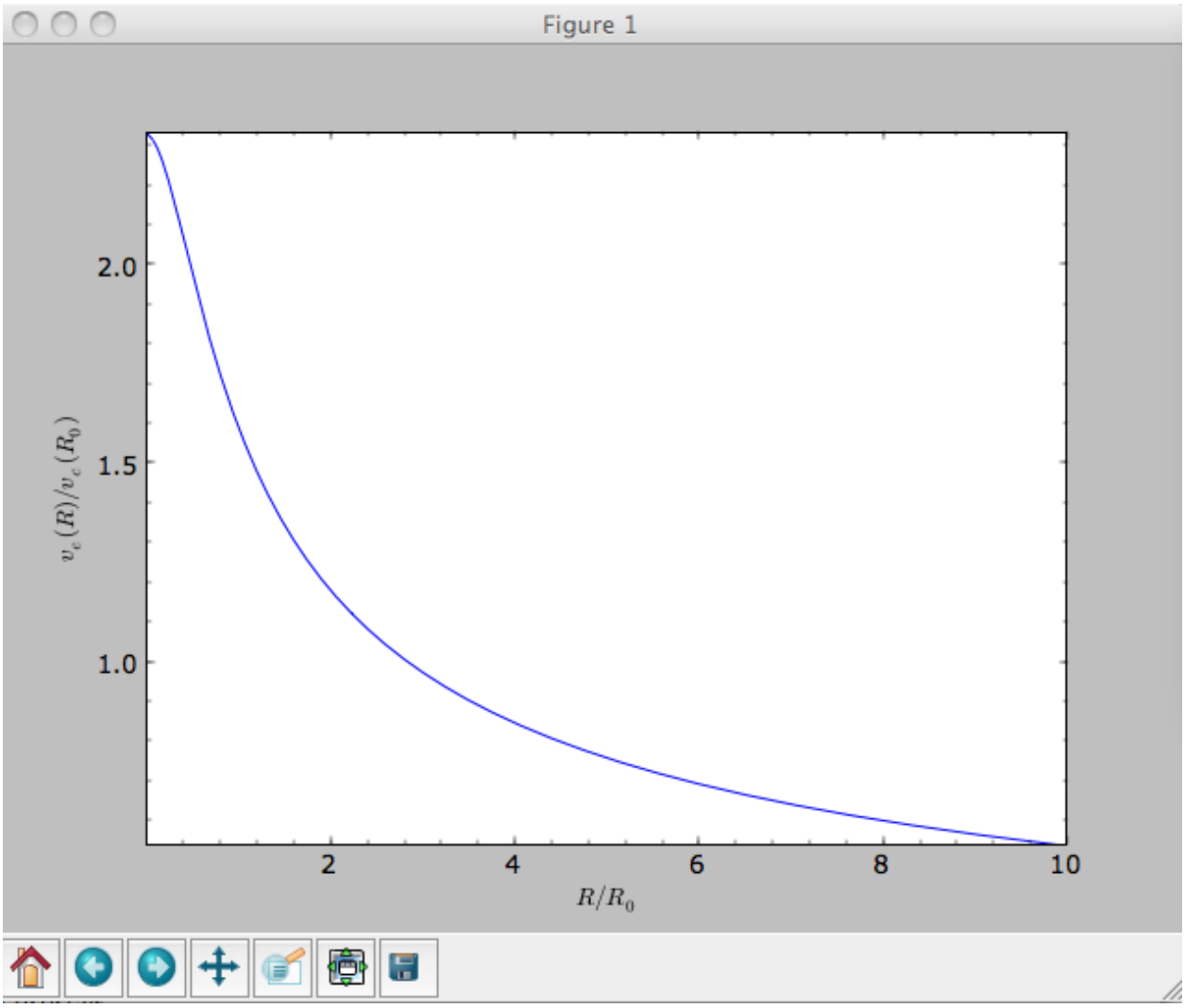
```
>>> o.plotE(xlabel=r'$t$',ylabel=r'$E(t)/E(0)$')
```



1.3.4 Escape velocity curves

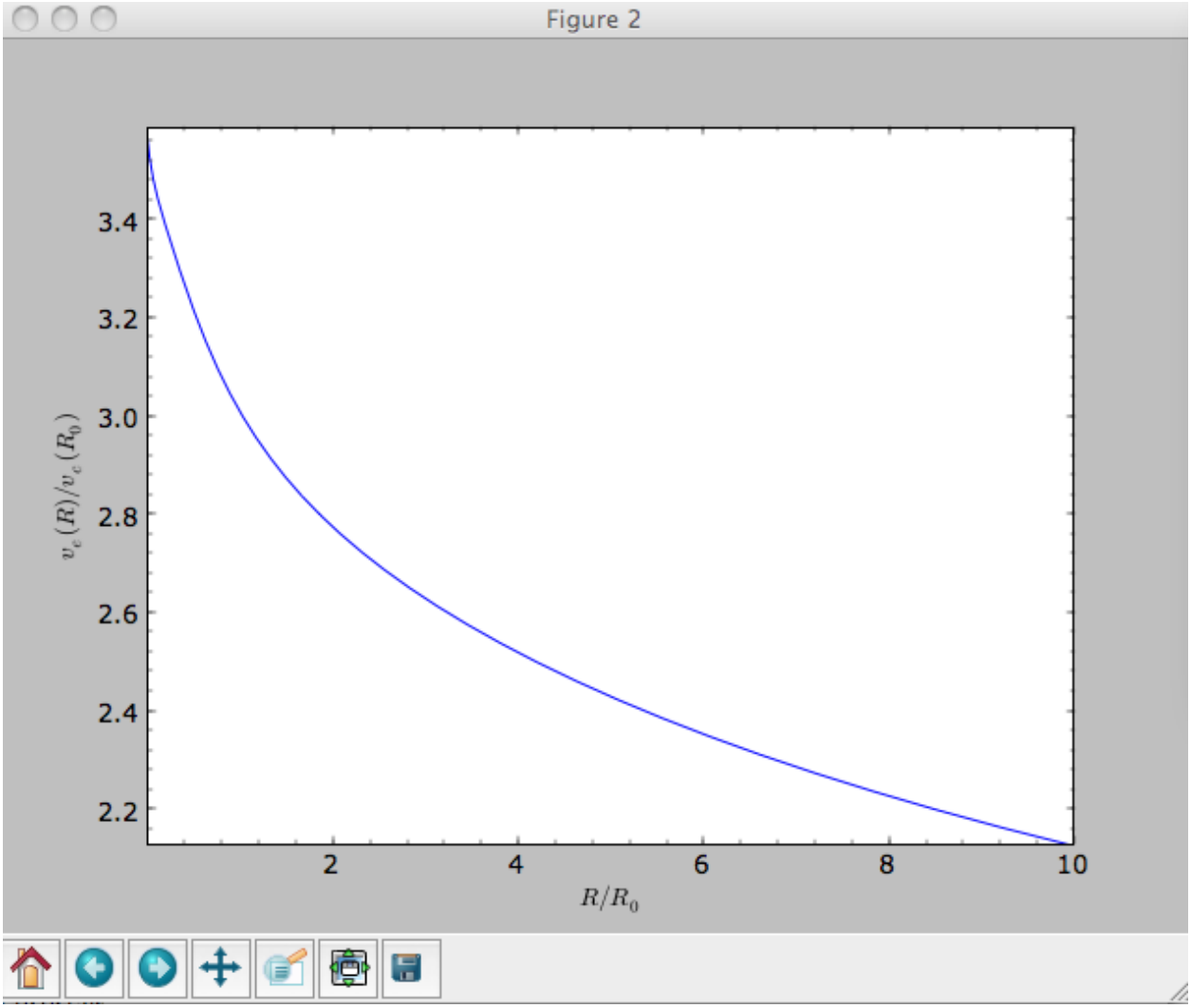
Just like we can plot the rotation curve for a potential or a combination of potentials, we can plot the escape velocity curve. For example, the escape velocity curve for the Miyamoto-Nagai disk defined above

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



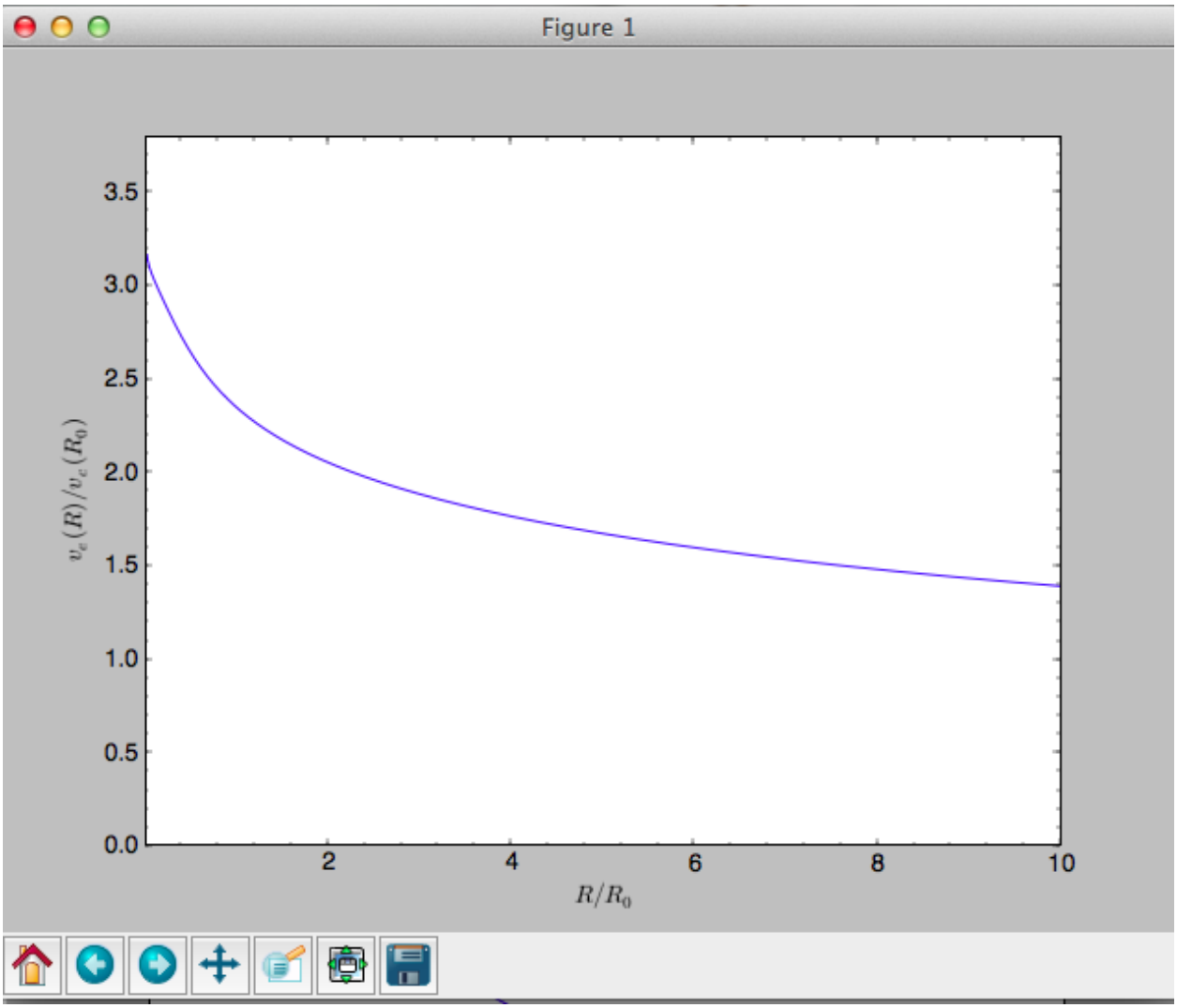
or of the combination of potentials defined above

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



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

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



At the solar radius, the escape velocity is

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

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

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

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

1.4 Potentials in galpy

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

1.4.1 Potentials and forces

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

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

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

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

Most member functions of Potential instances have corresponding functions in the galpy.potential module that allow them to be evaluated for lists of multiple Potential instances. `galpy.potential.MWPotential2014` is such a list of three Potential instances

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

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

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

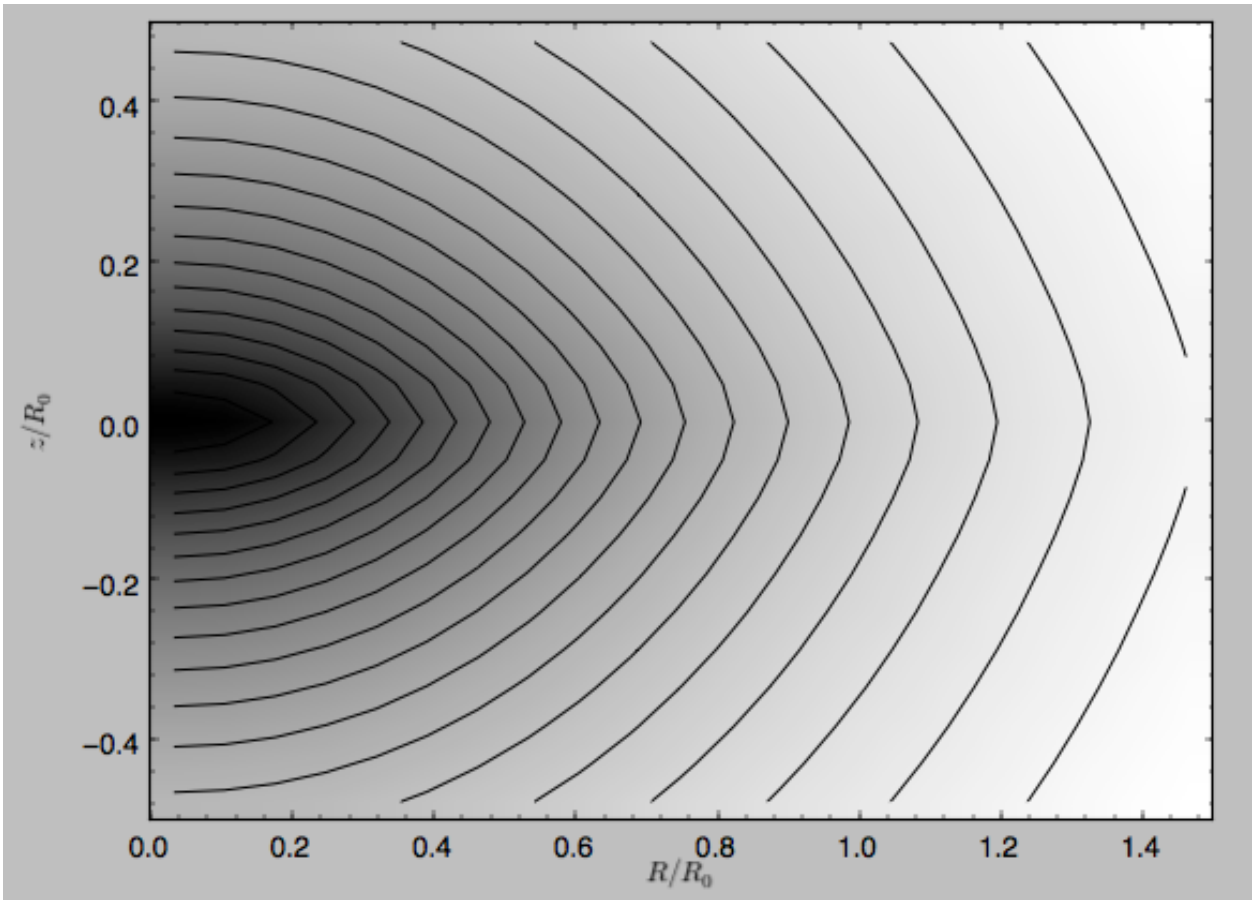
Warning: `galpy` potentials do *not* necessarily approach zero at infinity. To compute, for example, the escape velocity or whether or not an orbit is unbound, you need to take into account the value of the potential at infinity. E.g., $v_{\rm esc}(r) = \sqrt{2[\Phi(\infty) - \Phi(r)]}$.

Tip: As discussed in the section on *physical units*, potentials can be initialized and evaluated with arguments specified as a astropy Quantity with units. Use the configuration parameter `apy-units = True` to get output values as a Quantity. See also the subsection on *Initializing potentials with parameters with units* below.

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

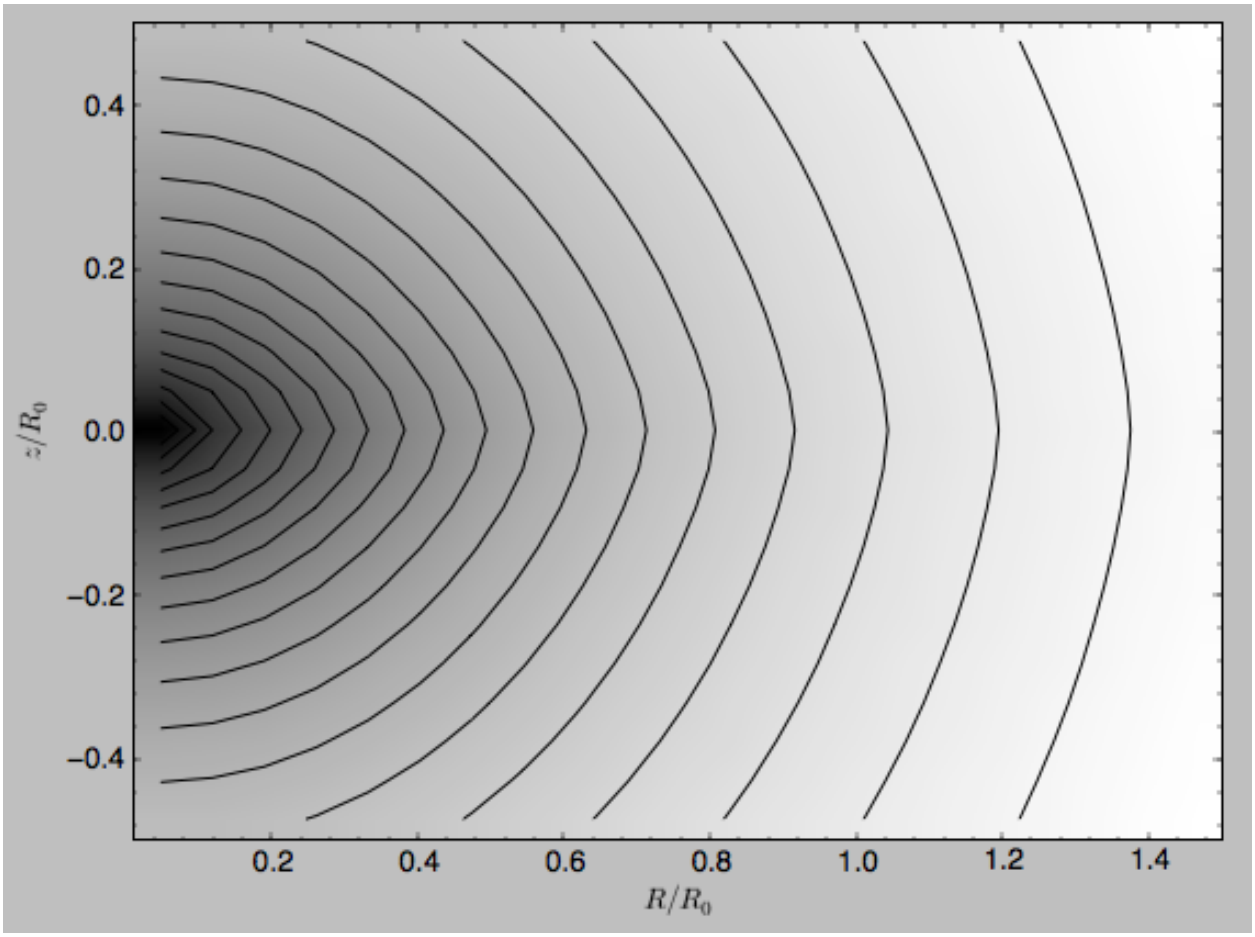
```
>>> mp.plot()
```

which produces the following plot



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

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



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

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

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

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

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

but not further from the mid-plane

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

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

```
>>> 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.)
# -119.72021771473301 #2 \pi G Msol / pc^2
```

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

```
>>> from galpy.potential import evaluateRforces
>>> evaluateRforces(MWPotential2014,1.,0.)
# -1.0
>>> from galpy.potential import evaluatezforces
>>> evaluatezforces(MWPotential2014,1.,0.125)*bovy_conversion.force_in_
↪2piGmsolpc2(220.,8.)
>>> -69.680720137571114 #2 \pi G Msol / pc^2
```

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

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

1.4.2 Densities

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

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

and we can also calculate this using the Poisson equation

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

which are the same to machine precision

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

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

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

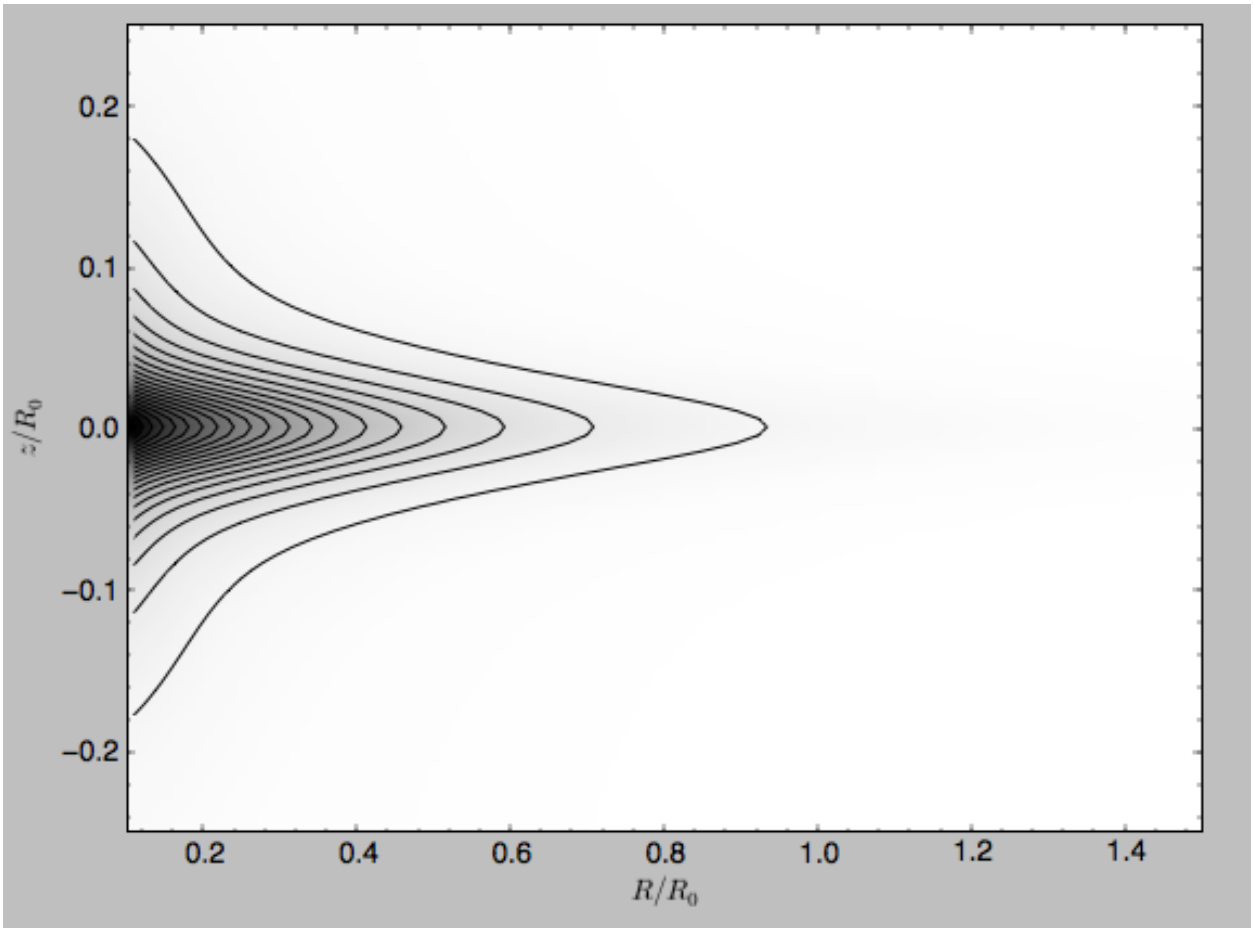
In physical coordinates, this becomes

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

We can also plot densities

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

which gives



Another example of this is for an exponential disk potential

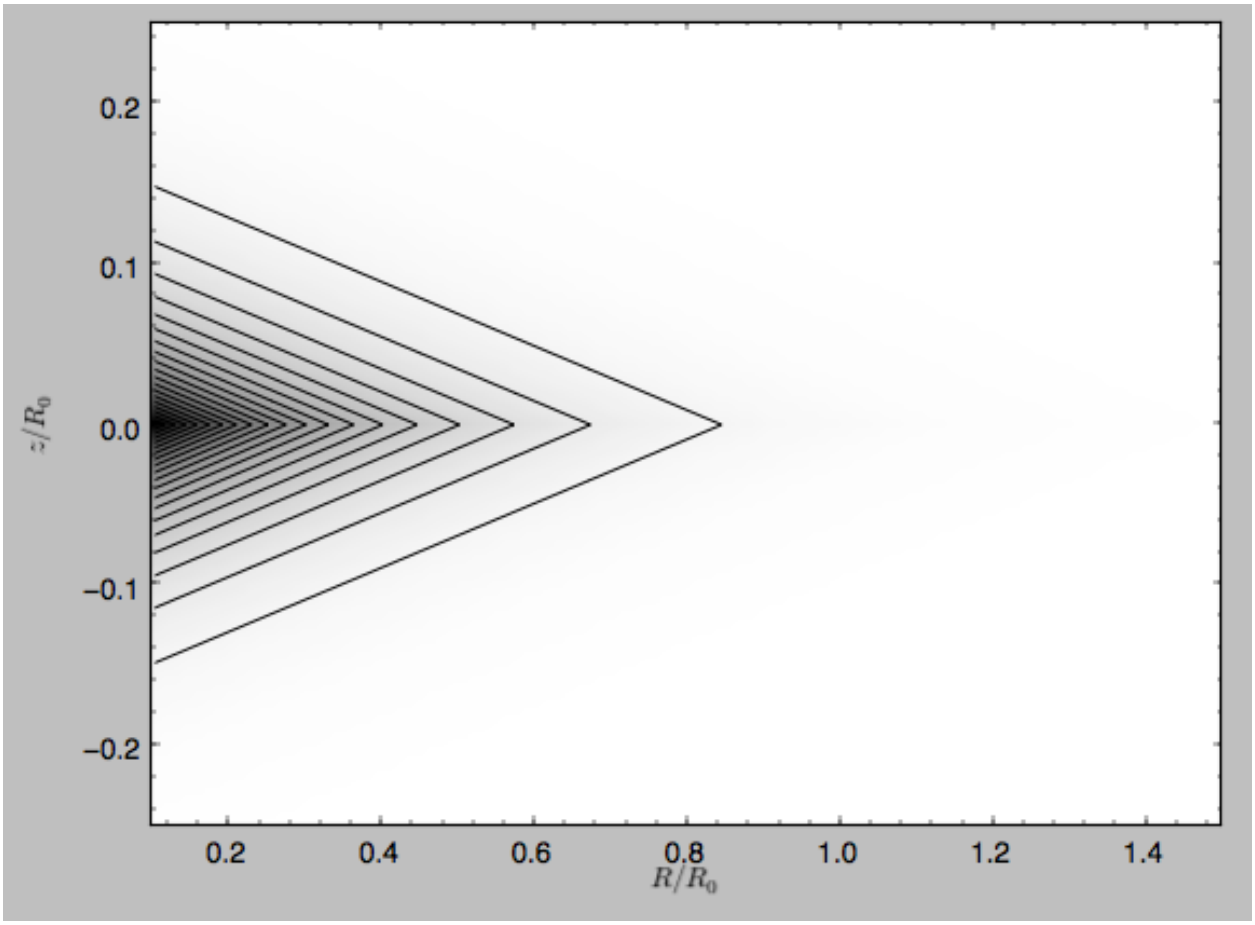
```
>>> from galpy.potential import DoubleExponentialDiskPotential
>>> dp= DoubleExponentialDiskPotential(hr=1./4.,hz=1./20.,normalize=1.)
```

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

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

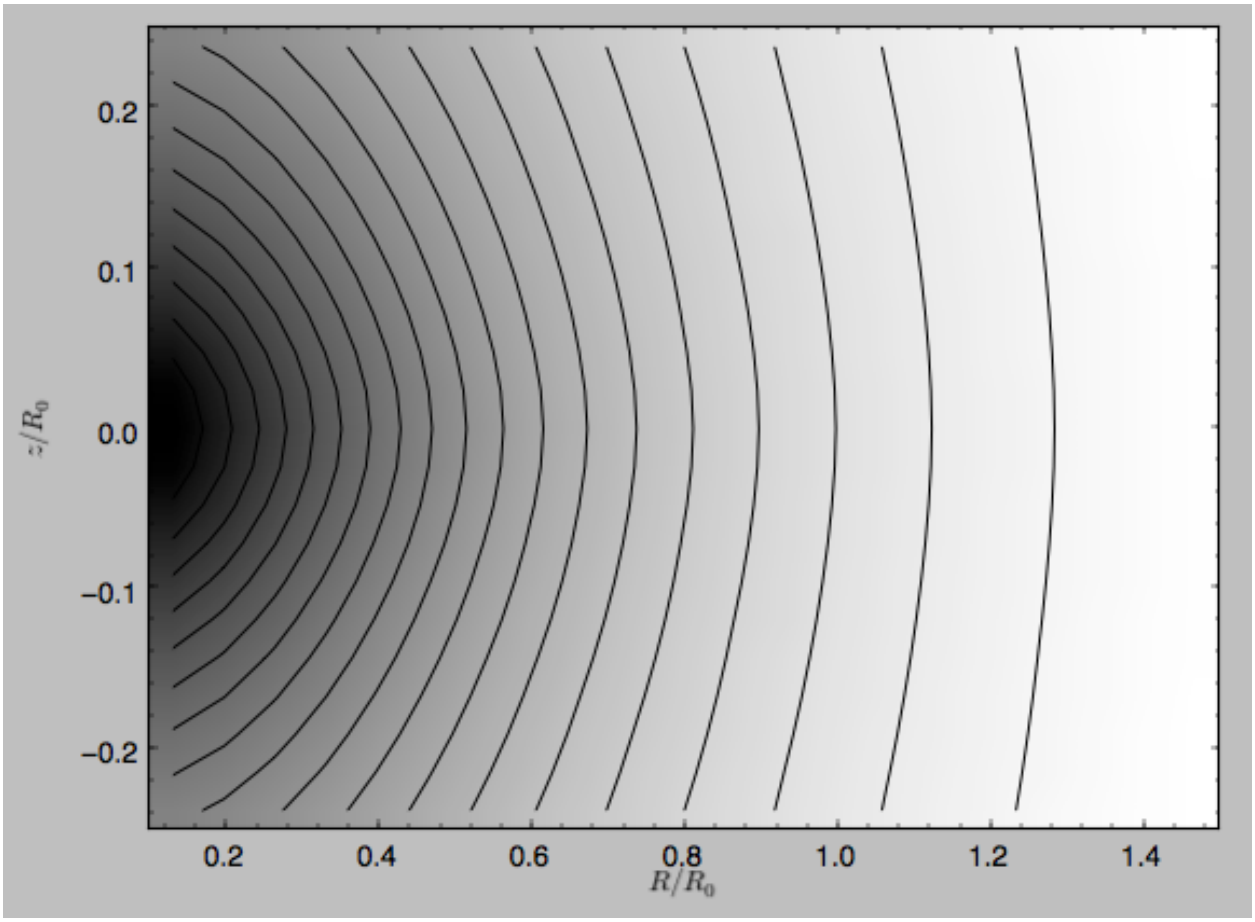
The density is

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



and the potential is

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



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

1.4.3 NEW in v1.3: Modifying potential instances using wrappers

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

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

and then wrap it

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

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

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

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

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

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

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

Compare for example

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

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

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

1.4.4 Close-to-circular orbits and orbital frequencies

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

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

or, for lists of Potential instances

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

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

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

and the epicycle frequency

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

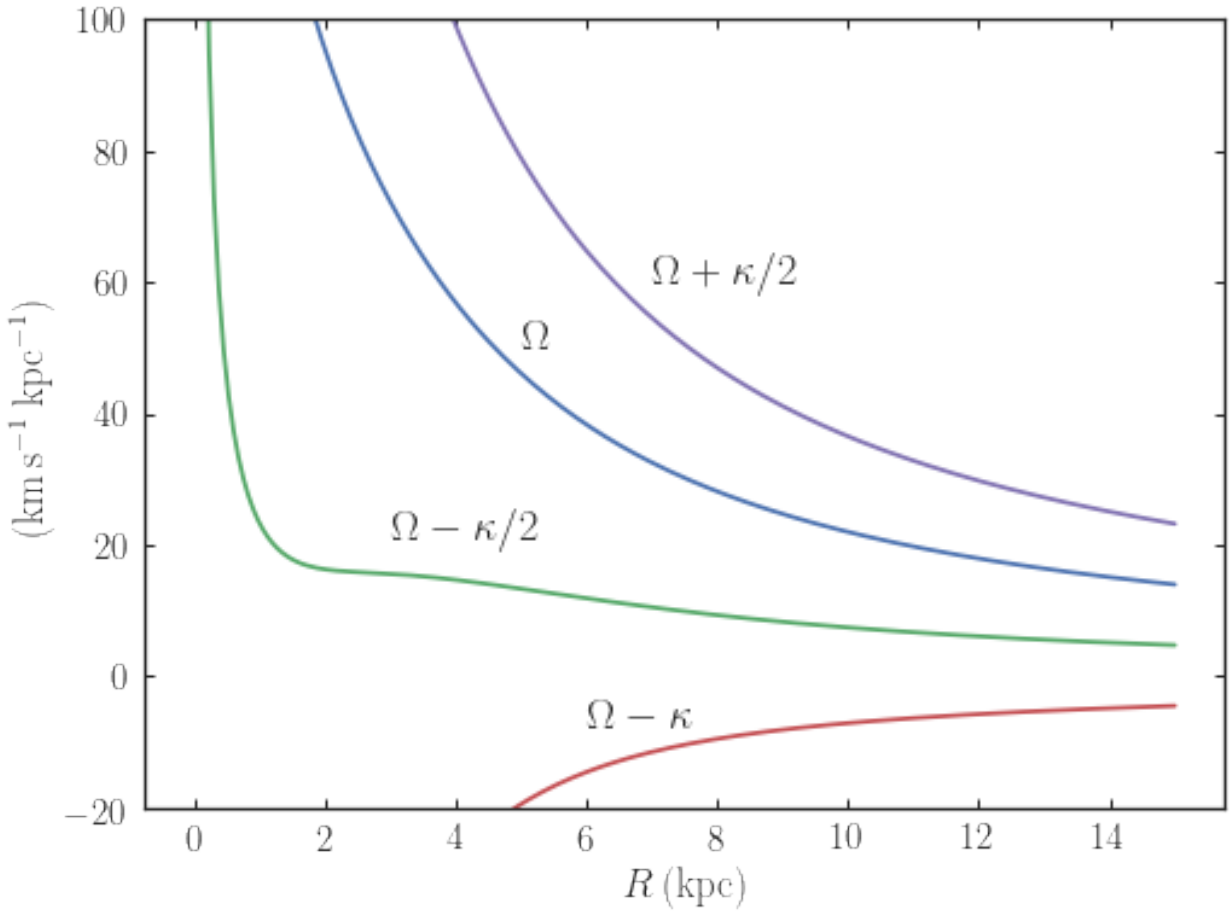
as well as the vertical frequency

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

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

```
>>> def OmegaMinusKappa(pot,Rs,n,m,ro=8.,vo=220.): # ro,vo for physical units
        return omegac(pot,Rs,ro=ro,vo=vo)-n/m*epifreq(pot,Rs,ro=ro,vo=vo)
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,0,1))
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,1,2))
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,1,1))
>>> plot(Rs,OmegaMinusKappa(MWPotential2014,Rs,1,-2))
>>> ylim(-20.,100.)
>>> xlabel(r'$R\,(\mathrm{kpc})$')
>>> ylabel(r'$(\mathrm{km\,s}^{-1}\,\mathrm{kpc}^{-1})$')
>>> text(3.,21.,r'$\Omega-\kappa/2$',size=18.)
>>> text(5.,50.,r'$\Omega$',size=18.)
>>> text(7.,60.,r'$\Omega+\kappa/2$',size=18.)
>>> text(6.,-7.,r'$\Omega-\kappa$',size=18.)
```

which gives



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

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

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

1.4.5 Using interpolations of potentials

`galpy` contains a general `Potential` class `interpRZPotential` that can be used to generate interpolations of potentials that can be used in their stead to speed up calculations when the calculation of the original potential is computationally expensive (for example, for the `DoubleExponentialDiskPotential`). Full details on how to set this up are given *here*. Interpolated potentials can be used anywhere that general three-dimensional galpy potentials can be used. Some care must be taken with outside-the-interpolation-grid evaluations for functions that use `C` to speed up computations.

1.4.6 Initializing potentials with parameters with units

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

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

1.4.7 UPDATED in v1.3: General density/potential pairs with basis-function expansions

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

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

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

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

As a more complicated example, consider a prolate NFW potential

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

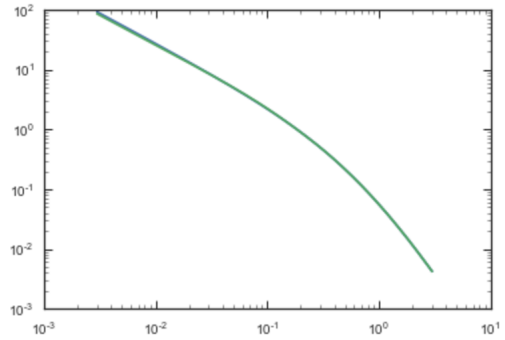
and we compute the coefficients using the axisymmetric `scf_compute_coeffs_axi`

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

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

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

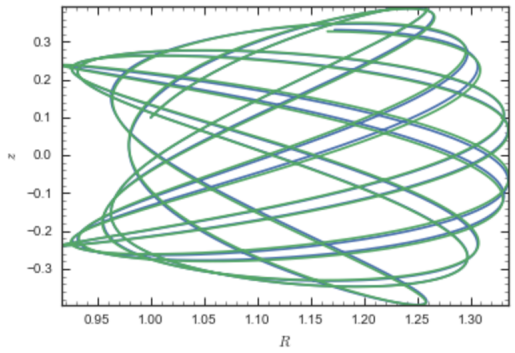
we get



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

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

which gives



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

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

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

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

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

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

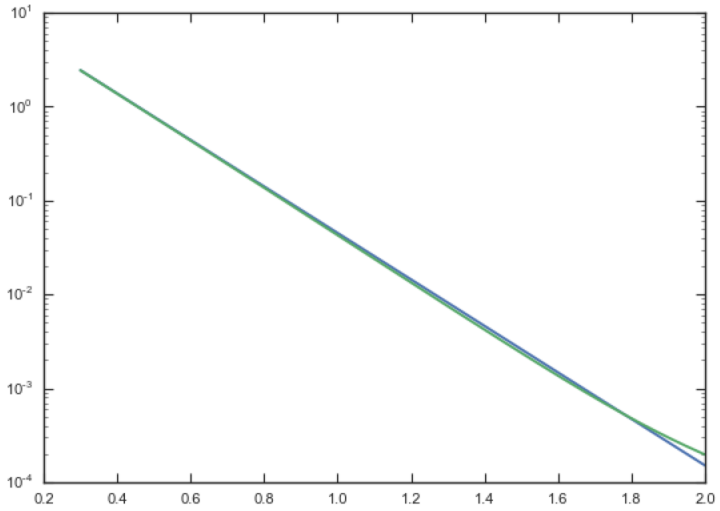
and then setup the `DiskSCFPotential` approximation to this as

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

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

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

which gives



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

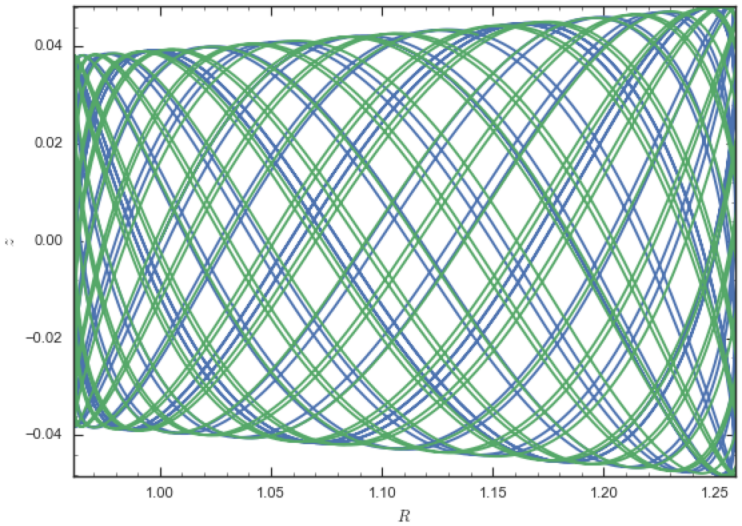
```
>>> from galpy.orbit import Orbit
>>> o= Orbit([1.,0.1,0.9,0.,0.1,0.])
>>> ts= numpy.linspace(0.,100.,10001)
```

(continues on next page)

(continued from previous page)

```
>>> o.integrate(ts,dp)
>>> o.plot()
>>> o.integrate(ts,dscfp)
>>> o.plot(overplot=True)
```

which gives



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

```
>>> timeit(o.integrate(ts,dp))
# 1 loops, best of 3: 5.83 s per loop
>>> timeit(o.integrate(ts,dscfp))
# 1 loops, best of 3: 286 ms per loop
```

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

1.4.8 The potential of N-body simulations

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

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

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

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

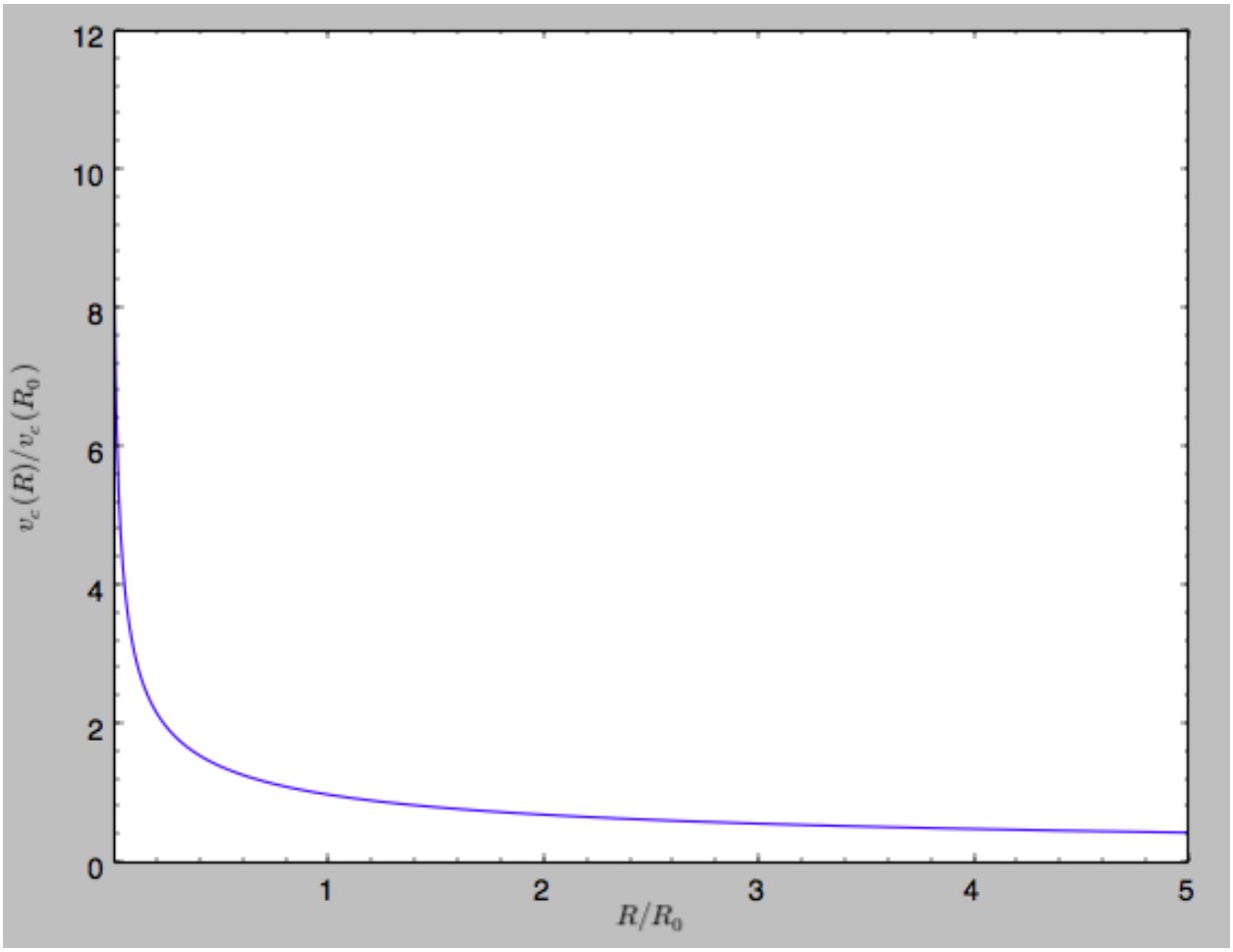
```
>>> from galpy.potential import SnapshotRZPotential
>>> sp= SnapshotRZPotential(s,num_threads=1)
```

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

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

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

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



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

download this snapshot.

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

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

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

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

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

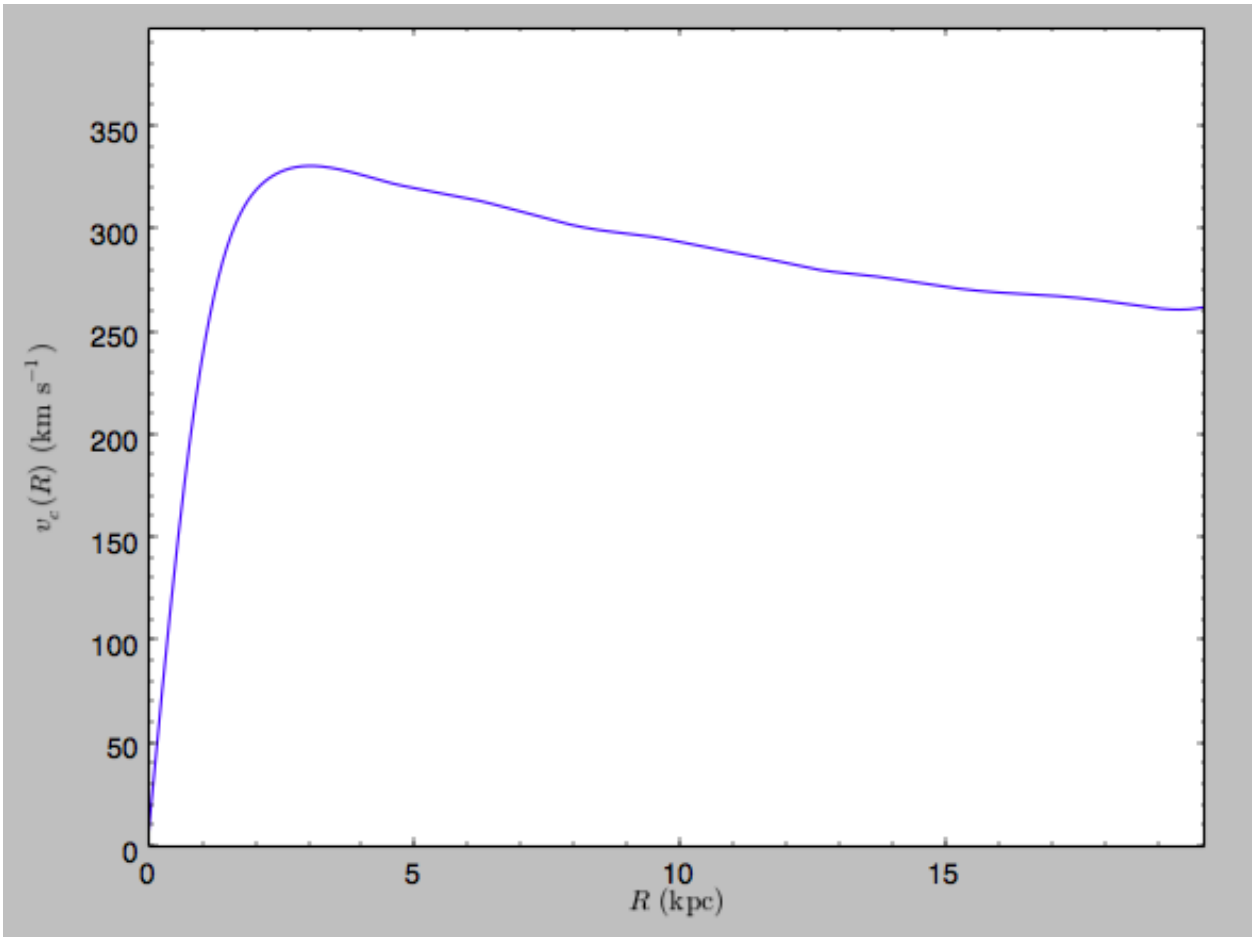
```
>>> s.physical_units()
>>> from galpy.util.bovy_conversion import _G
>>> g= pynbody.array.SimArray(_G/1000.)
>>> g.units= 'kpc Msol**-1 km**2 s**-2 G**-1'
>>> s._arrays['mass']= s._arrays['mass']*g
```

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

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

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

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



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

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

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

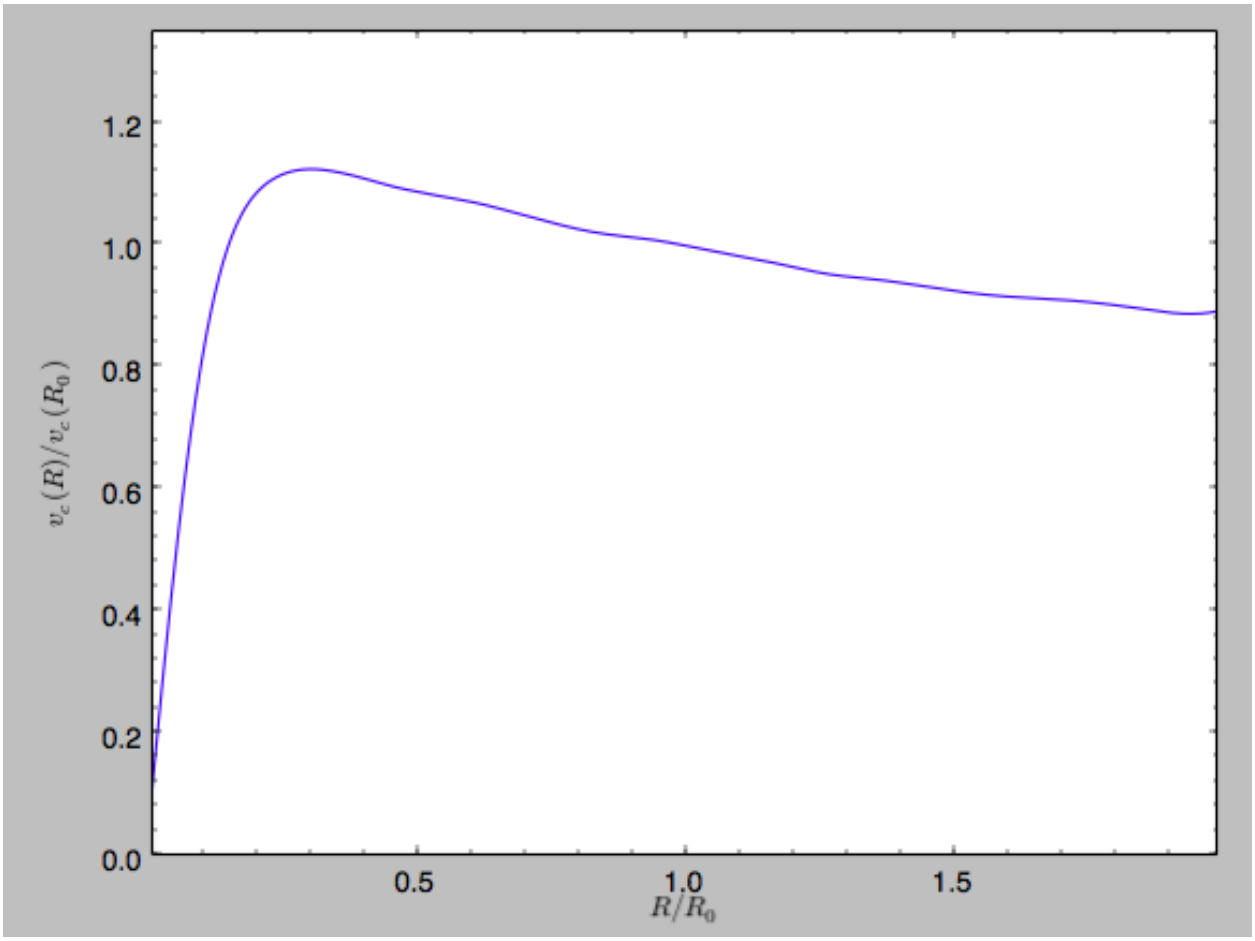
To convert to *natural units* we do

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

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

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

which gives

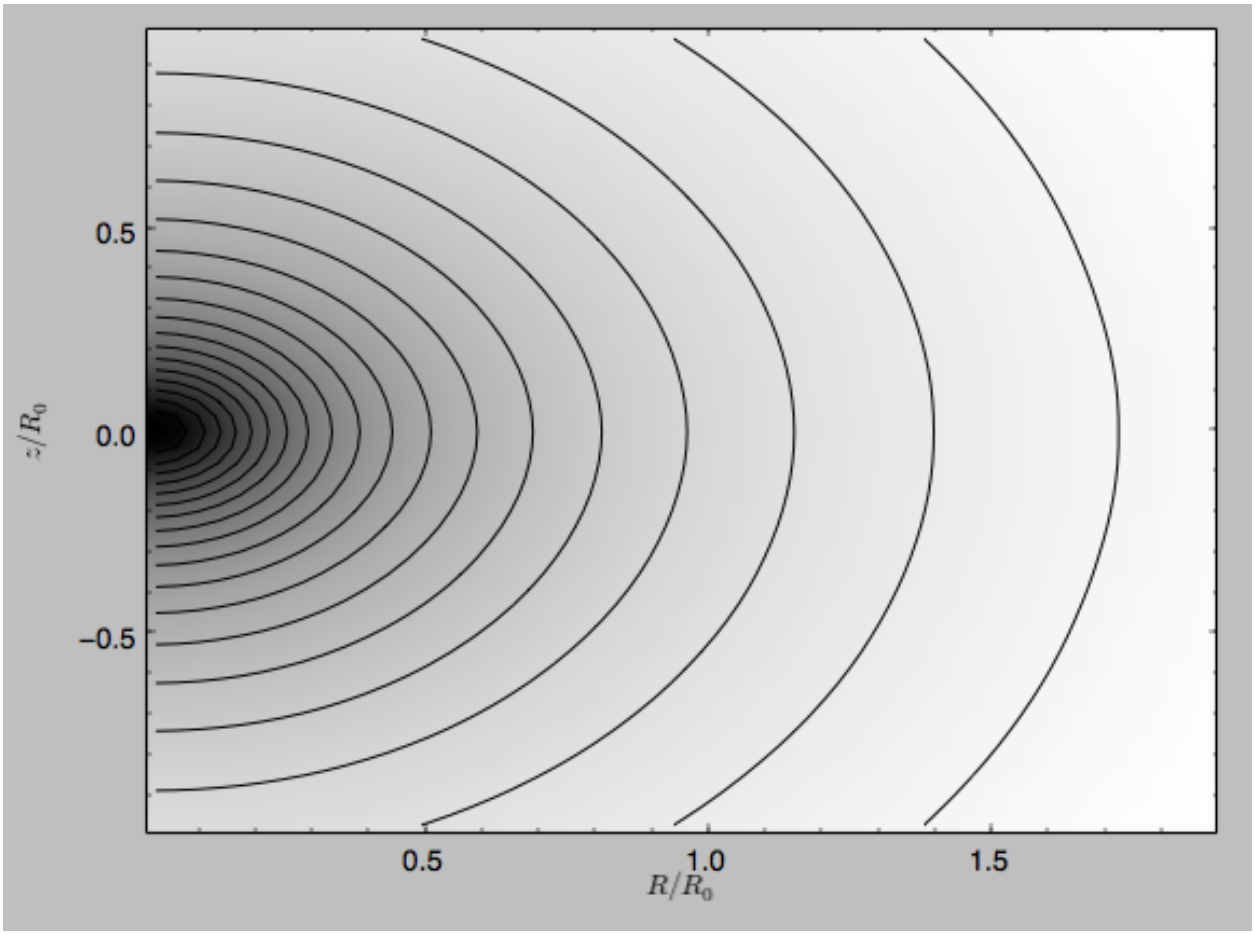


in particular

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

We can also plot the potential

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



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

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

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

1.4.9 Conversion to NEMO potentials

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

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

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

and its parameters as

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

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

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

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

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

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

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

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

1.4.10 Adding potentials to the galpy framework

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

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

tem of units; the amp_units= keyword specifies the physical units of the amplitude parameter (e.g., amp_units='velocity2' when the units of the amplitude are velocity-squared) To add support for normalizing the potential to standard galpy units, one can call the galpy.potential.Potential.normalize function at the end of the __init__ function.

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

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

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

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

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.

The __init__ method should be written in such a way that a relevant object can be initialized using Classname() (i.e., there have to be reasonable defaults given for all parameters, including the amplitude); doing this allows the nose tests for potentials to automatically check that your Potential's potential

function, force functions, second derivatives, and density (through the Poisson equation) are correctly implemented (if they are implemented). The continuous-integration platform that builds the galpy codebase upon code pushes will then automatically test all of this, streamlining push requests of new potentials.

A few atrributes need to be set depending on the potential: `hasC=True` for potentials for which the forces and potential are implemented in C (see below); `self.hasC_dxdv=True` for potentials for which the (planar) second derivatives are implemented in C; `self.isNonAxi=True` for non-axisymmetric potentials.

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). If you have implemented the necessary second derivatives for integrating phase-space volumes, also add `self.hasC_dxdv=True`.

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

1.4.11 NEW in v1.3: Adding wrapper potentials to the galpy framework

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

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

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

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

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

where `smooth(t)` returns the smoothing function of the amplitude. When any of the basic `Potential` functions are called (`_evaluate`, `_Rforce`, etc.), `_wrap` gets called by the superclass `WrapperPotential`, and the `_wrap` function returns the corresponding function for the wrapped potentials with the amplitude modified by `smooth(t)`. Therefore, one does not need to implement each of the `_evaluate`, `_Rforce`, etc. functions like for regular potential. The rest of the `DehnenSmoothWrapperPotential` is essentially (slightly simplified in non-crucial aspects)

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

def _smooth(self,t):
    #Calculate relevant time
```

(continues on next page)

(continued from previous page)

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

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

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

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

1.5 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.

1.5.1 Types of disk distribution functions

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

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

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

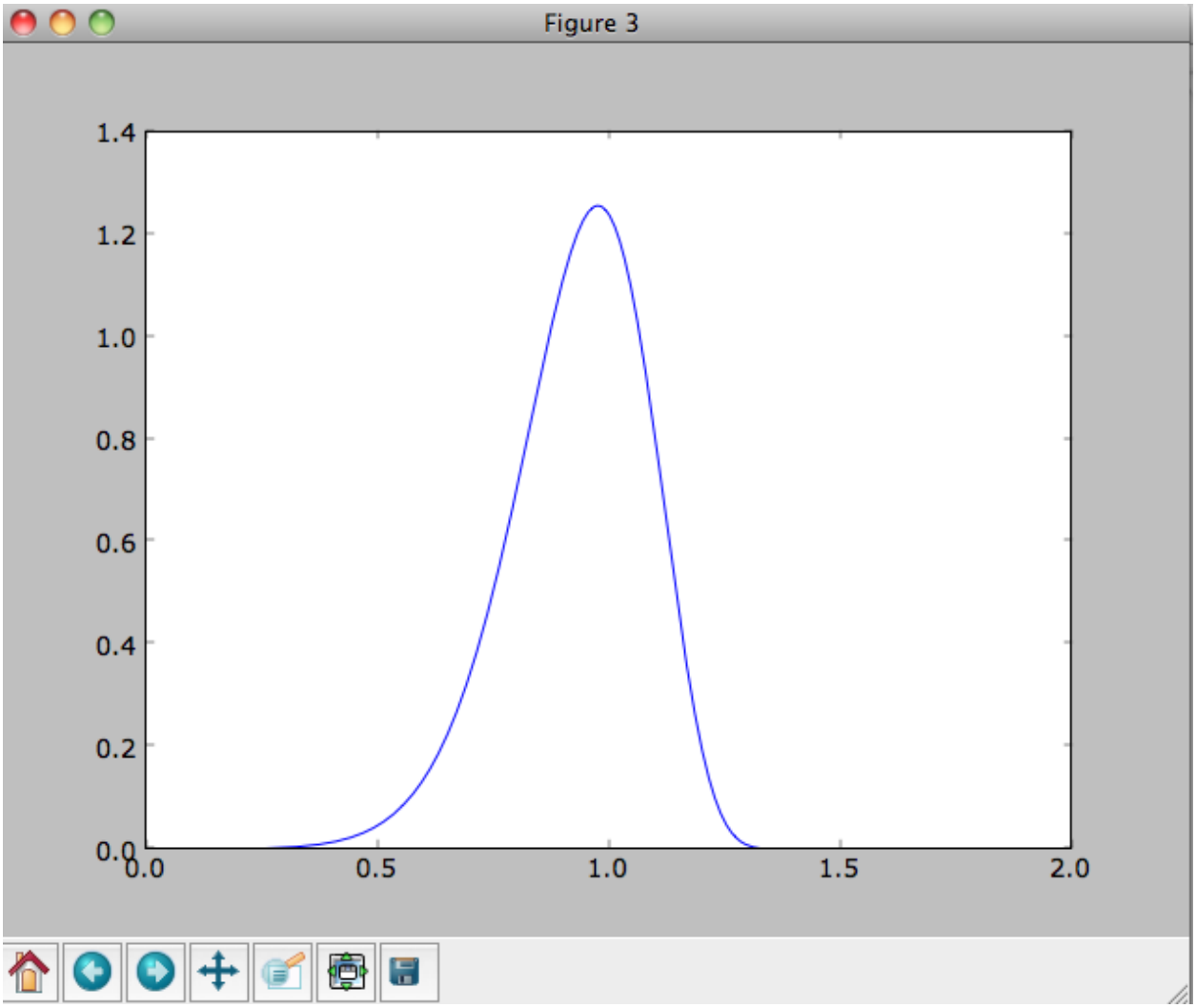
This initializes a `dehnendf` instance based on an exponential surface-mass profile with scale-length 1/3 and an exponential radial-velocity-dispersion profile with scale-length 1 and a value of 0.2 at R=1. Different parameters for these profiles can be provided as an initialization keyword. For example,

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

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

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

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



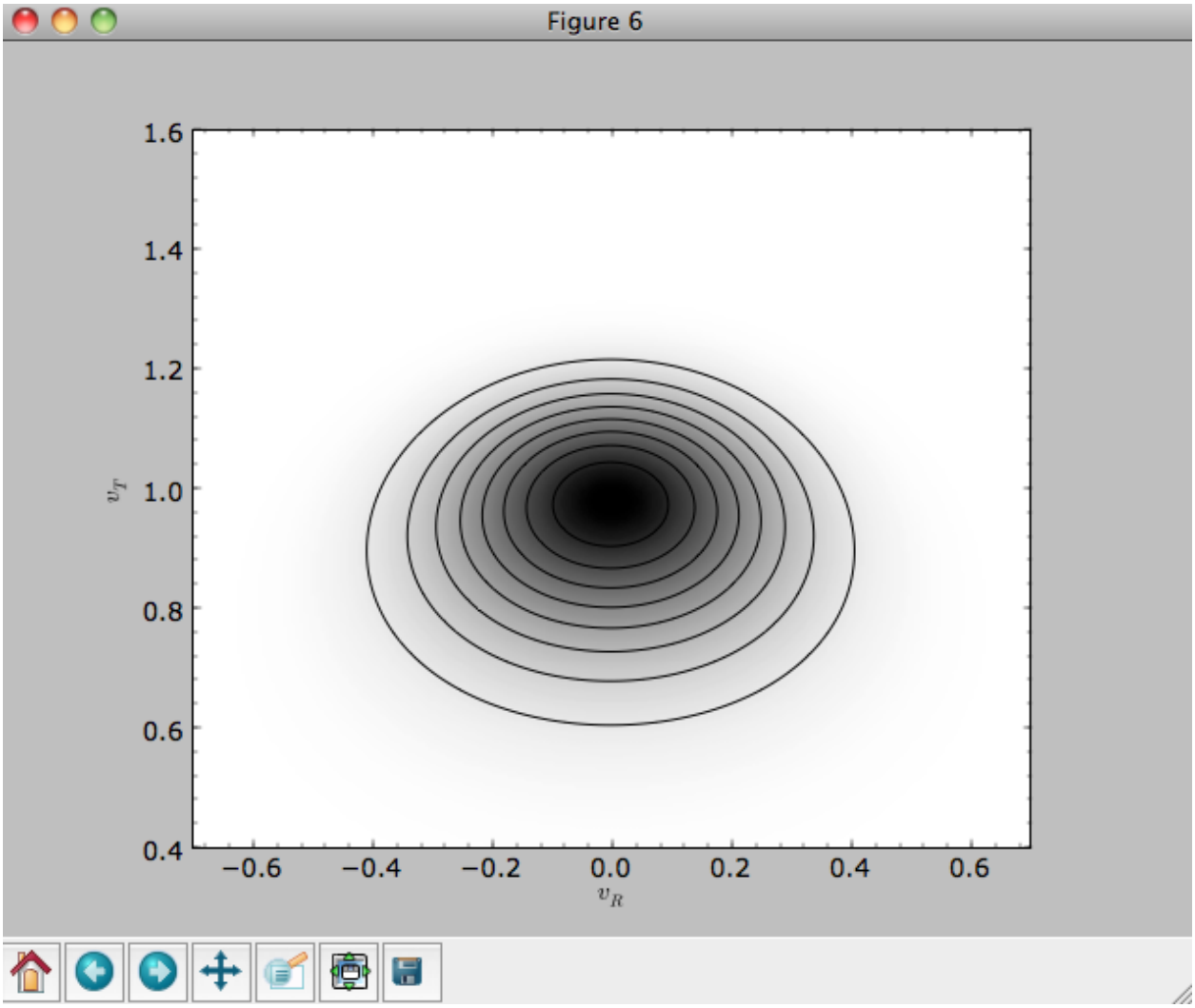
Or we can plot the two-dimensional density at R=1.

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

(continues on next page)

(continued from previous page)

```
>>> from galpy.util.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],xlabel=r'$v_R$',ylabel=r'$v_T$')
```



1.5.2 Evaluating moments of the DF

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

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

and the velocity dispersions

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

(continues on next page)

(continued from previous page)

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

and their ratio

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

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

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

We can also calculate higher order moments

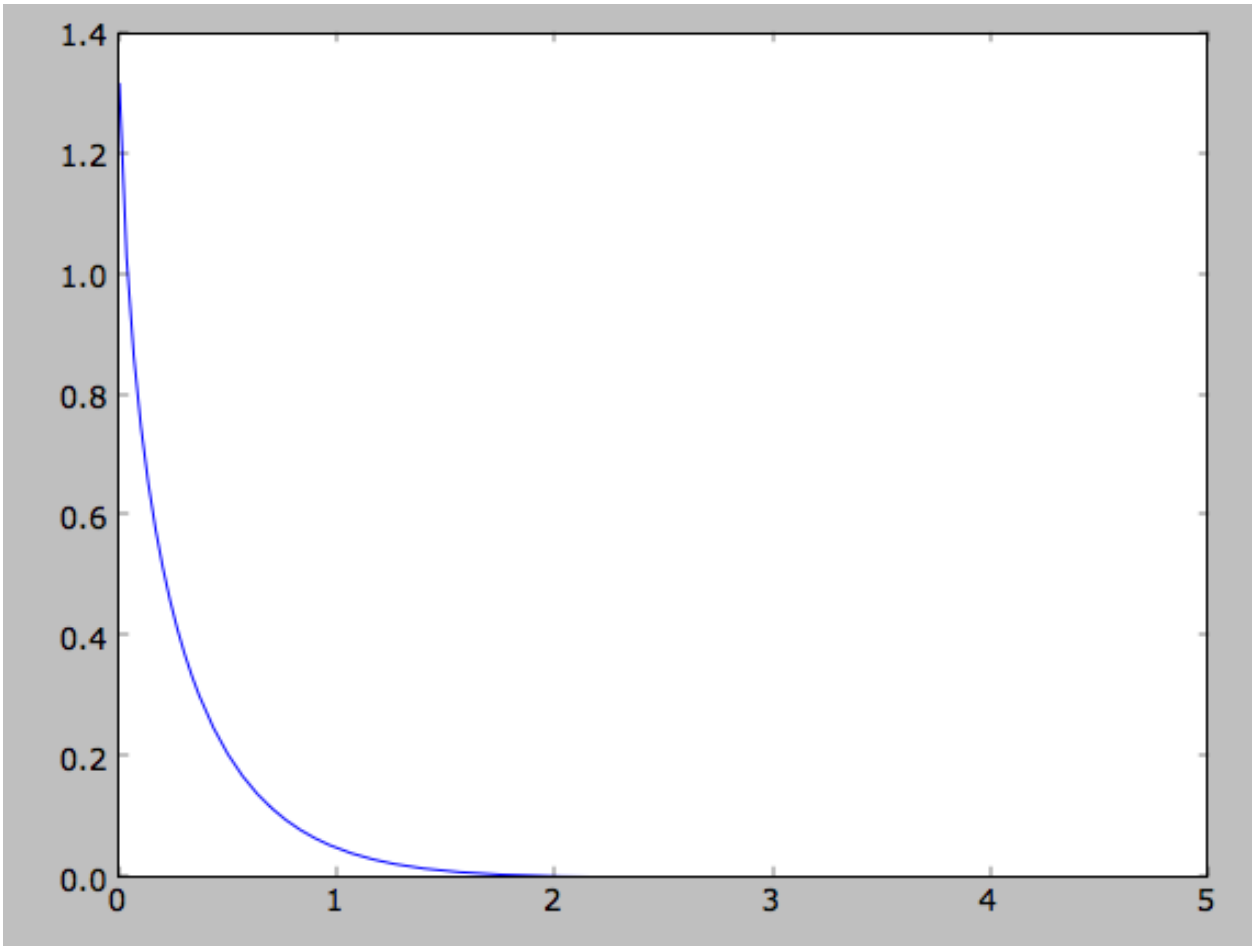
```
>>> dfc.skewvT(1.)
# -0.48617143862047763
>>> dfc.kurtosisvT(1.)
# 0.13338978593181494
>>> dfc.kurtosisvR(1.)
# -0.12159407676394096
```

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

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

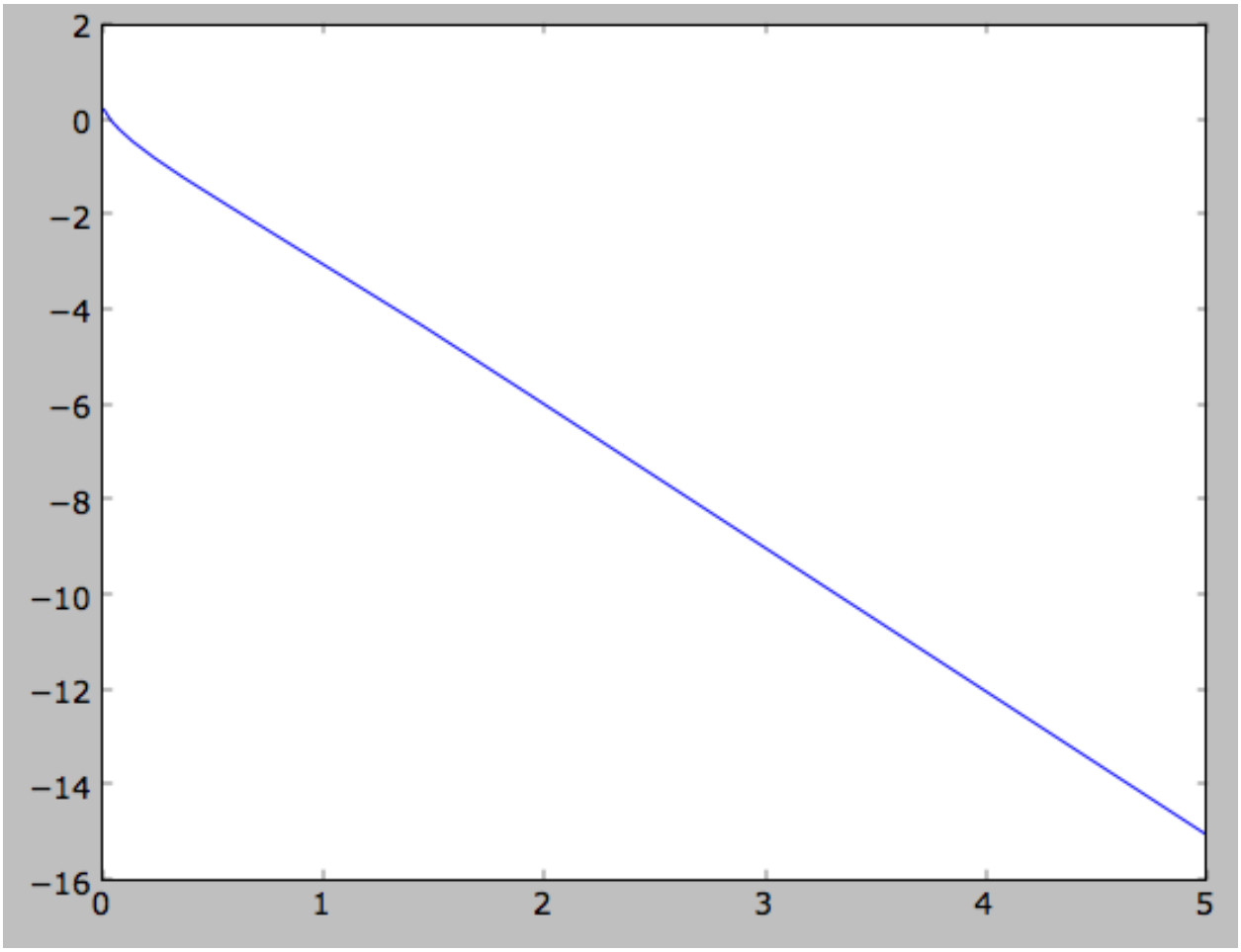
We can plot the surface-mass density as follows

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



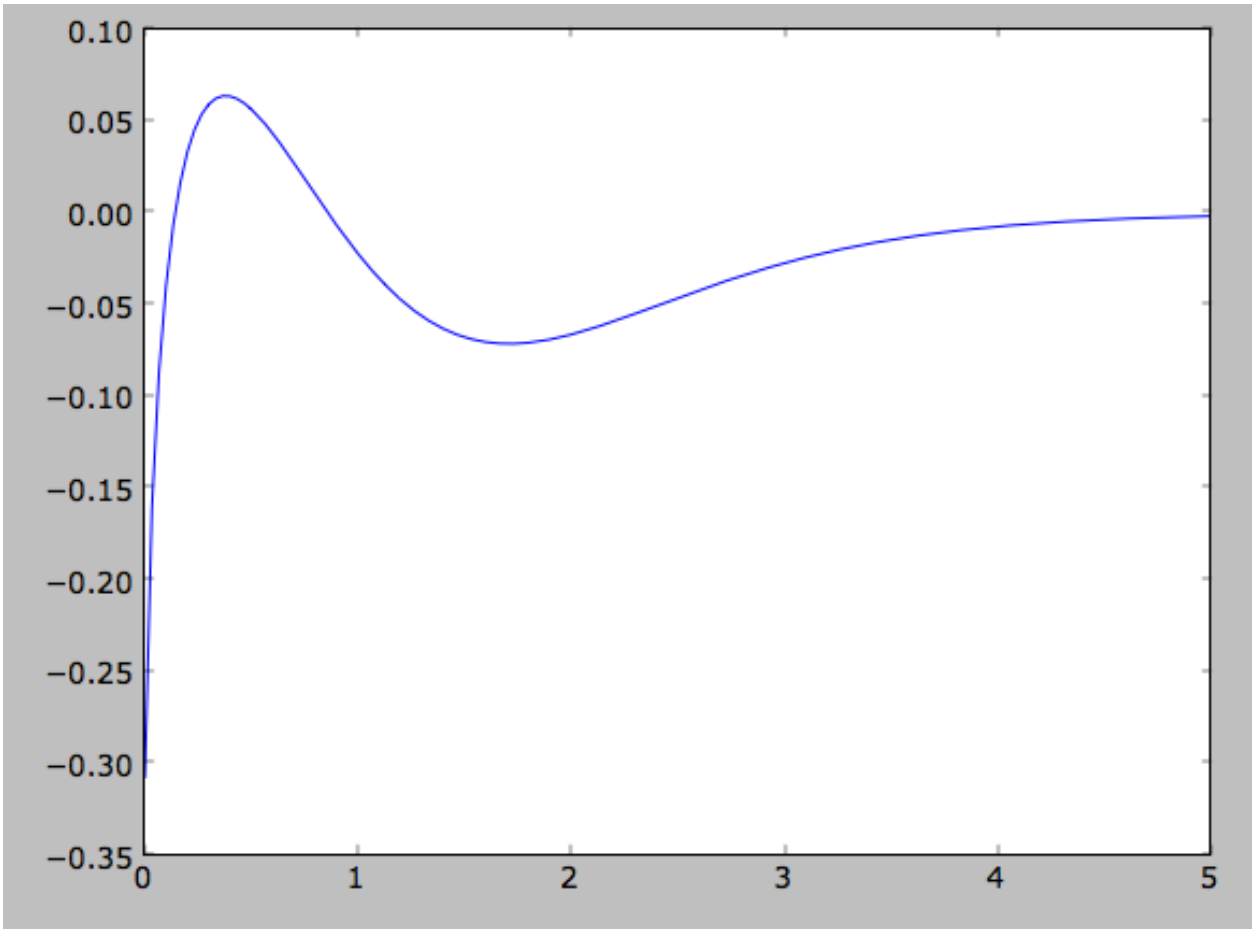
or

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



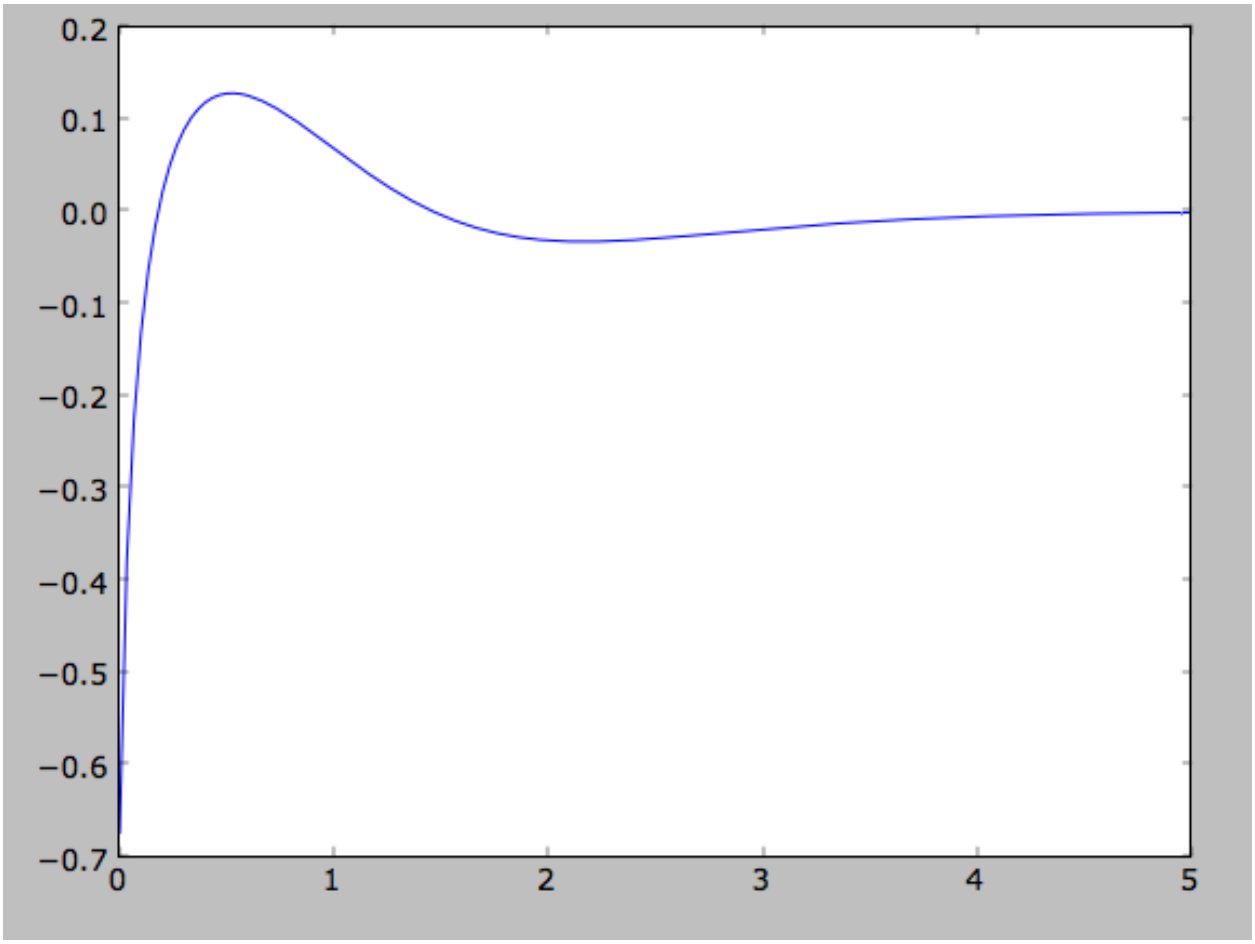
which shows the exponential behavior expected for an exponential disk. We can compare this to the input surface-mass density

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



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

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



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

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

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

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

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

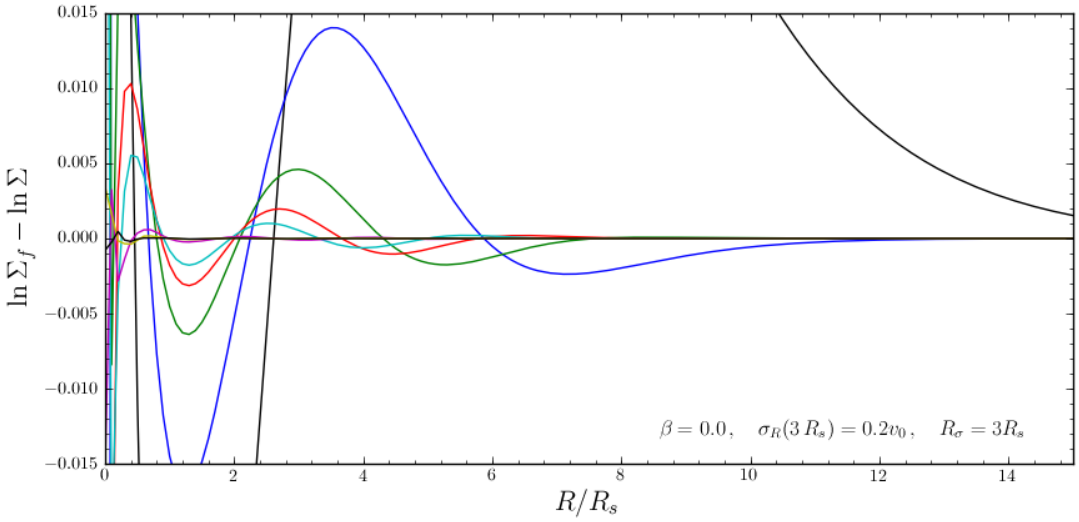
1.5.3 Using corrected disk distribution functions

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

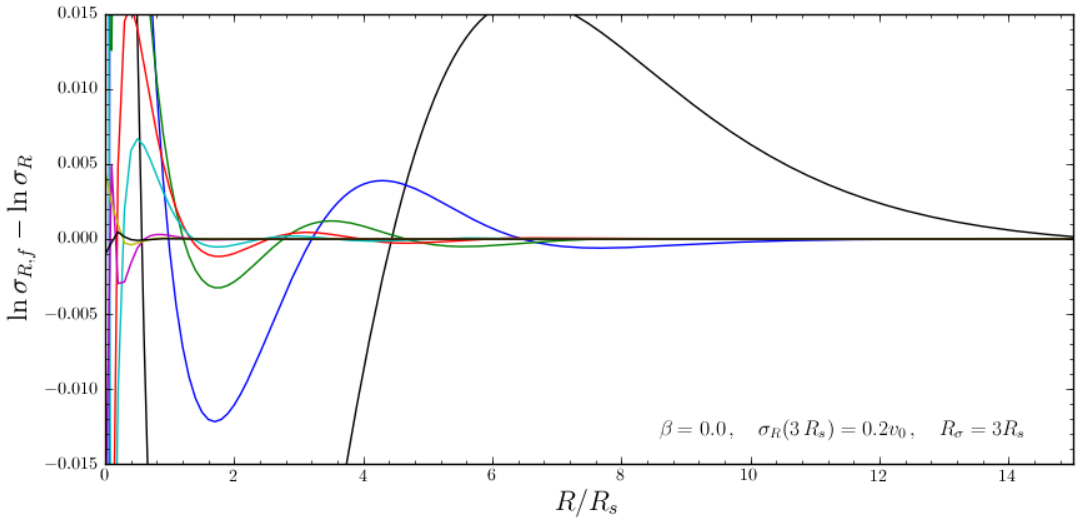
set of *corrections* to the input profiles such that the output profiles more closely resemble the desired profiles (see 1999AJ....118.1201D). galpy supports the calculation of these corrections, and comes with some pre-calculated corrections (these can be found here). For example, the following initializes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

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

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



and the same for the velocity-dispersion profile



galpy will automatically save any new corrections that you calculate.

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

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

and the mean rotation velocity is

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

and (correct) asymmetric drift

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

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

1.5.4 Oort constants and functions

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

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

The K and C Oort constants are zero for axisymmetric DFs

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

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

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

For the cold DF from above the approximation is much better

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

1.5.5 Sampling data from the DF

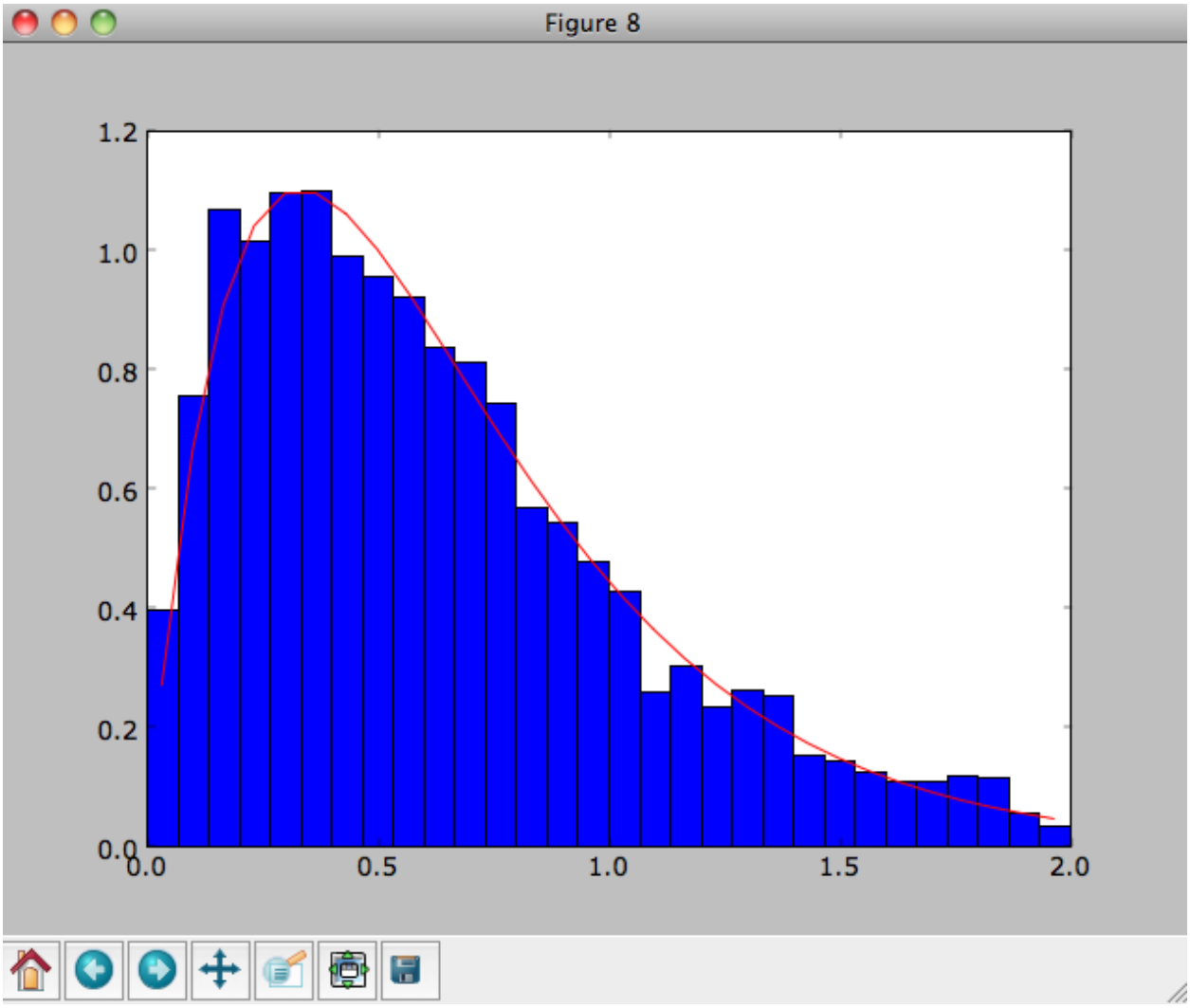
We can sample from the disk distribution functions using `sample`. `sample` can return either an energy–angular-momentum pair, or a full orbit initialization. We can sample 4000 orbits for example as (could take two minutes)

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

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

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

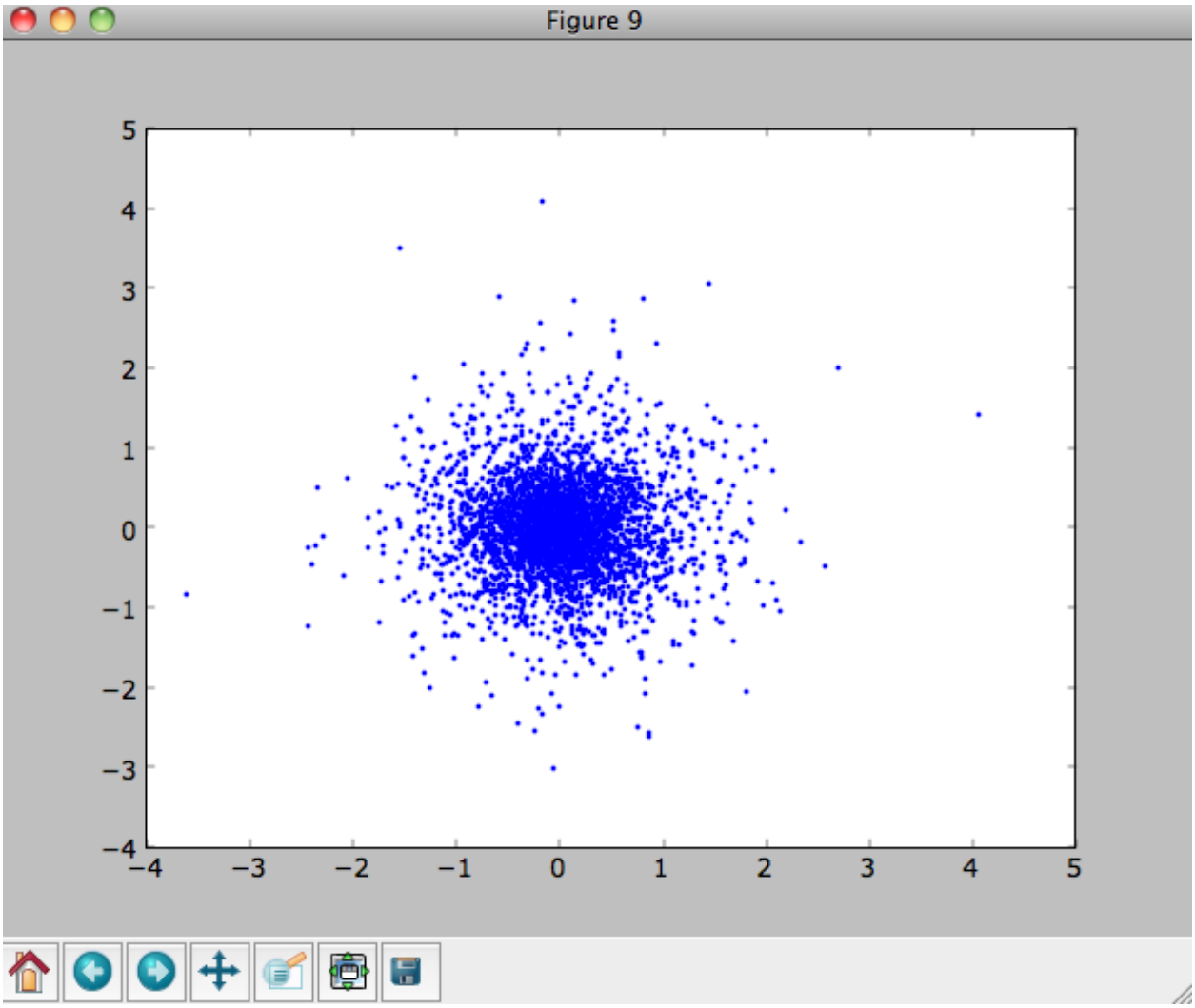
E.g.,



We can also plot the spatial distribution of the sampled disk

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

E.g.,

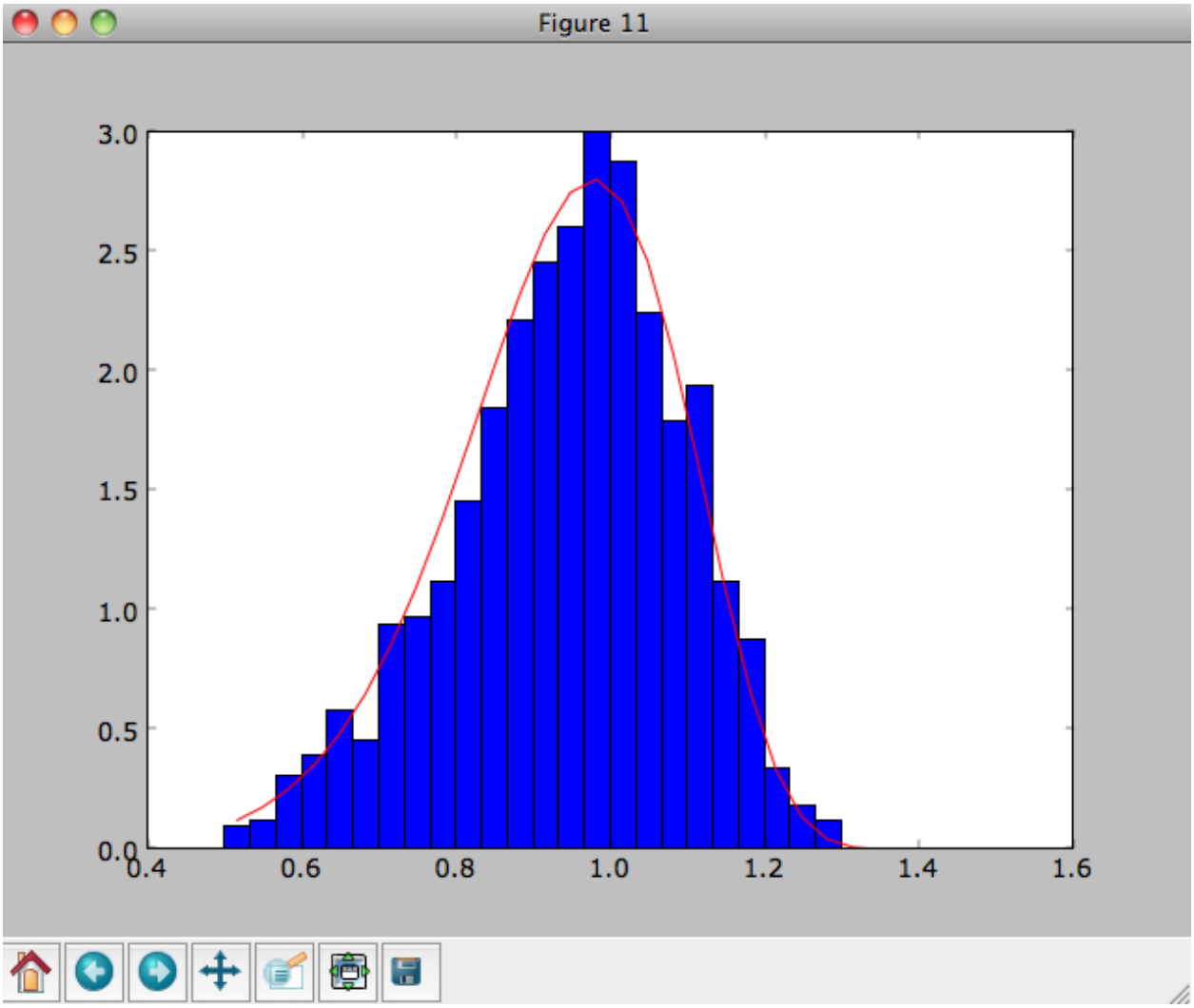


We can also sample points in a specific radial range (might take a few minutes)

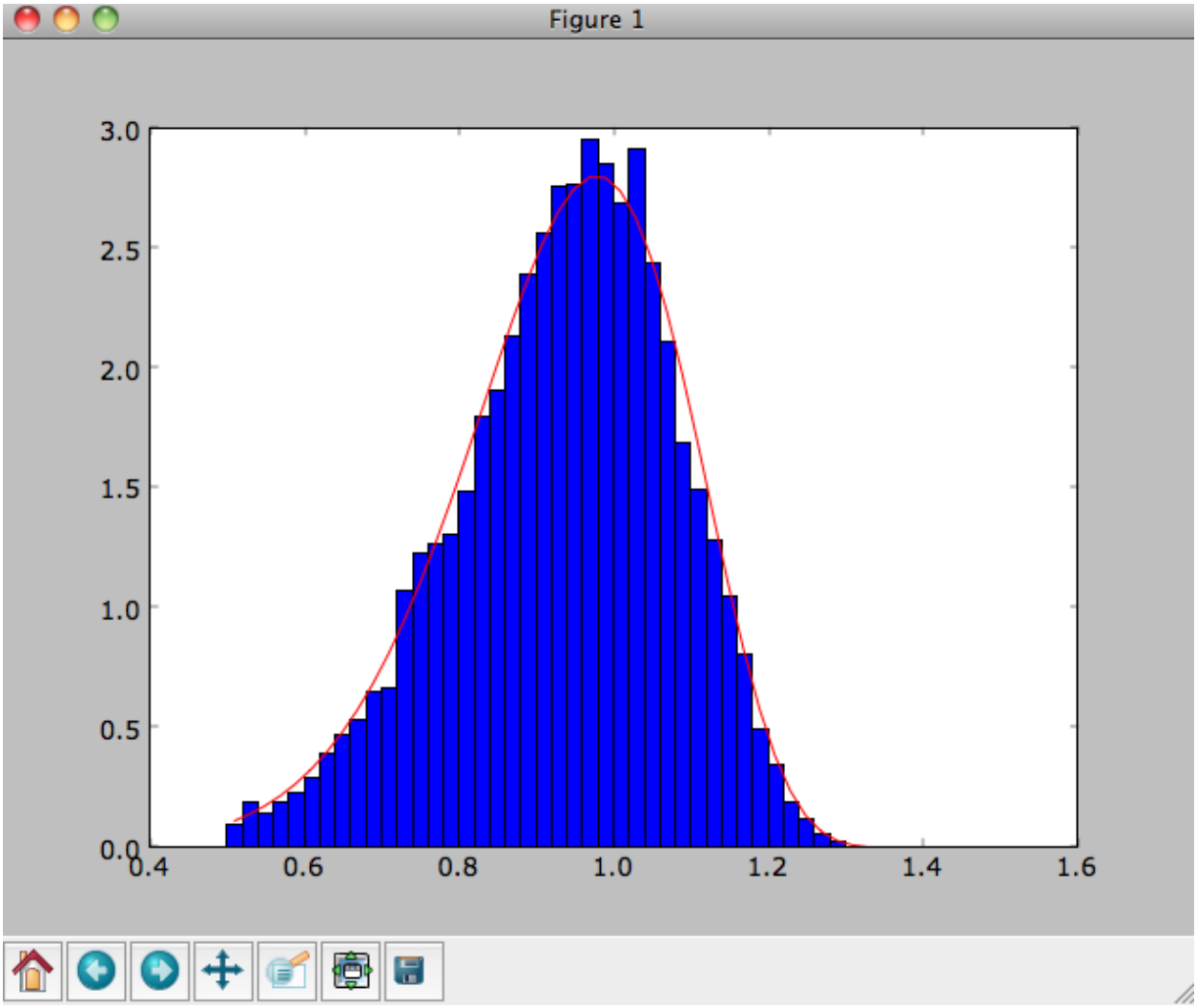
```
>>> o= dfc.sample(n=1000,returnOrbit=True,nphi=1,rrange=[0.8,1.2])
```

and we can plot the distribution of tangential velocities

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

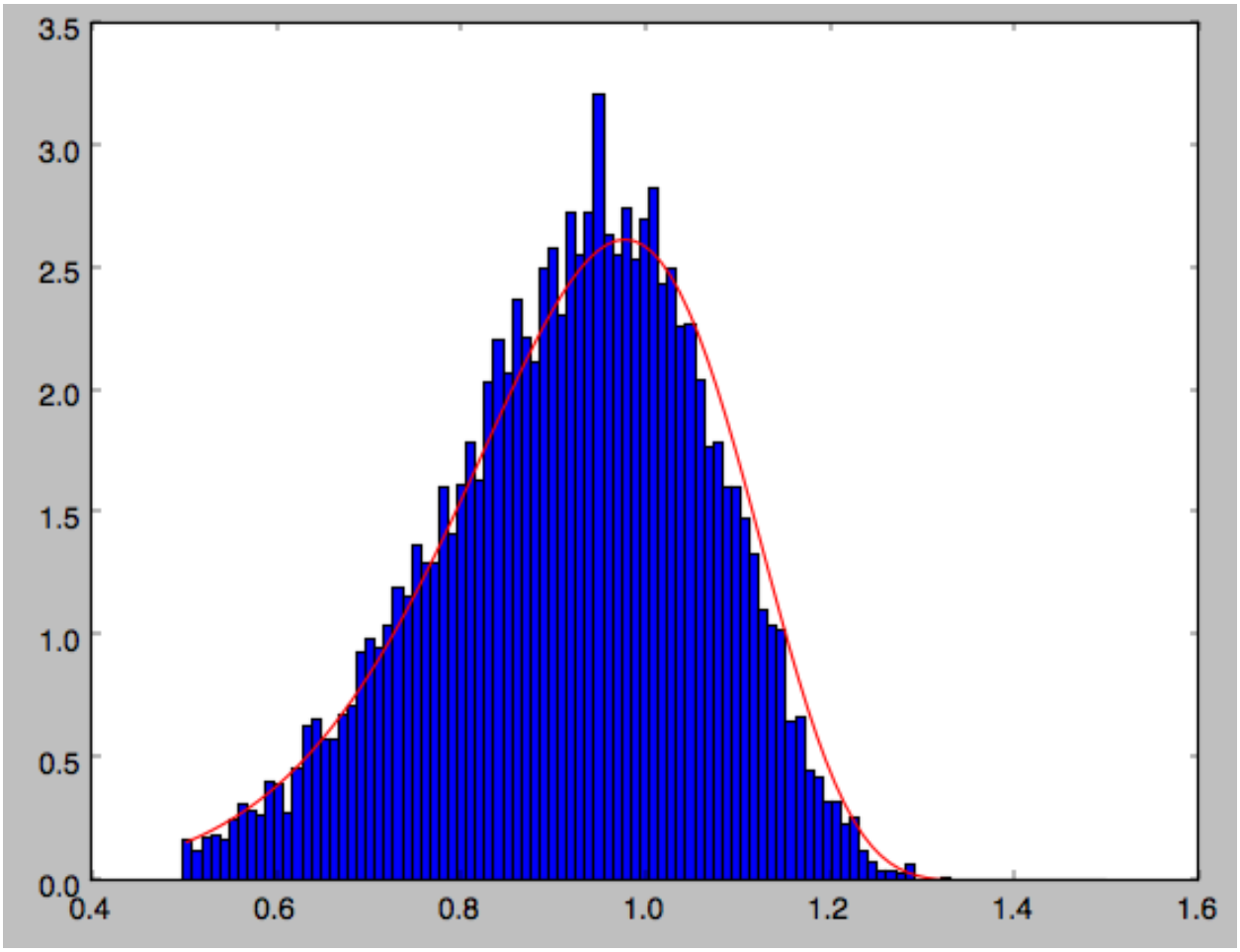


The agreement between the sampled distribution and the theoretical curve is not as good because the sampled distribution has a finite radial range. If we sample 10,000 points in `rrange=[0.95,1.05]` the agreement is better (this takes a long time):



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

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



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

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

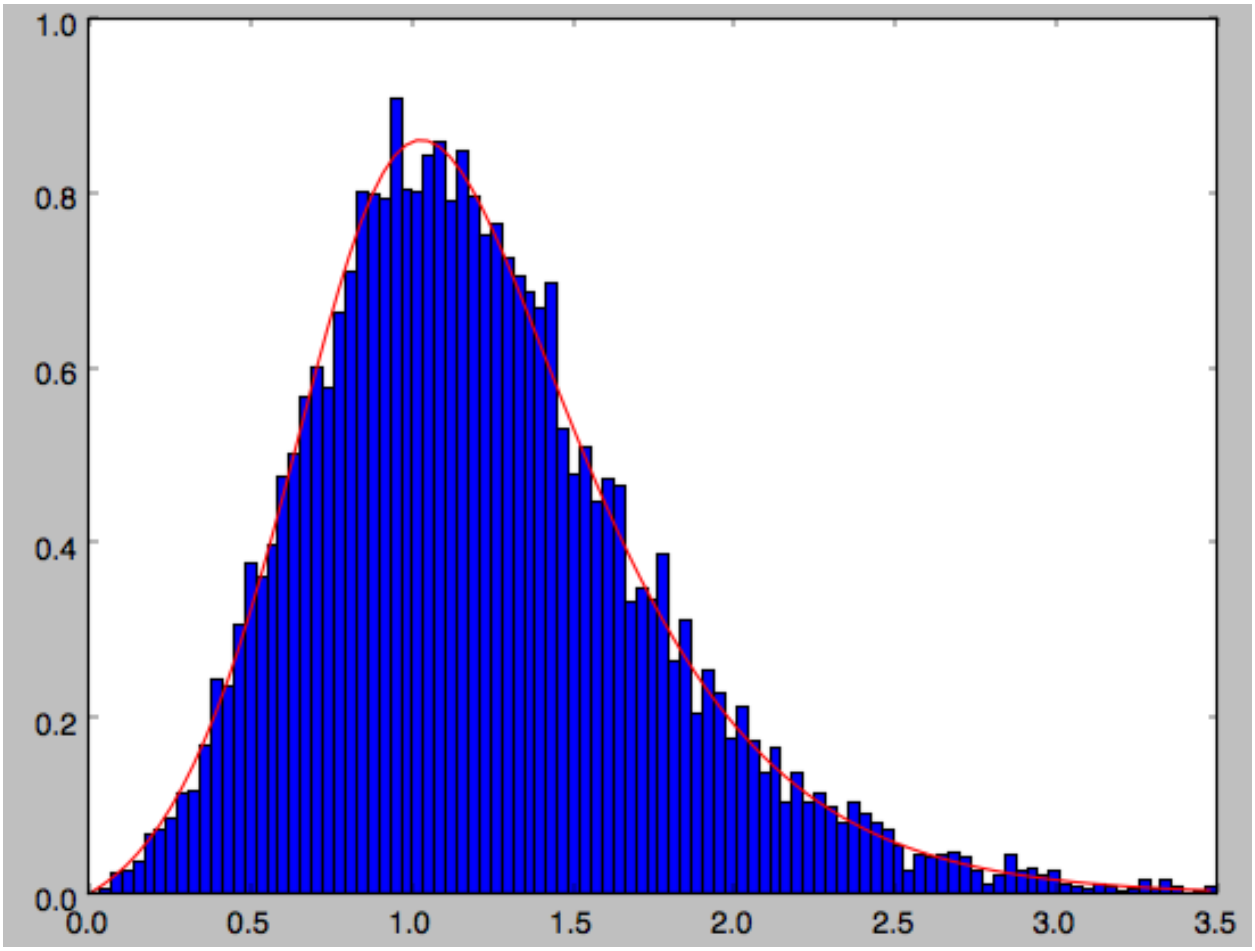
which is very fast. We can histogram these

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

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

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

which shows very good agreement with the sampled distances



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

1.5.6 Non-axisymmetric, time-dependent disk distribution functions

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

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

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

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

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

and then set up the `evolveddiskdf`

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

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

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

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

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

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

The warm response is slightly smaller in amplitude

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

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

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

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

which returns the mean rotational velocity, and

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

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

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

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

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

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

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

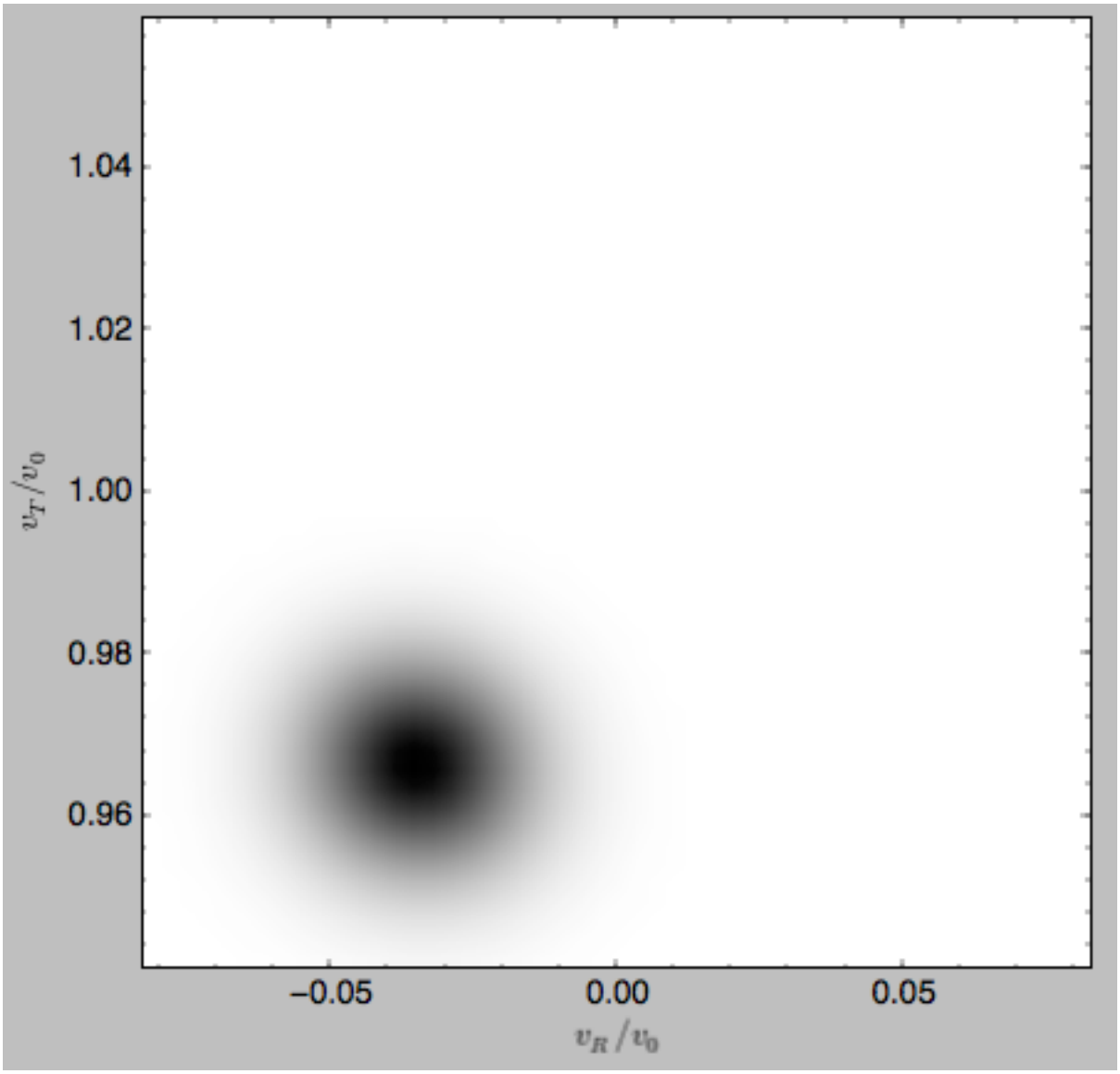
```
>>> print(edfcold.oortB(0.9,phi=22.5,deg=True,t=0.,grid=gridcold,
↪derivphiGrid=gridphicold,derivRGrid=gridrcold))
# -0.574674310521
>>> print(edfwarm.oortB(0.9,phi=22.5,deg=True,t=0.,grid=gridwarm,
↪derivphiGrid=gridphiwarm,derivRGrid=gridrwarm))
# -0.555546911144
```

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

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

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

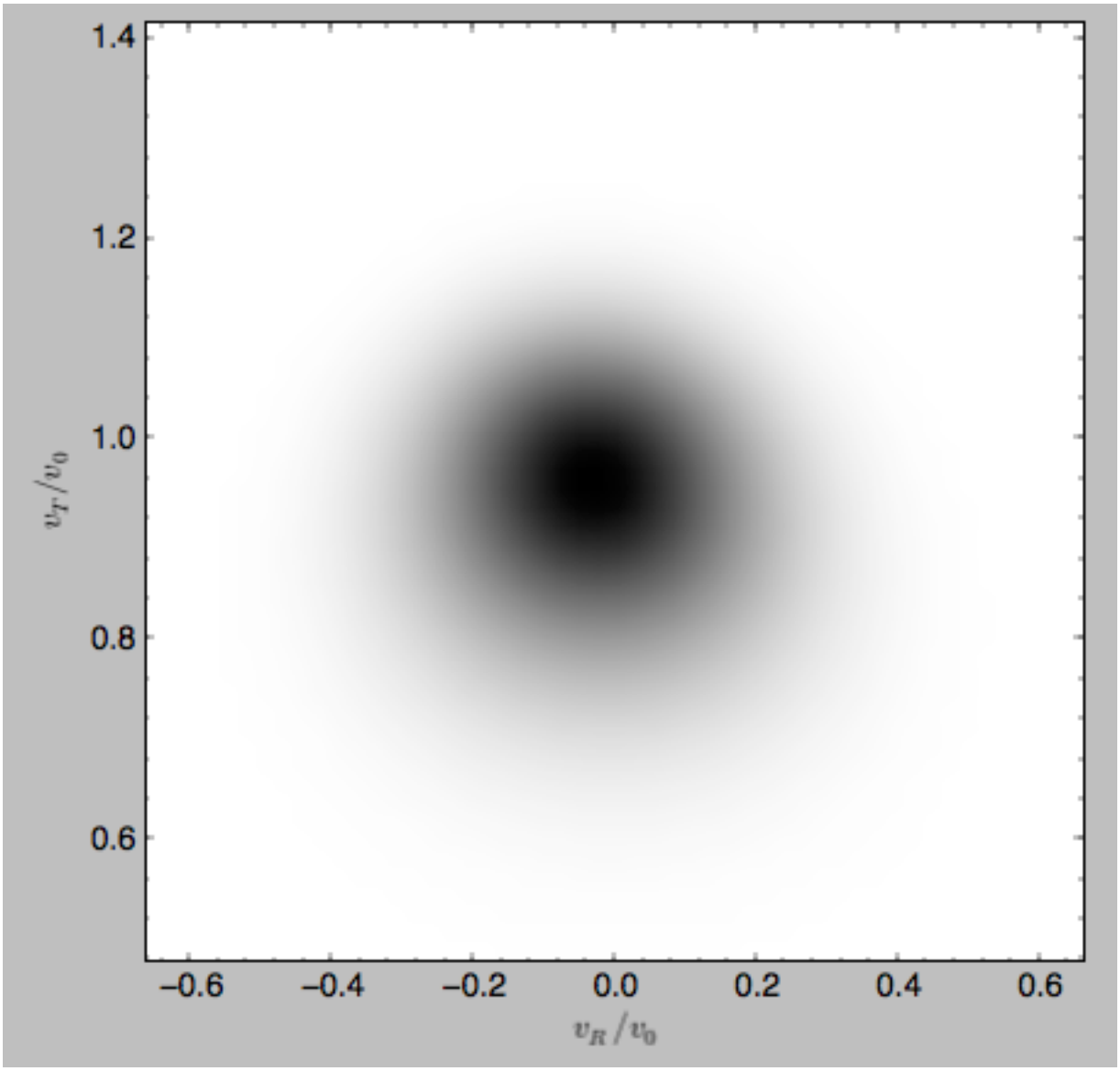
gives



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

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

which returns



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

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

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

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

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

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

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

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

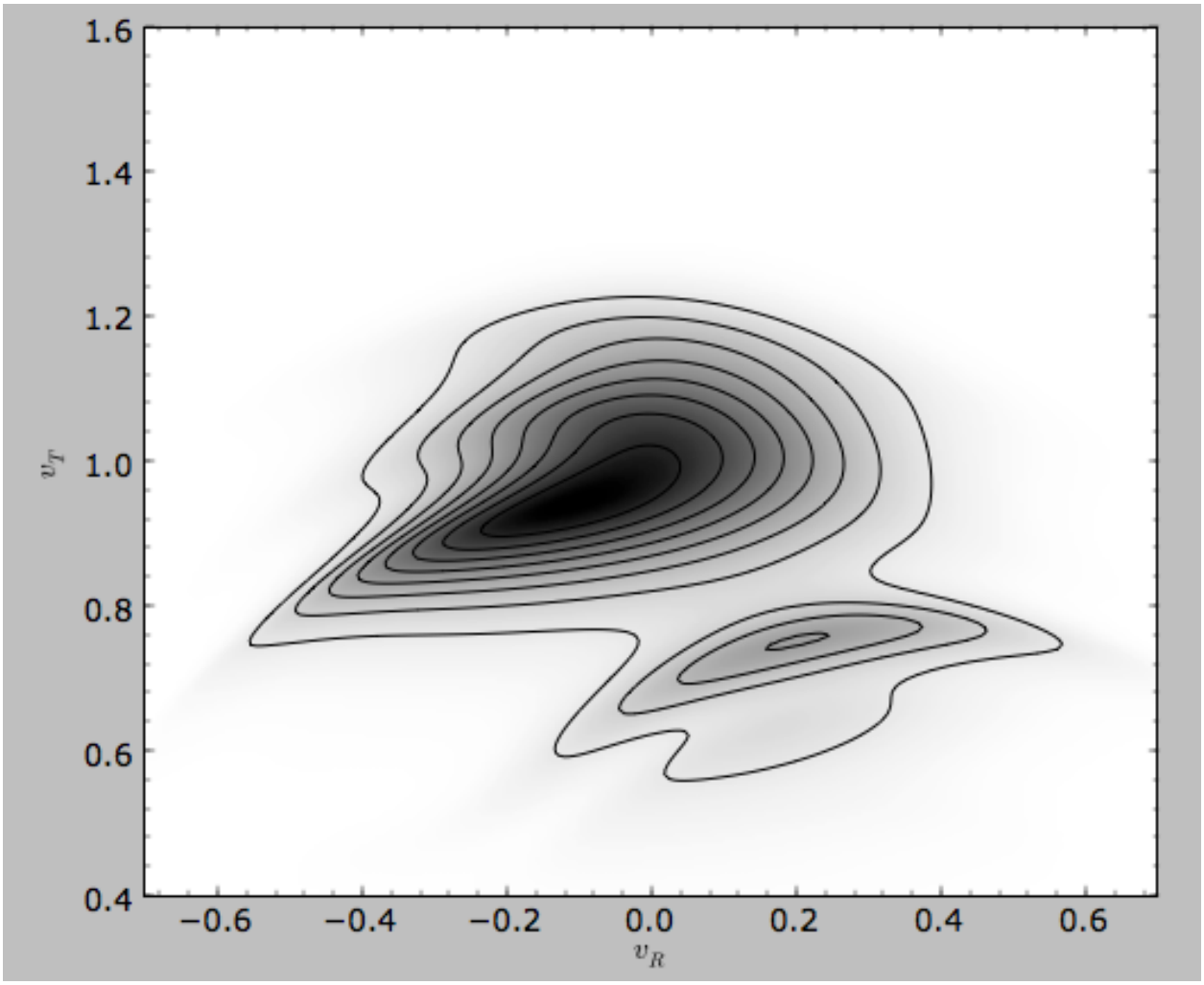
```
>>> ins=[[Orbit([1.,-0.7+1.4/100*jj,1.-0.6+1.2/100*ii,0.]) for jj in range(101)] for␣
↪ii in range(101)]
>>> int=[[o.integrate(ts,[lp,dp]) for o in j] for j in ins]
```

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 `dp.tform()` for each of the orbits

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

This gives

```
>>> from galpy.util.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],xlabel=r'$v_R$',ylabel=r'$v_T$')
```

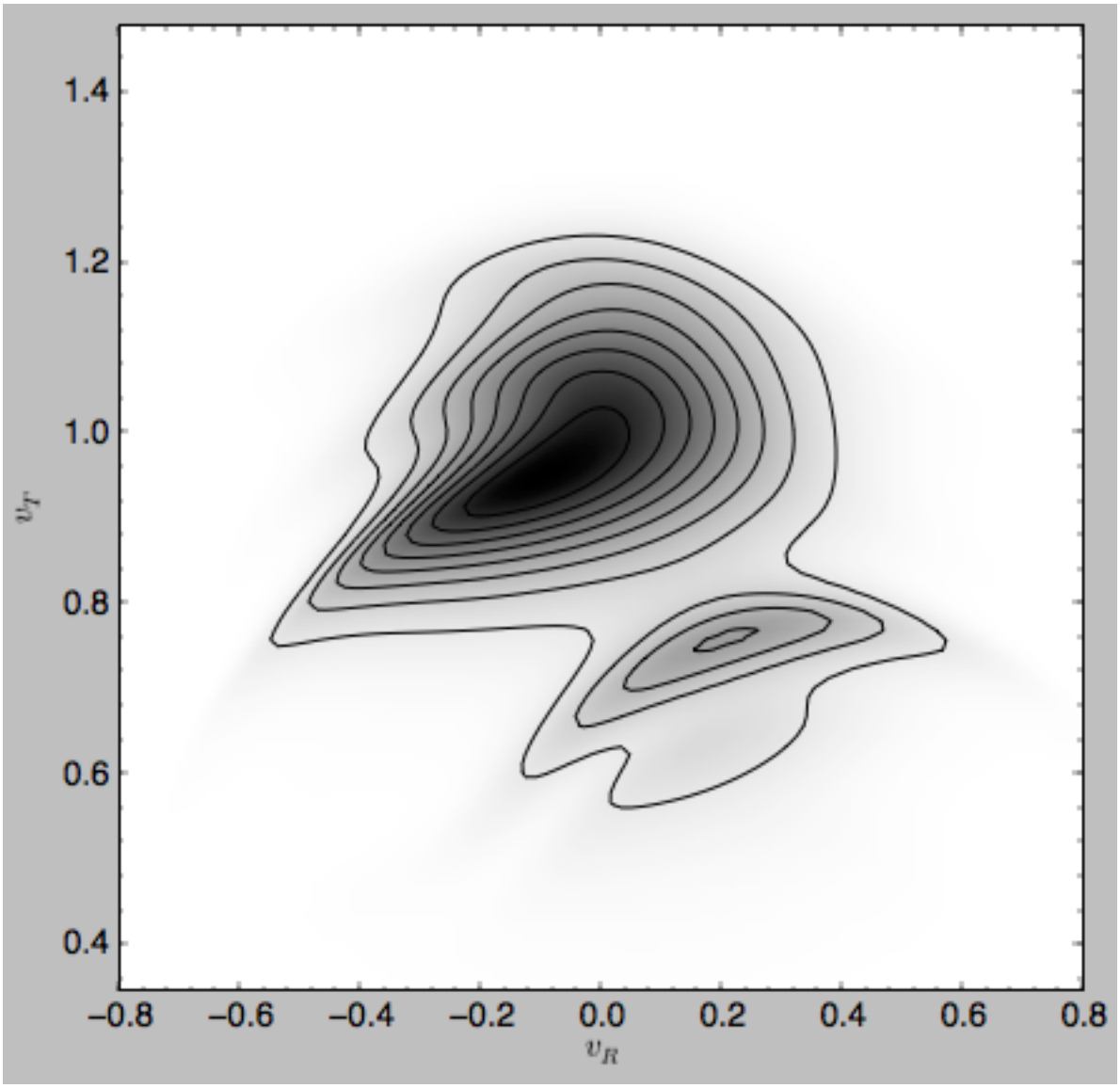


Now that `galpy` contains the `evolveddiskdf` described above, this whole calculation is encapsulated in this module and can be done much more easily as

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

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

```
>>> bovy_dens2d(grid.df.T,origin='lower',cmap='gist_yarg',contours=True,xrange=[grid.
↪vRgrid[0],grid.vRgrid[-1]],yrange=[grid.vTgrid[0],grid.vTgrid[-1]],xlabel=r'$v_R$',
↪ylabel=r'$v_T$')
```



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

1.6 A closer look at orbit integration

1.6.1 Orbit initialization

Standard initialization

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

one specifies the circular velocity, and initial velocities are best expressed as fractions of this circular velocity. For example,

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

initializes a fully three-dimensional orbit, while

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

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

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

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

Initialization with physical units

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

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

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

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

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

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

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

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

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

Initialization from observed coordinates

For orbit integration and characterization of observed stars or clusters, initial conditions can also be specified directly as observed quantities when `radec=True` is set. In this case a full three-dimensional orbit is initialized as `o= Orbit(vxvv=[RA,Dec,distance,pmRA,pmDec,Vlos],radec=True)` where RA and Dec are expressed in degrees, the distance is expressed in kpc, proper motions are expressed in mas/yr (pmra =

pmra' * cos[Dec]), and `Vlos` is the heliocentric line-of-sight velocity given in km/s. The observed epoch is currently assumed to be J2000.00. These observed coordinates are translated to the Galactocentric cylindrical coordinate frame by assuming a Solar motion that can be specified as either `solarmotion=hogg` (default; 2005ApJ...629..268H), `solarmotion=dehnen` (1998MNRAS.298..387D) or `solarmotion=schoenrich` (2010MNRAS.403.1829S). A circular velocity can be specified as `vo=220` in km/s and a value for the distance between the Galactic center and the Sun can be given as `ro=8.0` in kpc (e.g., 2012ApJ...759..131B). While the inputs are given in physical units, the orbit is initialized assuming a circular velocity of one at the distance of the Sun (that is, the orbit's position and velocity is scaled to galpy's *natural* units after converting to the Galactocentric coordinate frame, using the specified `ro=` and `vo=`). The parameters of the coordinate transformations are stored internally, such that they are automatically used for relevant outputs (for example, when the RA of an orbit is requested). An example of all of this is:

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

However, the internally stored position/velocity vector is

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

and is therefore in *natural* units.

Tip: Initialization using observed coordinates can also use units. So, for example, proper motions can be specified as `2*units.mas/units.yr`.

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

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

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

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

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

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

1.6.2 Orbit integration

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

```
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> o= Orbit(vxvv=[1.,0.1,1.1,0.,0.1,0.])
```

(continues on next page)

(continued from previous page)

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

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

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

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

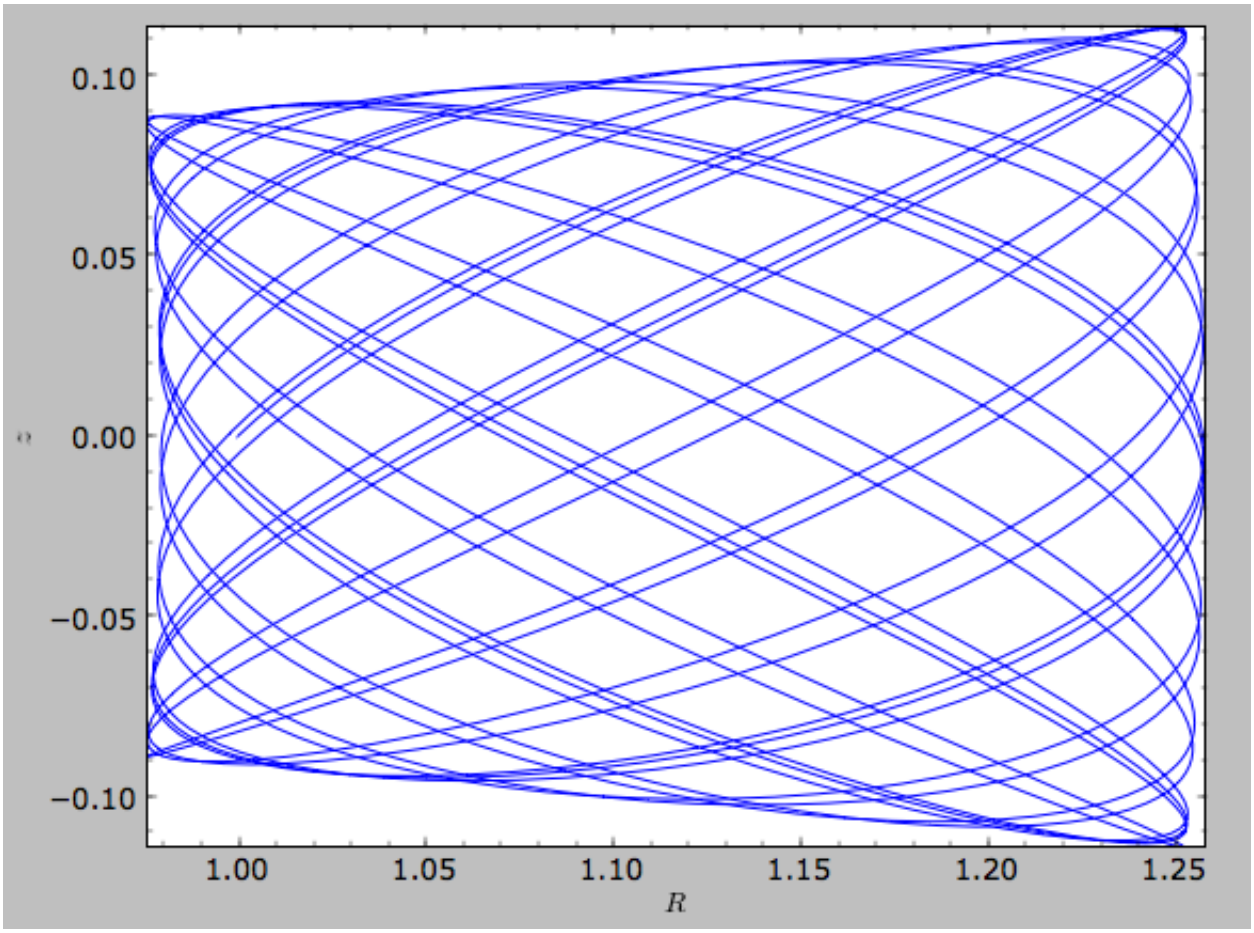
```
>>> op= Orbit(vxvv=[1.,0.1,1.1,0.,0.1,0.],ro=8.,vo=220.) #Use Vc=220 km/s at R= 8 kpc␣
↪as the normalization
>>> op.integrate(ts,lp)
```

1.6.3 Displaying the orbit

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

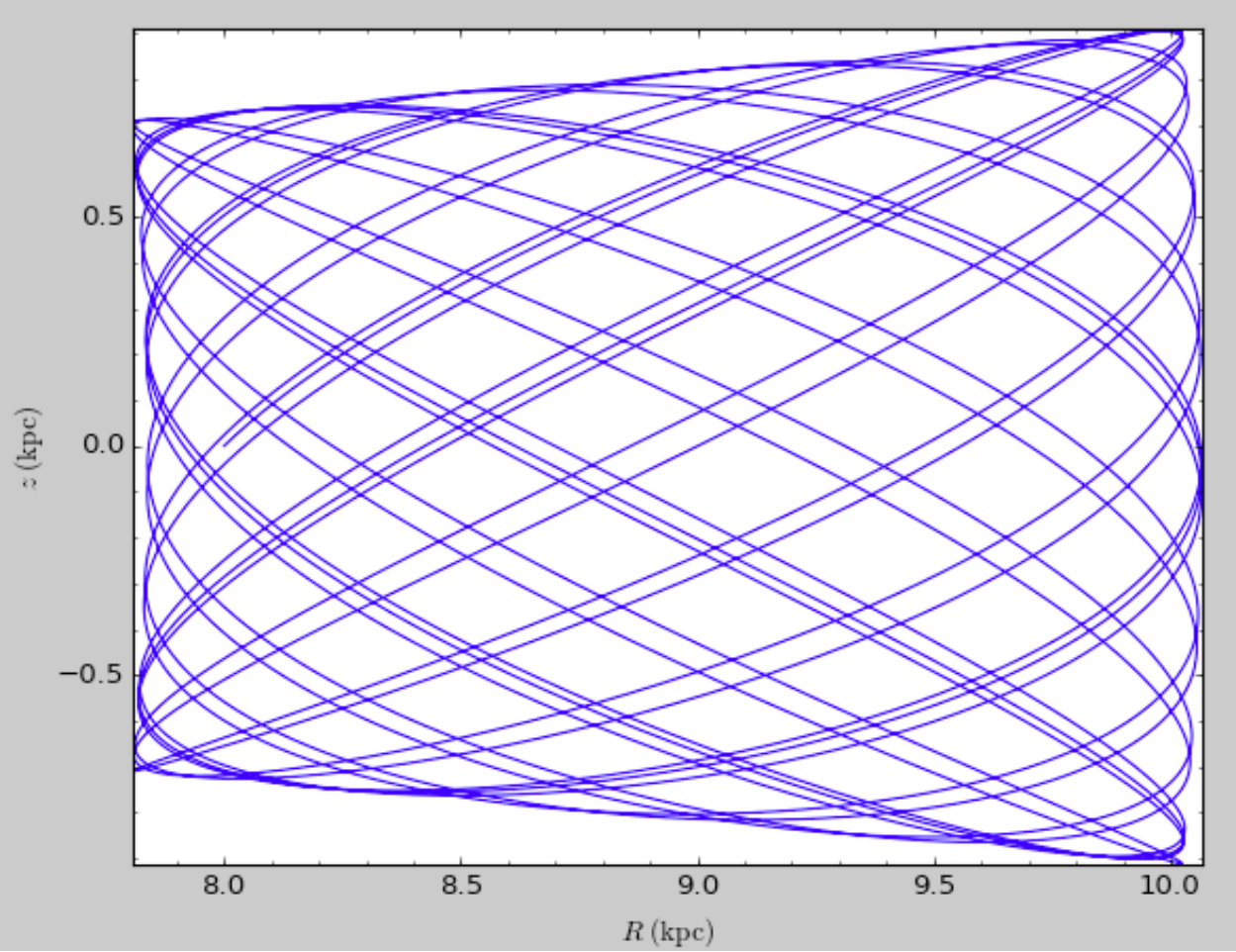
```
>>> o.plot()
```

gives



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

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

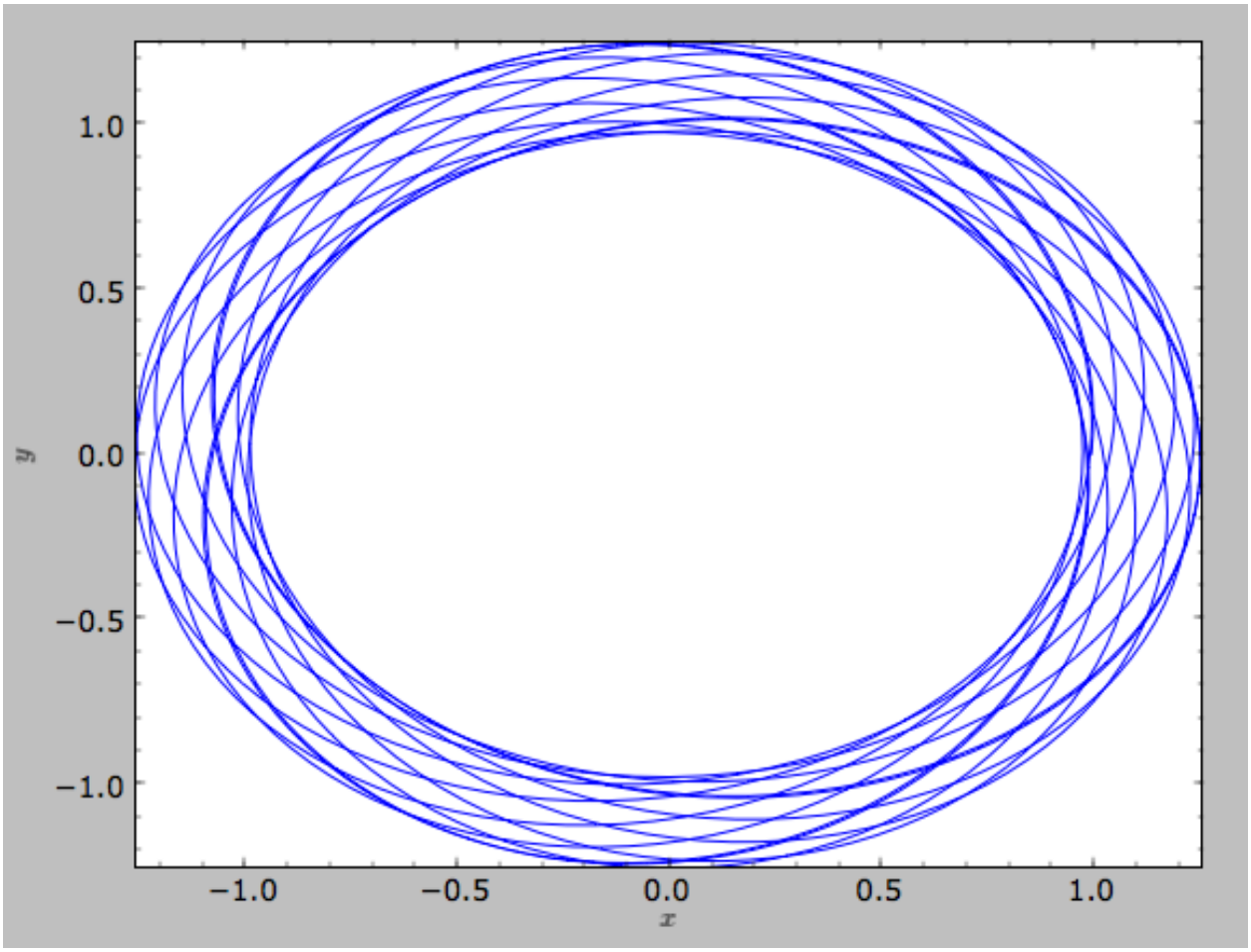


If we call `op.plot(use_physical=False)`, the quantities will be displayed in natural galpy coordinates.

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

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

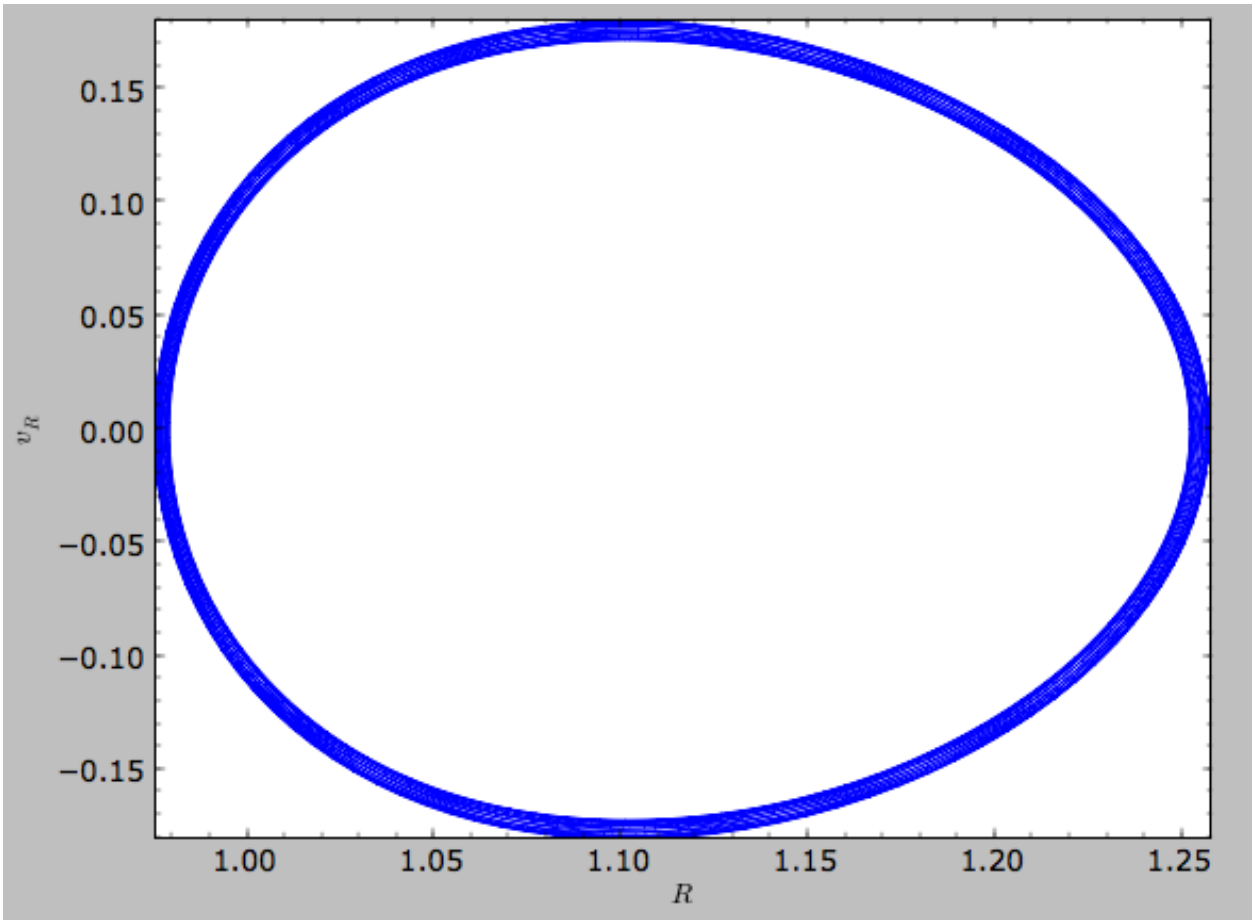
gives the projection onto the plane of the orbit:



while

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

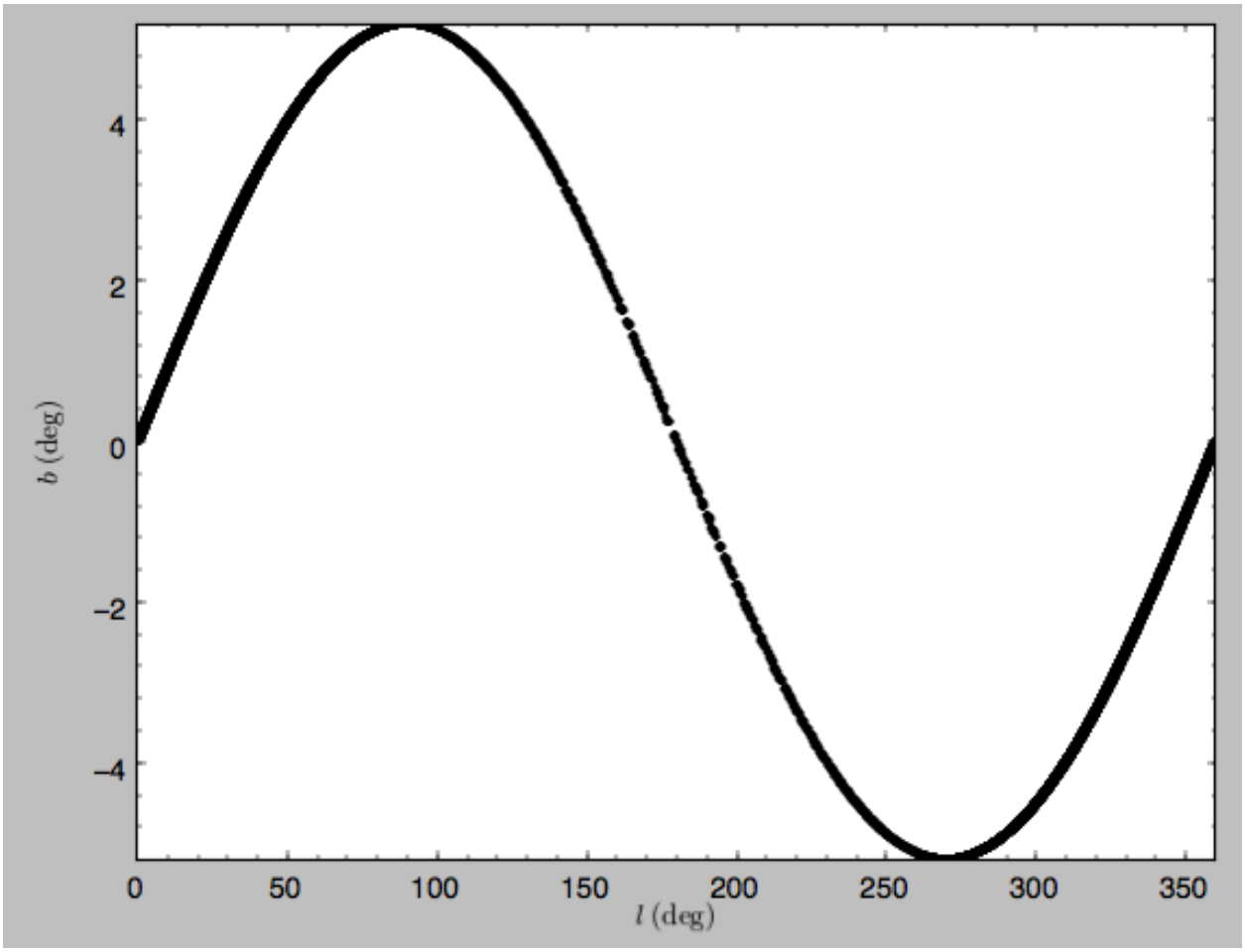
gives the projection onto (R,vR):



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

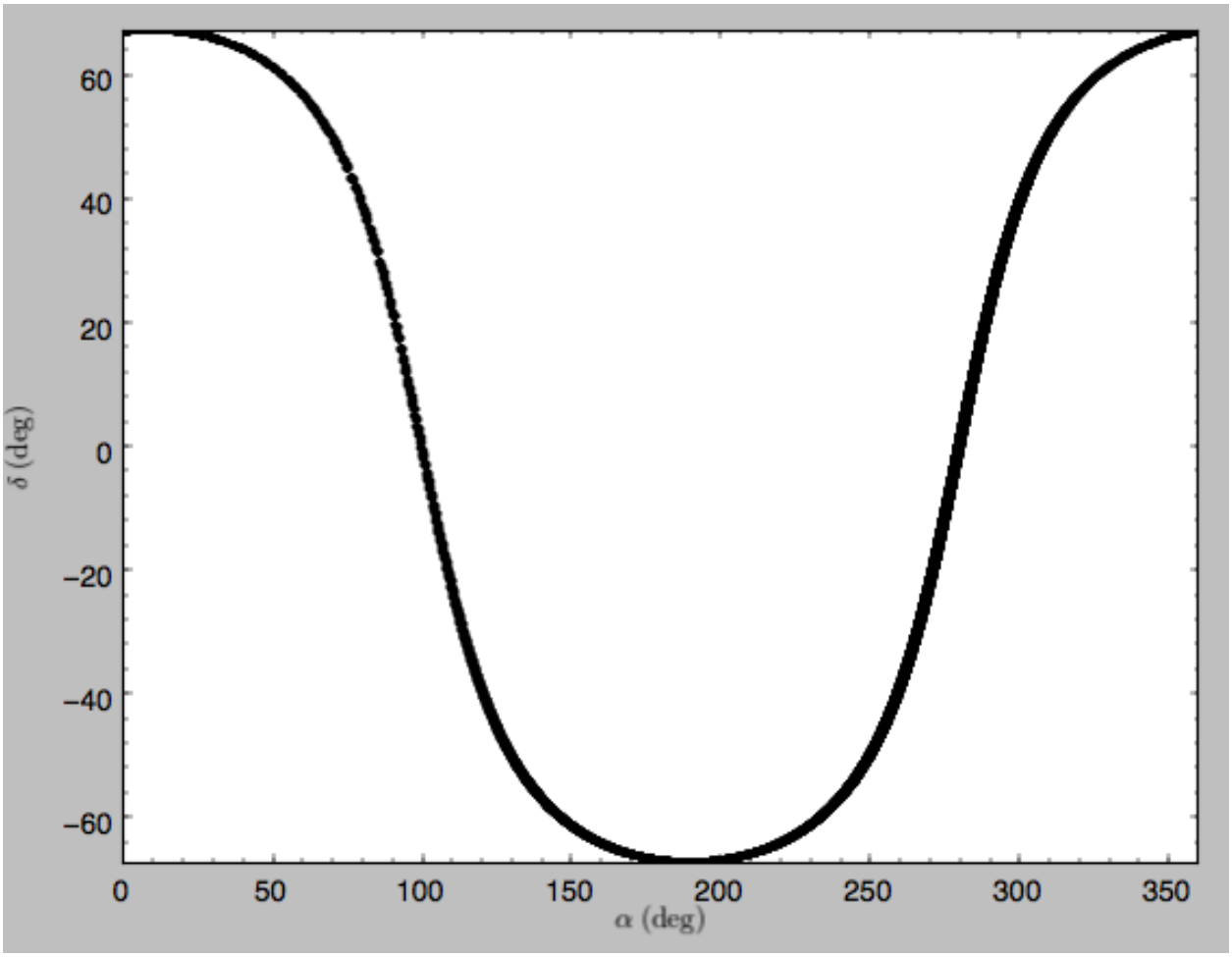
```
>>> o.plot('k.',d1='ll',d2='bb')
```

which shows



or RA and Dec

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

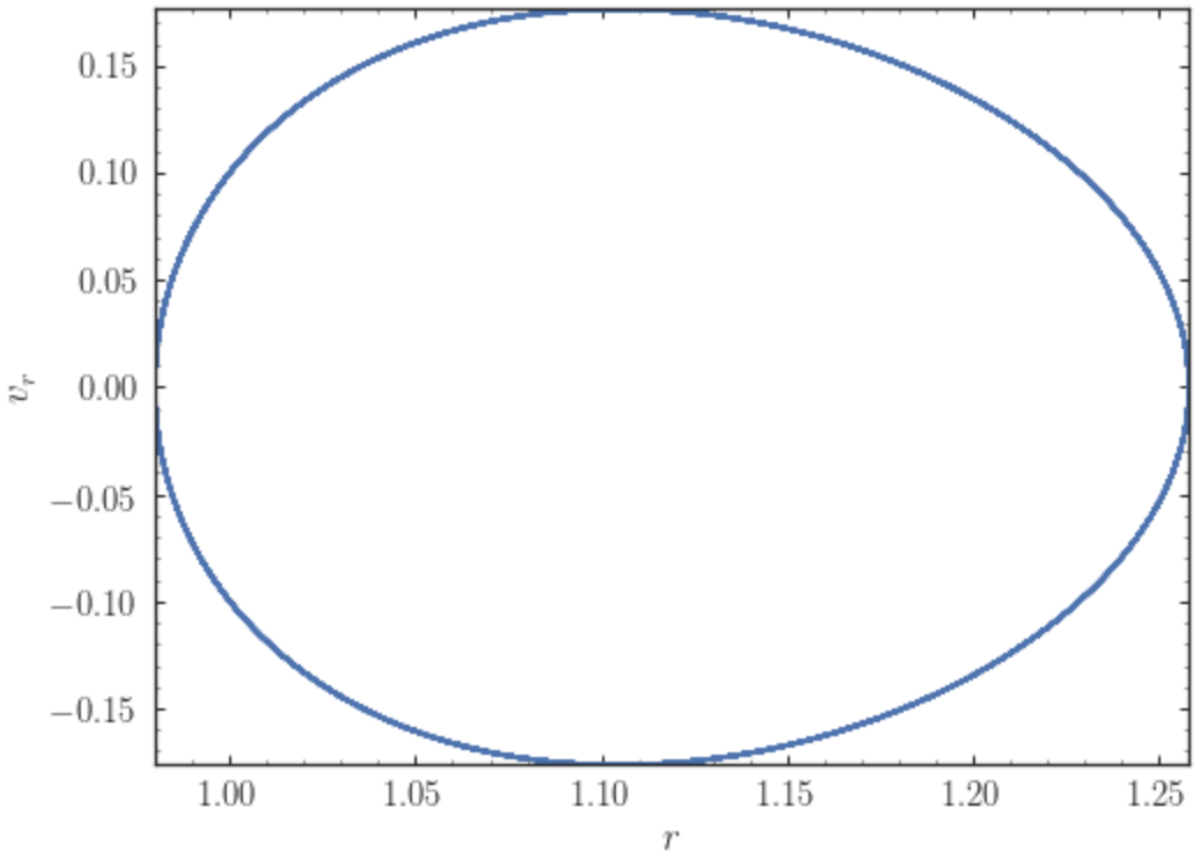


See the documentation of the o.plot function and the o.ra(), o.ll(), etc. functions on how to provide the necessary parameters for the coordinate transformations.

Finally, it is also possible to plot arbitrary functions of time with `Orbit.plot`, by specifying `d1=` or `d2=` as a function. This is for example useful if you want to display the orbit in a different coordinate system. For example, to display the orbital velocity in the spherical radial direction (which is currently not a pre-defined option), you can do the following

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

where `d2=` converts the velocity to spherical coordinates. This gives the following orbit (which is closed in this projection, because we are using a spherical potential):



1.6.4 NEW in v1.3: Animating the orbit

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

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

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

This will create the following animation

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

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

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

which gives

1.6.5 Orbit characterization

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

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

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

```
>>> o.rap(analytic=True), o.rperi(analytic=True), o.e(analytic=True), o.
↪zmax(analytic=True)
# (1.2581448917376636,0.97981640959995842,0.12436697719989584,0.11390708640305315)
```

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

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

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

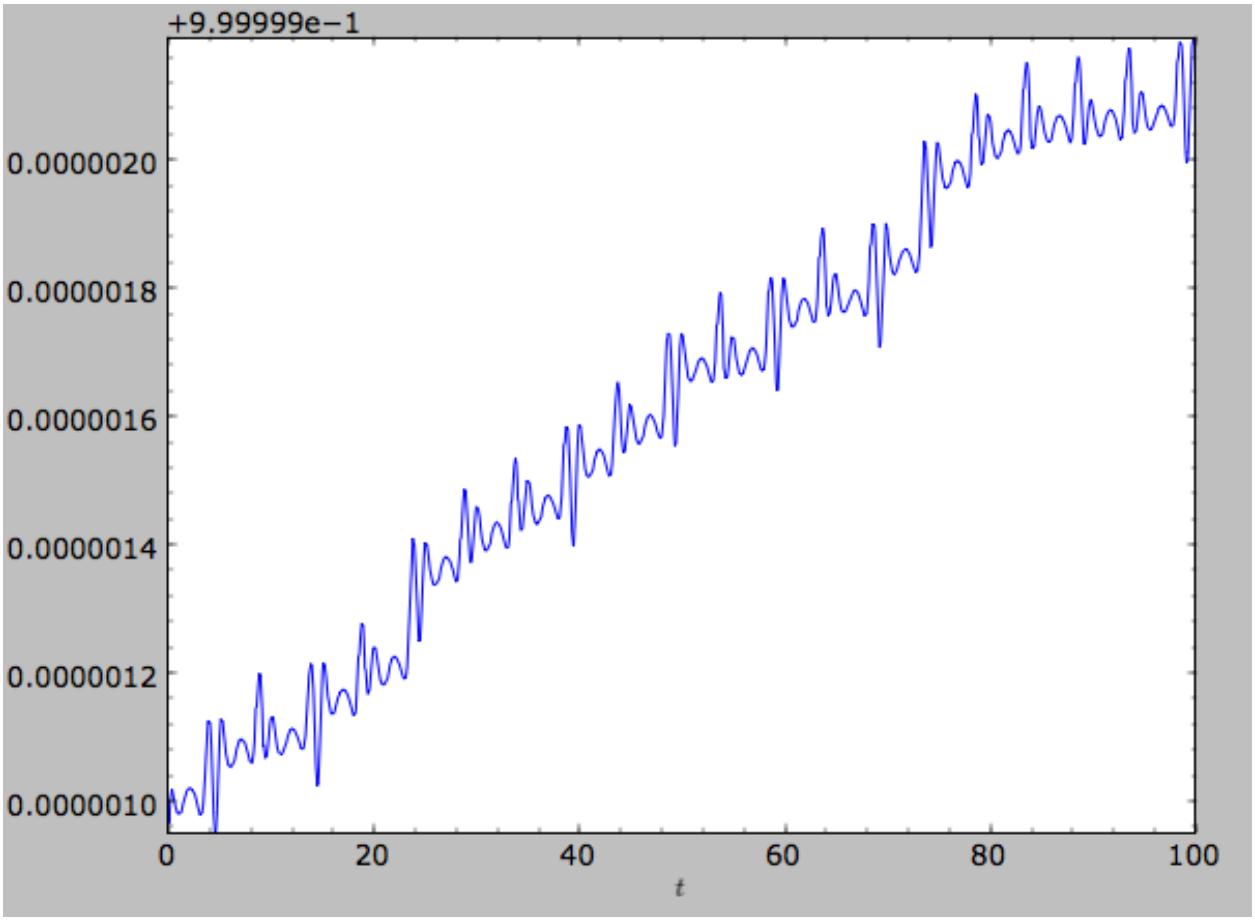
For the Orbit `op` that was initialized above with a distance scale `ro=` and a velocity scale `vo=`, these outputs are all in physical units

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

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

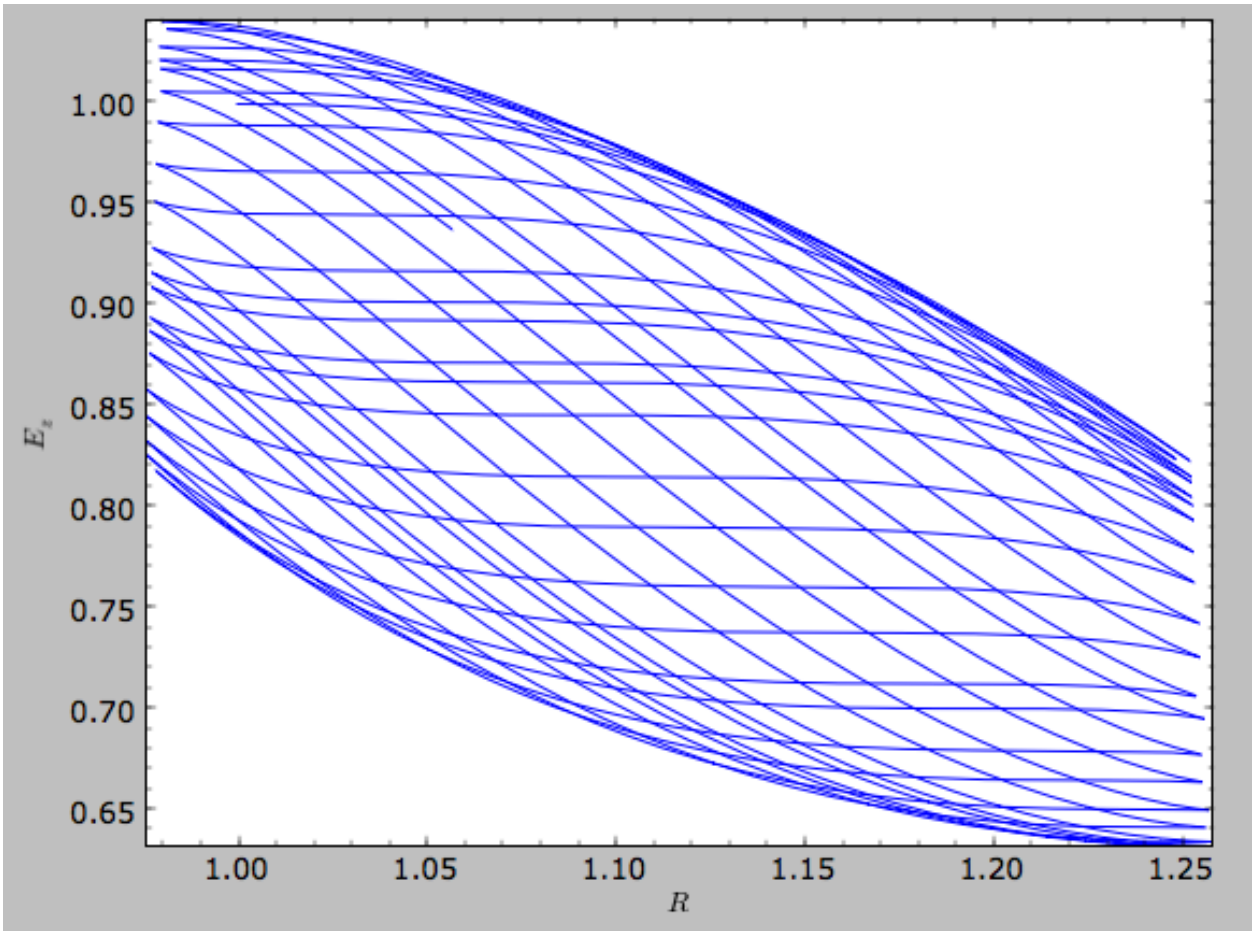
```
>>> o.plotE(normed=True)
```

gives



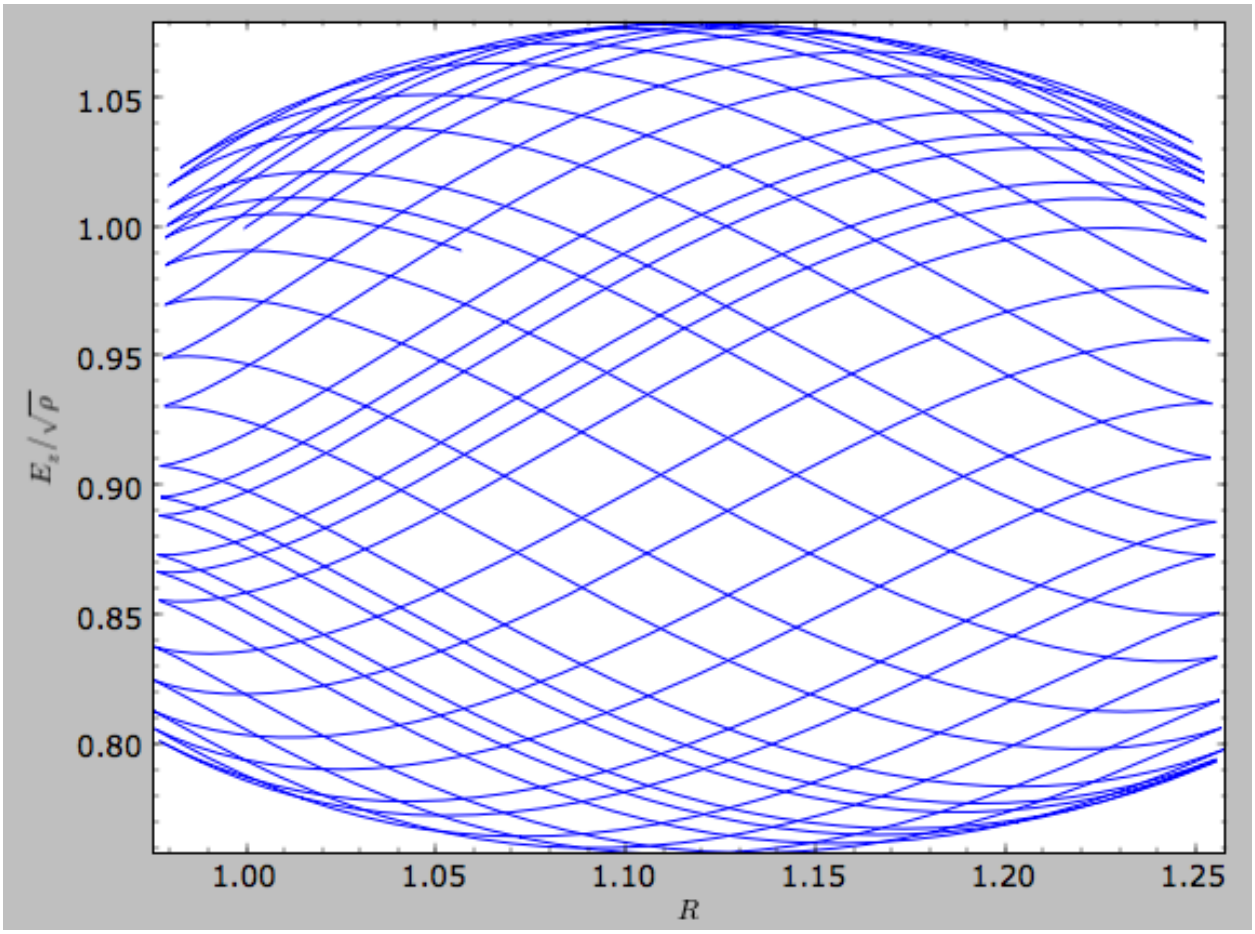
We can specify another quantity to plot the energy against by specifying `d1=`. We can also show the vertical energy, for example, as a function of R

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



Often, a better approximation to an integral of the motion is given by Ez/sqrt(density[R]). We refer to this quantity as `EzJz` and we can plot its behavior

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



1.6.6 NEW in v1.3 Fast orbit characterization

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

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

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

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

This interface automatically estimates the necessary delta parameter based on the initial condition of the `Orbit` object.

While this is useful and fast for individual `Orbit` objects, it is likely that users will want to rapidly evaluate the orbit parameters of large numbers of objects. It is possible to perform the orbital parameter estimation above through the *actionAngle* interface. To do this, we need arrays of the phase-space points `R`, `vR`, `vT`, `z`, `vz`, and `phi` for the objects. The orbit parameters are then calculated by first specifying an `actionAngleStaeckel` instance (this requires a single `delta` focal-length parameter, see *the documentation of the actionAngleStaeckel class*), then using the `EccZmaxRperiRap` method with the data points:

```
>>> aAS = actionAngleStaeckel(pot=mp, delta=0.4)
>>> e, Zmax, rperi, rap = aAS.EccZmaxRperiRap(R, vR, vT, z, vz, phi)
```

Alternatively, you can specify an array for `delta` when calling `aAS.EccZmaxRperiRap`, for example by first estimating good `delta` parameters as follows:

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

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

```
>>> e, Zmax, rperi, rap = aAS.EccZmaxRperiRap(R, vR, vT, z, vz, phi, delta=delta)
```

Th `EccZmaxRperiRap` method also exists for the `actionAngleIsochrone`, `actionAngleSpherical`, and `actionAngleAdiabatic` modules.

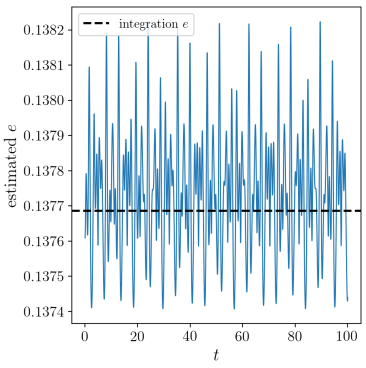
We can test the speed of this method in iPython by finding the parameters at 100000 steps along an orbit in MWPotential2014, like this

```
>>> o= Orbit(vxvv=[1.,0.1,1.1,0.,0.1,0.])
>>> ts = numpy.linspace(0,100,100000)
>>> o.integrate(ts,MWPotential2014)
>>> aAS = actionAngleStaeckel(pot=MWPotential2014,delta=0.3)
>>> R, vR, vT, z, vz, phi = o.getOrbit().T
>>> delta = estimateDeltaStaeckel(MWPotential2014, R, z, no_median=True)
>>> %timeit -n 10 es, zms, rps, ras = aAS.EccZmaxRperiRap(R,vR,vT,z,vz,phi,
↪delta=delta)
#10 loops, best of 3: 899 ms per loop
```

you can see that in this potential, each phase space point is calculated in roughly 9µs. further speed-ups can be gained by using the `actionAngleStaeckelGrid` module, which first calculates the parameters using a grid-based interpolation

```
>>> from galpy.actionAngle import actionAngleStaeckelGrid
>>> aASG= actionAngleStaeckelGrid(pot=mp,delta=0.4,nE=51,npsi=51,nLz=61,c=True,
↪interpecc=True)
>>> %timeit -n 10 es, zms, rps, ras = aASG.EccZmaxRperiRap(R,vR,vT,z,vz,phi)
#10 loops, best of 3: 587 ms per loop
```

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



1.6.7 Accessing the raw orbit

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

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

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

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

which we can also access in natural coordinates as

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

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

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

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

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

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

The whole orbit can also be obtained using the function getOrbit

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

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

1.6.8 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

```
>>> 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 `symplec4_c`, which is speedy and reliable. For example, compare

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

As this example shows, galpy will issue a warning that C is being used.

1.6.9 Integration of the phase-space volume

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

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

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

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

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

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

This determinant should be equal to one

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

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

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

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

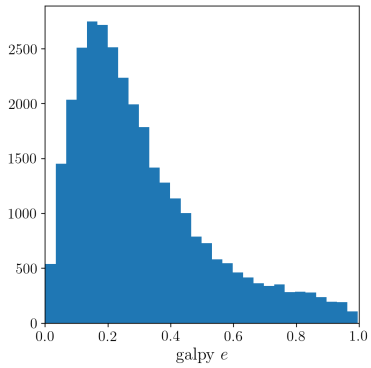
A straightforward application of galpy's orbit initialization and integration capabilities is to derive the eccentricity distribution of a set of thick disk stars. We start by downloading the sample of SDSS SEGUE (2009AJ....137.4377Y) thick disk stars compiled by Dierickx et al. (2010arXiv1009.1616D)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

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

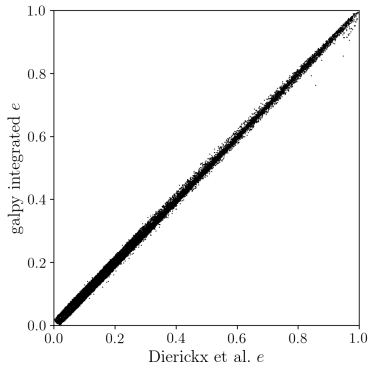
After reading in the data (RA,Dec,distance,pmRA,pmDec,vlos; see above) as a vector `vxvv` with dimensions [6,ndata] we (a) define the potential in which we want to integrate the orbits, and (b) integrate each orbit and save its eccentricity as calculated analytically following the *Staeckel approximation method* and by orbit integration (running this for all 30,000-ish stars will take about half an hour)

```
>>> from galpy.actionAngle import UnboundError
>>> ts= np.linspace(0.,20.,10000)
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> e_ana = numpy.zeros(len(vxvv))
>>> e_int = numpy.zeros(len(vxvv))
>>> for i in range(len(vxvv)):
...     #calculate analytic e estimate, catch any 'unbound' orbits
...     try:
...         orbit = Orbit(vxvv[i], radec=True, vo=220., ro=8.)
...         e_ana[i] = orbit.e(analytic=True, pot=lp, c=True)
...     except UnboundError:
...         #parameters cannot be estimated analytically
...         e_ana[i] = np.nan
...     #integrate the orbit and return the numerical e value
...     orbit.integrate(ts, lp)
...     e_int[i] = orbit.e(analytic=False)
```

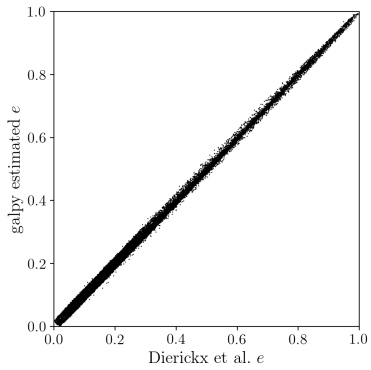
We then find the following eccentricity distribution (from the numerical eccentricities)



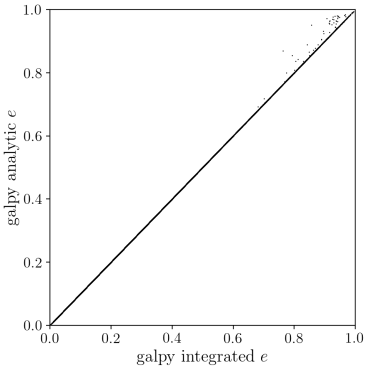
The eccentricity calculated by integration in galpy compare well with those calculated by Dierickx et al., except for a few objects



and the analytical estimates are equally as good:



In comparing the analytic and integrated eccentricity estimates - one can see that in this case the estimation is almost exact, due to the spherical symmetry of the chosen potential:



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

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

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

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

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

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

1.7 Action-angle coordinates

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

Since v1.2, galpy can also compute positions and velocities corresponding to a given set of actions and angles for axisymmetric potentials using the TorusMapper code of Binney & McMillan (2016). This is described in *this section* below. The interface for this is different than for the other action-angle classes, because the transformations are generally different.

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

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

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

- __call__: returns the actions

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

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

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

1.7.1 Action-angle coordinates for the isochrone potential

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

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

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

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

or for a more eccentric orbit

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

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

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

We can likewise calculate the frequencies as well

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

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

We obtain the angles as well by calling

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

(continues on next page)

(continued from previous page)

```
#  array([ 0.57101518]),
#  array([ 5.96238847]),
#  array([ 1.24999949]))
```

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

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

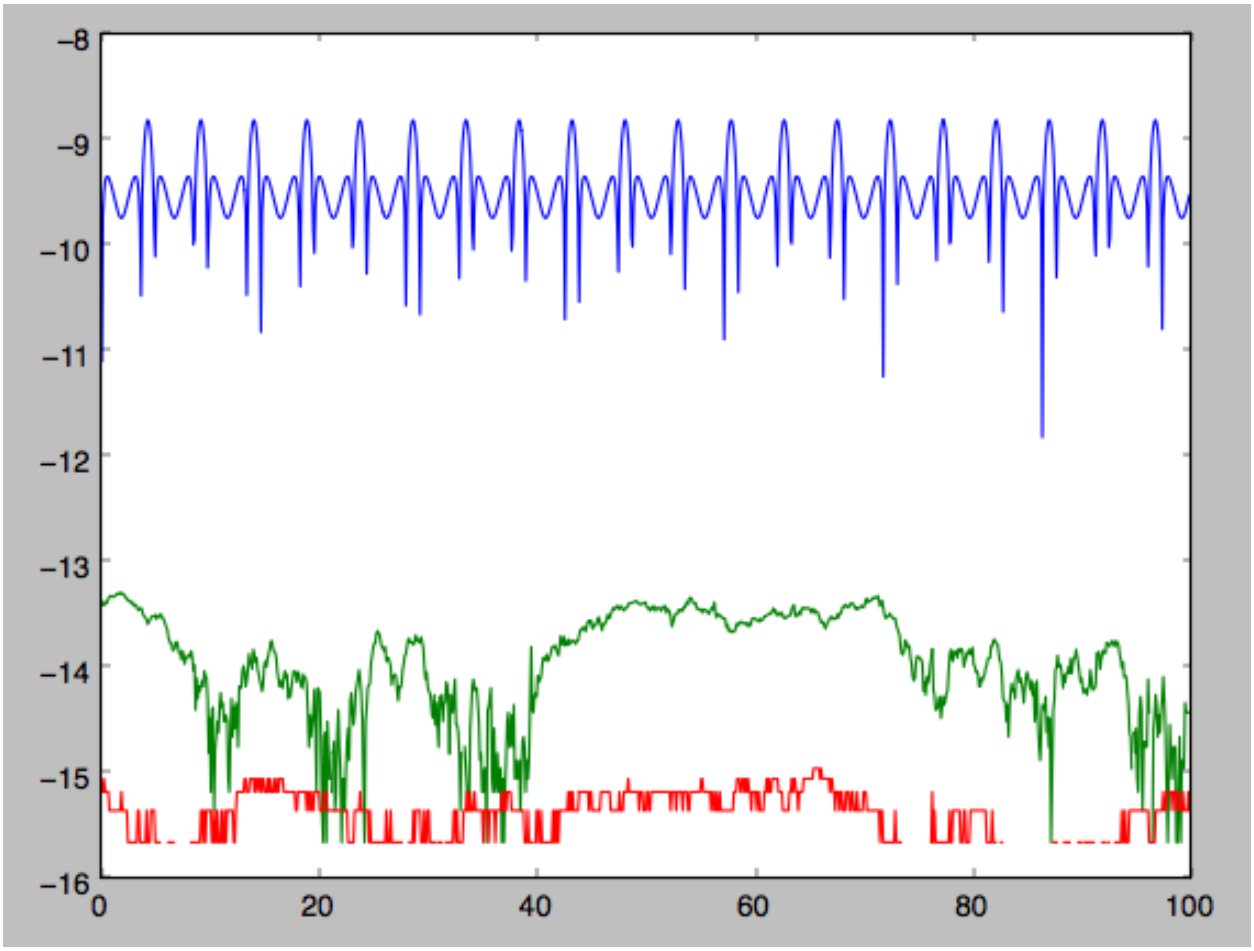
```
>>> from galpy.orbit import Orbit
>>> o= Orbit([1.,0.5,1.3,0.2,0.1,0.])
>>> ts= numpy.linspace(0.,100.,1001)
>>> o.integrate(ts,ip)
>>> jfa= aAI.actionsFreqsAngles(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),o.phi(ts))
```

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

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

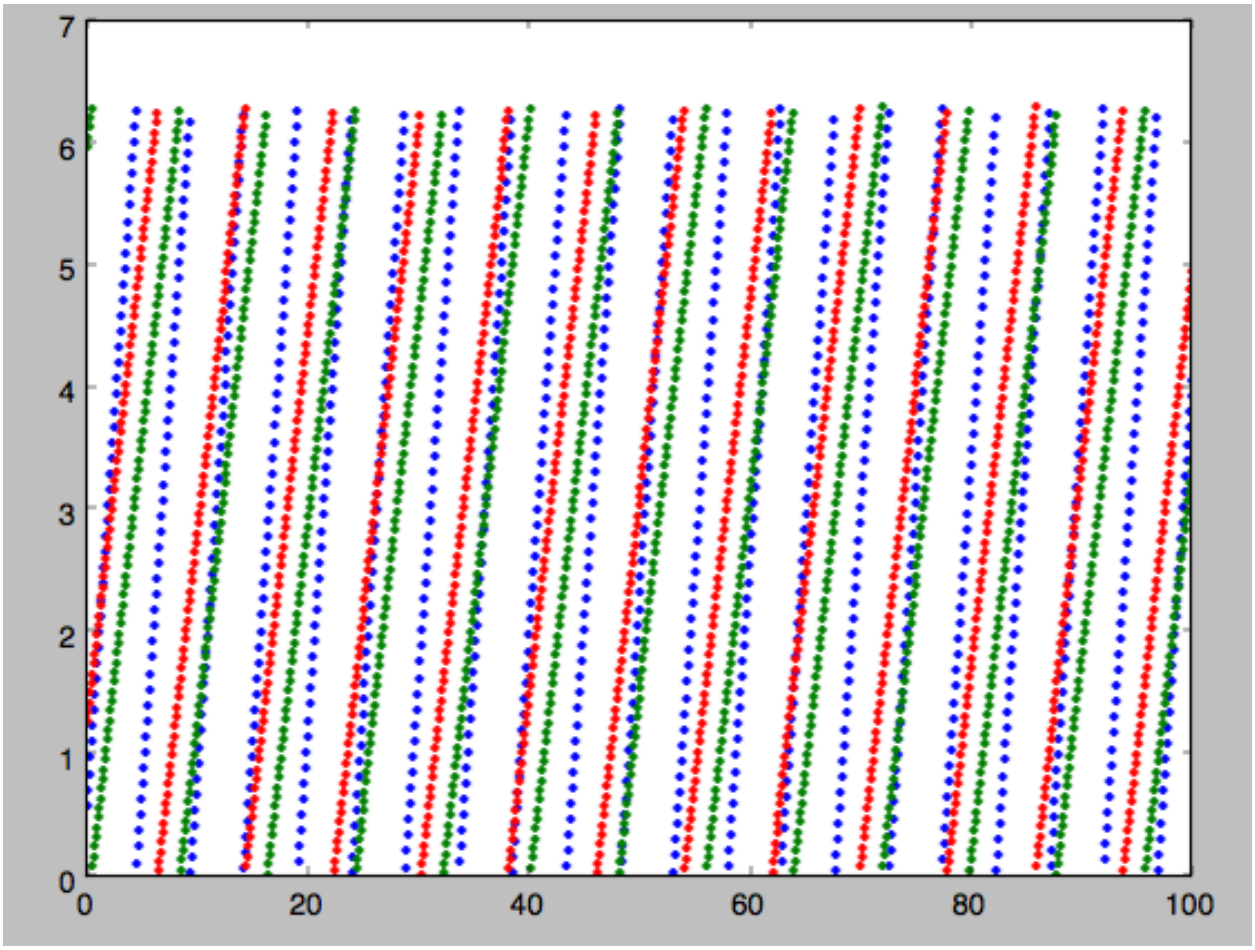
```
>>> plot(ts,numpy.log10(numpy.fabs((jfa[0]-numpy.mean(jfa[0])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[1]-numpy.mean(jfa[1])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[2]-numpy.mean(jfa[2])))))
```

which gives



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

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



1.7.2 Action-angle coordinates for spherical potentials

Action-angle coordinates for any spherical potential can be calculated using a few orbit integrations. These are implemented in galpy in the `actionAngleSpherical` module. For example, we can do

```
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> from galpy.actionAngle import actionAngleSpherical
>>> aAS= actionAngleSpherical(pot=lp)
```

For the same eccentric orbit as above we find

```
>>> aAS(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.22022112]), array([ 1.3]), array([ 0.02574507]))
>>> aAS.actionsFreqs(1.,0.5,1.3,0.2,0.1,0.)
# (array([ 0.22022112]),
#  array([ 1.3]),
#  array([ 0.02574507]),
#  array([ 0.87630459]),
```

(continues on next page)

(continued from previous page)

```
#  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([ 0.02574507]),
#  array([ 0.87630459]),
#  array([ 0.60872881]),
#  array([ 0.60872881]),
#  array([ 0.40443857]),
#  array([ 5.85965048]),
#  array([ 1.1472615]))
```

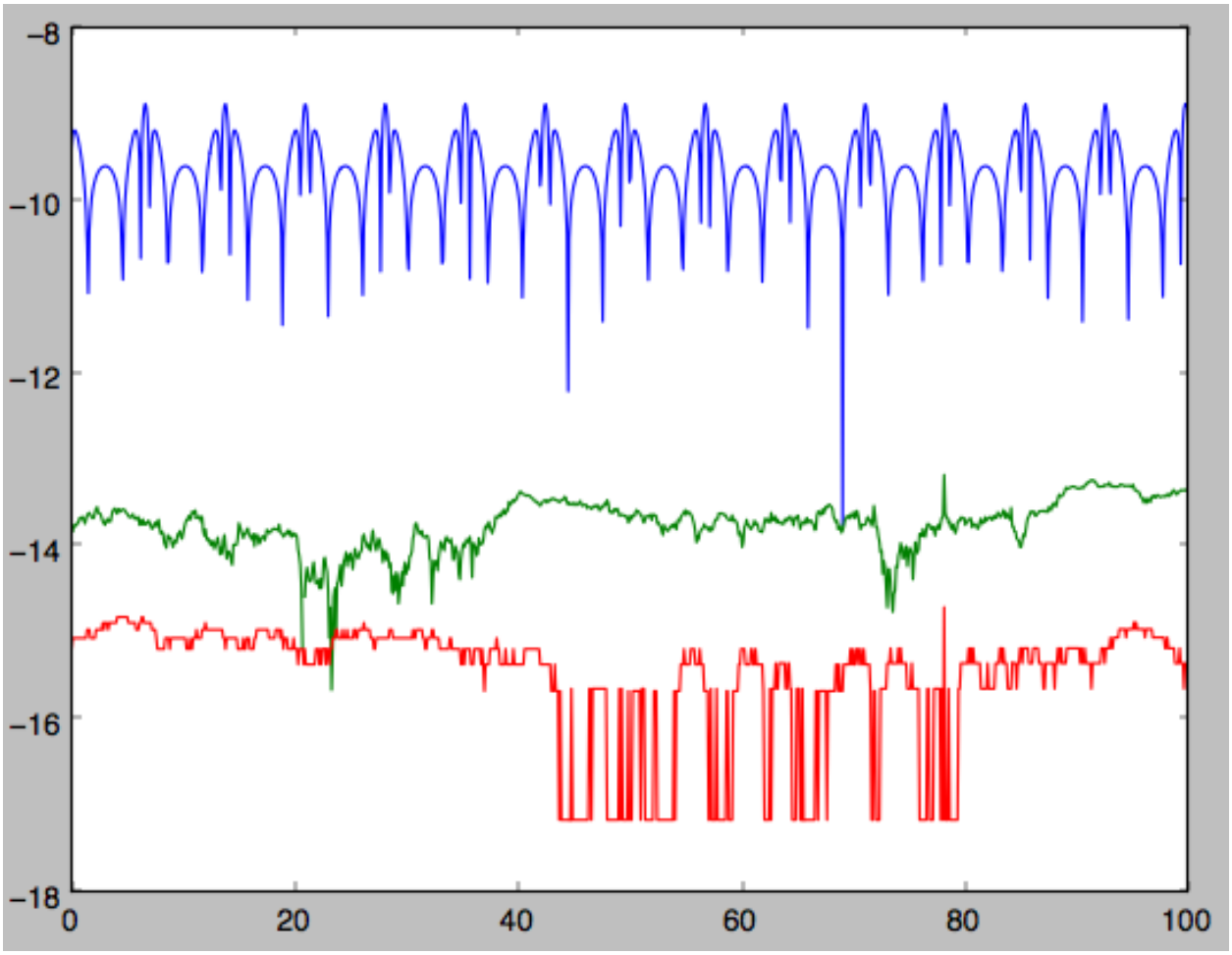
We can again check that the actions are conserved along the orbit and that the angles increase linearly with time:

```
>>> o.integrate(ts,lp)
>>> jfa= aAS.actionsFreqsAngles(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),o.phi(ts),
↪fixed_quad=True)
```

where we use `fixed_quad=True` for a faster evaluation of the required one-dimensional integrals using Gaussian quadrature. We then plot the action fluctuations

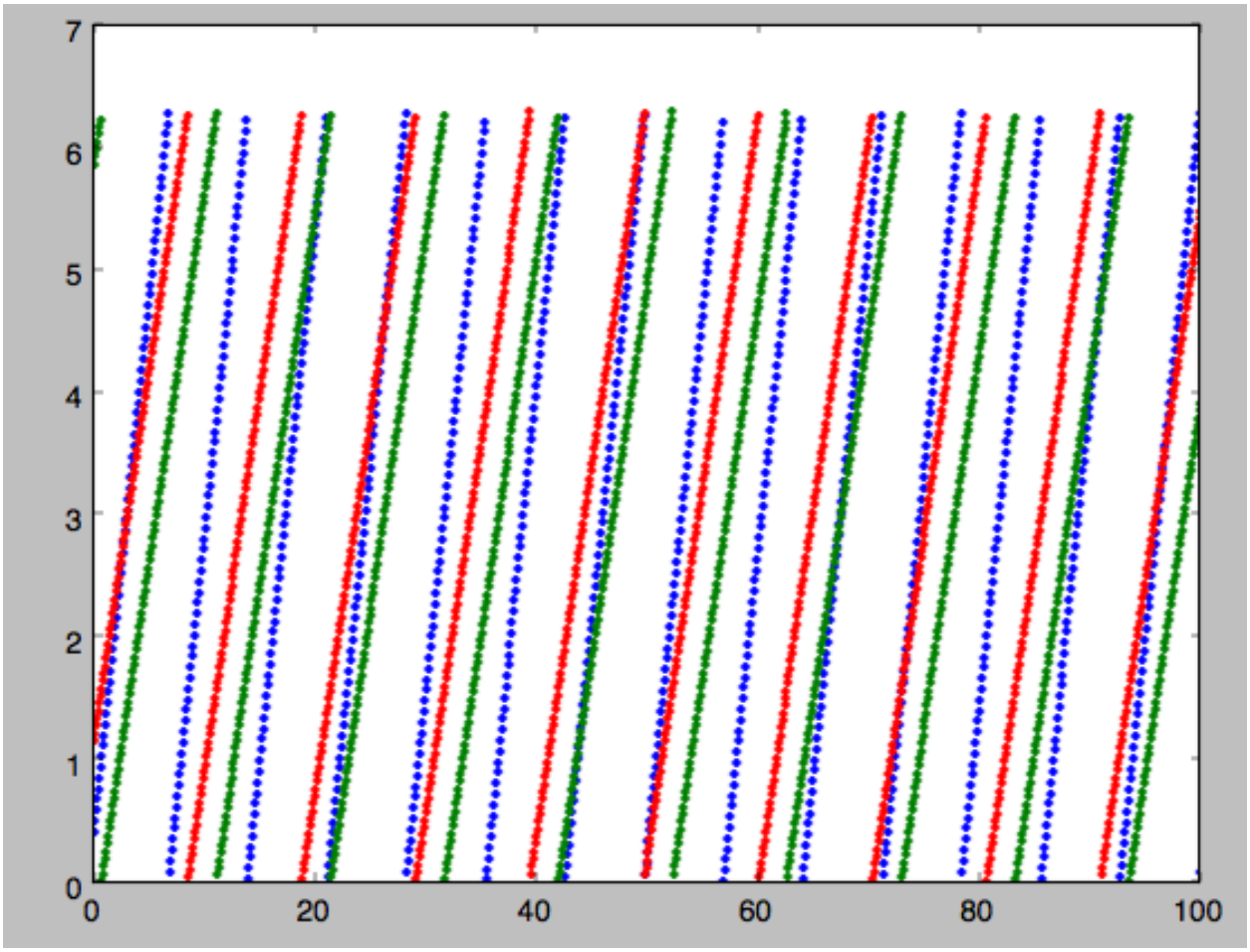
```
>>> plot(ts,numpy.log10(numpy.fabs((jfa[0]-numpy.mean(jfa[0])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[1]-numpy.mean(jfa[1])))))
>>> plot(ts,numpy.log10(numpy.fabs((jfa[2]-numpy.mean(jfa[2])))))
```

which gives



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

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



We can check the spherical action-angle calculations against the analytical calculations for the isochrone potential. Starting again from the isochrone potential used in the previous section

```
>>> ip= IsochronePotential(b=1.,normalize=1.)
>>> aAI= actionAngleIsochrone(ip=ip)
>>> aAS= actionAngleSpherical(pot=ip)
```

we can compare the actions, frequencies, and angles computed using both

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

(continues on next page)

(continued from previous page)

```
#  array([ 0.79093738]),
#  array([ 0.57101518]),
#  array([ 5.96238838]),
#  array([ 1.2499994]))
```

or more explicitly comparing the two

```
>>> [r-s for r,s in zip(aAI.actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.),aAS.
↪actionsFreqsAngles(1.,0.5,1.3,0.2,0.1,0.))]
# [array([  6.66133815e-16]),
#  array([ 0.]),
#  array([ 0.]),
#  array([ -4.53851845e-10]),
#  array([  4.74775219e-10]),
#  array([  4.74775219e-10]),
#  array([ -1.65965242e-10]),
#  array([  9.04759645e-08]),
#  array([  9.04759649e-08])]
```

1.7.3 Action-angle coordinates using the adiabatic approximation

For non-spherical, axisymmetric potentials galpy contains multiple methods for calculating approximate action–angle coordinates. The simplest of those is the adiabatic approximation, which works well for disk orbits that do not go too far from the plane, as it assumes that the vertical motion is decoupled from that in the plane (e.g., 2010MNRAS.401.2318B).

Setup is similar as for other actionAngle objects

```
>>> from galpy.potential import MWPotential2014
>>> from galpy.actionAngle import actionAngleAdiabatic
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014)
```

and evaluation then proceeds similarly as before

```
>>> aAA(1.,0.1,1.1,0.,0.05)
# (0.01351896260559274, 1.1, 0.0004690133479435352)
```

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

```
>>> from galpy.orbit import Orbit
>>> ts=numpy.linspace(0.,100.,1001)
>>> o= Orbit([1.,0.1,1.1,0.,0.05])
>>> o.integrate(ts,MWPotential2014)
>>> js= aAA(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts))
```

This takes a while. The adiabatic approximation is also implemented in C, which leads to great speed-ups. Here is how to use it

```
>>> timeit(aAA(1.,0.1,1.1,0.,0.05))
# 10 loops, best of 3: 73.7 ms per loop
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
>>> timeit(aAA(1.,0.1,1.1,0.,0.05))
# 1000 loops, best of 3: 1.3 ms per loop
```

or about a *50 times* speed-up. For arrays the speed-up is even more impressive

```
>>> s= numpy.ones(100)
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 10 loops, best of 3: 37.8 ms per loop
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014) #back to no C
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 1 loops, best of 3: 7.71 s per loop
```

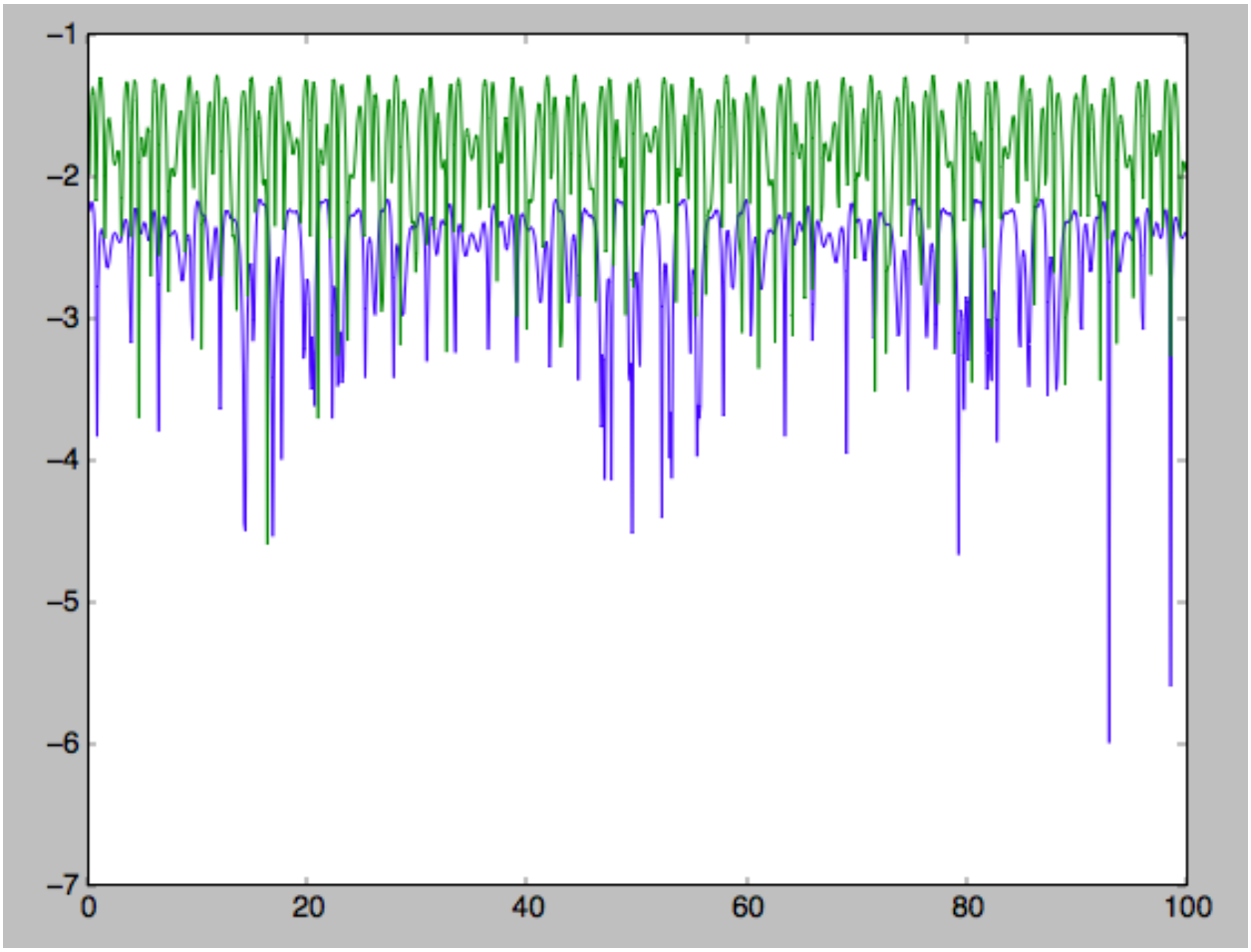
or a speed-up of 200! Back to the previous example, you can run it with `c=True` to speed up the computation

```
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
>>> js= aAA(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts))
```

We can plot the radial- and vertical-action fluctuation as a function of time

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

which gives



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

Another way to speed up the calculation of actions using the adiabatic approximation is to tabulate the actions on a grid in (approximate) integrals of the motion and evaluating new actions by interpolating on this grid. How this is done in practice is described in detail in the galpy paper. To setup this grid-based interpolation method, which is contained in `actionAngleAdiabaticGrid`, do

```
>>> from galpy.actionAngle import actionAngleAdiabaticGrid
>>> aAG= actionAngleAdiabaticGrid(pot=MWPotential2014,nR=31,nEz=31,nEr=51,nLz=51,
↪c=True)
```

where `c=True` specifies that we use the C implementation of `actionAngleAdiabatic` for speed. We can now evaluate in the same was as before, for example

```
>>> aAA(1.,0.1,1.1,0.,0.05), aAG(1.,0.1,1.1,0.,0.05)
# ((array([ 0.01352523]), array([ 1.1]), array([ 0.00046909])),
#  (0.013527010324238781, 1.1, 0.00047747359874375148))
```

which agree very well. To look at the timings, we first switch back to not using C and then list all of the relevant timings:

```
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=False)
# Not using C, direct calculation
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 1 loops, best of 3: 9.05 s per loop
>>> aAA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
# Using C, direct calculation
>>> timeit(aAA(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 10 loops, best of 3: 39.7 ms per loop
# Grid-based calculation
>>> timeit(aAG(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 1000 loops, best of 3: 1.09 ms per loop
```

Thus, in this example (and more generally) the grid-based calculation is significantly faster than even the direct implementation in C. The overall speed up between the direct Python version and the grid-based version is larger than 8,000; the speed up between the direct C version and the grid-based version is 36. For larger arrays of input phase-space positions, the latter speed up can increase to 150. For simpler, fully analytical potentials the speed up will be slightly less, but for `MWPotential2014` and other more complicated potentials (such as those involving a double-exponential disk), the overhead of setting up the grid is worth it when evaluating more than a few thousand actions.

The adiabatic approximation works well for orbits that stay close to the plane. The orbit we have been considering so far only reaches a height two percent of R_0, or about 150 pc for $R_0 = 8$ kpc.

```
>>> o.zmax()*8.
# 0.17903686455491979
```

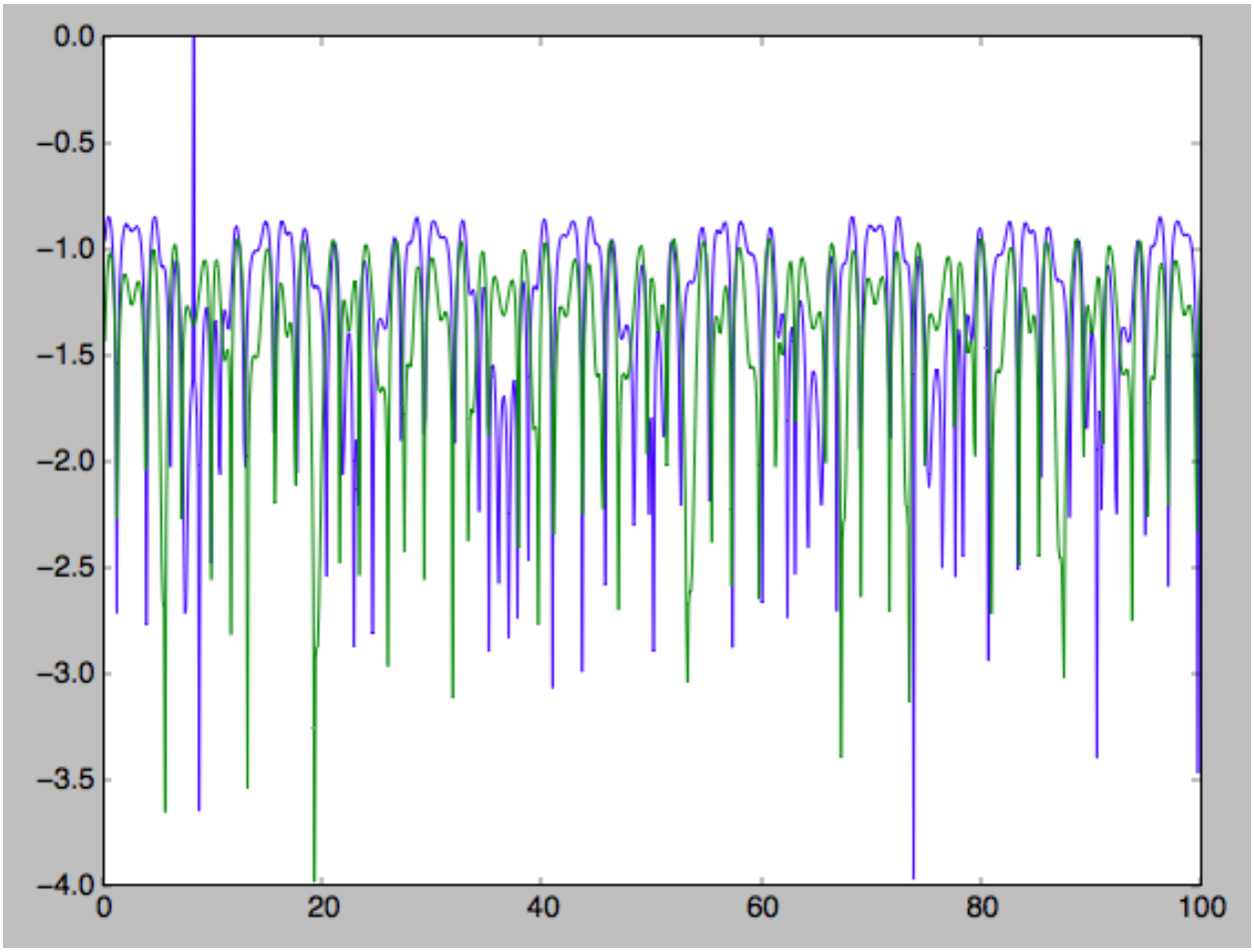
For orbits that reach distances of a kpc and more from the plane, the adiabatic approximation does not work as well. For example,

```
>>> o= Orbit([1.,0.1,1.1,0.,0.25])
>>> o.integrate(ts,MWPotential2014)
>>> o.zmax()*8.
# 1.3506059038621048
```

and we can again calculate the actions along the orbit

```
>>> js= aAA(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts))
>>> plot(ts,numpy.log10(numpy.fabs((js[0]-numpy.mean(js[0]))/numpy.mean(js[0]))))
>>> plot(ts,numpy.log10(numpy.fabs((js[2]-numpy.mean(js[2]))/numpy.mean(js[2]))))
```

which gives



The radial action is now only conserved to about ten percent and the vertical action to approximately five percent.

Warning: Frequencies and angles using the adiabatic approximation are not implemented at this time.

1.7.4 Action-angle coordinates using the Staeckel approximation

A better approximation than the adiabatic one is to locally approximate the potential as a Staeckel potential, for which actions, frequencies, and angles can be calculated through numerical integration. galpy contains an implementation of the algorithm of Binney (2012; 2012MNRAS.426.1324B), which accomplishes the Staeckel approximation for disk-like (i.e., oblate) potentials without explicitly fitting a Staeckel potential. For all intents and purposes the adiabatic approximation is made obsolete by this new method, which is as fast and more precise. The only advantage of the adiabatic approximation over the Staeckel approximation is that the Staeckel approximation requires the user to specify a *focal length* Δ to be used in the Staeckel approximation. However, this focal length can be easily estimated from the second derivatives of the potential (see Sanders 2012; 2012MNRAS.426..128S).

Starting from the second orbit example in the adiabatic section above, we first estimate a good focal length of the `MWPotential2014` to use in the Staeckel approximation. We do this by averaging (through the median) estimates at positions around the orbit (which we integrated in the example above)

```
>>> from galpy.actionAngle import estimateDeltaStaeckel
>>> estimateDeltaStaeckel(MWPotential2014,o.R(ts),o.z(ts))
# 0.40272708556203662
```

We will use $\Delta = 0.4$ in what follows. We set up the `actionAngleStaeckel` object

```
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=False) #c=True is the␣
↪default
```

and calculate the actions

```
>>> aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz())
# (0.019212848866725911, 1.1000000000000001, 0.015274597971510892)
```

The adiabatic approximation from above gives

```
>>> aAA(o.R(),o.vR(),o.vT(),o.z(),o.vz())
# (array([ 0.01686478]), array([ 1.1]), array([ 0.01590001]))
```

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

```
>>> aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz(),fixed_quad=True)
# (0.01922167296633687, 1.1000000000000001, 0.015276825017286706)
```

which in itself leads to a ten times speed up

```
>>> timeit(aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz(),fixed_quad=False))
# 10 loops, best of 3: 129 ms per loop
>>> timeit(aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz(),fixed_quad=True))
# 100 loops, best of 3: 10.3 ms per loop
```

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

```
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=True)
>>> s= numpy.ones(100)
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 10 loops, best of 3: 35.1 ms per loop
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=False) #back to no C
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s,fixed_quad=True))
# 1 loops, best of 3: 496 ms per loop
```

or a fifteen times speed up. The speed up is not that large because the bulge model in `MWPotential2014` requires expensive special functions to be evaluated. Computations could be sped up ten times more when using a simpler bulge model.

Similar to `actionAngleAdiabaticGrid`, we can also tabulate the actions on a grid of (approximate) integrals of the motion and interpolate over this look-up table when evaluating new actions. The details of how this look-up table is setup and used are again fully explained in the galpy paper. To use this grid-based Staeckel approximation, contained in `actionAngleStaeckelGrid`, do

```
>>> from galpy.actionAngle import actionAngleStaeckelGrid
>>> aASG= actionAngleStaeckelGrid(pot=MWPotential2014,delta=0.4,nE=51,npsi=51,nLz=61,
↪c=True)
```

where `c=True` makes sure that we use the C implementation of the Staeckel method to calculate the grid. Because this is a fully three-dimensional grid, setting up the grid takes longer than it does for the adiabatic method (which only uses two two-dimensional grids). We can then evaluate actions as before

```
>>> aAS(o.R(),o.vR(),o.vT(),o.z(),o.vz()), aASG(o.R(),o.vR(),o.vT(),o.z(),o.vz())
# ((0.019212848866725911, 1.1000000000000001, 0.015274597971510892),
#  (0.019221119033345408, 1.1000000000000001, 0.015022528662310393))
```

These actions agree very well. We can compare the timings of these methods as above

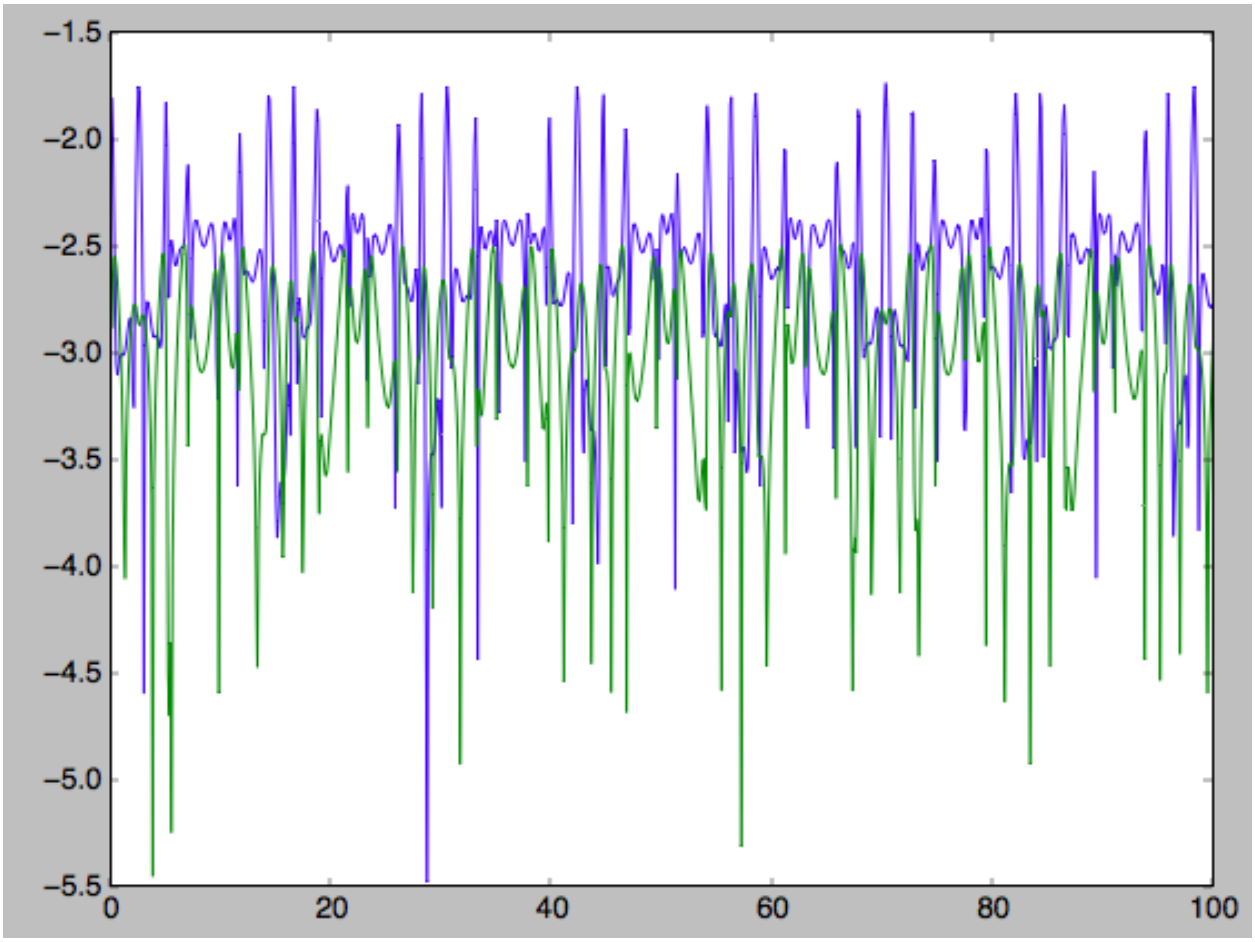
```
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s,fixed_quad=True))
# 1 loops, best of 3: 576 ms per loop # Not using C, direct calculation
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=True)
>>> timeit(aAS(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 100 loops, best of 3: 17.8 ms per loop # Using C, direct calculation
>>> timeit(aASG(1.*s,0.1*s,1.1*s,0.*s,0.05*s))
# 100 loops, best of 3: 3.45 ms per loop # Grid-based calculation
```

This demonstrates that the grid-based interpolation again leeds to a significant speed up, even over the C implementation of the direct calculation. This speed up becomes more significant for larger array input, although it saturates at about 25 times (at least for `MWPotential2014`).

We can now go back to checking that the actions are conserved along the orbit (going back to the `c=False` version of `actionAngleStaeckel`)

```
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=False)
>>> js= aAS(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),fixed_quad=True)
>>> plot(ts,numpy.log10(numpy.fabs((js[0]-numpy.mean(js[0]))/numpy.mean(js[0]))))
>>> plot(ts,numpy.log10(numpy.fabs((js[2]-numpy.mean(js[2]))/numpy.mean(js[2]))))
```

which gives



The radial action is now conserved to better than a percent and the vertical action to only a fraction of a percent. Clearly, this is much better than the five to ten percent errors found for the adiabatic approximation above.

For the Staeckel approximation we can also calculate frequencies and angles through the `actionsFreqs` and `actionsFreqsAngles` methods.

Warning: Frequencies and angles using the Staeckel approximation are *only* implemented in C. So use `c=True` in the setup of the actionAngleStaeckel object.

Warning: Angles using the Staeckel approximation in galpy are such that (a) the radial angle starts at zero at pericenter and increases then going toward apocenter; (b) the vertical angle starts at zero at $z=0$ and increases toward positive zmax. The latter is a different convention from that in Binney (2012), but is consistent with that in actionAngleIsochrone and actionAngleSpherical.

```
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.4,c=True)
>>> o= Orbit([1.,0.1,1.1,0.,0.25,0.]) #need to specify phi for angles
>>> aAS.actionsFreqsAngles(o.R(),o.vR(),o.vT(),o.z(),o.vz(),o.phi())
# (array([ 0.01922167]),
#  array([ 1.1]),
#  array([ 0.01527683]),
#  array([ 1.11317796]),
```

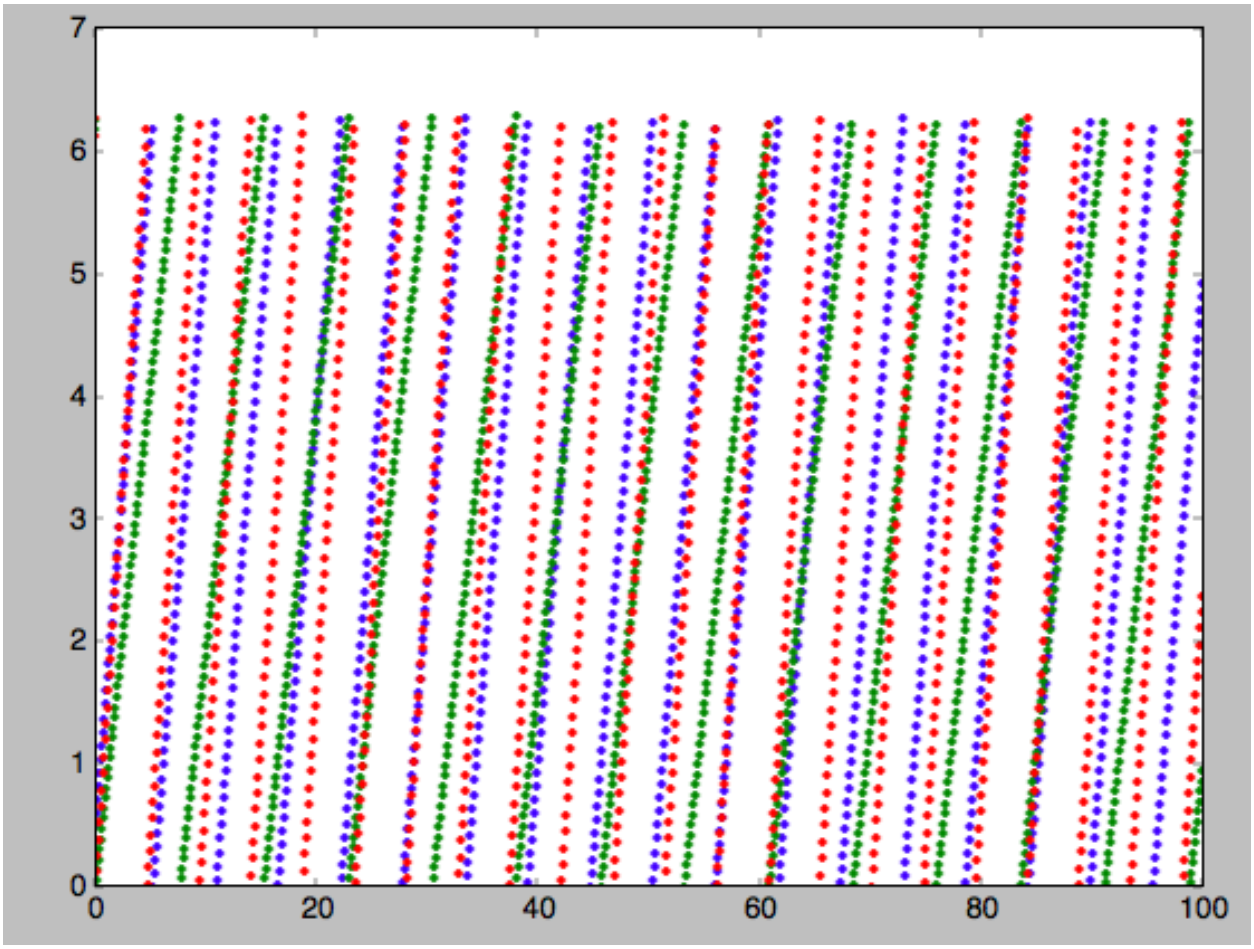
(continues on next page)

(continued from previous page)

```
#  array([ 0.82538032]),
#  array([ 1.34126138]),
#  array([ 0.37758087]),
#  array([ 6.17833493]),
#  array([ 6.13368239]))
```

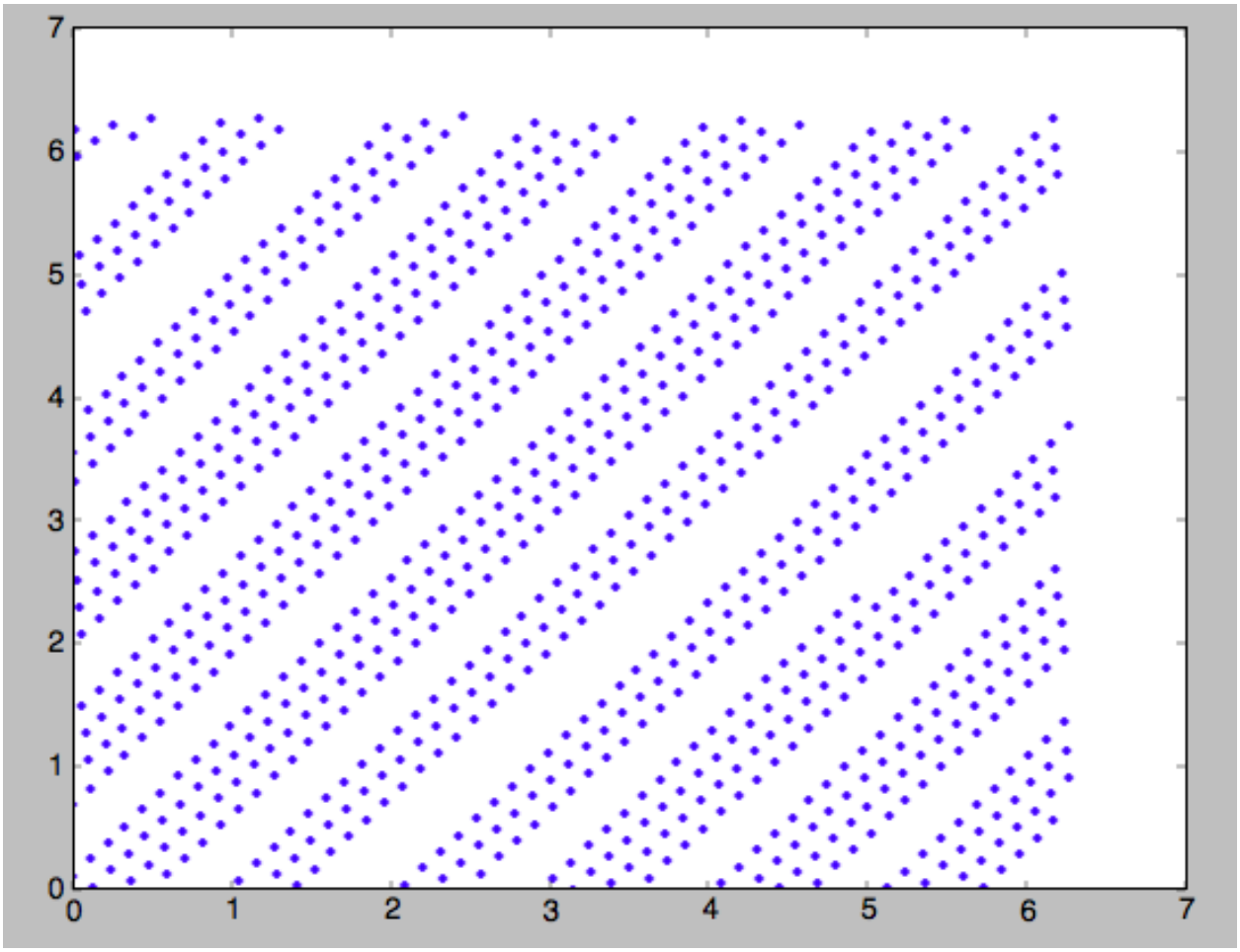
and we can check that the angles increase linearly along the orbit

```
>>> o.integrate(ts,MWPotential2014)
>>> jfa= aAS.actionsFreqsAngles(o.R(ts),o.vR(ts),o.vT(ts),o.z(ts),o.vz(ts),o.phi(ts))
>>> plot(ts,jfa[6],'b.')
>>> plot(ts,jfa[7],'g.')
>>> plot(ts,jfa[8],'r.')
```



or

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



1.7.5 Action-angle coordinates using an orbit-integration-based approximation

The adiabatic and Staeckel approximations used above are good for stars on close-to-circular orbits, but they break down for more eccentric orbits (specifically, orbits for which the radial and/or vertical action is of a similar magnitude as the angular momentum). This is because the approximations made to the potential in these methods (that it is separable in R and z for the adiabatic approximation and that it is close to a Staeckel potential for the Staeckel approximation) break down for such orbits. Unfortunately, these methods cannot be refined to provide better approximations for eccentric orbits.

galpy contains a new method for calculating actions, frequencies, and angles that is completely general for any static potential. It can calculate the actions to any desired precision for any orbit in such potentials. The method works by employing an auxiliary isochrone potential and calculates action-angle variables by arithmetic operations on the actions and angles calculated in the auxiliary potential along an orbit (integrated in the true potential). Full details can be found in Appendix A of Bovy (2014).

We setup this method for a logarithmic potential as follows

```
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.potential import LogarithmicHaloPotential
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

`b=0.8` here sets the scale parameter of the auxiliary isochrone potential; this potential can also be specified as an

IsochronePotential instance through `ip=`). We can now calculate the actions for an orbit similar to that of the GD-1 stream

```
>>> obs= numpy.array([1.56148083,0.35081535,-1.15481504,0.88719443,-0.47713334,0.
↪12019596]) #orbit similar to GD-1
>>> aAIA(*obs)
# (array([ 0.16605011]), array([-1.80322155]), array([ 0.50704439]))
```

An essential requirement of this method is that the angles calculated in the auxiliary potential go through the full range $[0, 2\pi]$. If this is not the case, galpy will raise a warning

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=10.8)
>>> aAIA(*obs)
# galpyWarning: Full radial angle range not covered for at least one object; actions␣
↪are likely not reliable
# (array([ 0.08985167]), array([-1.80322155]), array([ 0.50849276]))
```

Therefore, some care should be taken to choosing a good auxiliary potential. galpy contains a method to estimate a decent scale parameter for the auxiliary scale parameter, which works similar to `estimateDeltaStaeckel` above except that it also gives a minimum and maximum b if multiple R and z are given

```
>>> from galpy.actionAngle import estimateBIsochrone
>>> from galpy.orbit import Orbit
>>> o= Orbit(obs)
>>> ts= numpy.linspace(0.,100.,1001)
>>> o.integrate(ts,lp)
>>> estimateBIsochrone(lp,o.R(ts),o.z(ts))
# (0.78065062339131952, 1.2265541473461612, 1.4899326335155412) #bmin,bmedian,bmax␣
↪over the orbit
```

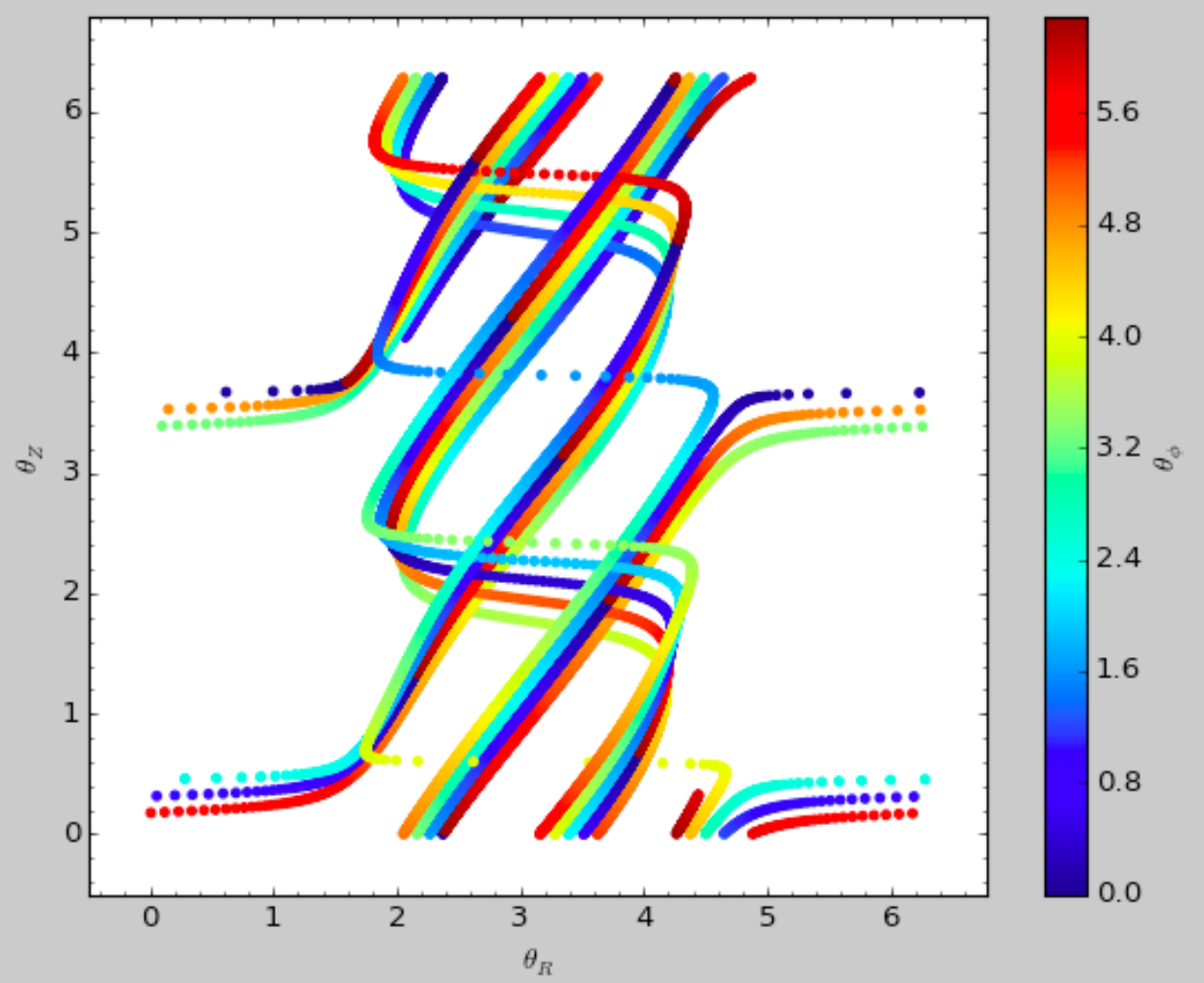
Experience shows that a scale parameter somewhere in the range returned by this function makes sure that the angles go through the full $[0, 2\pi]$ range. However, even if the angles go through the full range, the closer the angles increase to linear, the better the converenge of the algorithm is (and especially, the more accurate the calculation of the frequencies and angles is, see below). For example, for the scale parameter at the upper and of the range

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=1.5)
>>> aAIA(*obs)
# (array([ 0.01120145]), array([-1.80322155]), array([ 0.50788893]))
```

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

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

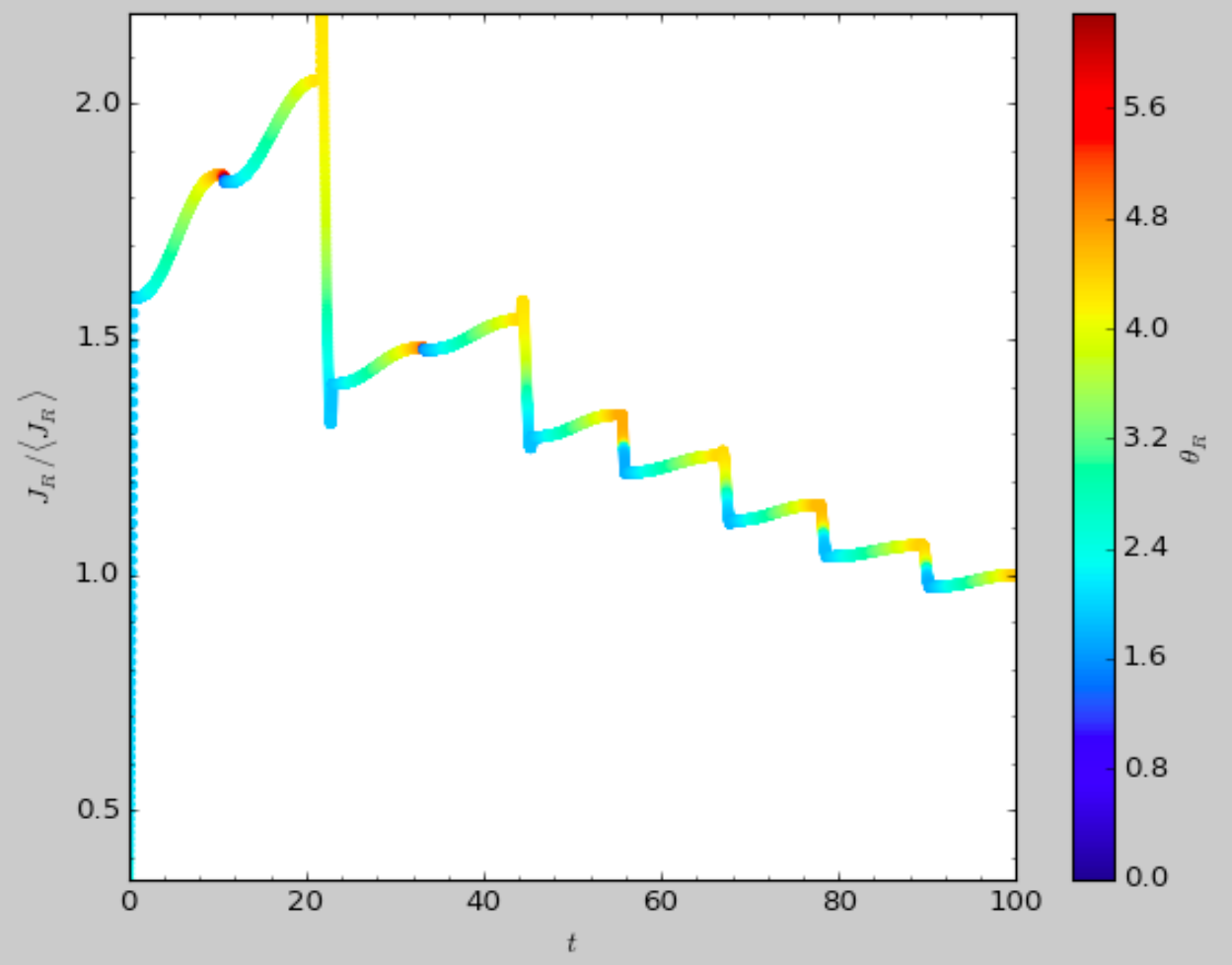
which gives



and this clearly shows that the angles increase *very* non-linearly, because the auxiliary isochrone potential used is too far from the real potential. This causes the actions to converge only very slowly. For example, for the radial action we can plot the converge as a function of integration time

```
>>> aAIA.plot(*obs,type='jr')
```

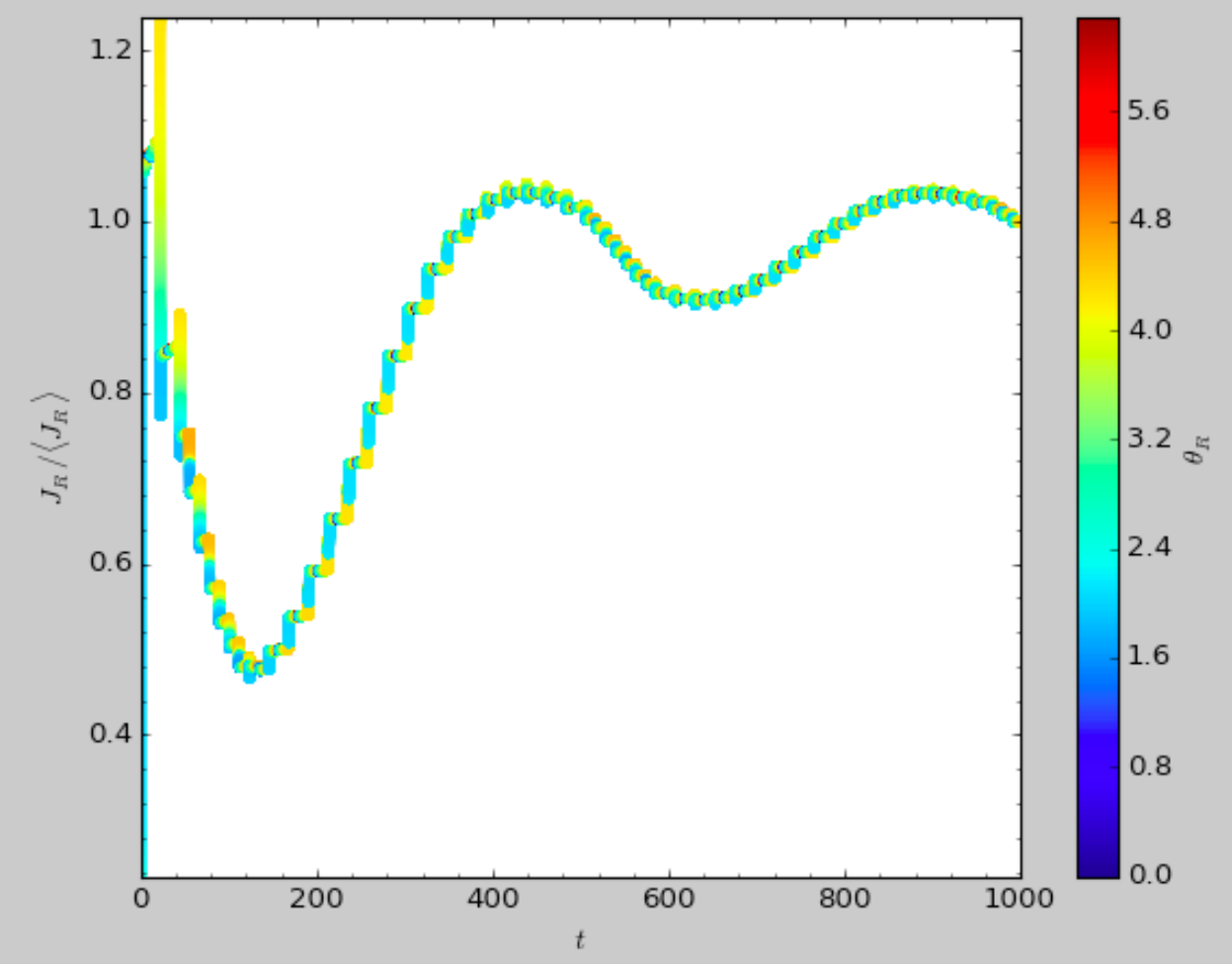
which gives



This Figure clearly shows that the radial action has not converged yet. We need to integrate *much* longer in this auxiliary potential to obtain convergence and because the angles increase so non-linearly, we also need to integrate the orbit much more finely:

```
>>> aAIA= actionAngleIsochroneApprox(pot=lp,b=1.5,tintJ=1000,ntintJ=800000)
>>> aAIA(*obs)
# (array([ 0.01711635]), array([-1.80322155]), array([ 0.51008058]))
>>> aAIA.plot(*obs,type='jr')
```

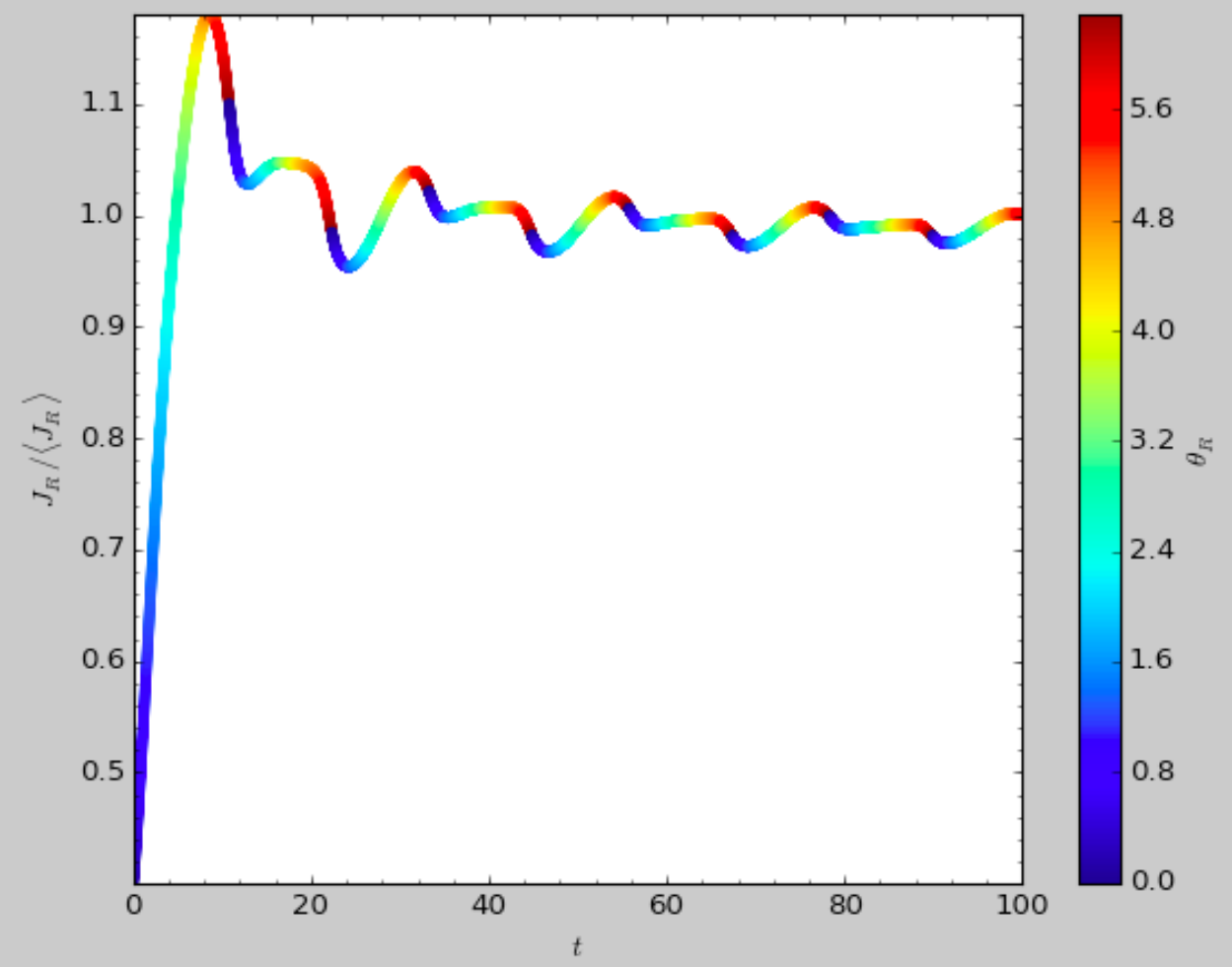
which shows slow convergence



Finding a better auxiliary potential makes convergence *much* faster and also allows the frequencies and the angles to be calculated by removing the small wiggles in the auxiliary angles vs. time (in the angle plot above, the wiggles are much larger, such that removing them is hard). The auxiliary potential used above had `b=0.8`, which shows very quick converenge and good behavior of the angles

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

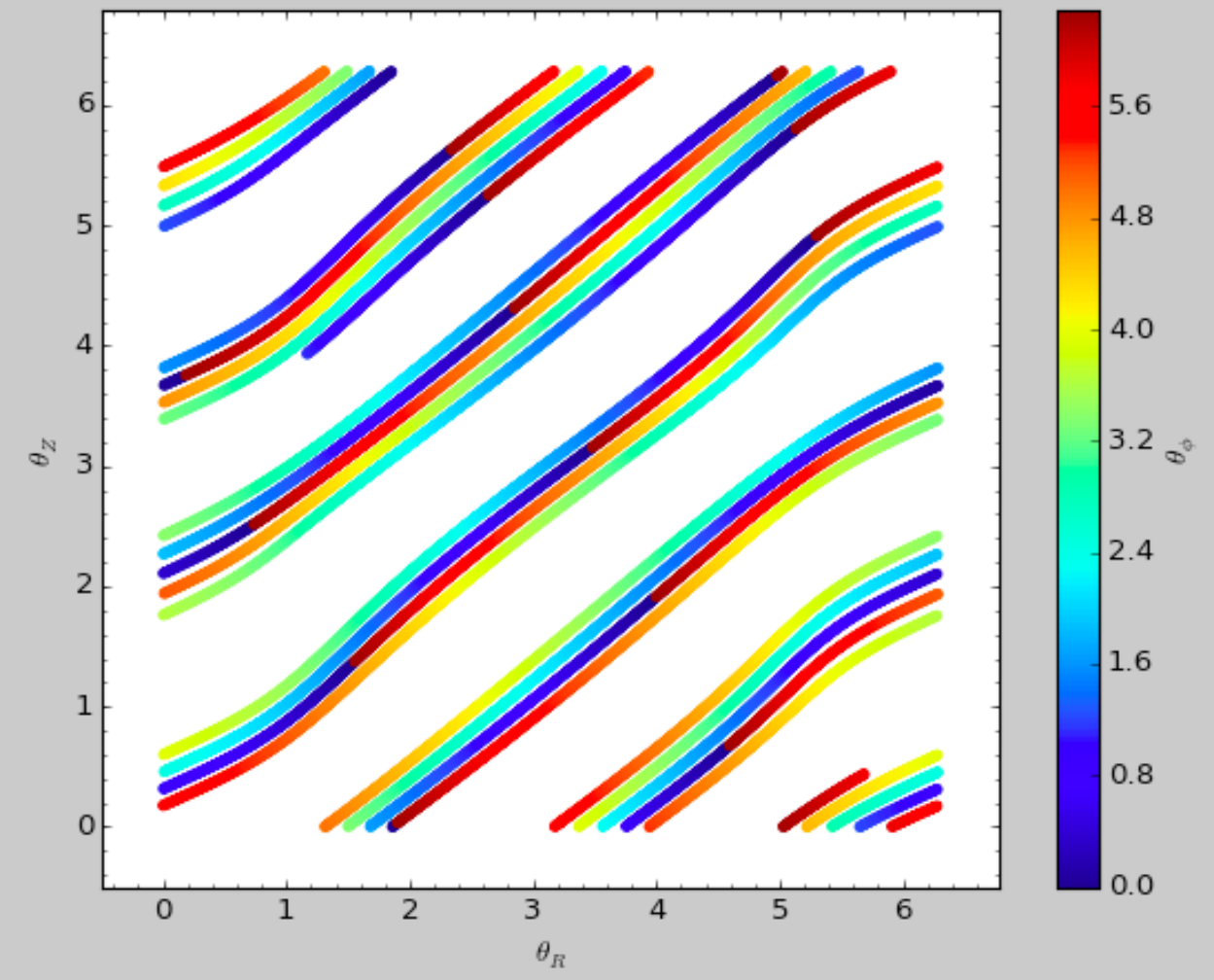
gives



and

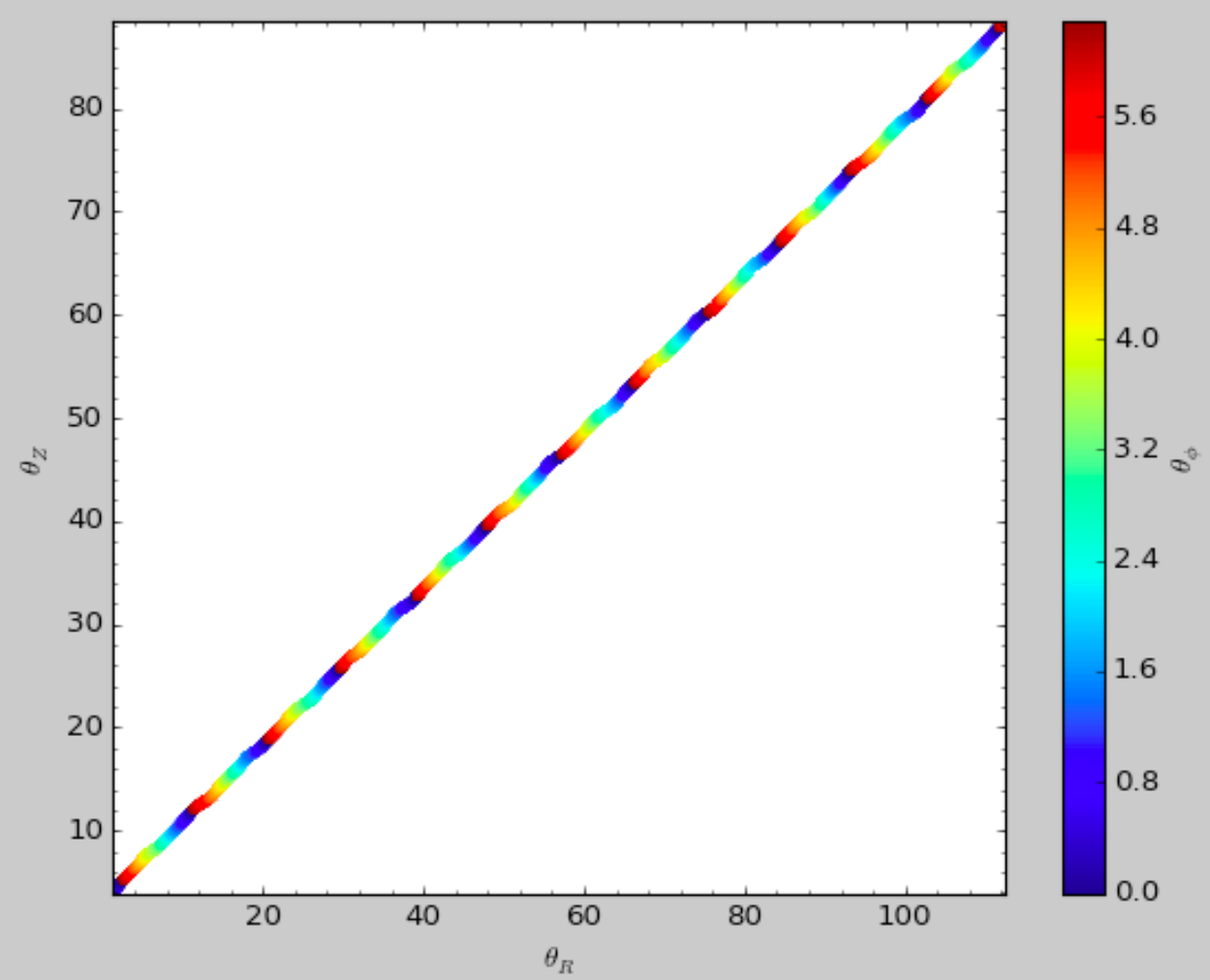
```
>>> aAIA.plot(*obs,type='araz')
```

gives



We can remove the periodic behavior from the angles, which clearly shows that they increase close-to-linear with time

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



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

```
>>> aAIA.actionsFreqsAngles(*obs)
# (array([ 0.16392384]),
#  array([-1.80322155]),
#  array([ 0.50999882]),
#  array([ 0.55808933]),
#  array([-0.38475753]),
#  array([ 0.42199713]),
#  array([ 0.18739688]),
#  array([ 0.3131815]),
#  array([ 2.18425661]))
```

This function takes as an argument `maxn=` the maximum n for which to remove sinusoidal wiggles. So we can raise this, for example to 4 from 3

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

(continues on next page)

(continued from previous page)

```
#  array([ 0.18732009]),
#  array([ 0.31318534]),
#  array([ 2.18421296]))
```

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

This technique also works for triaxial potentials, but using those requires the code to also use the azimuthal angle variable in the auxiliary potential (this is unnecessary in axisymmetric potentials as the z component of the angular momentum is conserved). We can calculate actions for triaxial potentials by specifying that `nonaxi=True`:

```
>>> aAIA(*obs,nonaxi=True)
# (array([ 0.16605011]), array([-1.80322155]), array([ 0.50704439]))
```

1.7.6 Action-angle coordinates using the TorusMapper code

All of the methods described so far allow one to compute the actions, angles, and frequencies for a given phase-space location. `galpy` also contains some support for computing the inverse transformation by using an interface to the TorusMapper code. Currently, this is limited to axisymmetric potentials, because the TorusMapper code is limited to such potentials.

The basic use of this part of `galpy` is to compute an orbit $(R, v_R, v_T, z, v_z, \phi)$ for a given torus, specified by three actions (J_R, L_Z, J_Z) and as many angles along a torus as you want. First we set up an `actionAngleTorus` object

```
>>> from galpy.actionAngle import actionAngleTorus
>>> from galpy.potential import MWPotential2014
>>> aAT= actionAngleTorus(pot=MWPotential2014)
```

To compute an orbit, we first need to compute the frequencies, which we do as follows

```
>>> jr,lz,jz= 0.1,1.1,0.2
>>> Om= aAT.Freqs(jr,lz,jz)
```

This set consists of $(\Omega_R, \Omega_\phi, \Omega_Z, \mathrm{TMerr})$, where the last entry is the exit code of the TorusMapper code (will be printed as a warning when it is non-zero). Then we compute a set of angles that fall along an orbit as $\theta(t) = \theta_0 + \mathbf{\Omega}\, t$ for a set of times t

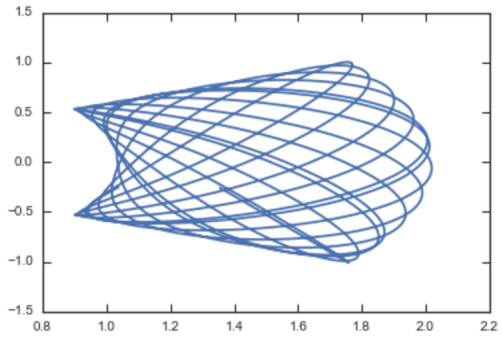
```
>>> times= numpy.linspace(0.,100.,10001)
>>> init_angle= numpy.array([1.,2.,3.])
>>> angles= numpy.tile(init_angle,(len(times),1))+Om[:3]*numpy.tile(times,(3,1)).T
```

Then we can compute the orbit by transforming the orbit in action-angle coordinates to configuration space as follows

```
>>> RvR,_,_,_,_= aAT.xvFreqs(jr,lz,jz,angles[:,0],angles[:,1],angles[:,2])
```

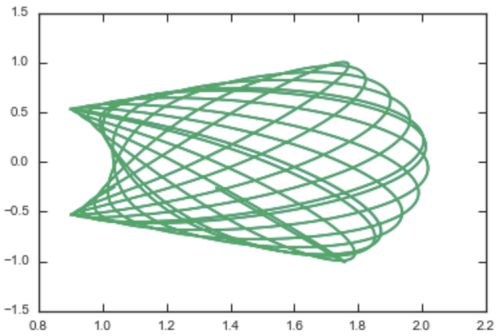
Note that the frequency is also always computed and returned by this method, because it can be obtained at zero cost. The `RvR` array has shape `(ntimes,6)` and the six phase-space coordinates are arranged in the usual `(R,vR,vT,z,vz,phi)` order. The orbit in (R, Z) is then given by

```
>>> plot(RvR[:,0],RvR[:,3])
```



We can compare this to the direct numerical orbit integration. We integrate the orbit, starting at the position and velocity of the initial angle `RvR[0]`

```
>>> from galpy.orbit import Orbit
>>> orb= Orbit(RvR[0])
>>> orb.integrate(times,MWPotential2014)
>>> orb.plot(overplot=True)
```

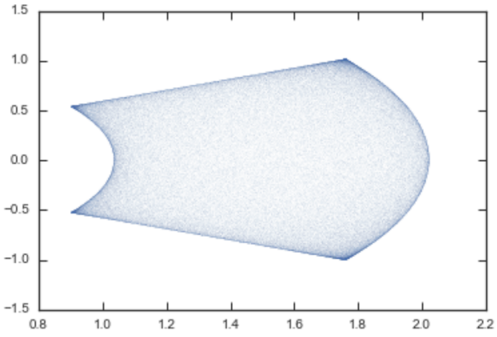


The two orbits are exactly the same.

Of course, we do not have to follow the path of an orbit to map the entire orbital torus and thus reveal the orbital building blocks of galaxies. To directly map a torus, we can do (don't worry, this doesn't take very long)

```
>>> nangles= 200001
>>> angler= numpy.random.uniform(size=nangles)*2.*numpy.pi
>>> anglep= numpy.random.uniform(size=nangles)*2.*numpy.pi
>>> anglez= numpy.random.uniform(size=nangles)*2.*numpy.pi
>>> RvR,_,_,_,_= aAT.xvFreqs(jr,lz,jz,angler,anglep,anglez)
>>> plot(RvR[:,0],RvR[:,3],',',alpha=0.02)
```

which directly shows where the orbit spends most of its time:



`actionAngleTorus` has additional methods documented on the action-angle API page for computing Hessians and Jacobians of the transformation between action-angle and configuration space coordinates.

1.7.7 Accessing action-angle coordinates for Orbit instances

While the 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

```
>>> from galpy.orbit import Orbit
>>> from galpy.potential import MWPotential2014
>>> o= Orbit([1.,0.1,1.1,0.,0.25,0.])
```

and we can then calculate the actions (default is to use the staeckel approximation with an automatically-estimated delta parameter, but this can be adjusted)

```
>>> o.jr(MWPotential2014), o.jp(MWPotential2014), o.jz(MWPotential2014)
# (0.018194068808944613,1.1,0.01540155584446606)
```

`o.jp` here gives the azimuthal action (which is the z component of the angular momentum for axisymmetric potentials). We can also use the other methods described above or adjust the parameters of the approximation (see above):

```
>>> o.jr(MWPotential2014,type='staeckel',delta=0.4), o.jp(MWPotential2014,type=
↪'staeckel',delta=0.4), o.jz(MWPotential2014,type='staeckel',delta=0.4)
# (0.019221672966336707, 1.1, 0.015276825017286827)
>>> o.jr(MWPotential2014,type='adiabatic'), o.jp(MWPotential2014,type='adiabatic'), o.
↪jz(MWPotential2014,type='adiabatic')
# (0.016856430059017123, 1.1, 0.015897730620467752)
>>> o.jr(MWPotential2014,type='isochroneApprox',b=0.8), o.jp(MWPotential2014,type=
↪'isochroneApprox',b=0.8), o.jz(MWPotential2014,type='isochroneApprox',b=0.8)
# (0.019066091295488922, 1.1, 0.015280492319332751)
```

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

```
>>> (o.jr(MWPotential2014,type='staeckel',delta=0.4)-o.jr(MWPotential2014,type=
↪'isochroneApprox',b=0.8))/o.jr(MWPotential2014,type='isochroneApprox',b=0.8)
# 0.00816012408818143
>>> (o.jz(MWPotential2014,type='staeckel',delta=0.4)-o.jz(MWPotential2014,type=
↪'isochroneApprox',b=0.8))/o.jz(MWPotential2014,type='isochroneApprox',b=0.8)
# 0.00023999894566772273
```

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.resetaA() 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

```
>>> o.Or(MWPotential2014,type='staeckel',delta=0.4)
# 1.1131779637307115
>>> o.Or(MWPotential2014,type='isochroneApprox',b=0.8)
# 1.1134635974560649
```

and the radial angle

```
>>> o.wr(MWPotential2014,type='staeckel',delta=0.4)
# 0.37758086786371969
>>> o.wr(MWPotential2014,type='isochroneApprox',b=0.8)
# 0.38159809018175395
```

which again agree to 1%. We can also calculate the other frequencies, angles, as well as periods using the functions `o.Op`, `o.oz`, `o.wp`, `o.wz`, `o.Tr`, `o.Tp`, `o.Tz`.

1.7.8 Example: Evidence for a Lindblad resonance in the Solar neighborhood

We can use galpy to calculate action-angle coordinates for a set of stars in the Solar neighborhood and look for unexplained features. For this we download the data from the Geneva-Copenhagen Survey (2009A&A...501..941H; data available at viZier). Since the velocities in this catalog are given as U,V, and W, we use the `radec` and `UVW` keywords to initialize the orbits from the raw data. For each object `ii`

```
>>> o= Orbit(vxvv[ii,:],radec=True,uvw=True,vo=220.,ro=8.)
```

We then calculate the actions and angles for each object in a flat rotation curve potential

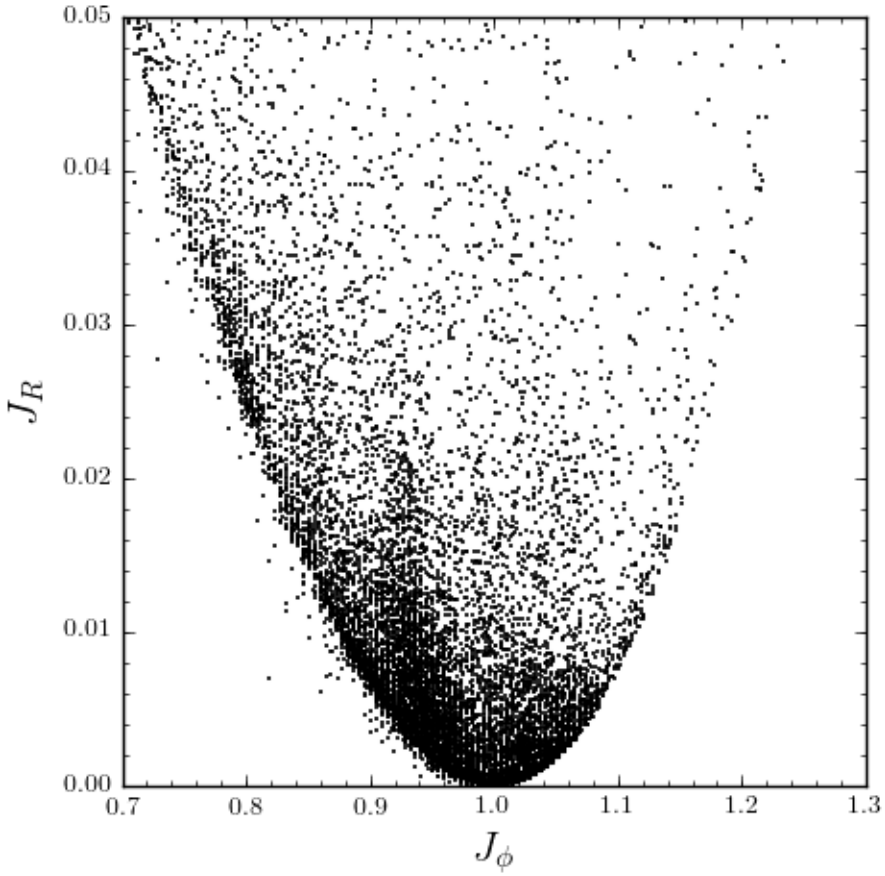
```
>>> lp= LogarithmicHaloPotential(normalize=1.)
>>> myjr[ii]= o.jr(lp)
```

etc.

Plotting the radial action versus the angular momentum

```
>>> plot.bovy_plot(myjp,myjr,'k.',ms=2.,xlabel=r'$J_{\phi}$',ylabel=r'$J_R$',
↪xrange=[0.7,1.3],yrange=[0.,0.05])
```

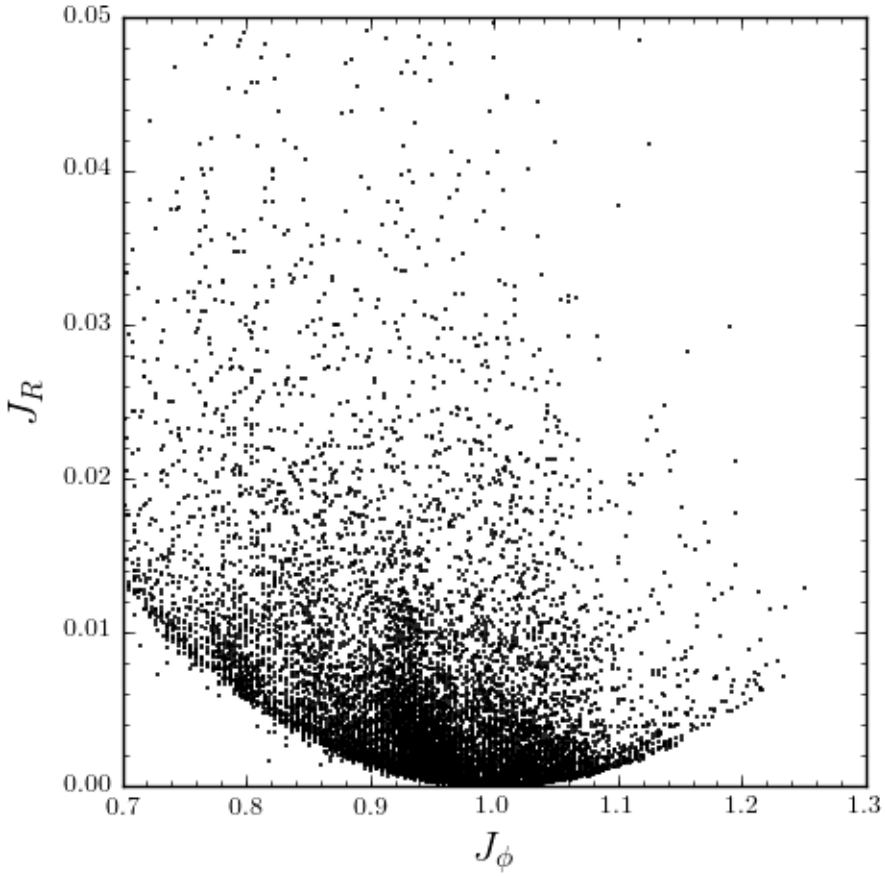
shows a feature in the distribution



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

```
>>> pp= PowerSphericalPotential(normalize=1.,alpha=-2.)
>>> myjr[ii]= o.jr(pp)
```

We find that the distribution is stretched, but the feature remains



Code for this example can be found `here` (note that this code uses a particular download of the GCS data set; if you use your own version, you will need to modify the part of the code that reads the data). For more information see 2010MNRAS.409..145S.

1.7.9 Example: actions in an N-body simulation

To illustrate how we can use `galpy` to calculate actions in a snapshot of an N-body simulation, we again look at the `g15784` snapshot in the `pynbody` test suite, discussed in *The potential of N-body simulations*. Please look at that section for information on how to setup the potential of this snapshot in `galpy`. One change is that we should set `enable_c=True` in the instantiation of the `InterpSnapshotRZPotential` object

```
>>> spi= InterpSnapshotRZPotential(h1,rgrid=(numpy.log(0.01),numpy.log(20.),101),
↪logR=True,zgrid=(0.,10.,101),interpPot=True,zsym=True,enable_c=True)
>>> spi.normalize(R0=10.)
```

where we again normalize the potential to use galpy's *natural units*.

We first load a pristine copy of the simulation (because the normalization above leads to some inconsistent behavior in pynbody)

```
>>> sc = pynbody.load('Repos/pynbody-testdata/g15784.lr.01024.gz'); hc = sc.halos();␣
↪hc1= hc[1]; pynbody.analysis.halo.center(hc1,mode='hyb'); pynbody.analysis.angmom.
↪faceon(hc1, cen=(0,0,0),mode='ssc'); sc.physical_units()
```

and then select particles near R=*8* kpc by doing

```
>>> sn= pynbody.filt.BandPass('rxy','7 kpc','9 kpc')
>>> R,vR,vT,z,vz = [numpy.ascontiguousarray(hc1.s[sn][x]) for x in ('rxy','vr','vt','z
↪','vz')]
```

These have physical units, so we normalize them (the velocity normalization is the circular velocity at R=*10* kpc, see *here*).

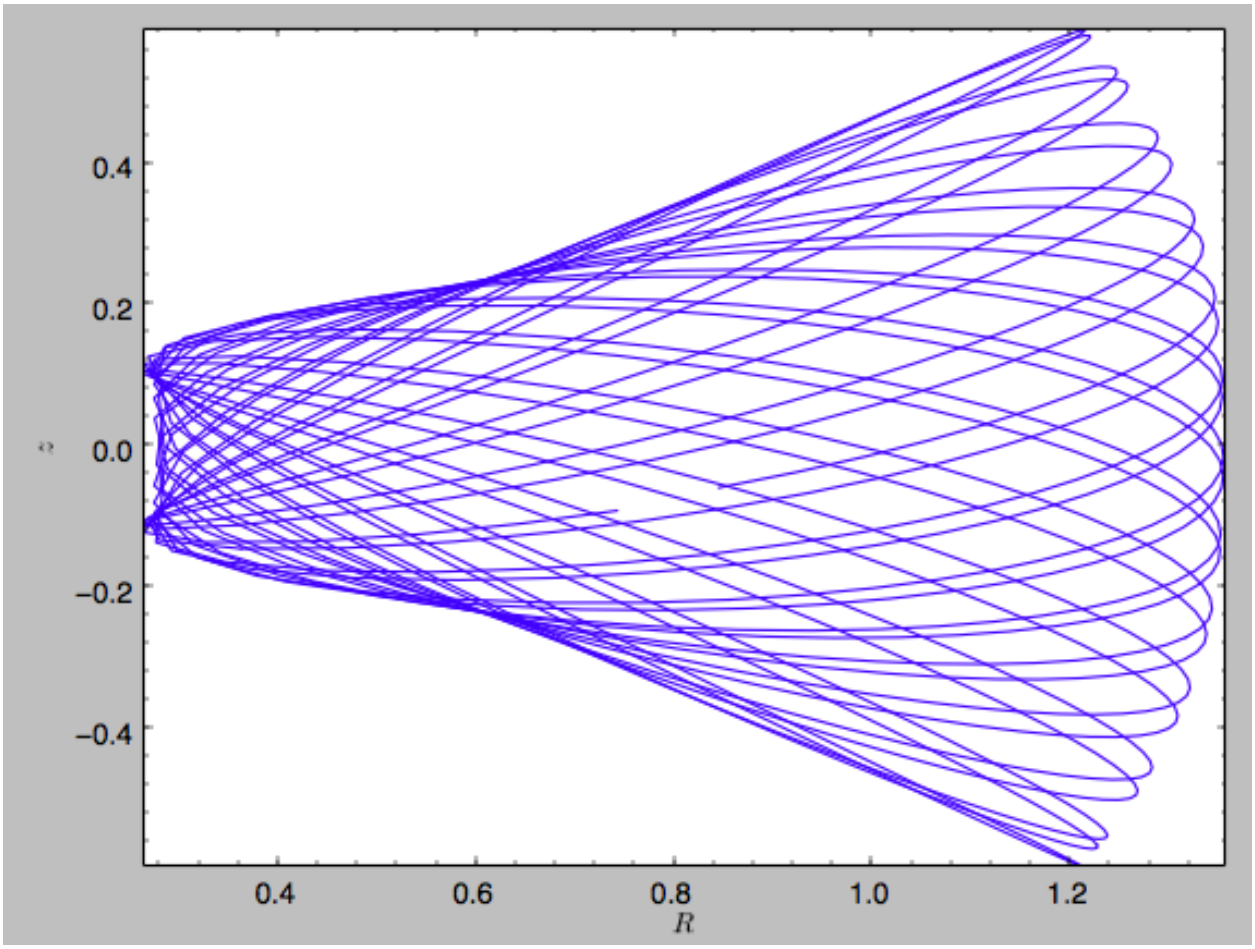
```
>>> ro, vo= 10., 294.62723076942245
>>> R/= ro
>>> z/= ro
>>> vR/= vo
>>> vT/= vo
>>> vz/= vo
```

We will calculate actions using `actionAngleStaeckel` above. We can first integrate a random orbit in this potential

```
>>> from galpy.orbit import Orbit
>>> numpy.random.seed(1)
>>> ii= numpy.random.permutation(len(R))[0]
>>> o= Orbit([R[ii],vR[ii],vT[ii],z[ii],vz[ii]])
>>> ts= numpy.linspace(0.,100.,1001)
>>> o.integrate(ts,spi)
```

This orbit looks like this

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



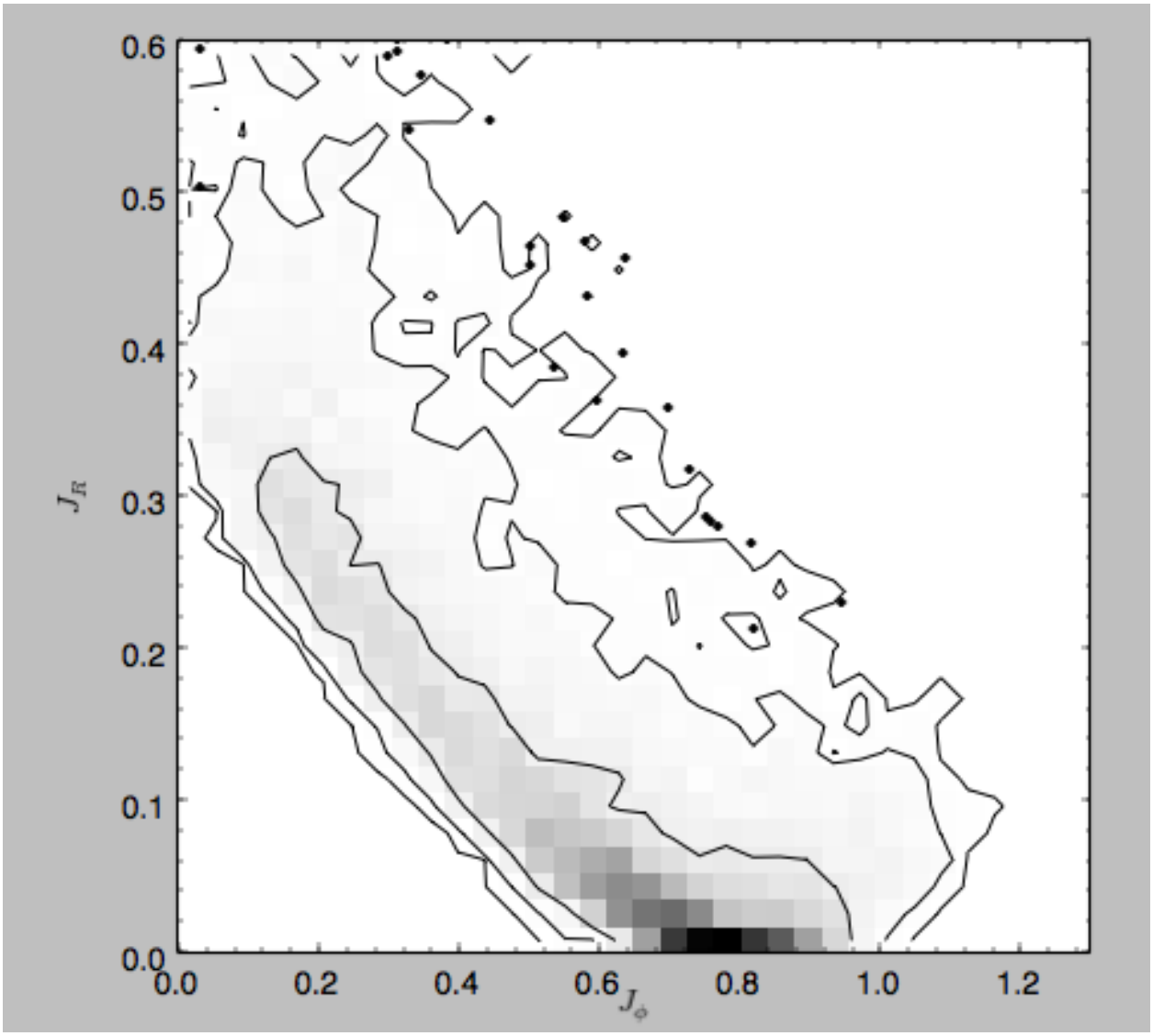
We can now calculate the actions by doing

```
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=spi,delta=0.45,c=True)
>>> jr,lz,jz= aAS(R,vR,vT,z,vz)
```

These actions are also in *natural units*; you can obtain physical units by multiplying with *ro*vo*. We can now plot these actions

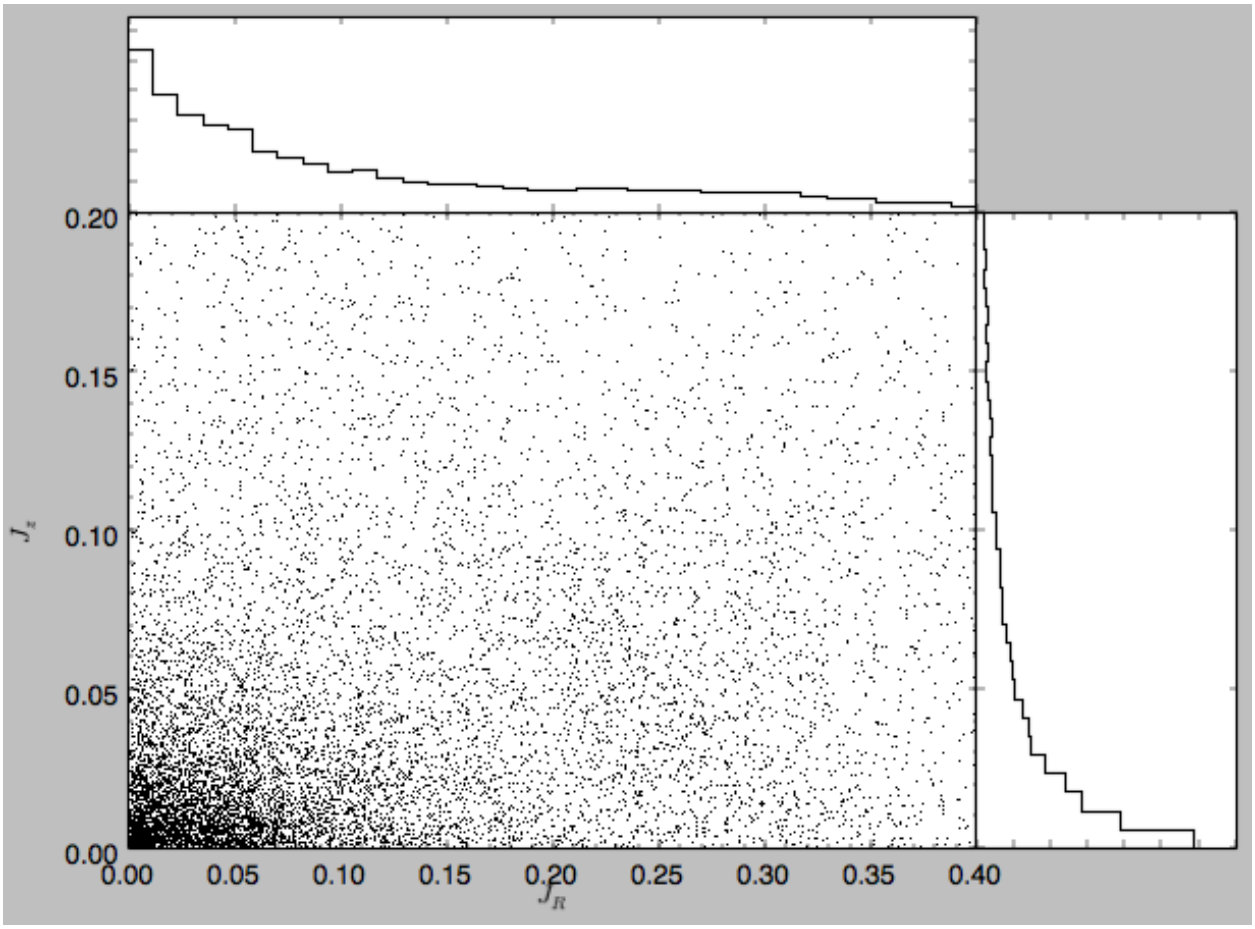
```
>>> from galpy.util import bovy_plot
>>> bovy_plot.scatterplot(lz,jr,'k.',xlabel=r'$J_\phi$',ylabel=r'$J_R$',xrange=[0.,1.
↪3],yrange=[0.,.6])
```

which gives



Note the similarity between this figure and the GCS figure above. The curve shape is due to the selection (low angular momentum stars can only enter the selected radial ring if they are very elliptical and therefore have large radial action) and the density gradient in angular momentum is due to the falling surface density of the disk. We can also look at the distribution of radial and vertical actions.

```
>>> bovy_plot.bovy_plot(jr,jz,'k,',xlabel=r'$J_R$',ylabel=r'$J_z$',xrange=[0.,.4],
↪yrange=[0.,0.2],onedhists=True)
```



With the other methods in the actionAngle module we can also calculate frequencies and angles.

1.8 Three-dimensional disk distribution functions

galpy contains a fully three-dimensional disk distribution: `galpy.df.quasiisothermaldf`, which is an approximately isothermal distribution function expressed in terms of action–angle variables (see 2010MNRAS.401.2318B and 2011MNRAS.413.1889B). Recent research shows that this distribution function provides a good model for the DF of mono-abundance sub-populations (MAPs) of the Milky Way disk (see 2013MNRAS.434..652T and 2013ApJ...779..115B). This distribution function family requires action-angle coordinates to evaluate the DF, so `galpy.df.quasiisothermaldf` makes heavy use of the routines in `galpy.actionAngle` (in particular those in `galpy.actionAngleAdiabatic` and `galpy.actionAngle.actionAngleStaeckel`).

1.8.1 Setting up the DF and basic properties

The quasi-isothermal DF is defined by a gravitational potential and a set of parameters describing the radial surface-density profile and the radial and vertical velocity dispersion as a function of radius. In addition, we have to provide an instance of a `galpy.actionAngle` class to calculate the actions for a given position and velocity. For example, for a `galpy.potential.MWPotential2014` potential using the adiabatic approximation for the actions, we import and define the following

```
>>> from galpy.potential import MWPotential2014
>>> from galpy.actionAngle import actionAngleAdiabatic
>>> from galpy.df import quasiisothermaldf
>>> aA= actionAngleAdiabatic(pot=MWPotential2014,c=True)
```

and then setup the `quasiisothermaldf` instance

```
>>> qdf= quasiisothermaldf(1./3.,0.2,0.1,1.,1.,pot=MWPotential2014,aA=aA,
↪cutcounter=True)
```

which sets up a DF instance with a radial scale length of $R_0/3$, a local radial and vertical velocity dispersion of $0.2\,V_c(R_0)$ and $0.1\,V_c(R_0)$, respectively, and a radial scale lengths of the velocity dispersions of R_0. `cutcounter=True` specifies that counter-rotating stars are explicitly excluded (normally these are just exponentially suppressed). As for the two-dimensional disk DFs, these parameters are merely input (or target) parameters; the true density and velocity dispersion profiles calculated by evaluating the relevant moments of the DF (see below) are not exactly exponential and have scale lengths and local normalizations that deviate slightly from these input parameters. We can estimate the DF's actual radial scale length near R_0 as

```
>>> qdf.estimate_hr(1.)
# 0.32908034635647182
```

which is quite close to the input value of 1/3. Similarly, we can estimate the scale lengths of the dispersions

```
>>> qdf.estimate_hsr(1.)
# 1.1913935820372923
>>> qdf.estimate_hsz(1.)
# 1.0506918075359255
```

The vertical profile is fully specified by the velocity dispersions and radial density / dispersion profiles under the assumption of dynamical equilibrium. We can estimate the scale height of this DF at a given radius and height as follows

```
>>> qdf.estimate_hz(1.,0.125)
# 0.021389597757156088
```

Near the mid-plane this vertical scale height becomes very large because the vertical profile flattens, e.g.,

```
>>> qdf.estimate_hz(1.,0.125/100.)
# 1.006386030587223
```

or even

```
>>> qdf.estimate_hz(1.,0.)
# 187649.98447377066
```

which is basically infinity.

1.8.2 Evaluating moments

We can evaluate various moments of the DF giving the density, mean velocities, and velocity dispersions. For example, the mean radial velocity is again everywhere zero because the potential and the DF are axisymmetric

```
>>> qdf.meanvR(1.,0.)
# 0.0
```

Likewise, the mean vertical velocity is everywhere zero

```
>>> qdf.meanvz(1.,0.)
# 0.0
```

The mean rotational velocity has a more interesting dependence on position. Near the plane, this is the same as that calculated for a similar two-dimensional disk DF (see *Evaluating moments of the DF*)

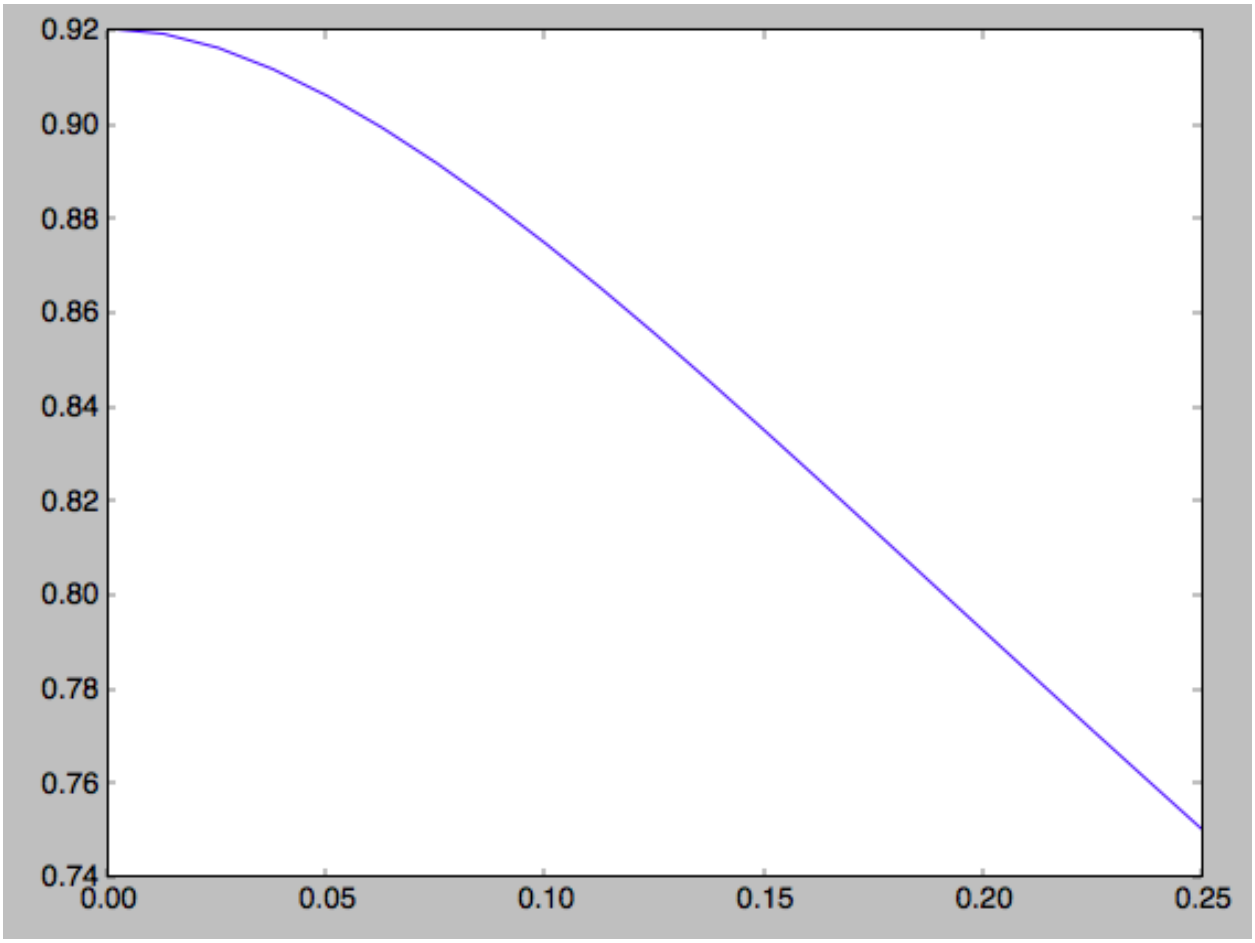
```
>>> qdf.meanvT(1.,0.)
# 0.91988346380781227
```

However, this value decreases as one moves further from the plane. The `quasiisothermaldf` allows us to calculate the average rotational velocity as a function of height above the plane. For example,

```
>>> zs= numpy.linspace(0.,0.25,21)
>>> mvts= numpy.array([qdf.meanvT(1.,z) for z in zs])
```

which gives

```
>>> plot(zs,mvts)
```



We can also calculate the second moments of the DF. We can check whether the radial and velocity dispersions at R_0 are close to their input values

```
>>> numpy.sqrt(qdf.sigmaR2(1.,0.))
# 0.20807112565801389
```

(continues on next page)

(continued from previous page)

```
>>> numpy.sqrt(qdf.sigmaz2(1.,0.))
# 0.090453510526130904
```

and they are pretty close. We can also calculate the mixed R and z moment, for example,

```
>>> qdf.sigmaRz(1.,0.125)
# 0.0
```

or expressed as an angle (the *tilt of the velocity ellipsoid*)

```
>>> qdf.tilt(1.,0.125)
# 0.0
```

This tilt is zero because we are using the adiabatic approximation. As this approximation assumes that the motions in the plane are decoupled from the vertical motions of stars, the mixed moment is zero. However, this approximation is invalid for stars that go far above the plane. By using the Staeckel approximation to calculate the actions, we can model this coupling better. Setting up a `quasiisothermaldf` instance with the Staeckel approximation

```
>>> from galpy.actionAngle import actionAngleStaeckel
>>> aAS= actionAngleStaeckel(pot=MWPotential2014,delta=0.45,c=True)
>>> qdfS= quasiisothermaldf(1./3.,0.2,0.1,1.,1.,pot=MWPotential2014,aA=aAS,
↪cutcounter=True)
```

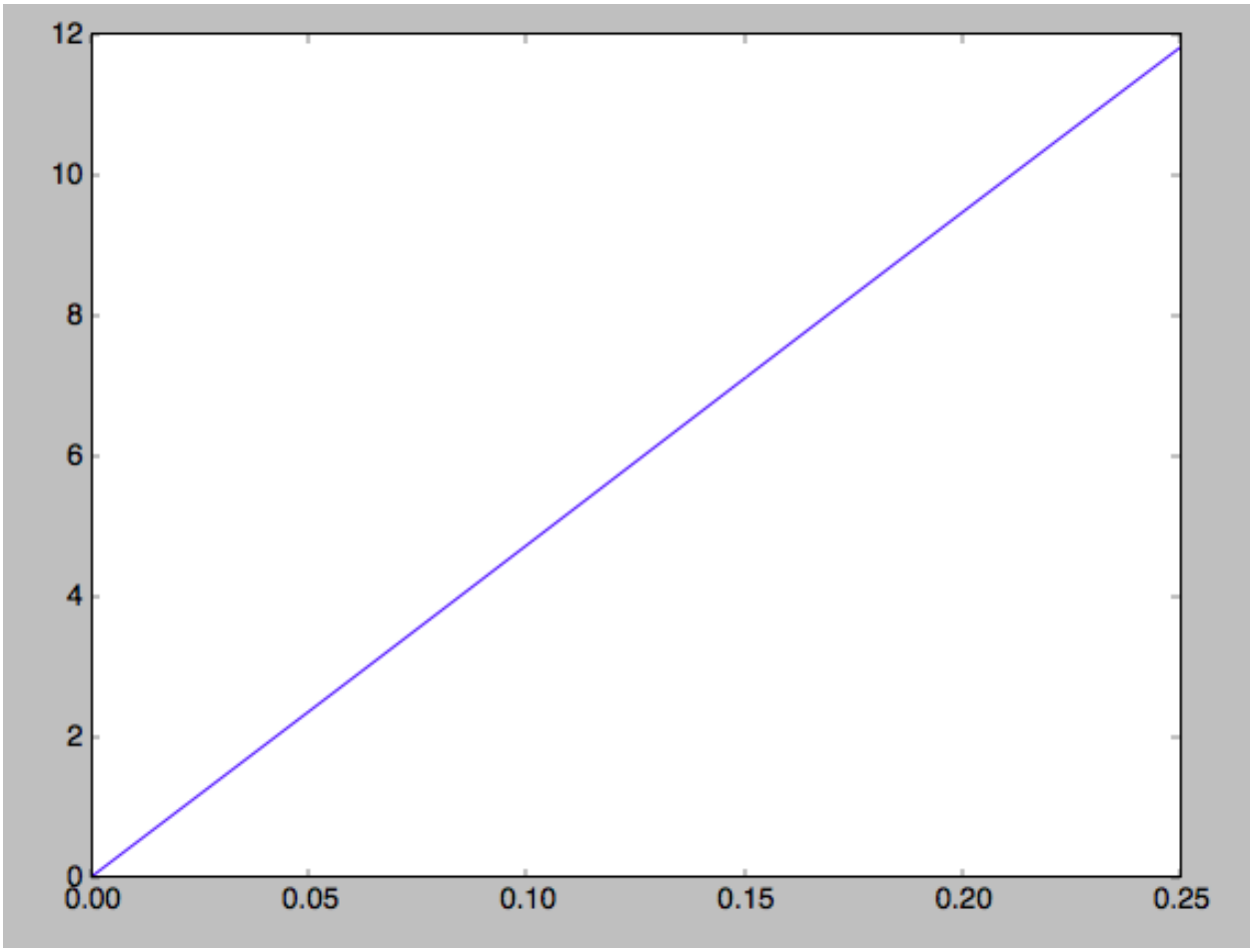
we can similarly calculate the tilt

```
>>> qdfS.tilt(1.,0.125)
# 0.10314272868452541
```

or about 5 degrees (the returned value has units of rad). As a function of height, we find

```
>>> tilts= numpy.array([qdfS.tilt(1.,z) for z in zs])
>>> plot(zs,tilts*180./numpy.pi)
```

which gives



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

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

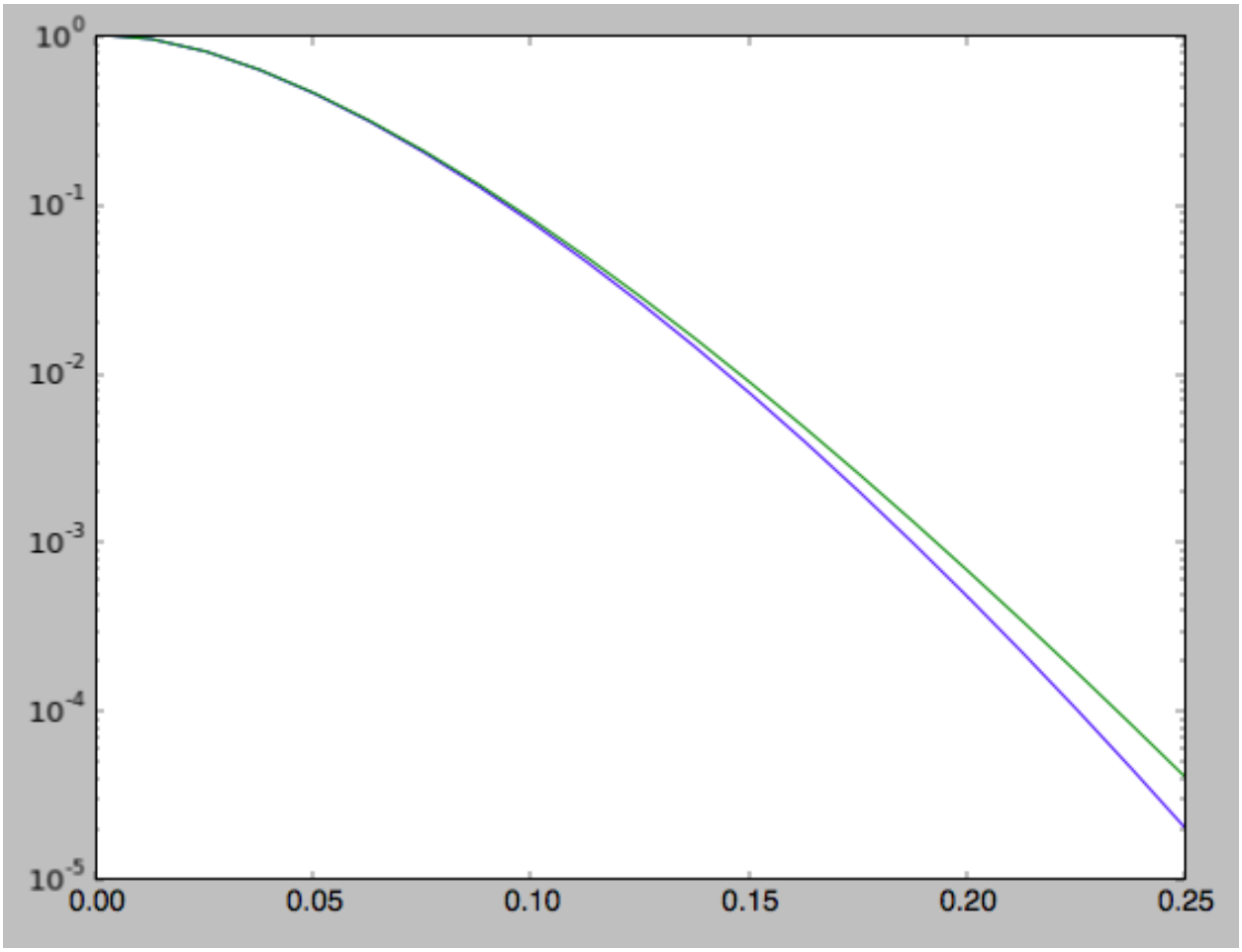
and

```
>>> denszS= numpy.array([qdfS.density(1.,z) for z in zs])
```

We can compare the vertical profiles calculated using the adiabatic and Staeckel action-angle approximations

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

which gives



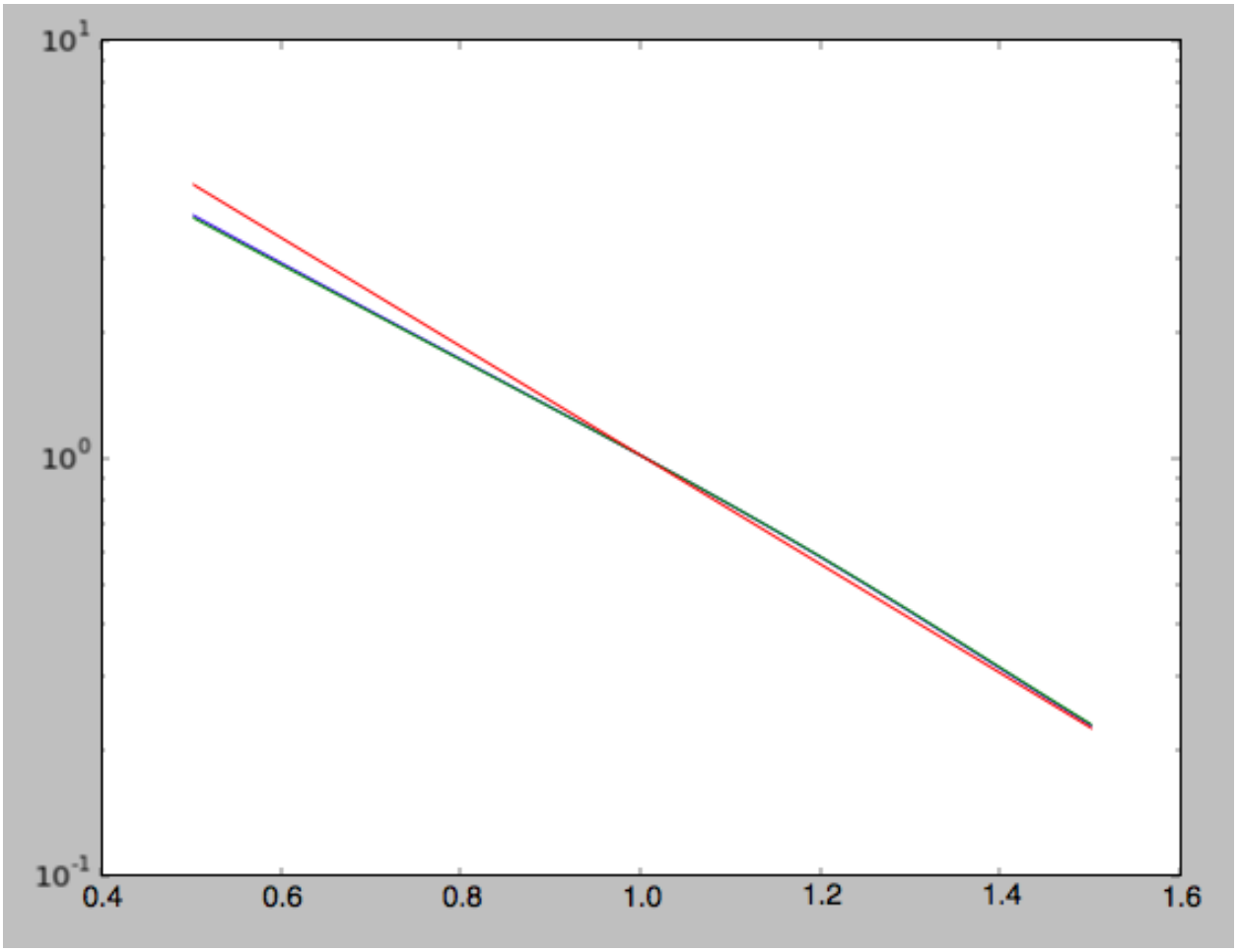
Similarly, we can calculate the radial profile of the surface density

```
>>> rs= numpy.linspace(0.5,1.5,21)
>>> surfr= numpy.array([qdf.surfacemass_z(r) for r in rs])
>>> surfrS= numpy.array([qdfS.surfacemass_z(r) for r in rs])
```

and compare them with each other and an exponential with scale length 1/3

```
>>> semilogy(rs,surfr/surfr[10])
>>> semilogy(rs,surfrS/surfrS[10])
>>> semilogy(rs,numpy.exp(-(rs-1.)/(1./3.)))
```

which gives



The two radial profiles are almost indistinguishable and are very close, if somewhat shallower, than the pure exponential profile.

General velocity moments, including all higher order moments, are implemented in `quasiisothermaldf.vmomentdensity`.

1.8.3 Evaluating and sampling the full probability distribution function

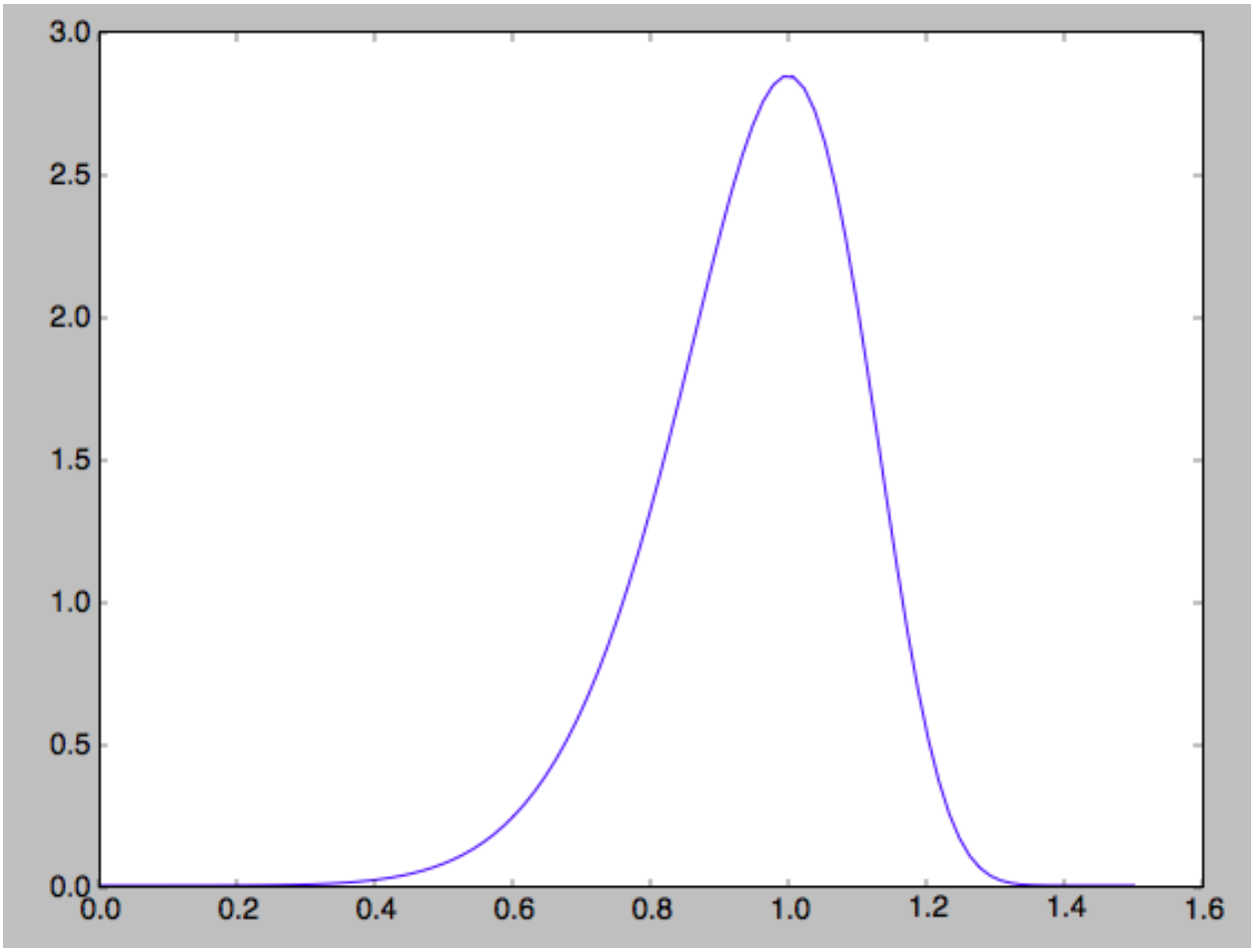
We can evaluate the distribution itself by calling the object, e.g.,

```
>>> qdf(1.,0.1,1.1,0.1,0.) #input: R,vR,vT,z,vz
# array([ 16.86790643])
```

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

```
>>> vts= numpy.linspace(0.,1.5,101)
>>> pvt= numpy.array([qdfS(1.,0.,vt,0.,0.) for vt in vts])
>>> plot(vts,pvt/numpy.sum(pvt)/(vts[1]-vts[0]))
```

which gives



This is, however, not the true distribution of rotational velocities at R =0 and z =0, because it is conditioned on zero radial and vertical velocities. We can calculate the distribution of rotational velocities marginalized over the radial and vertical velocities as

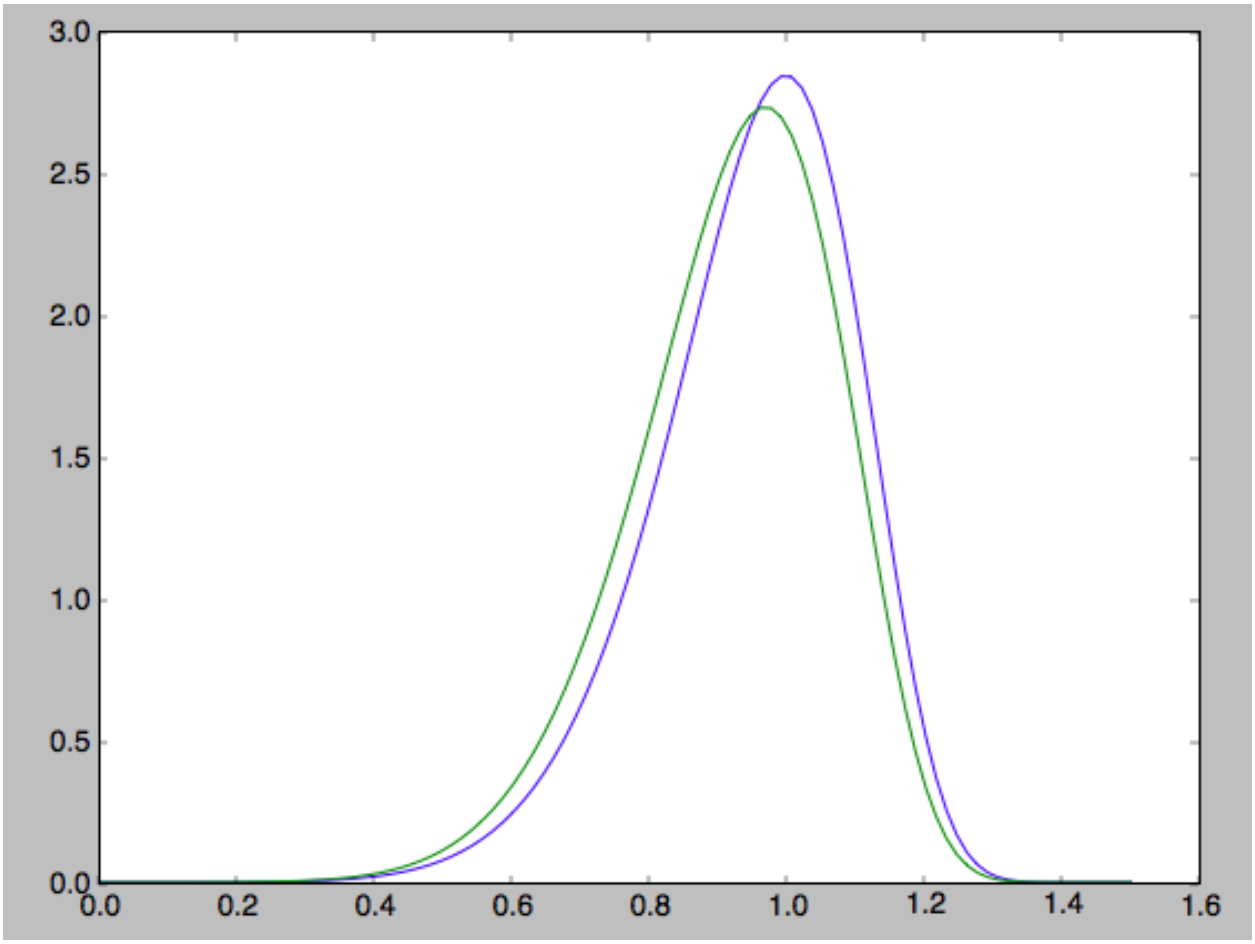
```
>>> qdfS.pvT(1.,1.,0.) #input vT,R,z
# 14.677231196899195
```

or as a function of rotational velocity

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

overplotting this over the previous distribution gives

```
>>> plot(vts,pvt/numpy.sum(pvt)/(vts[1]-vts[0]))
```

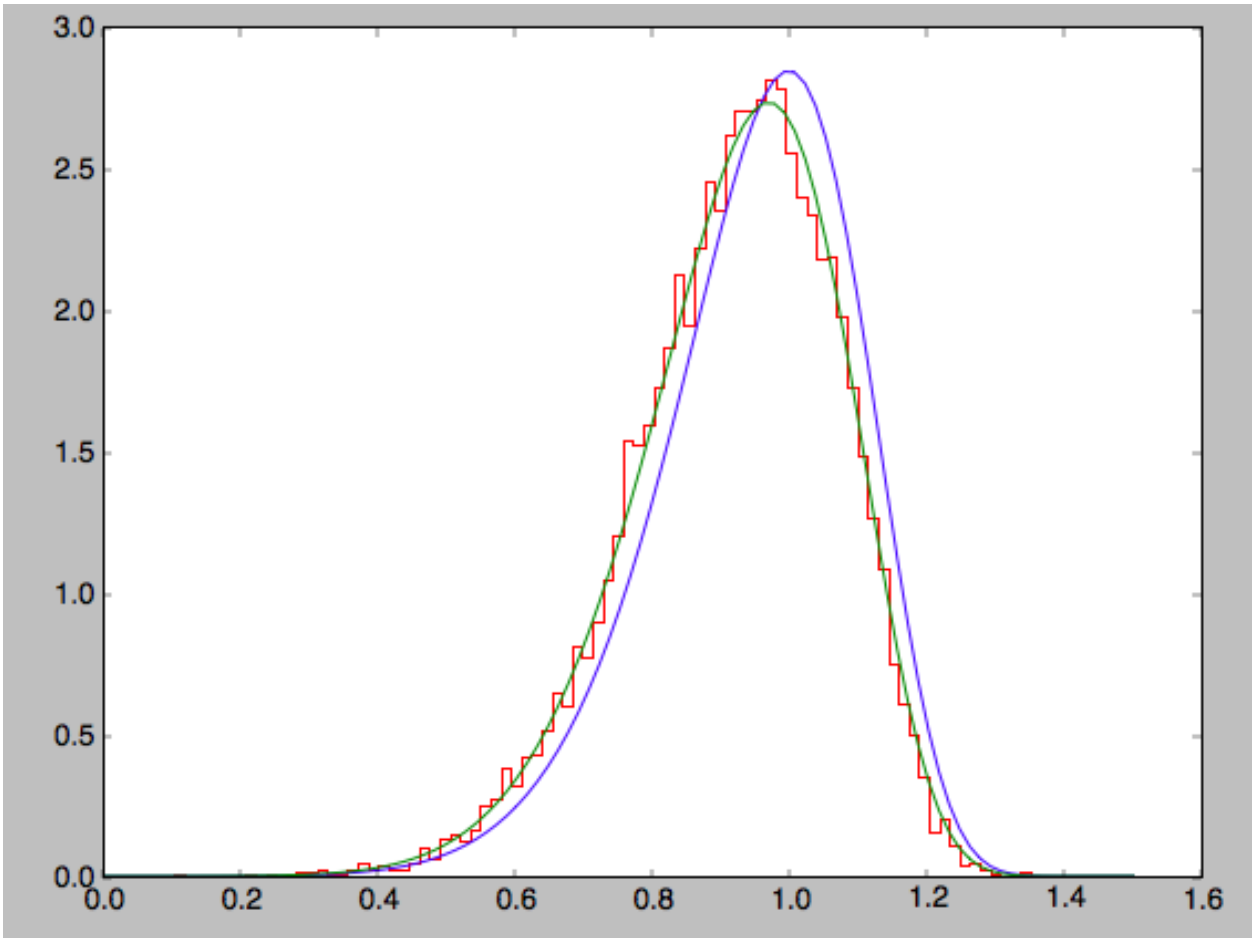


which is slightly different from the conditioned distribution. Similarly, we can calculate marginalized velocity probabilities `pvR, pvz, pvRvT, pvRvz, and pvTvz. These are all multiplied with the density, such that marginalizing these over the remaining velocity component results in the density.

We can sample velocities at a given location using `quasiisothermaldf.sampleV` (there is currently no support for sampling locations from the density profile, although that is rather trivial):

```
>>> vs= qdfS.sampleV(1.,0.,n=10000)
>>> hist(vs[:,1],normed=True,histtype='step',bins=101,range=[0.,1.5])
```

gives



which shows very good agreement with the green (marginalized over vR and vz) curve (as it should).

CHAPTER 2

Tutorials

2.1 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; 2014ApJ...795...95B). This movie shows the disruption of a cluster on a GD-1-like orbit around the Milky Way:

The blue line is the orbit of the progenitor cluster and the black points are cluster members. The disruption is shown in an approximate orbital plane and the movie is comoving with the progenitor cluster.

Streams can be represented by simple dynamical models in action-angle coordinates. In action-angle coordinates, stream members are stripped from the progenitor cluster onto orbits specified by a set of actions (J_R, J_ϕ, J_Z), which remain constant after the stars have been stripped. This is shown in the following movie, which shows the generation of the stream in action space

The color-coding gives the angular momentum J_ϕ and the black dot shows the progenitor orbit. These actions were calculated using `galpy.actionAngle.actionAngleIsochroneApprox`. The points move slightly because of small errors in the action calculation (these are correlated, so the cloud of points moves coherently because of calculation errors). The same movie that also shows the actions of stars in the cluster can be found here. This shows that the actions of stars in the cluster are not conserved (because the self-gravity of the cluster is important), but that the actions of stream members freeze once they are stripped. The angle difference between stars in a stream and the progenitor increases linearly with time, which is shown in the following movie:

where the radial and vertical angle difference with respect to the progenitor (co-moving at $(\theta_R, \theta_\phi, \theta_Z) = (\pi, \pi, \pi)$) is shown for each snapshot (the color-coding gives θ_ϕ).

One last movie provides further insight in how a stream evolves over time. The following movie shows the evolution of the stream in the two dimensional plane of frequency and angle along the stream (that is, both are projections of the three dimensional frequencies or angles onto the angle direction along the stream). The points are color-coded by the time at which they were removed from the progenitor cluster.

It is clear that disruption happens in bursts (at pericenter passages) and that the initial frequency distribution at the time of removal does not change (much) with time. However, stars removed at larger frequency difference move away from the cluster faster, such that the end of the stream is primarily made up of stars with large frequency differences

with respect to the progenitor. This leads to a gradient in the typical orbit in the stream, and the stream is on average *not* on a single orbit.

2.1.1 Modeling streams in galpy

In galpy we can model streams using the tools in `galpy.df.streamdf`. We setup a streamdf instance by specifying the host gravitational potential `pot=`, an actionAngle method (typically `galpy.actionAngle.actionAngleIsochroneApprox`), a `galpy.orbit.Orbit` instance with the position of the progenitor, a parameter related to the velocity dispersion of the progenitor, and the time since disruption began. We first import all of the necessary modules

```
>>> from galpy.df import streamdf
>>> from galpy.orbit import Orbit
>>> from galpy.potential import LogarithmicHaloPotential
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.util import bovy_conversion #for unit conversions
```

setup the potential and actionAngle instances

```
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAI= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

define a progenitor Orbit instance

```
>>> obs= Orbit([1.56148083,0.35081535,-1.15481504,0.88719443,-0.47713334,0.12019596])
```

and instantiate the streamdf model

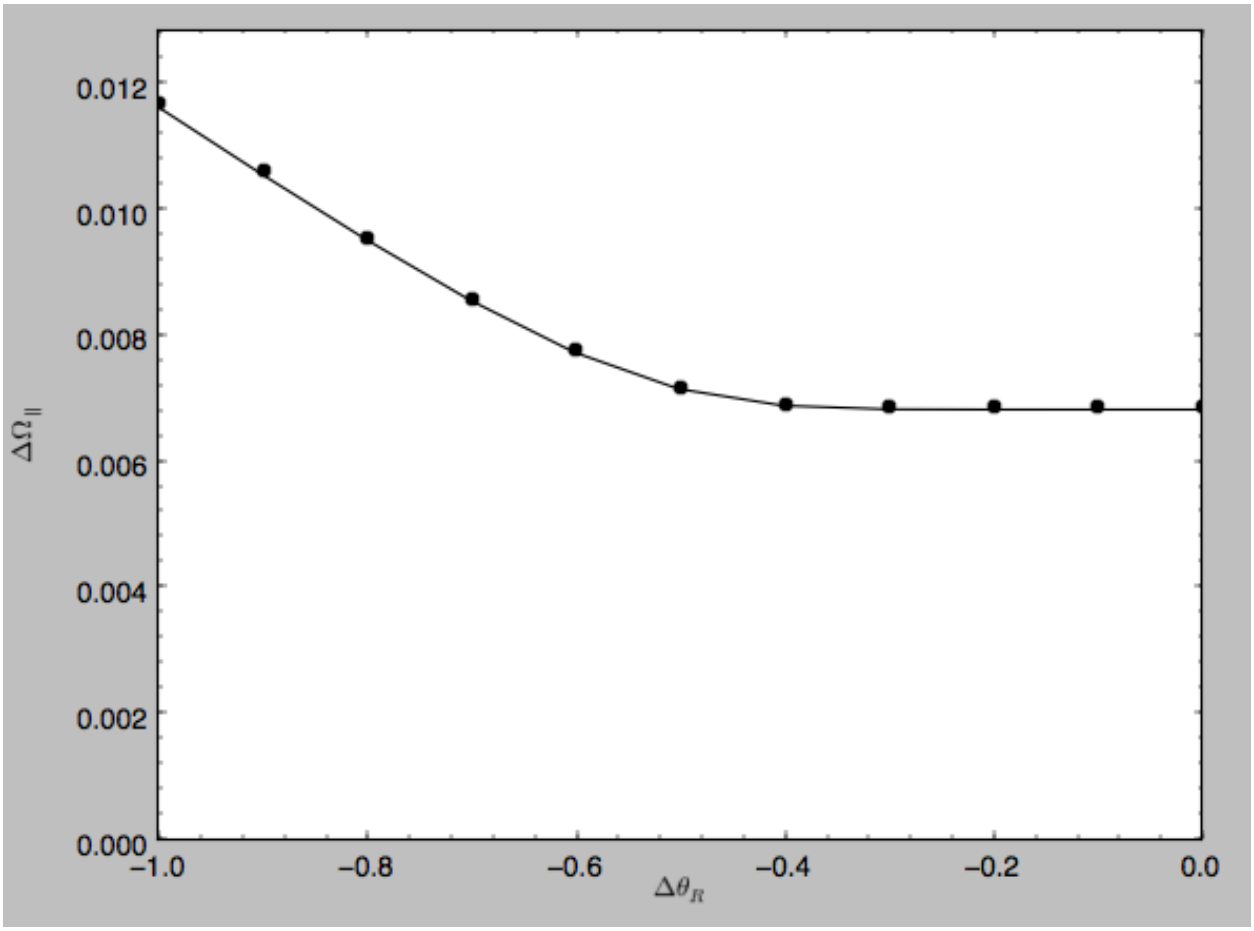
```
>>> sigv= 0.365 #km/s
>>> sdf= streamdf(sigv/220.,progenitor=obs,pot=lp,aA=aAI,leading=True,nTrackChunks=11,
↪tdisrupt=4.5/bovy_conversion.time_in_Gyr(220.,8.))
```

for a leading stream. This runs in about half a minute on a 2011 Macbook Air.

Bovy (2014) discusses how the calculation of the track needs to be iterated for potentials where there is a large offset between the track and a single orbit. One can increase the default number of iterations by specifying `nTrackIterations=` in the streamdf initialization (the default is set based on the angle between the track's frequency vector and the progenitor orbit's frequency vector; you can access the number of iterations used as `sdf.nTrackIterations`). To check whether the track is calculated accurately, one can use the following

```
>>> sdf.plotCompareTrackAAModel()
```

which in this case gives



This displays the stream model's track in frequency offset (y axis) versus angle offset (x axis) as the solid line; this is the track that the model should have if it is calculated correctly. The points are the frequency and angle offset calculated from the calculated track's $(\mathbf{x}, \mathbf{v})$. For a properly computed track these should line up, as they do in this figure. If they do not line up, increasing `nTrackIterations` is necessary.

We can calculate some simple properties of the stream, such as the ratio of the largest and second-to-largest eigenvalue of the Hessian $\partial\mathbf{\Omega}/\partial\mathbf{J}$

```
>>> sdf.freqEigvalRatio(isotropic=True)
# 34.450028399901434
```

or the model's ratio of the largest and second-to-largest eigenvalue of the model frequency variance matrix

```
>>> sdf.freqEigvalRatio()
# 29.625538344985291
```

The fact that this ratio is so large means that an approximately one dimensional stream will form.

Similarly, we can calculate the angle between the frequency vector of the progenitor and of the model mean frequency vector

```
>>> sdf.misalignment()
# 0.0086441947505973005
```

which returns this angle in radians. We can also calculate the angle between the frequency vector of the progenitor and the principal eigenvector of $\partial\mathbf{\Omega}/\partial\mathbf{J}$

```
>>> sdf.misalignment(isotropic=True)
# 0.02238411611147997
```

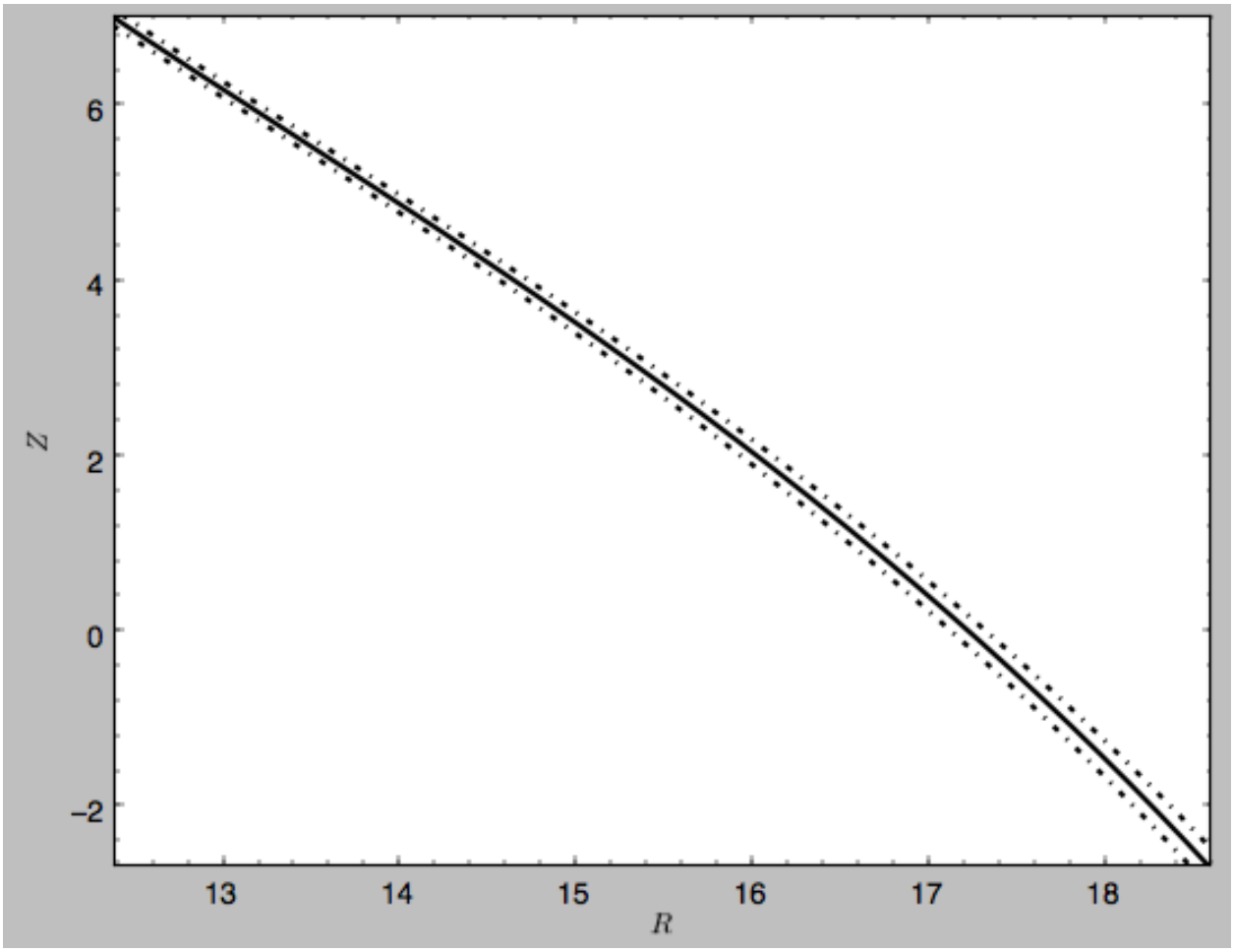
(the reason these are obtained by specifying `isotropic=True` is that these would be the ratio of the eigenvalues or the angle if we assumed that the disrupted materials action distribution were isotropic).

2.1.2 Calculating the average stream location (track)

We can display the stream track in various coordinate systems as follows

```
>>> sdf.plotTrack(d1='r',d2='z',interp=True,color='k',spread=2,overplot=False,lw=2.,
↪scaleToPhysical=True)
```

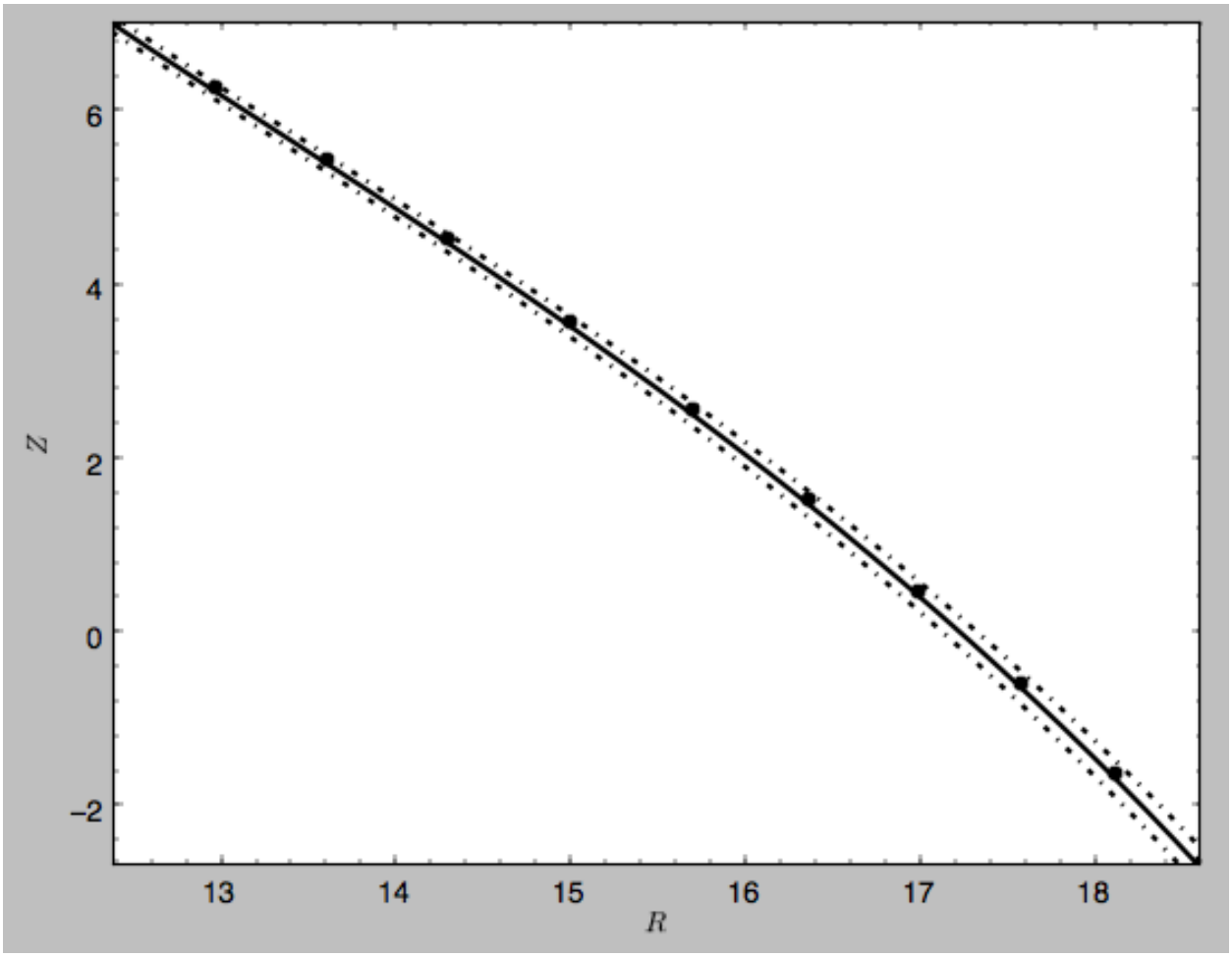
which gives



which shows the track in Galactocentric R and Z coordinates as well as an estimate of the spread around the track as the dash-dotted line. We can overplot the points along the track along which the $(\mathbf{x}, \mathbf{v}) \rightarrow (\mathbf{\Omega}, \theta)$ transformation and the track position is explicitly calculated, by turning off the interpolation

```
>>> sdf.plotTrack(d1='r',d2='z',interp=False,color='k',spread=0,overplot=True,ls='none
↪',marker='o',scaleToPhysical=True)
```

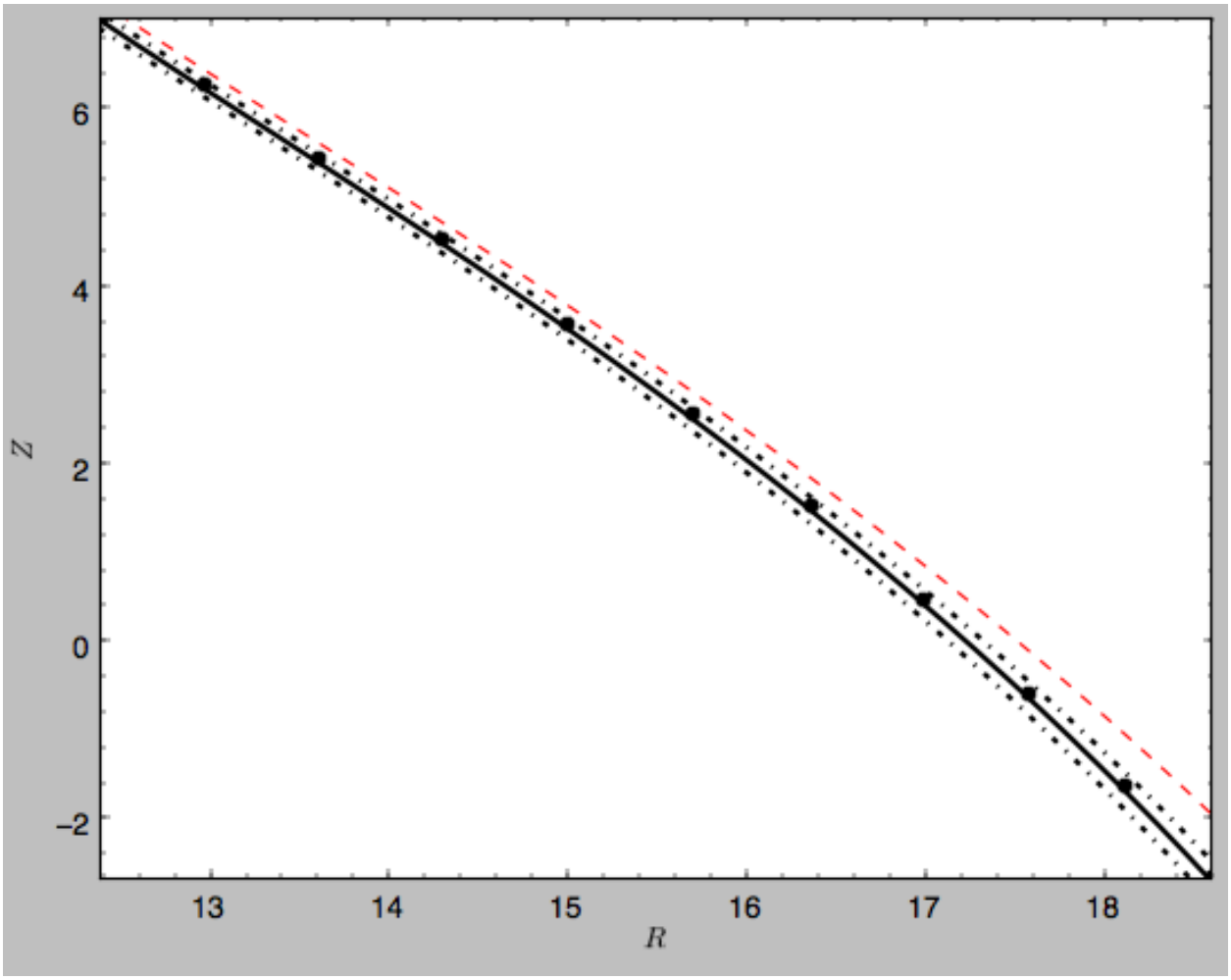
which gives



We can also overplot the orbit of the progenitor

```
>>> sdf.plotProgenitor(d1='r',d2='z',color='r',overplot=True,ls='--',
↪scaleToPhysical=True)
```

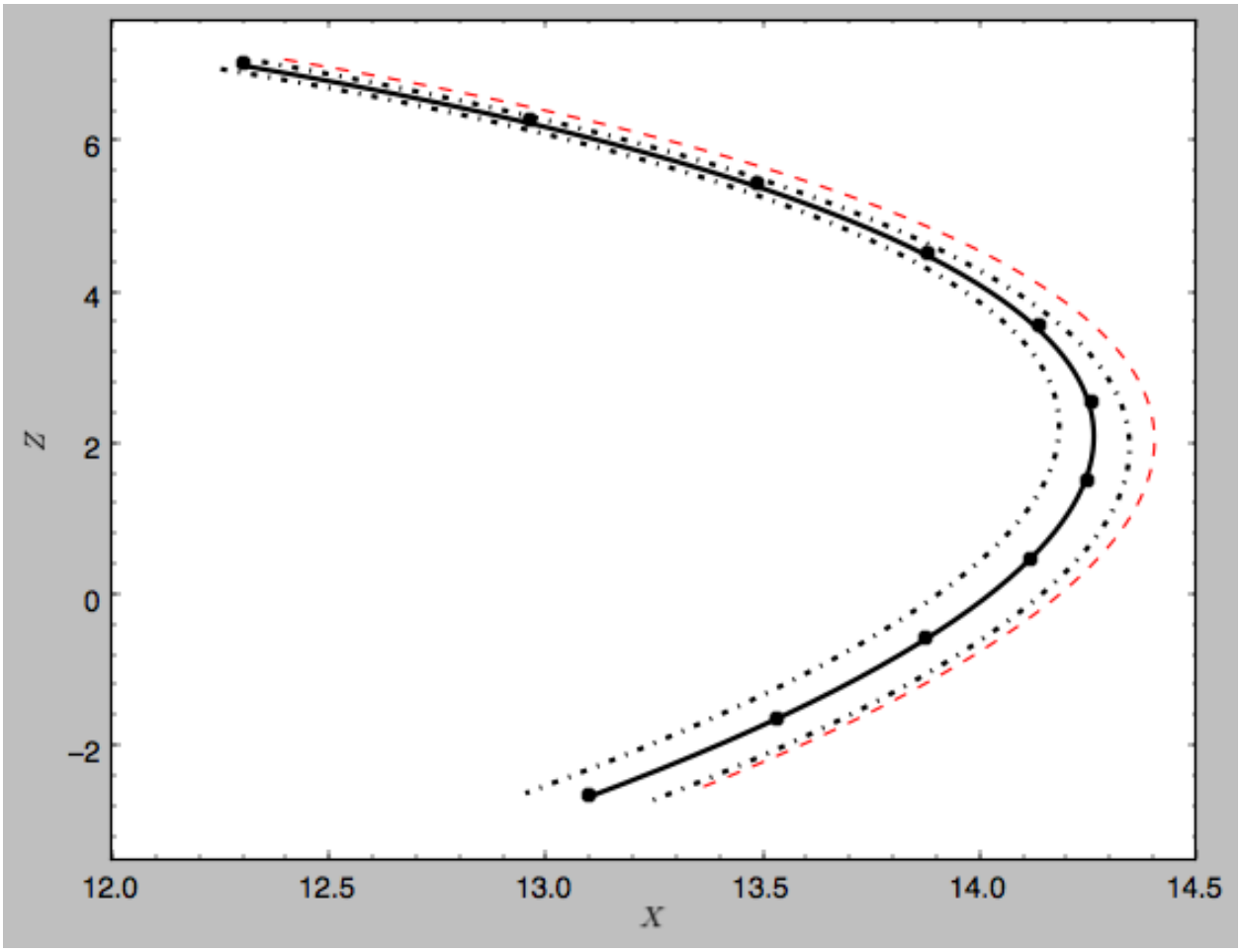
to give



We can do the same in other coordinate systems, for example X and Z (as in Figure 1 of Bovy 2014)

```
>>> sdf.plotTrack(d1='x',d2='z',interp=True,color='k',spread=2,overplot=False,lw=2.,
↪scaleToPhysical=True)
>>> sdf.plotTrack(d1='x',d2='z',interp=False,color='k',spread=0,overplot=True,ls='none
↪',marker='o',scaleToPhysical=True)
>>> sdf.plotProgenitor(d1='x',d2='z',color='r',overplot=True,ls='--',
↪scaleToPhysical=True)
>>> xlim(12.,14.5); ylim(-3.5,7.6)
```

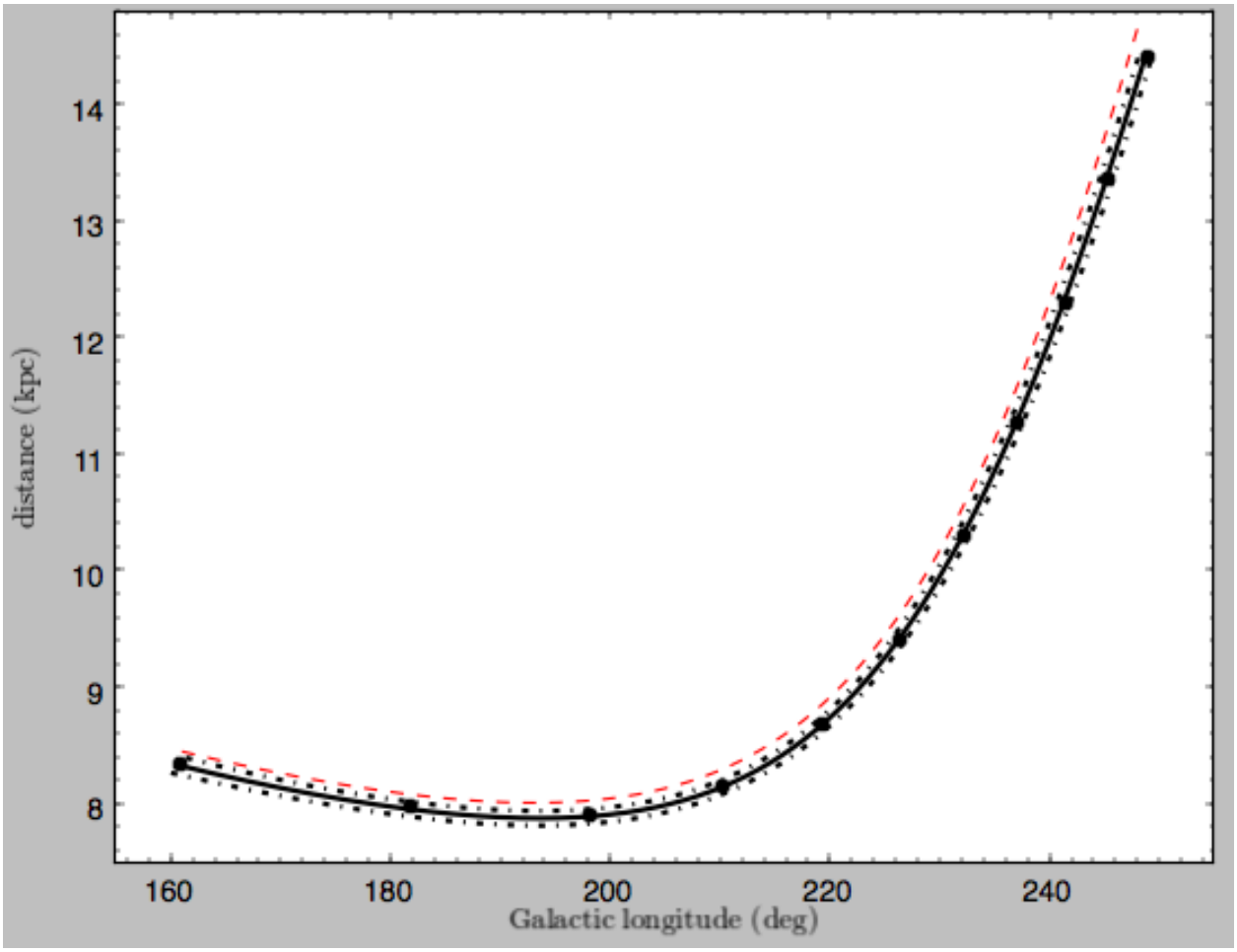
which gives



or we can calculate the track in observable coordinates, e.g.,

```
>>> sdf.plotTrack(d1='ll',d2='dist',interp=True,color='k',spread=2,overplot=False,
↪lw=2.)
>>> sdf.plotTrack(d1='ll',d2='dist',interp=False,color='k',spread=0,overplot=True,ls=
↪'none',marker='o')
>>> sdf.plotProgenitor(d1='ll',d2='dist',color='r',overplot=True,ls='--')
>>> xlim(155.,255.); ylim(7.5,14.8)
```

which displays



Coordinate transformations to physical coordinates are done using parameters set when initializing the `sdf` instance. See the help for `?streamdf` for a complete list of initialization parameters.

2.1.3 Mock stream data generation

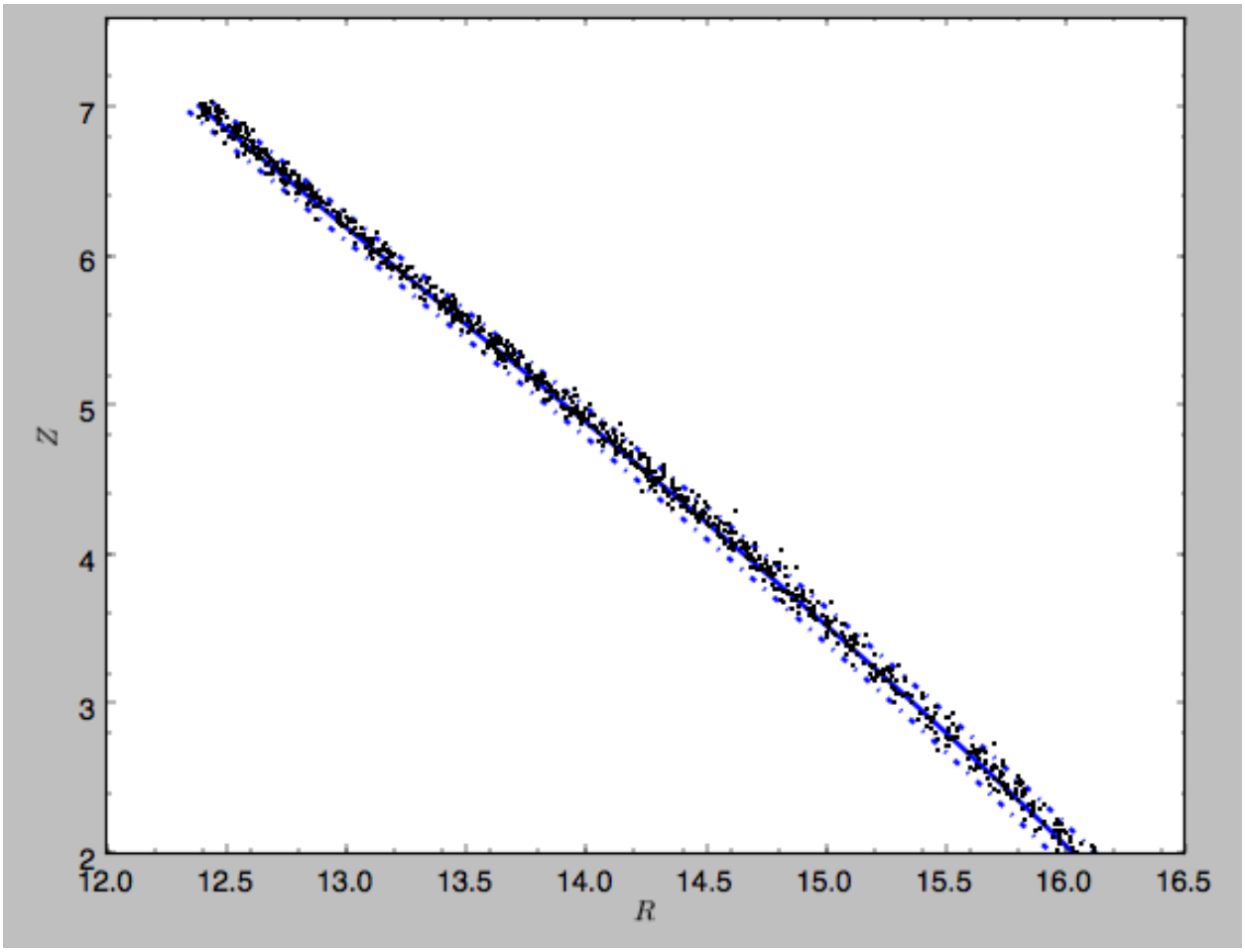
We can also easily generate mock data from the stream model. This uses `streamdf.sample`. For example,

```
>>> RvR= sdf.sample(n=1000)
```

which returns the sampled points as a set $(R, v_R, v_T, Z, v_Z, \phi)$ in natural galpy coordinates. We can plot these and compare them to the track location

```
>>> sdf.plotTrack(d1='r',d2='z',interp=True,color='b',spread=2,overplot=False,lw=2.,
↪scaleToPhysical=True)
>>> plot(RvR[0]*8.,RvR[3]*8.,'k.',ms=2.) #multiply by the physical distance scale
>>> xlim(12.,16.5); ylim(2.,7.6)
```

which gives



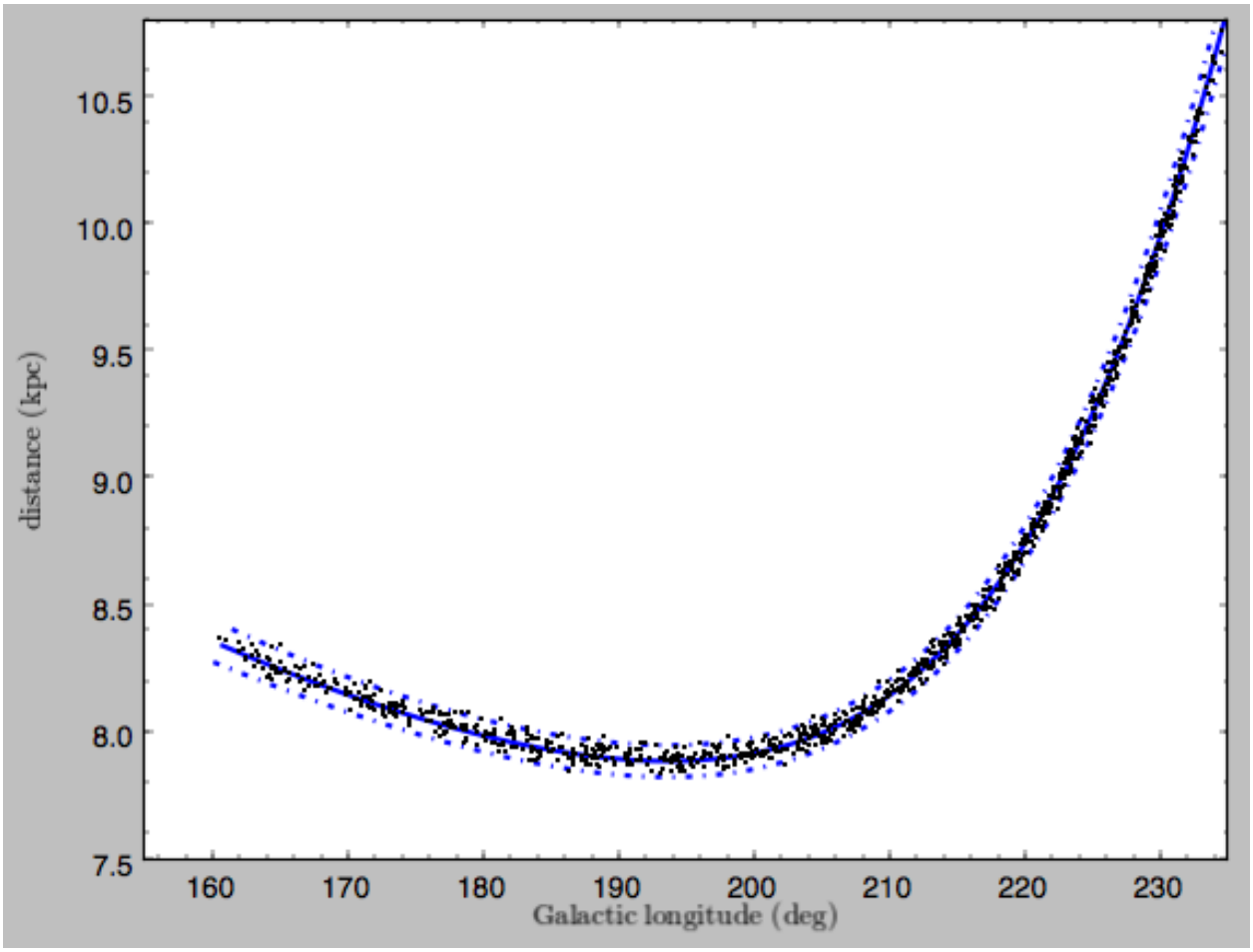
Similarly, we can generate mock data in observable coordinates

```
>>> lb= sdf.sample(n=1000,lb=True)
```

and plot it

```
>>> sdf.plotTrack(d1='ll',d2='dist',interp=True,color='b',spread=2,overplot=False,
↪lw=2.)
>>> plot(lb[0],lb[2],'k.',ms=2.)
>>> xlim(155.,235.); ylim(7.5,10.8)
```

which displays

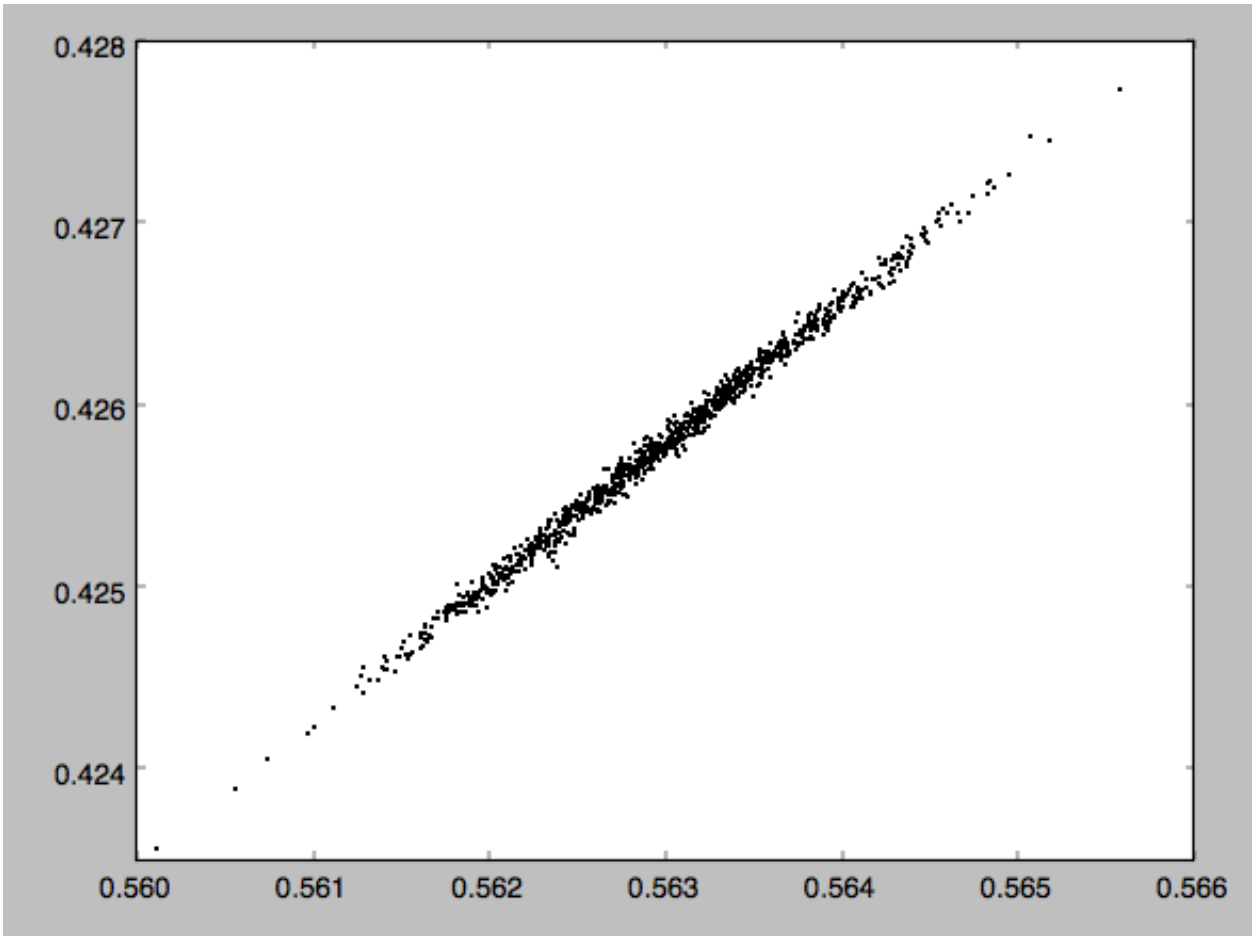


We can also just generate mock stream data in frequency-angle coordinates

```
>>> mockaA= sdf.sample(n=1000,returnaAdt=True)
```

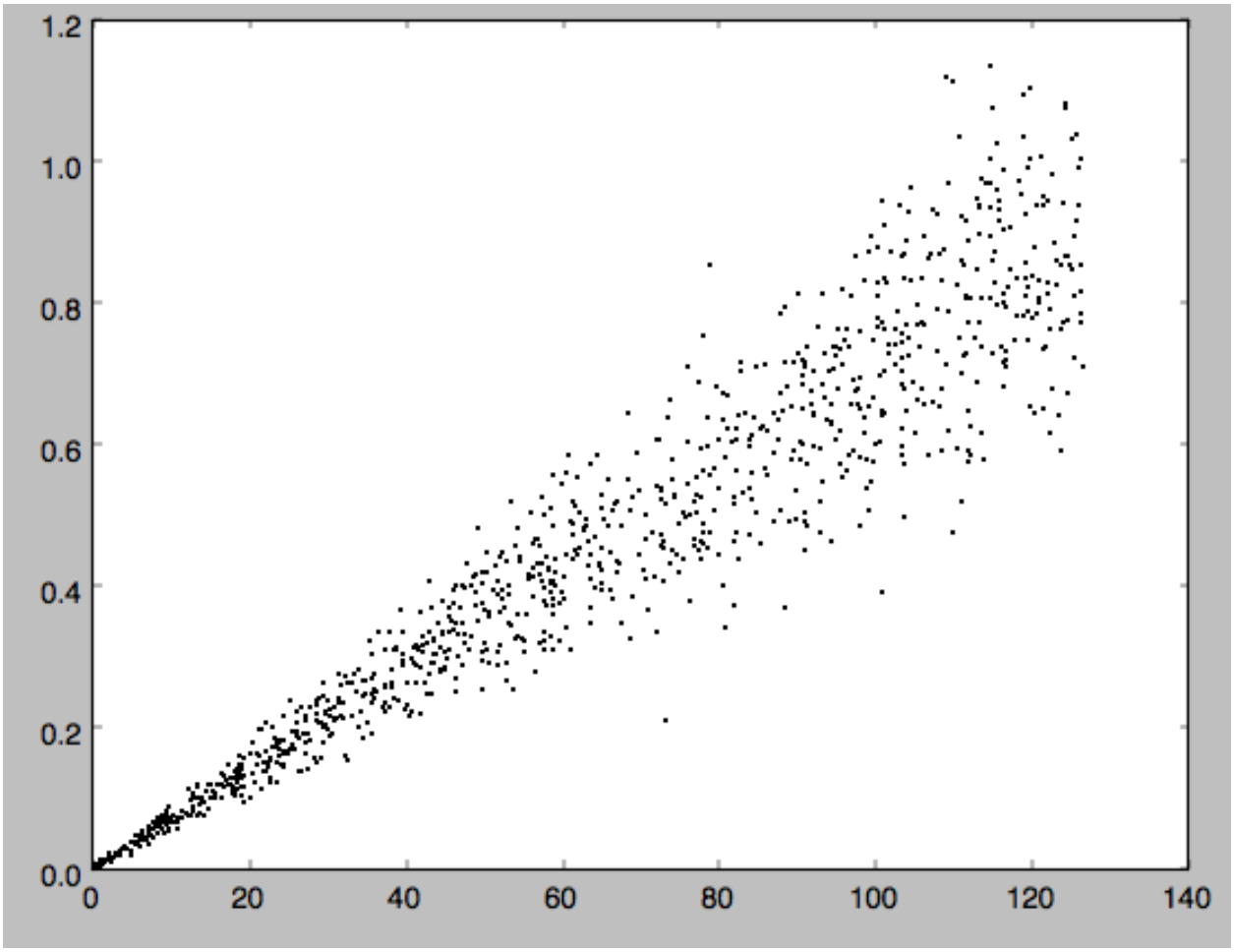
which returns a tuple with three components: an array with shape [3,N] of frequency vectors $(\Omega_R, \Omega_\phi, \Omega_Z)$, an array with shape [3,N] of angle vectors $(\theta_R, \theta_\phi, \theta_Z)$ and t_s, the stripping time. We can plot the vertical versus the radial frequency

```
>>> plot(mockaA[0][0],mockaA[0][2],'k.',ms=2.)
```



or we can plot the magnitude of the angle offset as a function of stripping time

```
>>> plot(mockaA[2],numpy.sqrt(numpy.sum((mockaA[1]-numpy.tile(sdf._progenitor_angle,
↪(1000,1)).T)**2.,axis=0)),'k.',ms=2.)
```



2.1.4 Evaluating and marginalizing the full PDF

We can also evaluate the stream PDF, the probability of a $(\mathbf{x}, \mathbf{v})$ phase-space position in the stream. We can evaluate the PDF, for example, at the location of the progenitor

```
>>> sdf(obs.R(),obs.vR(),obs.vT(),obs.z(),obs.vz(),obs.phi())
# array([-33.16985861])
```

which returns the natural log of the PDF. If we go to slightly higher in Z and slightly smaller in R, the PDF becomes zero

```
>>> sdf(obs.R()-0.1,obs.vR(),obs.vT(),obs.z()+0.1,obs.vz(),obs.phi())
# array([-inf])
```

because this phase-space position cannot be reached by a leading stream star. We can also marginalize the PDF over unobserved directions. For example, similar to Figure 10 in Bovy (2014), we can evaluate the PDF $p(X|Z)$ near a point on the track, say near Z =2 kpc (=0.25 in natural units. We first find the approximate Gaussian PDF near this point, calculated from the stream track and dispersion (see above)

```
>>> meanp, varp= sdf.gaussApprox([None,None,2./8.,None,None,None])
```

where the input is a array with entries [X,Y,Z,vX,vY,vZ] and we substitute None for directions that we want to establish the approximate PDF for. So the above expression returns an approximation to $p(X, Y, v_X, v_Y, v_Z|Z)$. This

approximation allows us to get a sense of where the PDF peaks and what its width is

```
>>> meanp[0]*8.
# 14.267559400127833
>>> numpy.sqrt(varp[0,0])*8.
# 0.04152968631186698
```

We can now evaluate the PDF $p(X|Z)$ as a function of X near the peak

```
>>> xs= numpy.linspace(-3.*numpy.sqrt(varp[0,0]),3.*numpy.sqrt(varp[0,0]),21)+meanp[0]
>>> logps= numpy.array([sdf.callMarg([x,None,2./8.,None,None,None]) for x in xs])
>>> ps= numpy.exp(logps)
```

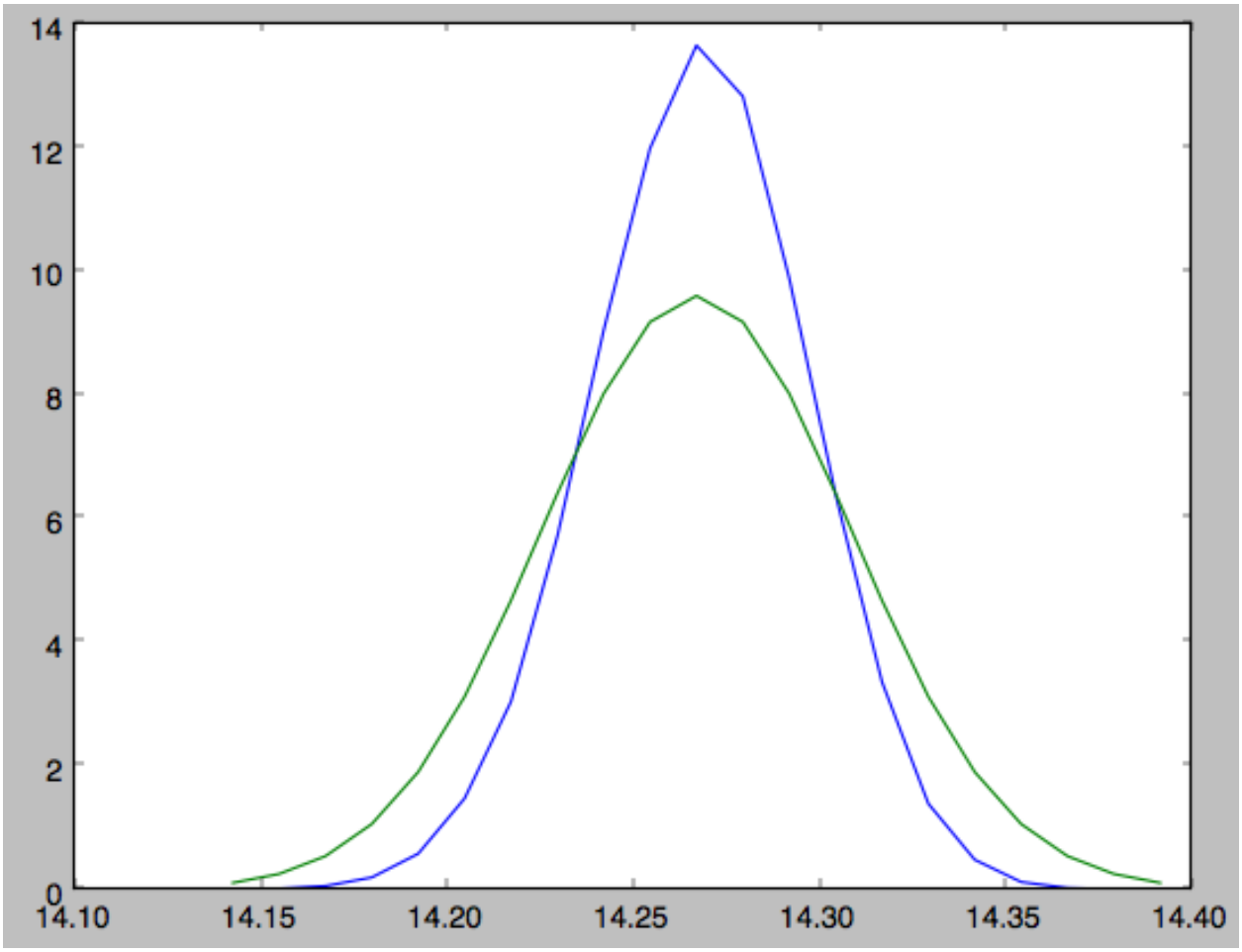
and we normalize the PDF

```
>>> ps/= numpy.sum(ps)*(xs[1]-xs[0])*8.
```

and plot it together with the Gaussian approximation

```
>>> plot(xs*8.,ps)
>>> plot(xs*8.,1./numpy.sqrt(2.*numpy.pi)/numpy.sqrt(varp[0,0])/8.*numpy.exp(-0.5*(xs-
↪meanp[0])**2./varp[0,0]))
```

which gives



Sometimes it is hard to automatically determine the closest point on the calculated track if only one phase-space

coordinate is given. For example, this happens when evaluating $p(Z|X)$ for $X > 13$ kpc here, where there are two branches of the track in Z (see the figure of the track above). In that case, we can determine the closest track point on one of the branches by hand and then provide this closest point as the basis of PDF calculations. The following example shows how this is done for the upper Z branch at $X = 13.5$ kpc, which is near $Z = 5$ kpc (Figure 10 in Bovy 2014).

```
>>> cindx= sdf.find_closest_trackpoint(13.5/8.,None,5.32/8.,None,None,None,xy=True)
```

gives the index of the closest point on the calculated track. This index can then be given as an argument for the PDF functions:

```
>>> meanp, varp= meanp, varp= sdf.gaussApprox([13.5/8.,None,None,None,None,None],
↪cindx=cindx)
```

computes the approximate $p(Y, Z, v_X, v_Y, v_Z|X)$ near the upper Z branch. In Z, this PDF has mean and dispersion

```
>>> meanp[1]*8.
# 5.4005530328542077
>>> numpy.sqrt(varp[1,1])*8.
# 0.05796023309510244
```

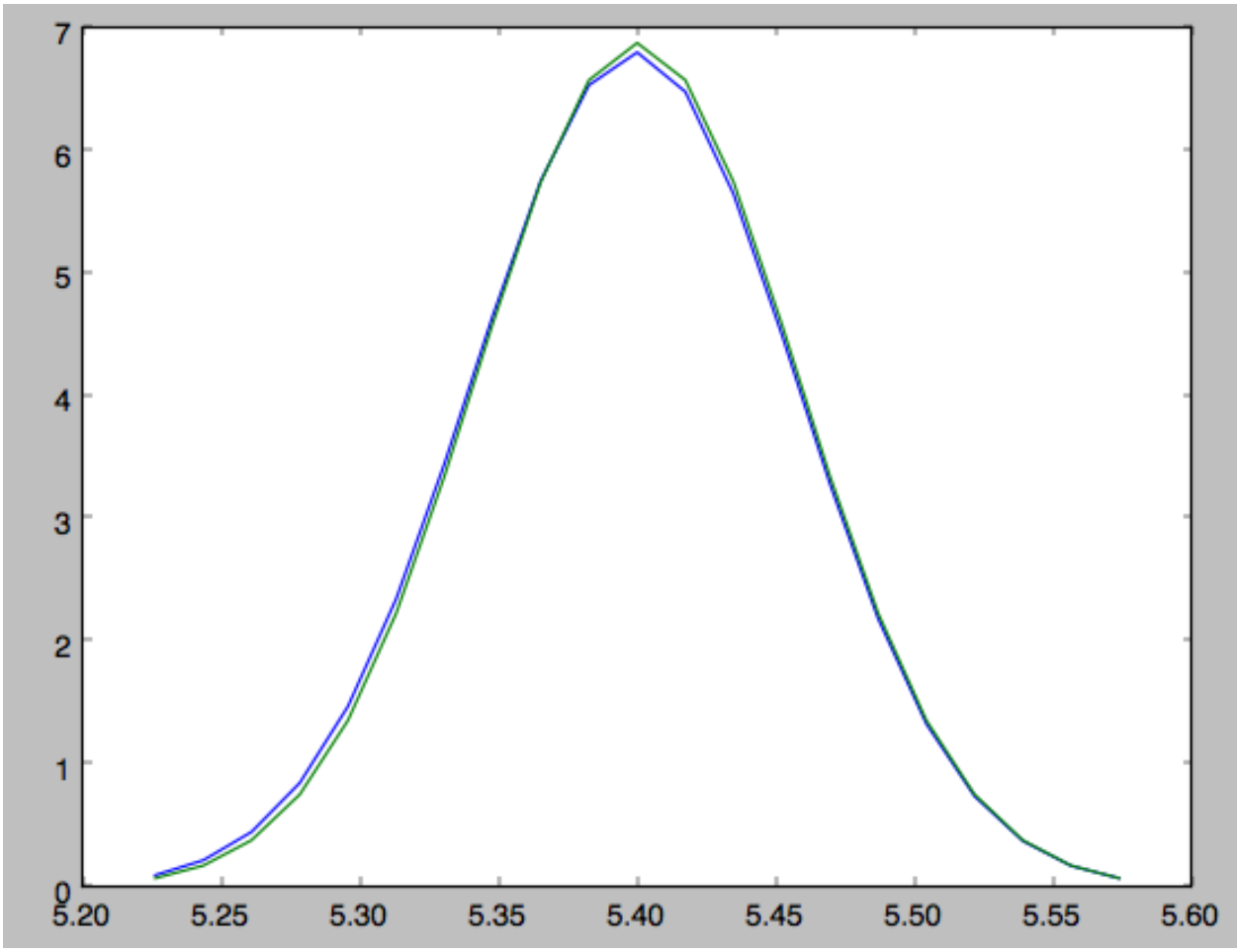
We can then evaluate $p(Z|X)$ for the upper branch as

```
>>> zs= numpy.linspace(-3.*numpy.sqrt(varp[1,1]),3.*numpy.sqrt(varp[1,1]),21)+meanp[1]
>>> logps= numpy.array([sdf.callMarg([13.5/8.,None,z,None,None,None],cindx=cindx) for␣
↪z in zs])
>>> ps= numpy.exp(logps)
>>> ps/= numpy.sum(ps)*(zs[1]-zs[0])*8.
```

and we can again plot this and the approximation

```
>>> plot(zs*8.,ps)
>>> plot(zs*8.,1./numpy.sqrt(2.*numpy.pi)/numpy.sqrt(varp[1,1])/8.*numpy.exp(-0.5*(zs-
↪meanp[1])**2./varp[1,1]))
```

which gives



The approximate PDF in this case is very close to the correct PDF. When supplying the closest track point, care needs to be taken that this really is the closest track point. Otherwise the approximate PDF will not be quite correct.

2.1.5 Modeling gaps in streams

`galpy` also contains tools to model the effect of impacts due to dark-matter subhalos on streams (see Sanders, Bovy, & Erkal 2015). This is implemented as a subclass `streamgapdf` of `streamdf`, because they share many of the same methods. Setting up a `streamgapdf` object requires the same arguments and keywords as setting up a `streamdf` instance (to specify the smooth underlying stream model and the Galactic potential) as well as parameters that specify the impact (impact parameter and velocity, location and time of closest approach, mass and structure of the subhalo, and helper keywords that specify how the impact should be calculated). An example used in the paper (but not that with the modifications in Sec. 6.1) is as follows. Imports:

```
>>> from galpy.df import streamdf, streamgapdf
>>> from galpy.orbit import Orbit
>>> from galpy.potential import LogarithmicHaloPotential
>>> from galpy.actionAngle import actionAngleIsochroneApprox
>>> from galpy.util import bovy_conversion
```

Parameters for the smooth stream and the potential:

```
>>> lp= LogarithmicHaloPotential(normalize=1.,q=0.9)
>>> aAI= actionAngleIsochroneApprox(pot=lp,b=0.8)
```

(continues on next page)

(continued from previous page)

```
>>> prog_unp_peri= Orbit([2.6556151742081835,
                          0.2183747276300308,
                          0.67876510797240575,
                          -2.0143395648974671,
                          -0.3273737682604374,
                          0.24218273922966019])
>>> V0, R0= 220., 8.
>>> sigv= 0.365*(10./2.)**(1./3.) # km/s
>>> tdisrupt= 10.88/bovy_conversion.time_in_Gyr(V0,R0)
```

and the parameters of the impact

```
>>> GM= 10.**-2./bovy_conversion.mass_in_1010msol(V0,R0)
>>> rs= 0.625/R0
>>> impactb= 0.
>>> subhalovel= numpy.array([6.82200571,132.7700529,149.4174464])/V0
>>> timpact= 0.88/bovy_conversion.time_in_Gyr(V0,R0)
>>> impact_angle= -2.34
```

The setup is then

```
>>> sdf_sanders15= streamgapdf(sigv/V0,progenitor=prog_unp_peri,pot=lp,aA=aAI,
                               leading=False,nTrackChunks=26,
                               nTrackIterations=1,
                               sigMeanOffset=4.5,
                               tdisrupt=tdisrupt,
                               Vnorm=V0,Rnorm=R0,
                               impactb=impactb,
                               subhalovel=subhalovel,
                               timpact=timpact,
                               impact_angle=impact_angle,
                               GM=GM,rs=rs)
```

The `streamgapdf` implementation is currently not entirely complete (for example, one cannot yet evaluate the full phase-space PDF), but the model can be sampled as in the paper above. To compare the perturbed model to the unperturbed model, we also set up an unperturbed model of the same stream

```
>>> sdf_sanders15_unp= streamdf(sigv/V0,progenitor=prog_unp_peri,pot=lp,aA=aAI,
                               leading=False,nTrackChunks=26,
                               nTrackIterations=1,
                               sigMeanOffset=4.5,
                               tdisrupt=tdisrupt,
                               Vnorm=V0,Rnorm=R0)
```

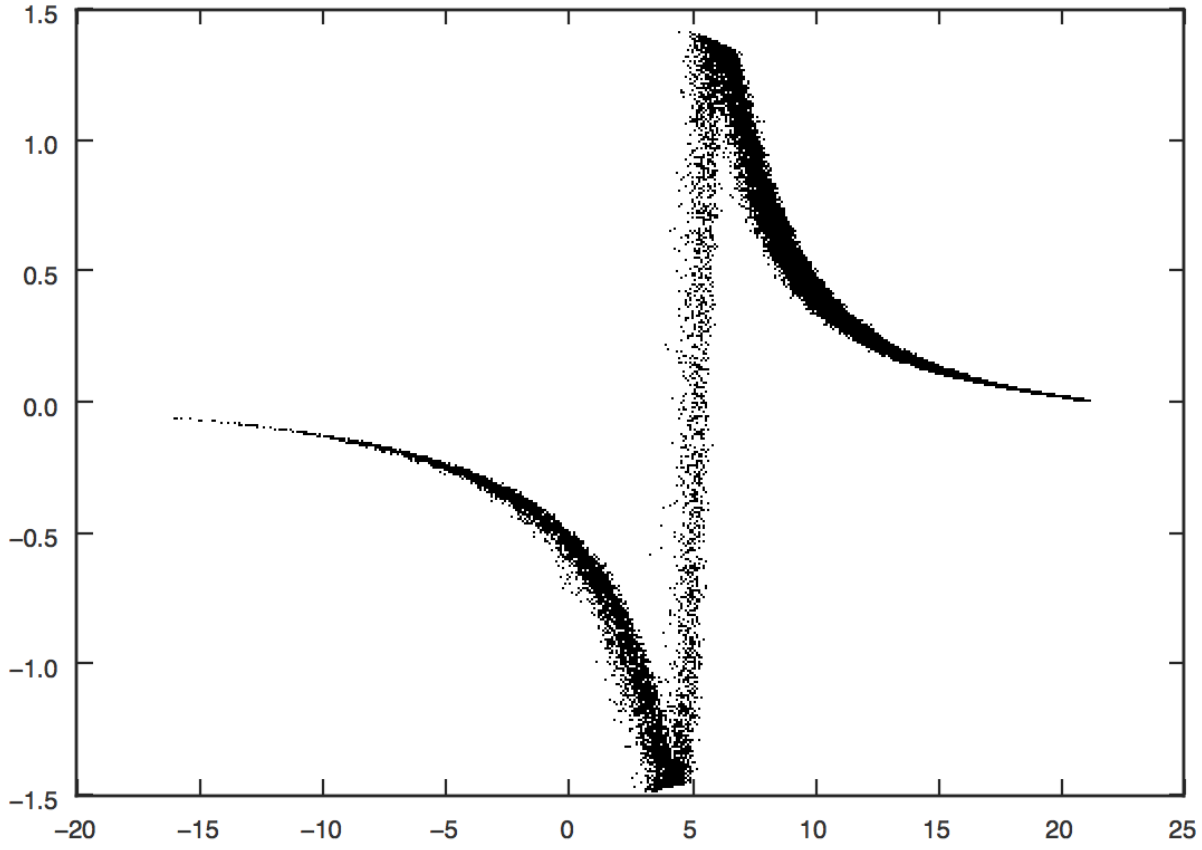
We can then sample positions and velocities for the perturbed and unperturbed prediction for the *same* particle by using the same random seed:

```
>>> numpy.random.seed(1)
>>> xv_mock_per= sdf_sanders15.sample(n=100000,xy=True).T
>>> numpy.random.seed(1) # should give same points
>>> xv_mock_unp= sdf_sanders15_unp.sample(n=100000,xy=True).T
```

and we can plot the offset due to the perturbation, for example,

```
>>> plot(xv_mock_unp[:,0]*R0,(xv_mock_per[:,0]-xv_mock_unp[:,0])*R0,'k,')
```

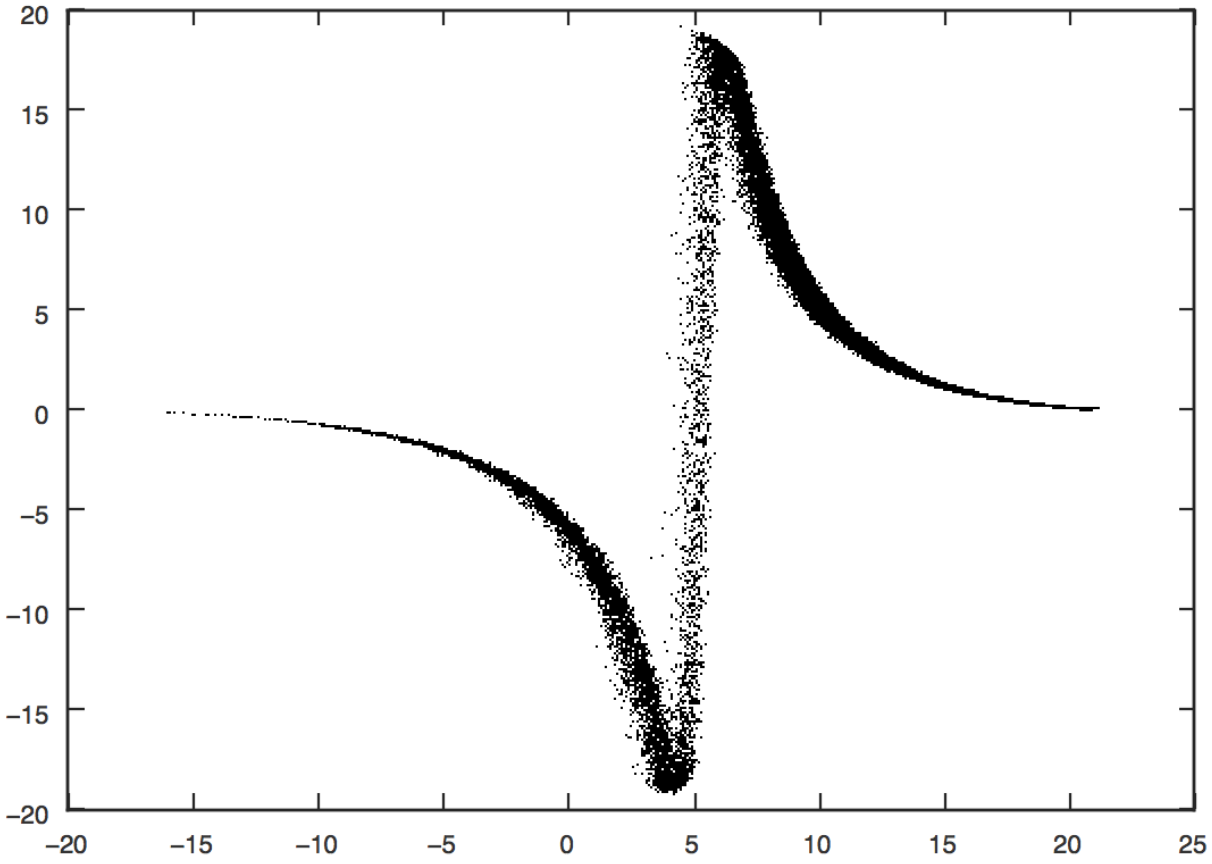
for the difference in X as a function of unperturbed X:



or

```
>>> plot(xv_mock_unp[:,0]*R0,(xv_mock_per[:,4]-xv_mock_unp[:,4])*V0,'k,')
```

for the difference in v_Y as a function of unperturbed X:


20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20
25

CHAPTER 3

Library reference

3.1 Orbit (`galpy.orbit`)

See *Orbit initialization* for a detailed explanation on how to set up Orbit instances.

3.1.1 Class

galpy.orbit.Orbit

3.1.2 Methods

galpy.orbit.Orbit.__add__

galpy.orbit.Orbit.__call__

galpy.orbit.Orbit.animate

galpy.orbit.Orbit.bb

galpy.orbit.Orbit.dec

galpy.orbit.Orbit.dist

galpy.orbit.Orbit.E

galpy.orbit.Orbit.e

galpy.orbit.Orbit.ER

galpy.orbit.Orbit.Ez

galpy.orbit.Orbit.fit

galpy.orbit.Orbit.flip

galpy.orbit.Orbit.integrate

galpy.orbit.Orbit.integrate_dxdv

Currently only supported for `planarOrbit` instances.

galpy.orbit.Orbit.getOrbit

galpy.orbit.Orbit.getOrbit_dxdv

`integrate_dxdv` is currently only supported for `planarOrbit` instances. `getOrbit_dxdv` is therefore also only supported for those types of `Orbit`.

galpy.orbit.Orbit.helioX

galpy.orbit.Orbit.helioY

galpy.orbit.Orbit.helioZ

galpy.orbit.Orbit.Jacobi

galpy.orbit.Orbit.jp

galpy.orbit.Orbit.jr

galpy.orbit.Orbit.jz

galpy.orbit.Orbit.ll

galpy.orbit.Orbit.L

galpy.orbit.Orbit.Op

galpy.orbit.Orbit.Or

galpy.orbit.Orbit.Oz

galpy.orbit.Orbit.phi

galpy.orbit.Orbit.plot

galpy.orbit.Orbit.plot3d

galpy.orbit.Orbit.plotE

galpy.orbit.Orbit.plotER

galpy.orbit.Orbit.toPlanar

galpy.orbit.Orbit.Tp

galpy.orbit.Orbit.Tr

galpy.orbit.Orbit.TrTp

galpy.orbit.Orbit.turn_physical_off

galpy.orbit.Orbit.turn_physical_on

galpy.orbit.Orbit.Tz

galpy.orbit.Orbit.U

galpy.orbit.Orbit.V

galpy.orbit.Orbit.vbb

galpy.orbit.Orbit.vdec

galpy.orbit.Orbit.vll

galpy.orbit.Orbit.vlos

galpy.orbit.Orbit.vphi

galpy.orbit.Orbit.vR

galpy.orbit.Orbit.vra

galpy.orbit.Orbit.vT

galpy.orbit.Orbit.vx

galpy.orbit.Orbit.vy

galpy.orbit.Orbit.vz

galpy.orbit.Orbit.W

galpy.orbit.Orbit.wp

galpy.orbit.Orbit.wr

galpy.orbit.Orbit.wz

galpy.orbit.Orbit.x

galpy.orbit.Orbit.y

galpy.orbit.Orbit.z

galpy.orbit.Orbit.zmax

3.2 Potential (galpy.potential)

3.2.1 3D potentials

General instance routines

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

galpy.potential.Potential.__call__

Warning: galpy potentials do *not* necessarily approach zero at infinity. To compute, for example, the escape velocity or whether or not an orbit is unbound, you need to take into account the value of the potential at infinity. E.g., $v_{\rm esc}(r) = \sqrt{2[\Phi(\infty) - \Phi(r)]}$.

galpy.potential.Potential.dens

galpy.potential.Potential.dvcircdR

galpy.potential.Potential.epifreq

galpy.potential.Potential.flattening

galpy.potential.Potential.lindbladR

galpy.potential.Potential.mass

galpy.potential.Potential.nemo_accname

galpy.potential.Potential.nemo_accpars

galpy.potential.Potential.omegac

galpy.potential.Potential.phiforce

galpy.potential.Potential.phi2deriv

galpy.potential.Potential.plot

galpy.potential.Potential.plotDensity

galpy.potential.Potential.plotEscapecurve

galpy.potential.Potential.plotRotcurve

galpy.potential.Potential.R2deriv

galpy.potential.Potential.Rzderiv

galpy.potential.Potential.Rforce

galpy.potential.Potential.rforce

galpy.potential.Potential.rl

galpy.planar.Potential.toPlanar

galpy.potential.Potential.toVertical

galpy.potential.Potential.turn_physical_off

galpy.potential.Potential.turn_physical_on

galpy.potential.Potential.vcirc

galpy.potential.Potential.verticalfreq

galpy.potential.Potential.vesc

galpy.potential.Potential.vterm

galpy.potential.Potential.z2deriv

galpy.potential.Potential.zforce

In addition to these, the `NFWPotential` also has methods to calculate virial quantities

galpy.potential.Potential.conc

galpy.potential.Potential.mvir

galpy.potential.NFWPotential.rvir

General 3D potential routines

Use as `method(...)`

galpy.potential.dvcircdR

galpy.potential.epifreq

galpy.potential.evaluateDensities

galpy.potential.evaluatephiforces

galpy.potential.evaluatePotentials

Warning: `galpy` potentials do *not* necessarily approach zero at infinity. To compute, for example, the escape velocity or whether or not an orbit is unbound, you need to take into account the value of the potential at infinity. E.g., $v_{\rm esc}(r) = \sqrt{2[\Phi(\infty) - \Phi(r)]}$.

galpy.potential.evaluateR2derivs

galpy.potential.evaluateRzderivs

galpy.potential.evaluateRforces

galpy.potential.evaluaterforces

galpy.potential.evaluatez2derivs

galpy.potential.evaluatezforces

galpy.potential.flattening

galpy.potential.lindbladR

galpy.potential.nemo_accname

galpy.potential.nemo_accpars

galpy.potential.omegac

galpy.potential.plotDensities

galpy.potential.plotEscapecurve

galpy.potential.plotPotentials

galpy.potential.plotRotcurve

galpy.potential.rl

galpy.potential.turn_physical_off

galpy.potential.turn_physical_on

galpy.potential.vcirc

galpy.potential.verticalfreq

galpy.potential.vesc

galpy.potential.vterm

In addition to these, the following methods are available to compute expansion coefficients for the `SCFPotential` class for a given density

galpy.potential.scf_compute_coeffs

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos and Asin by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$ and $Asin = 0$.

For a given $\rho(R, z, \phi)$ we can compute $Acos$ and $Asin$ through the following equation

$$\begin{bmatrix} Acos \\ Asin \end{bmatrix}_{nlm} = \frac{4a^3}{I_{nl}} \int_{\xi=0}^{\infty} \int_{\cos(\theta)=-1}^{1} \int_{\phi=0}^{2\pi} (1+\xi)^2 (1-\xi)^{-4} \rho(R,z,\phi) \Phi_{nlm}(\xi, \cos(\theta), \phi) d\phi d\cos(\theta) d\xi$$

Where

$$\Phi_{nlm}(\xi, \cos(\theta), \phi) = -\frac{\sqrt{2l+1}}{a 2^{2l+1}} \sqrt{\frac{(l-m)!}{(l+m)!}} (1+\xi)^l (1-\xi)^{l+1} C_n^{2l+3/2}(\xi) P_{lm}(\cos(\theta)) \begin{bmatrix} \cos(m\phi) \\ \sin(m\phi) \end{bmatrix}$$

$$I_{nl} = -K_{nl} \frac{4\pi}{a 2^{8l+6}} \frac{\Gamma(n+4l+3)}{n!(n+2l+3/2)[\Gamma(2l+3/2)]^2} \qquad K_{nl} = \frac{1}{2} n(n+4l+3) + (l+1)(2l+1)$$

P_{lm} is the Associated Legendre Polynomials whereas C_n^{α} is the Gegenbauer polynomial.

Also note $\xi = \frac{r-a}{r+a}$, and n, l and m are integers bounded by $0 <= n < N$, $0 <= l < L$, and $0 <= m <= l$

galpy.potential.scf_compute_coeffs_axi

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$.

Further note that this function is a specification of *scf_compute_coeffs* where $Acos_{nlm} = 0$ at $m \neq 0$ and $Asin_{nlm} = None$

For a given $\rho(R, z)$ we can compute $Acos$ and $Asin$ through the following equation

$$Acos_{nlm} = \frac{8\pi a^3}{I_{nl}} \int_{\xi=0}^{\infty} \int_{\cos(\theta)=-1}^{1} (1+\xi)^2 (1-\xi)^{-4} \rho(R,z) \Phi_{nlm}(\xi, \cos(\theta)) d\cos(\theta) d\xi \qquad Asin_{nlm} = None$$

Where

$$\Phi_{nlm}(\xi,\cos(\theta)) = -\frac{\sqrt{2l+1}}{a2^{2l+1}}(1+\xi)^l(1-\xi)^{l+1}C_n^{2l+3/2}(\xi)P_{l0}(\cos(\theta))\delta_{m0}$$

$$I_{nl} = -K_{nl}\frac{4\pi}{a2^{8l+6}}\frac{\Gamma(n+4l+3)}{n!(n+2l+3/2)[\Gamma(2l+3/2)]^2} \qquad K_{nl} = \frac{1}{2}n(n+4l+3)+(l+1)(2l+1)$$

P_{lm} is the Associated Legendre Polynomials whereas C_n^α is the Gegenbauer polynomial.

Also note $\xi = \frac{r-a}{r+a}$, and n, l and m are integers bounded by $0 <= n < N$, $0 <= l < L$, and $m = 0$

galpy.potential.scf_compute_coeffs_spherical

Note: This function computes Acos and Asin as defined in Hernquist & Ostriker (1992), except that we multiply Acos by 2 such that the density from *Galpy's Hernquist Potential* corresponds to $Acos = \delta_{0n}\delta_{0l}\delta_{0m}$.

Futher note that this function is a specification of *scf_compute_coeffs_axi* where $Acos_{nlm} = 0$ at $l \neq 0$

For a given $\rho(r)$ we can compute $Acos$ and $Asin$ through the following equation

$$Acos_{nlm} = \frac{16\pi a^3}{I_{nl}}\int_{\xi=0}^{\infty}(1+\xi)^2(1-\xi)^{-4}\rho(r)\Phi_{nlm}(\xi)d\xi \qquad Asin_{nlm} = None$$

Where

$$\Phi_{nlm}(\xi,\cos(\theta)) = -\frac{1}{2a}(1-\xi)C_n^{3/2}(\xi)\delta_{l0}\delta_{m0}$$

$$I_{n0} = -K_{n0}\frac{1}{4a}\frac{(n+2)(n+1)}{(n+3/2)} \qquad K_{nl} = \frac{1}{2}n(n+3)+1$$

C_n^α is the Gegenbauer polynomial.

Also note $\xi = \frac{r-a}{r+a}$, and n, l and m are integers bounded by $0 <= n < N$, $l = m = 0$

Specific potentials

All of the following potentials can also be modified by the specific `WrapperPotentials` listed *below*.

Spherical potentials

Burkert potential

Double power-law density spherical potential

Jaffe potential

Hernquist potential

Isochrone potential

Kepler potential

NFW potential

Plummer potential

Power-law density spherical potential

Power-law density spherical potential with an exponential cut-off

Pseudo-isothermal potential

Axisymmetric potentials

Double exponential disk potential

Flattened Power-law potential

Flattening is in the potential as in Evans (1994) rather than in the density

Interpolated axisymmetric potential

The `interpRZPotential` class provides a general interface to generate interpolated instances of general three-dimensional, axisymmetric potentials or lists of such potentials. This interpolated potential can be used in any function where other three-dimensional galpy potentials can be used. This includes functions that use `C` to speed up calculations, if the `interpRZPotential` instance was set up with `enable_c=True`. Initialize as

```
>>> from galpy import potential
>>> ip= potential.interpRZPotential(potential.MWPotential,interpPot=True)
```

which sets up an interpolation of the potential itself only. The potential and all different forces and functions (`dens`,''vcirc'', `epifreq`, `verticalfreq`, `dvcircdR`) are interpolated separately and one needs to specify that these need to be interpolated separately (so, for example, one needs to set `interpRforce=True` to interpolate the radial force, or `interpvcirc=True` to interpolate the circular velocity).

When points outside the grid are requested within the python code, the instance will fall back on the original (non-interpolated) potential. However, when the potential is used purely in `C`, like during orbit integration in `C` or during action–angle evaluations in `C`, there is no way for the potential to fall back onto the original potential and nonsense or NaNs will be returned. Therefore, when using `interpRZPotential` in `C`, one must make sure that the whole relevant part of the `(R,z)` plane is covered. One more time:

Warning: When an interpolated potential is used purely in `C`, like during orbit integration in `C` or during action–angle evaluations in `C`, there is no way for the potential to fall back onto the original potential and nonsense or NaNs will be returned. Therefore, when using `interpRZPotential` in `C`, one must make sure that the whole relevant part of the `(R,z)` plane is covered.

Interpolated SnapshotRZ potential

This class is built on the `interpRZPotential` class; see the documentation of that class *here* for additional information on how to setup objects of the `InterpSnapshotRZPotential` class.

Kuzmin disk potential

Kuzmin-Kutuzov Staeckel potential

Logarithmic halo potential

Miyamoto-Nagai potential

Three Miyamoto-Nagai disk approximation to an exponential disk

Razor-thin exponential disk potential

Axisymmetrized N-body snapshot potential

Triaxial, spiral, and bar potentials

Dehnen bar potential

Double power-law density triaxial potential

Ferrers potential

Moving object potential

Softened-needle bar potential

Spiral arms potential

Triaxial Jaffe potential

Triaxial Hernquist potential

Triaxial NFW potential

All `galpy` potentials can also be made to rotate using the `SolidBodyRotationWrapperPotential` listed in the section on wrapper potentials *below*.

General Poisson solvers for disks and halos

Disk potential using SCF basis-function-expansion

Hernquist & Ostriker Self-Consistent-Field-type potential

In addition to these classes, a simple Milky-Way-like potential fit to data on the Milky Way is included as `galpy.potential.MWPotential2014` (see the `galpy` paper for details). Note that this potential assumes a circular velocity of 220 km/s at the solar radius at 8 kpc; see arXiv/1412.3451 for full information on how this potential was fit. This potential is defined as

```
>>> bp= PowerSphericalPotentialwCutoff(alpha=1.8,rc=1.9/8.,normalize=0.05)
>>> mp= MiyamotoNagaiPotential(a=3./8.,b=0.28/8.,normalize=.6)
>>> np= NFWPotential(a=16/8.,normalize=.35)
>>> MWPotential2014= [bp,mp,np]
```

and can thus be used like any list of `Potentials`. If one wants to add the supermassive black hole at the Galactic center, this can be done by

```
>>> from galpy.potential import KeplerPotential
>>> from galpy.util import bovy_conversion
>>> MWPotential2014.append(KeplerPotential(amp=4*10**6./bovy_conversion.mass_in_
↪msol(220.,8.)))
```

for a black hole with a mass of $4 \times 10^6\,M_\odot$.

As explained in *this section*, *without* this black hole `MWPotential2014` can be used with Dehnen's gyrfalcON code using `accname=PowSphwCut+MiyamotoNagai+NFW` and `accpars=0,1001.79126907,1.8,1.9#0,306770.418682,3.0,0.28#0,16.0,162.958241887`.

An older version `galpy.potential.MWPotential` of a similar potential that was *not* fit to data on the Milky Way is defined as

```
>>> mp= MiyamotoNagaiPotential(a=0.5,b=0.0375,normalize=.6)
>>> np= NFWPotential(a=4.5,normalize=.35)
>>> hp= HernquistPotential(a=0.6/8,normalize=0.05)
>>> MWPotential= [mp,np,hp]
```

`galpy.potential.MWPotential2014` supersedes `galpy.potential.MWPotential`.

3.2.2 2D potentials

General instance routines

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

galpy.potential.planarPotential.__call__

galpy.potential.planarPotential.phiforce

galpy.potential.planarPotential.Rforce

galpy.potential.planarPotential.turn_physical_off

galpy.potential.planarPotential.turn_physical_on

General axisymmetric potential instance routines

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

galpy.potential.planarAxiPotential.epifreq

galpy.potential.planarAxiPotential.lindbladR

galpy.potential.planarAxiPotential.omegac

galpy.potential.planarAxiPotential.plot

galpy.potential.planarAxiPotential.plotEscapecurve

galpy.potential.planarAxiPotential.plotRotcurve

galpy.potential.planarAxiPotential.vcirc

galpy.potential.planarAxiPotential.vesc

General 2D potential routines

Use as `method(...)`

galpy.potential.evaluateplanarphiforces

galpy.potential.evaluateplanarPotentials

galpy.potential.evaluateplanarRforces

galpy.potential.evaluateplanarR2derivs

galpy.potential.LinShuReductionFactor

galpy.potential.plotplanarPotentials

Specific potentials

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

galpy.potential.toPlanarPotential

galpy.potential.RZToplanarPotential

In addition, a two-dimensional bar potential, two spiral potentials, the Henon & Heiles (1964) potential, and some static non-axisymmetric perturbations are included

Cos(m phi) disk potential

Generalization of the *lopsided* and *elliptical* disk potentials to any m and to allow for a break radius within which the radial dependence of the potential changes from R^p to R^-p.

Elliptical disk potential

Like in Kuijken & Tremaine. See *galpy.potential.CosmphiDiskPotential* for a more general version that allows for a break radius within which the radial dependence of the potential changes from R^p to R^-p (elliptical disk corresponds to `m=2`).

Henon-Heiles potential

Lopsided disk potential

Like in Kuijken & Tremaine, but for m=1. See *galpy.potential.CosmphiDiskPotential* for a more general version that allows for a break radius within which the radial dependence of the potential changes from R^p to R^-p (lopsided disk corresponds to `m=1`).

Steady-state logarithmic spiral potential

Transient logarithmic spiral potential

3.2.3 1D potentials

General instance routines

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

galpy.potential.linearPotential.__call__

galpy.potential.linearPotential.force

galpy.potential.linearPotential.plot

galpy.potential.linearPotential.turn_physical_off

galpy.potential.linearPotential.turn_physical_on

General 1D potential routines

Use as `method(...)`

galpy.potential.evaluatelinearForces

galpy.potential.evaluatelinearPotentials

galpy.potential.plotlinearPotentials

Specific potentials

Vertical Kuijken & Gilmore potential

One-dimensional potentials can also be derived from 3D axisymmetric potentials as the vertical potential at a certain Galactocentric radius

galpy.potential.RZToverticalPotential

3.2.4 Potential wrappers

Gravitational potentials in `galpy` can also be modified using wrappers, for example, to change their amplitude as a function of time. These wrappers can be applied to *any* `galpy` potential (although whether they can be used in C depends on whether the wrapper *and* all of the potentials that it wraps are implemented in C). Multiple wrappers can be applied to the same potential.

Specific wrappers

Dehnen-like smoothing wrapper potential

Solid-body rotation wrapper potential

3.3 actionAngle (`galpy.actionAngle`)

3.3.1 (x, v) –> (J, O, a)

General instance routines

Not necessarily supported for all different types of actionAngle calculations. These have extra arguments for different `actionAngle` modules, so check the documentation of the module-specific functions for more info (e.g., `?actionAngleIsochrone.__call__`)

galpy.actionAngle.actionAngle.__call__

galpy.actionAngle.actionAngle.actionsFreqs

galpy.actionAngle.actionAngle.actionsFreqsAngles

galpy.actionAngle.actionAngle.EccZmaxRperiRap

galpy.actionAngle.actionAngle.turn_physical_off

galpy.actionAngle.actionAngle.turn_physical_on

Specific actionAngle modules

actionAngleIsochrone

actionAngleSpherical

actionAngleAdiabatic

actionAngleAdiabaticGrid

actionAngleStaeckel

actionAngleStaeckelGrid

actionAngleIsochroneApprox

3.3.2 (J, a) –> (x, v, O)

General instance routines

Warning: While the `actionAngleTorus` code below can compute the Jacobian and Hessian of the (**J**, **a**) –> (**x**, **v**, **O**) transformation, the accuracy of these does not appear to be very good using the current interface to the TorusMapper code, so care should be taken when using these.

Currently, only the interface to the TorusMapper code supports going from (**J**, **a**) –> (**x**, **v**, **O**). Instance methods are

galpy.actionAngle.actionAngleTorus.__call__

galpy.actionAngle.actionAngleTorus.Freqs

galpy.actionAngle.actionAngleTorus.hessianFreqs

galpy.actionAngle.actionAngleTorus.xvFreqs

galpy.actionAngle.actionAngleTorus.xvJacobianFreqs

Specific actionAngle modules

actionAngleTorus

3.4 DF (`galpy.df`)

3.4.1 General instance routines for all df classes

galpy.actionAngle.actionAngle.turn_physical_off

galpy.actionAngle.actionAngle.turn_physical_on

3.4.2 Two-dimensional, axisymmetric disk distribution functions

Distribution function for orbits in the plane of a galactic disk.

General instance routines

galpy.df.diskdf.__call__

galpy.df.diskdf.asymmetricdrift

galpy.df.diskdf.kurtosisvR

galpy.df.diskdf.kurtosisvT

galpy.df.diskdf.meanvR

galpy.df.diskdf.meanvT

galpy.df.diskdf.oortA

galpy.df.diskdf.oortB

galpy.df.diskdf.oortC

galpy.df.diskdf.oortK

galpy.df.diskdf.sigma2surfacemass

galpy.df.diskdf.sigma2

galpy.df.diskdf.sigmaR2

galpy.df.diskdf.sigmaT2

galpy.df.diskdf.skewvR

galpy.df.diskdf.skewvT

galpy.df.diskdf.surfacemass

galpy.df.diskdf.surfacemassLOS

galpy.df.diskdf.targetSigma2

galpy.df.diskdf.targetSurfacemass

galpy.df.diskdf.targetSurfacemassLOS

galpy.df.diskdf._vmomentsurfacemass

Sampling routines

galpy.df.diskdf.sample

galpy.df.diskdf.sampledSurfacemassLOS

hhgalpy.df.diskdf.sampleLOS

galpy.df.diskdf.sampleVRVT

Specific distribution functions

Dehnen DF

Schwarzschild DF

Shu DF

3.4.3 Two-dimensional, non-axisymmetric disk distribution functions

Distribution function for orbits in the plane of a galactic disk in non-axisymmetric potentials. These are calculated using the technique of Dehnen 2000, where the DF at the current time is obtained as the evolution of an initially-axisymmetric DF at time `to` in the non-axisymmetric potential until the current time.

General instance routines

galpy.df.evolveddiskdf.__call__

The DF of a two-dimensional, non-axisymmetric disk

galpy.df.evolveddiskdf.meanvR

galpy.df.evolveddiskdf.meanvT

galpy.df.evolveddiskdf.oortA

galpy.df.evolveddiskdf.oortB

galpy.df.evolveddiskdf.oortC

galpy.df.evolveddiskdf.oortK

galpy.df.evolveddiskdf.sigmaR2

galpy.df.evolveddiskdf.sigmaRT

galpy.df.evolveddiskdf.sigmaT2

galpy.df.evolveddiskdf.vertexdev

galpy.df.evolveddiskdf.vmomentsurfacemass

3.4.4 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__

galpy.df.quasiisothermaldf.density

galpy.df.quasiisothermaldf.estimate_hr

galpy.df.quasiisothermaldf.estimate_hsr

galpy.df.quasiisothermaldf.estimate_hsz

galpy.df.quasiisothermaldf.estimate_hz

galpy.df.quasiisothermaldf._jmomentdensity

galpy.df.quasiisothermaldf.meanjr

galpy.df.quasiisothermaldf.meanjz

galpy.df.quasiisothermaldf.meanlz

galpy.df.quasiisothermaldf.meanvR

galpy.df.quasiisothermaldf.meanvT

galpy.df.quasiisothermaldf.meanvz

galpy.df.quasiisothermaldf.pvR

galpy.df.quasiisothermaldf.pvRvT

galpy.df.quasiisothermaldf.pvRvz

galpy.df.quasiisothermaldf.pvT

galpy.df.quasiisothermaldf.pvTvz

galpy.df.quasiisothermaldf.pvz

galpy.df.quasiisothermaldf.sampleV

galpy.df.quasiisothermaldf.sigmaR2

galpy.df.quasiisothermaldf.sigmaRz

galpy.df.quasiisothermaldf.sigmaT2

galpy.df.quasiisothermaldf.sigmaz2

galpy.df.quasiisothermaldf.surfacemass_z

galpy.df.quasiisothermaldf.tilt

galpy.df.quasiisothermaldf._vmomentdensity

Specific distribution functions

Quasi-isothermal DF

3.4.5 The distribution function of a tidal stream

From Bovy 2014; see *Dynamical modeling of tidal streams*.

General instance routines

galpy.df.streamdf.__call__

The stream DF

galpy.df.streamdf.calc_stream_lb

galpy.df.streamdf.callMarg

galpy.df.streamdf.density_par

galpy.df.streamdf.estimateTdisrupt

galpy.df.streamdf.find_closest_trackpoint

galpy.df.streamdf.find_closest_trackpointLB

galpy.df.streamdf.freqEigvalRatio

galpy.df.streamdf.gaussApprox

galpy.df.streamdf.length

galpy.df.streamdf.meanangledAngle

galpy.df.streamdf.meanOmega

galpy.df.streamdf.meantdAngle

galpy.df.streamdf.misalignment

galpy.df.streamdf.pangledAngle

galpy.df.streamdf.plotCompareTrackAAModel

galpy.df.streamdf.plotProgenitor

galpy.df.streamdf.plotTrack

galpy.df.streamdf.pOparapar

galpy.df.streamdf.ptdAngle

galpy.df.streamdf.sample

galpy.df.streamdf.sigangledAngle

galpy.df.streamdf.sigOmega

galpy.df.streamdf.sigtdAngle

galpy.df.streamdf.subhalo_encounters

3.4.6 The distribution function of a gap in a tidal stream

From Sanders, Bovy, & Erkal 2015; see *Modeling gaps in streams*. Implemented as a subclass of `streamdf`. No full implementation is available currently, but the model can be set up and sampled as in the above paper.

General instance routines

The stream gap DF

Helper routines to compute kicks

galpy.df.impulse_deltav_plummer

galpy.df.impulse_deltav_plummer_curvedstream

galpy.df.impulse_deltav_hernquist

galpy.df.impulse_deltav_hernquist_curvedstream

galpy.df.impulse_deltav_general

galpy.df.impulse_deltav_general_curvedstream

galpy.df.impulse_deltav_general_orbitintegration

galpy.df.impulse_deltav_general_fullplummerintegration

3.5 Utilities (`galpy.util`)

3.5.1 galpy.util.config

Configuration module

galpy.util.config.set_ro

`galpy.util.config.`**`set_ro`**(*ro*)

NAME: set_ro

PURPOSE: set the global configuration value of ro (distance scale)

INPUT: ro - scale in kpc or astropy Quantity

OUTPUT: (none)

HISTORY: 2016-01-05 - Written - Bovy (UofT)

galpy.util.config.set_vo

galpy.util.config.**set_vo**(*vo*)

NAME: set_vo

PURPOSE: set the global configuration value of vo (velocity scale)

INPUT: vo - scale in km/s or astropy Quantity

OUTPUT: (none)

HISTORY: 2016-01-05 - Written - Bovy (UofT)

3.5.2 galpy.util.bovy_plot

Warning: Importing `galpy.util.bovy_plot` (or having it be imported by other `galpy` routines) with `seaborn` installed may change the `seaborn` plot style. If you don't like this, set the configuration parameter `seaborn-plotting-defaults` to False in the *configuration file*

Various plotting routines:

galpy.util.bovy_plot.bovy_dens2d

galpy.util.bovy_plot.**bovy_dens2d**(*X*, ***kwargs*)

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)

conditional - normalize each column separately (for probability densities, i.e., cntrmass=True)

gcf=True does not start a new figure (does change the ranges and labels)

Contours:

justcontours - if True, only draw contours

contours - if True, draw contours (10 by default)

levels - contour-levels

cntrmass - if True, the density is a probability and the levels are probability masses contained within the contour

cntrcolors - colors for contours (single color or array)

cntrlabel - label the contours

cntrlw, cntrls - linewidths and linestyles for contour

cntrlabelsize, cntrlabelcolors,cntrinline - contour arguments

cntrSmooth - use ndimage.gaussian_filter to smooth before contouring

onedhists - if True, make one-d histograms on the sides

onedhistcolor - histogram color

retAxes= return all Axes instances

retCont= return the contour instance

OUTPUT:

plot to output device, Axes instances depending on input

HISTORY:

2010-03-09 - Written - Bovy (NYU)

galpy.util.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

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

+all pyplot.hist keywords

OUTPUT: (from the matplotlib docs: http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.hist)

The return value is a tuple (n, bins, patches) or ([n0, n1, ...], bins, [patches0, patches1,...]) if the input contains multiple data

HISTORY:

2009-12-23 - Written - Bovy (NYU)

galpy.util.bovy_plot.bovy_plot

galpy.util.bovy_plot.**bovy_plot**(**args*, ***kwargs*)

NAME:

bovy_plot

PURPOSE:

wrapper around matplotlib's plot function

INPUT:

see http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot

xlabel - (raw string!) x-axis label, LaTeX math mode, no $s needed

ylabel - (raw string!) y-axis label, LaTeX math mode, no $s needed

xrange

yrange

scatter= if True, use pyplot.scatter and its options etc.

colorbar= if True, and scatter==True, add colorbar

crange - range for colorbar of scatter==True

clabel= label for colorbar

overplot=True does not start a new figure and does not change the ranges and labels

gcf=True does not start a new figure (does change the ranges and labels)

onedhists - if True, make one-d histograms on the sides

onedhistcolor, onedhistfc, onedhistec

onedhistxnormed, onedhistynormed - normed keyword for one-d histograms

onedhistxweights, onedhistyweights - weights keyword for one-d histograms

bins= number of bins for onedhists

semilogx=, semilogy=, loglog= if True, plot logs

OUTPUT:

plot to output device, returns what pyplot.plot returns, or 3 Axes instances if onedhists=True

HISTORY:

2009-12-28 - Written - Bovy (NYU)

galpy.util.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

conditional - normalize each column separately (for probability densities, i.e., cntrmass=True)

gcf=True does not start a new figure (does change the ranges and labels)

contours - if False, don't plot contours

justcontours - if True, only draw contours, no density

cntrcolors - color of contours (can be array as for bovy_dens2d)

cntrlw, cntrls - linewidths and linestyles for contour

cntrSmooth - use ndimage.gaussian_filter to smooth before contouring

levels - contour-levels; data points outside of the last level will be individually shown (so, e.g., if this list is descending, contours and data points will be overplotted)

onedhists - if True, make one-d histograms on the sides

onedhistx - if True, make one-d histograms on the side of the x distribution

onedhisty - if True, make one-d histograms on the side of the y distribution

onedhistcolor, onedhistfc, onedhistec

onedhistxnormed, onedhistynormed - normed keyword for one-d histograms

onedhistxweights, onedhistyweights - weights keyword for one-d histograms

cmap= cmap for density plot

hist= and edges= - you can supply the histogram of the data yourself, this can be useful if you want to censor the data, both need to be set and calculated using scipy.histogramdd with the given range

retAxes= return all Axes instances

OUTPUT:

plot to output device, Axes instance(s) or not, depending on input

HISTORY:

2010-04-15 - Written - Bovy (NYU)

`galpy` also contains a new matplotlib projection `'galpolar'` that can be used when working with older versions of matplotlib like `'polar'` to create a polar plot in which the azimuth increases clockwise (like when looking at the Milky Way from the north Galactic pole). In newer versions of matplotlib, this does not work, but the `'polar'` projection now supports clockwise azimuths by doing, e.g.,

```
>>> ax= pyplot.subplot(111,projection='polar')
>>> ax.set_theta_direction(-1)
```

3.5.3 galpy.util.bovy_conversion

Utility functions that provide conversions between galpy's *natural* units and *physical* units. These can be used to translate galpy outputs in natural coordinates to physical units by multiplying with the appropriate function.

These could also be used to figure out the conversion between different units. For example, if you want to know how many $\mathrm{GeV\,cm^{-3}}$ correspond to $1\,M_\odot\,\mathrm{pc^{-3}}$, you can calculate

```
>>> from galpy.util import bovy_conversion
>>> bovy_conversion.dens_in_gevcc(1.,1.)/bovy_conversion.dens_in_msolpc3(1.,1.)
# 37.978342941703616
```

or $1\,M_\odot\,\mathrm{pc^{-3}} \approx 40\,\mathrm{GeV\,cm^{-3}}$.

Functions:

galpy.util.bovy_conversion.dens_in_criticaldens

galpy.util.bovy_conversion.**dens_in_criticaldens**(*vo*, *ro*, *H=70.0*)

NAME:

dens_in_criticaldens

PURPOSE:

convert density to units of the critical density

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

H= (default: 70) Hubble constant in km/s/Mpc

OUTPUT:

conversion from units where vo=1. at ro=1. to units of the critical density

HISTORY:

2014-01-28 - Written - Bovy (IAS)

galpy.util.bovy_conversion.dens_in_gevcc

galpy.util.bovy_conversion.**dens_in_gevcc**(*vo*, *ro*)

NAME:

dens_in_gevcc

PURPOSE:

convert density to GeV / cm^3

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1. to GeV/cm^3

HISTORY:

2014-06-16 - Written - Bovy (IAS)

galpy.util.bovy_conversion.dens_in_meanmatterdens

galpy.util.bovy_conversion.**dens_in_meanmatterdens**(*vo*, *ro*, *H=70.0*, *Om=0.3*)

NAME:

dens_in_meanmatterdens

PURPOSE:

convert density to units of the mean matter density

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

H= (default: 70) Hubble constant in km/s/Mpc

Om= (default: 0.3) Omega matter

OUTPUT:

conversion from units where vo=1. at ro=1. to units of the mean matter density

HISTORY:

2014-01-28 - Written - Bovy (IAS)

galpy.util.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**(*vo*, *ro*)

NAME:

force_in_pcMyr2

PURPOSE:

convert a force or acceleration to pc/Myr^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.force_in_10m13kms2

galpy.util.bovy_conversion.**force_in_10m13kms2**(*vo*, *ro*)

NAME:

force_in_10m13kms2

PURPOSE:

convert a force or acceleration to 10^(-13) km/s^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2014-01-22 - Written - Bovy (IAS)

galpy.util.bovy_conversion.force_in_kmsMyr

galpy.util.bovy_conversion.**force_in_kmsMyr**(*vo*, *ro*)

NAME:

force_in_kmsMyr

PURPOSE:

convert a force or acceleration to km/s/Myr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.freq_in_Gyr

galpy.util.bovy_conversion.**freq_in_Gyr**(*vo*, *ro*)

NAME:

freq_in_Gyr

PURPOSE:

convert a frequency to 1/Gyr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.freq_in_kmskpc

galpy.util.bovy_conversion.**freq_in_kmskpc**(*vo*, *ro*)

NAME:

freq_in_kmskpc

PURPOSE:

convert a frequency to km/s/kpc

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.surfdens_in_msolpc2

galpy.util.bovy_conversion.**surfdens_in_msolpc2**(*vo*, *ro*)

NAME:

surfdens_in_msolpc2

PURPOSE:

convert a surface density to Msolar / pc^2

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.mass_in_msol

galpy.util.bovy_conversion.**mass_in_msol**(*vo*, *ro*)

NAME:

mass_in_msol

PURPOSE:

convert a mass to Msolar

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.mass_in_1010msol

galpy.util.bovy_conversion.**mass_in_1010msol**(*vo*, *ro*)

NAME:

mass_in_1010msol

PURPOSE:

convert a mass to 10^10 x Msolar

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.time_in_Gyr

galpy.util.bovy_conversion.**time_in_Gyr**(*vo*, *ro*)

NAME:

time_in_Gyr

PURPOSE:

convert a time to Gyr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

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

galpy.util.bovy_conversion.velocity_in_kpcGyr

galpy.util.bovy_conversion.**velocity_in_kpcGyr**(*vo*, *ro*)

NAME:

velocity_in_kpcGyr

PURPOSE:

convert a velocity to kpc/Gyr

INPUT:

vo - velocity unit in km/s

ro - length unit in kpc

OUTPUT:

conversion from units where vo=1. at ro=1.

HISTORY:

2014-12-19 - Written - Bovy (IAS)

3.5.4 galpy.util.bovy_coords

Warning: `galpy` uses a left-handed coordinate frame, as is common in studies of the kinematics of the Milky Way. Care should be taken when using the coordinate transformation routines below for coordinates in a right-handed frame, the routines do not always apply and are only tested for the standard `galpy` left-handed frame.

Various coordinate transformation routines with fairly self-explanatory names:

galpy.util.bovy_coords.cov_dvrpmllbb_to_vxyz

galpy.util.bovy_coords.**cov_dvrpmllbb_to_vxyz**(*d*, *e_d*, *e_vr*, *pmll*, *pmbb*, *cov_pmllbb*, *l*, *b*, *plx=False*, *degree=False*)

NAME:

cov_dvrpmllbb_to_vxyz

PURPOSE:

propagate distance, radial velocity, and proper motion uncertainties to Galactic coordinates

INPUT:

d - distance [kpc, as/mas for plx]

e_d - distance uncertainty [kpc, [as/mas] for plx]

e_vr - low velocity uncertainty [km/s]

pmll - proper motion in l (*cos(b)) [[as/mas]/yr]

pmbb - proper motion in b [[as/mas]/yr]

cov_pmllbb - uncertainty covariance for proper motion [pmll is pmll x cos(b)]

l - Galactic longitude

b - Galactic lattitude

KEYWORDS:

plx - if True, d is a parallax, and e_d is a parallax uncertainty

degree - if True, l and b are given in degree

OUTPUT:

cov(vx,vy,vz) [3,3] or [:,3,3]

HISTORY:

2010-04-12 - Written - Bovy (NYU)

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 (multplied with cos(dec)) and dec [2,2] or [:,2,2]

ra - right ascension

dec - declination

degree - if True, ra and dec are given in degrees (default=False)

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

covar_pmllbb [2,2] or [:,2,2] [pmll here is pmll x cos(b)]

HISTORY:

2010-04-12 - Written - Bovy (NYU)

galpy.util.bovy_coords.cyl_to_rect

galpy.util.bovy_coords.**cyl_to_rect**(*R*, *phi*, *Z*)

NAME:

cyl_to_rect

PURPOSE:

convert from cylindrical to rectangular coordinates

INPUT:

R, phi, Z - cylindrical coordinates

OUTPUT:

X,Y,Z

HISTORY:

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

galpy.util.bovy_coords.cyl_to_rect_vec

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

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

galpy.util.bovy_coords.**galcencyl_to_XYZ**(*R*, *phi*, *Z*, *Xsun=1.0*, *Zsun=0.0*)

NAME:

galcencyl_to_XYZ

PURPOSE:

transform cylindrical Galactocentric coordinates to XYZ coordinates (wrt Sun)

INPUT:

R, phi, Z - Galactocentric cylindrical coordinates

Xsun - cylindrical distance to the GC (can be array of same length as R)

Zsun - Sun's height above the midplane (can be array of same length as R)

OUTPUT:

X,Y,Z

HISTORY:

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

2017-10-24 - Allowed Xsun/Zsun to be arrays - Bovy (UofT)

galpy.util.bovy_coords.galcencyl_to_vxvyvz

galpy.util.bovy_coords.**galcencyl_to_vxvyvz**(*vR, vT, vZ, phi, vsun=[0.0, 1.0, 0.0], Xsun=1.0, Zsun=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 in the GC frame ndarray[3] (can be array of same length as vRg; shape [3,N])

Xsun - cylindrical distance to the GC (can be array of same length as vRg)

Zsun - Sun's height above the midplane (can be array of same length as vRg)

OUTPUT:

vx,vy,vz

HISTORY:

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

2017-10-24 - Allowed vsun/Xsun/Zsun to be arrays - Bovy (NYU)

galpy.util.bovy_coords.galcenrect_to_XYZ

galpy.util.bovy_coords.**galcenrect_to_XYZ**(*X*, *Y*, *Z*, *Xsun=1.0*, *Zsun=0.0*)

NAME:

galcenrect_to_XYZ

PURPOSE:

transform rectangular Galactocentric to XYZ coordinates (wrt Sun) coordinates

INPUT:

X, Y, Z - Galactocentric rectangular coordinates

Xsun - cylindrical distance to the GC (can be array of same length as X)

Zsun - Sun's height above the midplane (can be array of same length as X)

OUTPUT:

(X, Y, Z)

HISTORY:

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

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

2017-10-24 - Allowed Xsun/Zsun to be arrays - Bovy (UofT)

galpy.util.bovy_coords.galcenrect_to_vxvyvz

galpy.util.bovy_coords.**galcenrect_to_vxvyvz**(*vXg, vYg, vZg, vsun=[0.0, 1.0, 0.0], Xsun=1.0, Zsun=0.0*)

NAME:

galcenrect_to_vxvyvz

PURPOSE:

transform rectangular Galactocentric coordinates to XYZ coordinates (wrt Sun) for velocities

INPUT:

vXg - Galactocentric x-velocity

vYg - Galactocentric y-velocity

vZg - Galactocentric z-velocity

vsun - velocity of the sun in the GC frame ndarray[3] (can be array of same length as vXg; shape [3,N])

Xsun - cylindrical distance to the GC (can be array of same length as vXg)

Zsun - Sun's height above the midplane (can be array of same length as vXg)

OUTPUT:

[:,3]= vx, vy, vz

HISTORY:

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

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

2017-10-24 - Allowed vsun/Xsun/Zsun to be arrays - Bovy (UofT)

galpy.util.bovy_coords.lb_to_radec

galpy.util.bovy_coords.**lb_to_radec**(*l*, *b*, *degree=False*, *epoch=2000.0*)

NAME:

lb_to_radec

PURPOSE:

transform from Galactic coordinates to equatorial coordinates

INPUT:

l - Galactic longitude

b - Galactic lattitude

degree - (Bool) if True, l and b are given in degree and ra and dec will be as well

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

ra,dec

For vector inputs [:,2]

HISTORY:

2010-04-07 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

2016-05-13 - Added support for using astropy's coordinate transformations and for non-standard epochs - Bovy (UofT)

galpy.util.bovy_coords.lb_to_radec

galpy.util.bovy_coords.**lbd_to_XYZ**(*l*, *b*, *d*, *degree=False*)

NAME:

lbd_to_XYZ

PURPOSE:

transform from spherical Galactic coordinates to rectangular Galactic coordinates (works with vector inputs)

INPUT:

l - Galactic longitude (rad)

b - Galactic lattitude (rad)

d - distance (arbitrary units)

degree - (bool) if True, l and b are in degrees

OUTPUT:

[X,Y,Z] in whatever units d was in

For vector inputs [:,3]

HISTORY:

2009-10-24- Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.bovy_coords.pmllpmbb_to_pmrapmdec

galpy.util.bovy_coords.**pmllpmbb_to_pmrapmdec**(*pmll*, *pmbb*, *l*, *b*, *degree=False*, *epoch=2000.0*)

NAME:

pmllpmbb_to_pmrapmdec

PURPOSE:

rotate proper motions in (l,b) into proper motions in (ra,dec)

INPUT:

pmll - proper motion in l (multplied with cos(b)) [mas/yr]

pmbb - proper motion in b [mas/yr]

l - Galactic longitude

b - Galactic lattitude

degree - if True, l and b are given in degrees (default=False)

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

(pmra x cos(dec),pmdec), for vector inputs [:,2]

HISTORY:

2010-04-07 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.bovy_coords.pmrapmdec_to_pmllpmbb

galpy.util.bovy_coords.**pmrapmdec_to_pmllpmbb**(*pmra*, *pmdec*, *ra*, *dec*, *degree=False*, *epoch=2000.0*)

NAME:

pmrapmdec_to_pmllpmbb

PURPOSE:

rotate proper motions in (ra,dec) into proper motions in (l,b)

INPUT:

pmra - proper motion in ra (multplied with cos(dec)) [mas/yr]

pmdec - proper motion in dec [mas/yr]

ra - right ascension

dec - declination

degree - if True, ra and dec are given in degrees (default=False)

epoch - epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

(pmll x cos(b),pmbb) for vector inputs [:,2]

HISTORY:

2010-04-07 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.bovy_coords.pmrapmdec_to_custom

galpy.util.bovy_coords.**pmrapmdec_to_custom**(*pmra*, *pmdec*, *ra*, *dec*, *T=None*, *degree=False*, *epoch=2000.0*)

NAME:

pmrapmdec_to_custom

PURPOSE:

rotate proper motions in (ra,dec) to proper motions in a custom set of sky coordinates (phi1,phi2)

INPUT:

pmra - proper motion in ra (multplied with cos(dec)) [mas/yr]

pmdec - proper motion in dec [mas/yr]

ra - right ascension

dec - declination

T= matrix defining the transformation: new_rect= T dot old_rect, where old_rect = [cos(dec)cos(ra),cos(dec)sin(ra),sin(dec)] and similar for new_rect

degree= (False) if True, ra and dec are given in degrees (default=False)

epoch= (2000.) epoch of ra,dec (right now only 2000.0 and 1950.0 are supported when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

(pmphi1 x cos(phi2),pmph2) for vector inputs [:,2]

HISTORY:

2016-10-24 - Written - Bovy (UofT/CCA)

galpy.util.bovy_coords.pupv_to_vRvz

galpy.util.bovy_coords.**pupv_to_vRvz**(*pu*, *pv*, *u*, *v*, *delta=1.0*, *oblate=False*)

NAME:

pupv_to_vRvz

PURPOSE:

calculate cylindrical vR and vz from momenta in prolate or oblate confocal u and v coordinates for a given focal length delta

INPUT:

pu - u momentum

pv - v momentum

u - u coordinate

v - v coordinate

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(vR,vz)

HISTORY:

2017-12-04 - Written - Bovy (UofT)

galpy.util.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 when not using astropy's transformations internally; when internally using astropy's coordinate transformations, epoch can be None for ICRS, 'JXXXX' for FK5, and 'BXXXX' for FK4)

OUTPUT:

l,b

For vector inputs [:,2]

HISTORY:

2009-11-12 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

2016-05-13 - Added support for using astropy's coordinate transformations and for non-standard epochs - Bovy (UofT)

galpy.util.bovy_coords.radec_to_custom

galpy.util.bovy_coords.**radec_to_custom**(*ra*, *dec*, *T=None*, *degree=False*, *epoch=2000.0*)

NAME:

radec_to_custom

PURPOSE:

transform from equatorial coordinates to a custom set of sky coordinates

INPUT:

ra - right ascension

dec - declination

T= matrix defining the transformation: new_rect= T dot old_rect, where old_rect = [cos(dec)cos(ra),cos(dec)sin(ra),sin(dec)] and similar for new_rect

degree - (Bool) if True, ra and dec are given in degree and l and b will be as well

OUTPUT:

custom longitude, custom latitude (with longitude -180 to 180)

For vector inputs [:,2]

HISTORY:

2009-11-12 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

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 x cos(b),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:

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

Y - Y

Z - Z

cyl - if True, X,Y,Z are already cylindrical

OUTPUT:

vR,vT,vz

HISTORY:

2010-09-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.rphi_to_dl_2d

galpy.util.bovy_coords.**rphi_to_dl_2d**(*R*, *phi*, *degree=False*, *ro=1.0*, *phio=0.0*)

NAME:

rphi_to_dl_2d

PURPOSE:

convert Galactocentric radius and azimuth to distance and Galactic longitude

INPUT:

R - Galactocentric radius

phi - Galactocentric azimuth [rad/deg if degree]

KEYWORDS:

degree= (False): phi is in degrees rather than rad

ro= (1) Galactocentric radius of the observer

phio= (0) Galactocentric azimuth of the observer [rad/deg if degree]

OUTPUT:

(d,l); phi in degree if degree

HISTORY:

2012-01-04 - Written - Bovy (IAS)

galpy.util.bovy_coords.Rz_to_coshucosv

galpy.util.bovy_coords.**Rz_to_coshucosv**(*R*, *z*, *delta=1.0*, *oblate=False*)

NAME:

Rz_to_coshucosv

PURPOSE:

calculate prolate confocal cosh(u) and cos(v) coordinates from R,z, and delta

INPUT:

R - radius

z - height

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(cosh(u),cos(v))

HISTORY:

2012-11-27 - Written - Bovy (IAS)

2017-10-11 - Added oblate coordinates - Bovy (UofT)

galpy.util.bovy_coords.Rz_to_uv

galpy.util.bovy_coords.**Rz_to_uv**(*R*, *z*, *delta=1.0*, *oblate=False*)

NAME:

Rz_to_uv

PURPOSE:

calculate prolate or oblate confocal u and v coordinates from R,z, and delta

INPUT:

R - radius

z - height

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(u,v)

HISTORY:

2012-11-27 - Written - Bovy (IAS)

2017-10-11 - Added oblate coordinates - Bovy (UofT)

galpy.util.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)

HISTORY:

2009-10-25 - Written - Bovy (NYU)

galpy.util.bovy_coords.uv_to_Rz

galpy.util.bovy_coords.**uv_to_Rz**(*u*, *v*, *delta=1.0*, *oblate=False*)

NAME:

uv_to_Rz

PURPOSE:

calculate R and z from prolate confocal u and v coordinates

INPUT:

u - confocal u

v - confocal v

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

OUTPUT:

(R,z)

HISTORY:

2012-11-27 - Written - Bovy (IAS)

2017-10-11 - Added oblate coordinates - Bovy (UofT)

galpy.util.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)

For vector inputs [:,3]

HISTORY:

2009-10-24 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.bovy_coords.vRvz_to_pupv

galpy.util.bovy_coords.**vRvz_to_pupv**(*vR*, *vz*, *R*, *z*, *delta=1.0*, *oblate=False*, *uv=False*)

NAME:

vRvz_to_pupv

PURPOSE:

calculate momenta in prolate or oblate confocal u and v coordinates from cylindrical velocities vR,vz for a given focal length delta

INPUT:

vR - radial velocity in cylindrical coordinates

vz - vertical velocity in cylindrical coordinates

R - radius

z - height

delta= focus

oblate= (False) if True, compute oblate confocal coordinates instead of prolate

uv= (False) if True, the given R,z are actually u,v

OUTPUT:

(pu,pv)

HISTORY:

2017-11-28 - Written - Bovy (UofT)

galpy.util.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], Xsun=1.0, Zsun=0.0, galcen=False*)

NAME:

vxvyvz_to_galcencyl

PURPOSE:

transform velocities in XYZ coordinates (wrt Sun) to cylindrical Galactocentric coordinates for velocities

INPUT:

vx - U

vy - V

vz - W

X - X in Galactocentric rectangular coordinates

Y - Y in Galactocentric rectangular coordinates

Z - Z in Galactocentric rectangular coordinates

vsun - velocity of the sun in the GC frame ndarray[3]

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

galcen - if True, then X,Y,Z are in cylindrical Galactocentric coordinates rather than rectangular coordinates

OUTPUT:

vRg, vTg, vZg

HISTORY:

2010-09-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.vxvyvz_to_galcenrect

galpy.util.bovy_coords.**vxvyvz_to_galcenrect**(*vx, vy, vz, vsun=[0.0, 1.0, 0.0], Xsun=1.0, Zsun=0.0*)

NAME:

vxvyvz_to_galcenrect

PURPOSE:

transform velocities in XYZ coordinates (wrt Sun) to rectangular Galactocentric coordinates for velocities

INPUT:

vx - U

vy - V

vz - W

vsun - velocity of the sun in the GC frame ndarray[3]

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

OUTPUT:

[:,3]= vXg, vYg, vZg

HISTORY:

2010-09-24 - Written - Bovy (NYU)

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

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 lattitude

d - distance (kpc)

XYZ - (bool) If True, then l,b,d is actually X,Y,Z (rectangular Galactic coordinates)

degree - (bool) if True, l and b are in degrees

OUTPUT:

(vr,pmll x cos(b),pmbb) in (km/s,mas/yr,mas/yr); pmll = mu_l * cos(b)

For vector inputs [:,3]

HISTORY:

2009-10-24 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

galpy.util.bovy_coords.XYZ_to_galcencyl

galpy.util.bovy_coords.**XYZ_to_galcencyl**(*X*, *Y*, *Z*, *Xsun=1.0*, *Zsun=0.0*)

NAME:

XYZ_to_galcencyl

PURPOSE:

transform XYZ coordinates (wrt Sun) to cylindrical Galactocentric coordinates

INPUT:

X - X

Y - Y

Z - Z

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

OUTPUT:

R,phi,z

HISTORY:

2010-09-24 - Written - Bovy (NYU)

galpy.util.bovy_coords.XYZ_to_galcenrect

galpy.util.bovy_coords.**XYZ_to_galcenrect**(*X*, *Y*, *Z*, *Xsun=1.0*, *Zsun=0.0*)

NAME:

XYZ_to_galcenrect

PURPOSE:

transform XYZ coordinates (wrt Sun) to rectangular Galactocentric coordinates

INPUT:

X - X

Y - Y

Z - Z

Xsun - cylindrical distance to the GC

Zsun - Sun's height above the midplane

OUTPUT:

(Xg, Yg, Zg)

HISTORY:

2010-09-24 - Written - Bovy (NYU)

2016-05-12 - Edited to properly take into account the Sun's vertical position; dropped Ysun keyword - Bovy (UofT)

galpy.util.bovy_coords.XYZ_to_lbd

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 or degree,rad or degree,kpc)

For vector inputs [:,3]

HISTORY:

2009-10-24 - Written - Bovy (NYU)

2014-06-14 - Re-written w/ numpy functions for speed and w/ decorators for beauty - Bovy (IAS)

3.5.5 galpy.util.bovy_ars.bovy_ars

galpy.util.bovy_ars.**bovy_ars**(*domain*, *isDomainFinite*, *abcissae*, *hx*, *hpx*, *nsamples=1*, *hxparams=()*, *maxn=100*)

bovy_ars: Implementation of the Adaptive-Rejection Sampling algorithm by Gilks & Wild (1992): Adaptive Rejection Sampling for Gibbs Sampling, Applied Statistics, 41, 337 Based on Wild & Gilks (1993), Algorithm AS 287: Adaptive Rejection Sampling from Log-concave Density Functions, Applied Statistics, 42, 701

Input:

domain - [.,.] upper and lower limit to the domain

isDomainFinite - [.,.] is there a lower/upper limit to the domain?

abcissae - initial list of abcissae (must lie on either side of the peak in hx if the domain is unbounded

hx - function that evaluates h(x) = ln g(x)

hpx - function that evaluates hp(x) = d h(x) / d x

nsamples - (optional) number of desired samples (default=1)

hxparams - (optional) a tuple of parameters for h(x) and h'(x)

maxn - (optional) maximum number of updates to the hull (default=100)

Output:

list with nsamples of samples from exp(h(x))

External dependencies:

math scipy scipy.stats

History: 2009-05-21 - Written - Bovy (NYU)

CHAPTER 4

Acknowledging galpy

If you use galpy in a publication, please cite the following paper

- *galpy: A Python Library for Galactic Dynamics*, Jo Bovy (2015), *Astrophys. J. Supp.*, **216**, 29 (arXiv/1412.3451).

and link to `http://github.com/jobovy/galpy`. Some of the code's functionality is introduced in separate papers:

- `galpy.actionAngle.EccZmaxRperiRap` and `galpy.orbit.Orbit` methods with `analytic=True`: Fast method for computing orbital parameters from *this section*: please cite Mackereth & Bovy (2018).
- `galpy.actionAngle.actionAngleAdiabatic`: please cite Binney (2010).
- `galpy.actionAngle.actionAngleStaeckel`: please cite Bovy & Rix (2013) and Binney (2012).
- `galpy.actionAngle.actionAngleIsochroneApprox`: please cite Bovy (2014).
- `galpy.df.streamdf`: please cite Bovy (2014).
- `galpy.df.streamgapdf`: please cite Sanders, Bovy, & Erkal (2016).

Please also send me a reference to the paper or send a pull request including your paper in the list of galpy papers on this page (this page is at doc/source/index.rst). Thanks!

CHAPTER 5

Papers using galpy

`galpy` is described in detail in this publication:

- *galpy: A Python Library for Galactic Dynamics*, Jo Bovy (2015), *Astrophys. J. Supp.*, **216**, 29 (2015ApJS..216...29B).

The following is a list of publications using `galpy`; please let me (bovy at astro dot utoronto dot ca) know if you make use of `galpy` in a publication.

1. ***Tracing the Hercules stream around the Galaxy*, Jo Bovy (2010), *Astrophys. J.* 725, 1676 (2010ApJ...725.1676B):**
   Uses what later became the orbit integration routines and Dehnen and Shu disk distribution functions.

2. ***The spatial structure of mono-abundance sub-populations of the Milky Way disk*, Jo Bovy, Hans-Walter Rix, Chao Liu, et al.**
   Employs galpy orbit integration in `galpy.potential.MWPotential` to characterize the orbits in the SEGUE G dwarf sample.

3. ***On the local dark matter density*, Jo Bovy & Scott Tremaine (2012), *Astrophys. J.* 756, 89 (2012ApJ...756...89B):**
   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.

4. ***The Milky Way's circular velocity curve between 4 and 14 kpc from APOGEE data*, Jo Bovy, Carlos Allende Prieto, Timothy**
   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.

5. ***A direct dynamical measurement of the Milky Way's disk surface density profile, disk scale length, and dark matter profile at 4***
   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.

6. ***The peculiar pulsar population of the central parsec*, Jason Dexter & Ryan M. O'Leary (2013), *Astrophys. J. Lett.*, 783, L7 (**
   Uses galpy for orbit integration of pulsars kicked out of the Galactic center.

7. ***Chemodynamics of the Milky Way. I. The first year of APOGEE data*, Friedrich Anders, Christina Chiappini, Basilio X. San**
   Employs galpy to perform orbit integrations in `galpy.potential.MWPotential` to characterize the orbits of stars in the APOGEE sample.

8. ***Dynamical modeling of tidal streams*, Jo Bovy (2014), *Astrophys. J.*, 795, 95 (2014ApJ...795...95B):**
   Introduces `galpy.df.streamdf` and `galpy.actionAngle.`

`actionAngleIsochroneApprox` for modeling tidal streams using simple models formulated in action-angle space (see the tutorial above).

9. ***The Milky Way Tomography with SDSS. V. Mapping the Dark Matter Halo*, Sarah R. Loebman, Zeljko Ivezic Thomas R. Qu**
   Uses `galpy.potential` functions to calculate the acceleration field of the best-fit potential in Bovy & Rix (2013) above.

10. ***The Proper Motion of the Galactic Center Pulsar Relative to Sagittarius A*, Geoffrey C. Bower, Adam Deller, Paul Demores**
    Utilizes `galpy.orbit` integration in Monte Carlo simulations of the possible origin of the pulsar PSR J1745-2900 near the black hole at the center of the Milky Way.

11. ***The power spectrum of the Milky Way: Velocity fluctuations in the Galactic disk*, Jo Bovy, Jonathan C. Bird, Ana E. Garcia P**
    Uses `galpy.df.evolveddiskdf` to calculate the mean non-axisymmetric velocity field due to different non-axisymmetric perturbations and compares it to APOGEE data.

12. ***The LMC geometry and outer stellar populations from early DES data*, Eduardo Balbinot, B. X. Santiago, L. Girardi, et al. (**
    Employs `galpy.potential.MWPotential` as a mass model for the Milky Way to constrain the mass of the LMC.

13. ***Generation of mock tidal streams*, Mark A. Fardal, Shuiyao Huang, & Martin D. Weinberg (2015), *Mon. Not. Roy. Astron.***
    Uses `galpy.potential` and `galpy.orbit` for orbit integration in creating a *particle-spray* model for tidal streams.

14. ***The nature and orbit of the Ophiuchus stream*, Branimir Sesar, Jo Bovy, Edouard J. Bernard, et al. (2015), *Astrophys. J.*, 80**
    Uses the `Orbit.fit` routine in `galpy.orbit` to fit the orbit of the Ophiuchus stream to newly obtained observational data and the routines in `galpy.df.streamdf` to model the creation of the stream.

15. ***Young Pulsars and the Galactic Center GeV Gamma-ray Excess*, Ryan M. O'Leary, Matthew D. Kistler, Matthew Kerr, & J**
    Uses galpy orbit integration and `galpy.potential.MWPotential2014` as part of a Monte Carlo simulation of the Galactic young-pulsar population.

16. ***Phase Wrapping of Epicyclic Perturbations in the Wobbly Galaxy*, Alexander de la Vega, Alice C. Quillen, Jeffrey L. Carlin,**
    Employs galpy orbit integration, `galpy.potential` functions, and `galpy.potential.MWPotential2014` to investigate epicyclic motions induced by the pericentric passage of a large dwarf galaxy and how these motions give rise to streaming motions in the vertical velocities of Milky Way disk stars.

17. ***Chemistry of the Most Metal-poor Stars in the Bulge and the z 10 Universe*, Andrew R. Casey & Kevin C. Schlaufman (2015**
    This paper employs galpy orbit integration in `MWPotential` to characterize the orbits of three very metal-poor stars in the Galactic bulge.

18. *The Phoenix stream: a cold stream in the Southern hemisphere*, E. Balbinot, B. Yanny, T. S. Li, et al. (2015), *Astrophys. J.*, **820**, 58 (2016ApJ...820...58B).

19. ***Discovery of a Stellar Overdensity in Eridanus-Phoenix in the Dark Energy Survey*, T. S. Li, E. Balbinot, N. Mondrik, et al. (**
    Both of these papers use galpy orbit integration to integrate the orbit of NGC 1261 to investigate a possible association of this cluster with the newly discovered Phoenix stream and Eridanus-Phoenix overdensity.

20. ***The Proper Motion of Palomar 5*, T. K. Fritz & N. Kallivayalil (2015), *Astrophys. J.*, 811, 123 (2015ApJ...811..123F):**
    This paper makes use of the `galpy.df.streamdf` model for tidal streams to constrain the Milky Way's gravitational potential using the kinematics of the Palomar 5 cluster and stream.

21. ***Spiral- and bar-driven peculiar velocities in Milky Way-sized galaxy simulations*, Robert J. J. Grand, Jo Bovy, Daisuke Kawa**
    Uses `galpy.df.evolveddiskdf` to calculate the mean non-axisymmetric velocity field due to the bar in different parts of the Milky Way.

22. ***Vertical kinematics of the thick disc at 4.5 R 9.5 kpc*, Kohei Hattori & Gerard Gilmore (2015), *Mon. Not. Roy. Astron. Soc.***
    This paper uses `galpy.potential` functions to set up a realistic Milky-Way potential for investigating the kinematics of stars in the thick disk.

23. ***Local Stellar Kinematics from RAVE data - VI. Metallicity Gradients Based on the F-G Main-sequence Stars*, O. Plevne, T. A**
This paper employs galpy orbit integration in `MWPotential2014` to calculate orbital parameters for a sample of RAVE F and G dwarfs to investigate the metallicity gradient in the Milky Way.

24. ***Dynamics of stream-subhalo interactions*, Jason L. Sanders, Jo Bovy, & Denis Erkal (2015), *Mon. Not. Roy. Astron. Soc.*, 45**
Uses and extends `galpy.df.streamdf` to build a generative model of the dynamical effect of sub-halo impacts on tidal streams. This new functionality is contained in `galpy.df.streamgapdf`, a subclass of `galpy.df.streamdf`, and can be used to efficiently model the effect of impacts on the present-day structure of streams in position and velocity space.

25. ***Extremely metal-poor stars from the cosmic dawn in the bulge of the Milky Way*, L. M. Howes, A. R. Casey, M. Asplund, et a**
Employs galpy orbit integration in `MWPotential2014` to characterize the orbits of a sample of extremely metal-poor stars found in the bulge of the Milky Way. This analysis demonstrates that the orbits of these metal-poor stars are always close to the center of the Milky Way and that these stars are therefore true bulge stars rather than halo stars passing through the bulge.

26. ***Detecting the disruption of dark-matter halos with stellar streams*, Jo Bovy (2016), *Phys. Rev. Lett.*, 116, 121301 (2016PhRvL**
Uses galpy functions in `galpy.df` to estimate the velocity kick imparted by a disrupting dark-matter halo on a stellar stream. Also employs `galpy.orbit` integration and `galpy.actionAngle` functions to analyze N-body simulations of such an interaction.

27. ***Identification of Globular Cluster Stars in RAVE data II: Extended tidal debris around NGC 3201*, B. Anguiano, G. M. De Si**
Employs `galpy.orbit` integration to study the orbits of potential tidal-debris members of NGC 3201.

28. ***Young and Millisecond Pulsar GeV Gamma-ray Fluxes from the Galactic Center and Beyond*, Ryan M. O'Leary, Matthew D**
Uses `galpy.orbit` integration in `MWPotential2014` for orbit integration of pulsars kicked out of the central region of the Milky Way.

29. ***Abundances and kinematics for ten anticentre open clusters*, T. Cantat-Gaudin, P. Donati, A. Vallenari, R. Sordo, A. Bragag**
Uses `galpy.orbit` integration in `MWPotential2014` to characterize the orbits of 10 open clusters located toward the Galactic anti-center, finding that the most distant clusters have high-eccentricity orbits.

30. ***A Magellanic Origin of the DES Dwarfs*, P. Jethwa, D. Erkal, & V. Belokurov (2016), *Mon. Not. Roy. Astron. Soc.*, 461, 221**
Employs the C implementations of `galpy.potentials` to compute forces in orbit integrations of the LMC's satellite-galaxy population.

31. *PSR J1024-0719: A Millisecond Pulsar in an Unusual Long-Period Orbit*, D. L. Kaplan, T. Kupfer, D. J. Nice, et al. (2016), *Astrophys. J.*, **826**, 86 (arXiv/1604.00131):

32. ***A millisecond pulsar in an extremely wide binary system*, C. G. Bassa, G. H. Janssen, B. W. Stappers, et al. (2016), *Mon. Not***
Both of these papers use `galpy.orbit` integration in `MWPotential2014` to determine the orbit of the milli-second pulsar PSR J10240719, a pulsar in an unusual binary system.

33. ***The first low-mass black hole X-ray binary identified in quiescence outside of a globular cluster*, B. E. Tetarenko, A. Bahrami**
This paper employs `galpy.orbit` integration of orbits within the position-velocity uncertainty ellipse of the radio source VLA J213002.08+120904 to help characterize its nature (specifically, to rule out that it is a magnetar based on its birth location).

34. ***Action-based Dynamical Modelling for the Milky Way Disk*, Wilma H. Trick, Jo Bovy, & Hans-Walter Rix (2016), *Astrophys***
Makes use of potential models, the Staeckel actionAngle modules, and the quasiisothermal DF to develop a robust dynamical modeling approach for recovering the Milky Way's gravitational potential from kinematics of disk stars.

35. ***A Dipole on the Sky: Predictions for Hypervelocity Stars from the Large Magellanic Cloud*, Douglas Boubert & N. W. Evans**
Uses `galpy.orbit` integration to investigate the orbits of hyper-velocity stars that could be ejected from the Large Magellanic Cloud and their distribution on the sky.

36. ***Linear perturbation theory for tidal streams and the small-scale CDM power spectrum*, Jo Bovy, Denis Erkal, & Jason L. San**
Uses and extends `galpy.df.streamdf` and `galpy.df.streamgapdf` to quickly compute the

effect of impacts from dark-matter subhalos on stellar streams and investigates the structure of perturbed streams and how this structure relates to the CDM subhalo mass spectrum.

37. ***Local Stellar Kinematics from RAVE data - VII. Metallicity Gradients from Red Clump Stars*, O. Onal Tas, S. Bilir, G. M. Sea**
Employs galpy orbit integration in `MWPotential2014` to calculate orbital parameters for a sample of red clump stars in RAVE to investigate the metallicity gradient in the Milky Way.

38. ***Study of Eclipsing Binary and Multiple Systems in OB Associations IV: Cas OB6 Member DN Cas*, V. Bakis, H. Bakis, S. Bili**
Uses galpy orbit integration in `MWPotential2014` to calculate the orbit and orbital parameters of the spectroscopic binary DN Cas in the Milky Way.

39. ***The shape of the inner Milky Way halo from observations of the Pal 5 and GD-1 stellar streams*, Jo Bovy, Anita Bahmanyar,**
Makes use of the `galpy.df.streamdf` model for a tidal stream to constrain the shape and mass of the Milky Way's dark-matter halo. Introduced `galpy.potential.TriaxialNFWPotential`.

40. ***The Rotation-Metallicity Relation for the Galactic Disk as Measured in the Gaia DR1 TGAS and APOGEE Data*, Carlos Alle**
Employs orbit integration in `MWPotential2014` to calculate the orbits of a sample of stars in common between Gaia DR1's TGAS and APOGEE to study the rotation-metallicity relation for the Galactic disk.

41. ***Detection of a dearth of stars with zero angular momentum in the solar neighbourhood*, Jason A. S. Hunt, Jo Bovy, & Raym**
Uses `galpy.orbit` integration in `MWPotential2014` plus a hard Galactic core to calculate the orbits of stars in the solar neighborhood and predict how many of them should be lost to chaos.

42. ***Differences in the rotational properties of multiple stellar populations in M 13: a faster rotation for the "extreme" chemical su***
Employs `galpy.orbit` integration in `MWPotential2014` to investigate the orbit of the globular cluster M13 and in particular whether escaping stars from the cluster could contaminate the measurement of the rotation of different populations in the cluster.

43. ***Using the Multi-Object Adaptive Optics demonstrator RAVEN to observe metal-poor stars in and towards the Galactic Centre*,**
Uses `galpy.orbit` integration in `MWPotential2014` to characterize the orbits of three very metal-poor stars observed toward the Galactic center, to determine whether they are likely bulge members.

44. ***The Radial Velocity Experiment (RAVE): Fifth Data Release*, Andrea Kunder, Georges Kordopatis, Matthias Steinmetz, et a**
Employs `galpy.orbit` integration to characterize the orbits of stars in the RAVE survey.

45. ***The Proper Motion of Pyxis: the first use of Adaptive Optics in tandem with HST on a faint halo object*, Tobias K. Fritz, Sean**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the orbit of the globular cluster Pyxis using its newly measured proper motion and to search for potential streams associated with the cluster.

46. ***The Galactic distribution of X-ray binaries and its implications for compact object formation and natal kicks*, Serena Repetto,**
Uses `galpy.orbit` integration in `MWPotential2014` and that of Paczynski (1990) to study the orbits of X-ray binaries under different assumptions about their formation mechanism and natal velocity kicks.

47. ***Kinematics of Subluminous O and B Stars by Surface Helium Abundance*, P. Martin, C. S. Jeffery, Naslim N., & V. M. Wool**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the orbits of different types of low-mass core-helium-burning stars.

48. ***Is there a disk of satellites around the Milky Way?*, Moupiya Maji, Qirong Zhu, Federico Marinacci, & Yuexing Li (2017), s**
Employs `galpy.orbit` integration in `MWPotential2014` to predict the future paths of 11 classical Milky-Way satellites to investigate whether they remain in a disk configuration.

49. ***The devil is in the tails: the role of globular cluster mass evolution on stream properties*, Eduardo Balbinot & Mark Gieles (2**
Uses `galpy.orbit` integration in `MWPotential2014` of globular clusters in the Milky-Way halo. These integrations are used to investigate the clusters' mass loss due to tidal stripping, taking the effects of collisional dynamics in the cluster into account, and to evaluate the visibility of their (potential) tidal tails.

50. ***Absolute Ages and Distances of 22 GCs using Monte Carlo Main-Sequence Fitting*, Erin M. O'Malley, Christina Gilligan, &**
Employs `galpy.orbit` integration in `MWPotential2014` of globular clusters in the Milky Way, to study their orbits and classify them as disk or halo clusters.

51. **Siriusly, *a newly identified intermediate-age Milky Way stellar cluster: A spectroscopic study of* Gaia *1*, J. D. Simpson, G. M.**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the orbit in the Milky Way potential of a newly-confirmed stellar cluster found in the Gaia data.

52. ***Action-based Dynamical Modeling for the Milky Way Disk: The Influence of Spiral Arms*, Wilma H. Trick, Jo Bovy, Elena D**
Uses various potential models, the Staeckel actionAngle modules, and the quasiisothermal DF to test a robust dynamical modeling approach for recovering the Milky Way's gravitational potential from kinematics of disk stars against numerical simulations with spiral arms.

53. ***A spectroscopic study of the elusive globular cluster ESO452-SC11 and its surroundings*, Andreas Koch, Camilla Juul Hanse**
Employs `galpy.orbit` integration in `MWPotential2014` to investigate the orbit in the Milky Way potential of two candidate cluster members of the bulge globular cluster ESO452-SC11.

54. ***A Halo Substructure in Gaia Data Release 1*, G. C. Myeong, N. W. Evans, V. Belokurov, S. E. Koposov, & J. L. Sanders (201**
Uses `galpy.actionAngle.actionAngleAdiabatic` routines to compute the actions using the adiabatic approximation for 268,588 stars in *Gaia* DR1 *TGAS* with line-of-sight velocities from spectroscopic surveys. Detects a co-moving group of 14 stars on strongly radial orbits and computes their orbits using `MWPotential2014`.

55. ***An artificial neural network to discover Hypervelocity stars: Candidates in* Gaia *DR1/* TGAS, T. Marchetti, E. M. Rossi, G. F**
Uses `galpy.orbit` integration in a custom Milky-Way-like potential built from `galpy.potential` models to investigate the orbits of hypervelocity-star candidates in *Gaia* DR1.

56. ***GalRotpy: an educational tool to understand and parametrize the rotation curve and gravitational potential of disk-like galaxi***
These authors build an interactive tool to decompose observed rotation curves into bulge, disk (Miyamoto-Nagai or exponential), and NFW halo components on top of `galpy.potential` routines.

57. ***The AMBRE Project: formation and evolution of the Milky Way disc*, V. Grisoni, E. Spitoni, F. Matteucci, A. Recio-Blanco, I**
Uses `galpy` to compute orbital parameters for stars in the AMBRE sample of high-resolution spectra and uses these orbital parameters to aid in the comparison between the data and chemical-evolution models.

58. ***ESO452-SC11: The lowest mass globular cluster with a potential chemical inhomogeneity*, Jeffrey D. Simpson, Gayandhi De**
Uses `galpy.orbit` in `MWPotential2014` to compute the orbit of the MW bulge globular cluster ESO452-SC11.

59. ***Detailed chemical abundance analysis of the thick disk star cluster Gaia 1*, Andreas Koch, Terese T. Hansen, & Andrea Kun**
Employs `galpy.orbit` integration to compute the orbits of four red-giant members of the *Gaia 1* Milky Way star cluster, finding that the orbits of these stars are similar to those of the oldest stars in the Milky Way's disk.

60. ***Proper motions in the VVV Survey: Results for more than 15 million stars across NGC 6544*, R. Contreras Ramos, M. Zocca**
Uses `galpy.orbit` integration in `MWPotential2014` to calculate the orbit of NGC 6544, a Milky-Way globular cluster, using a newly determined proper motion, finding that it is likely a halo globular cluster based on its orbit.

61. ***How to make a mature accreting magnetar*, A. P. Igoshev & S. B. Popov (2017) *Mon. Not. Roy. Astron. Soc.*, in press (arXiv/**
Employs `galpy.orbit` integration of the magnetar candidate 4U 0114+65 in the potential model from Irrgang et al. (2013) to aid in the determination of its likely age.

62. ***iota Horologii is unlikely to be an evaporated Hyades star*, I. Ramirez, D. Yong, E. Gutierrez, M. Endl, D. L. Lambert, J.-D.**
Uses `galpy.orbit` integration in `MWPotential2014` to determine the approximate orbit of the star iota Horologii, a planet-hosting suspected former member of the Hyades cluster, to investigate whether it could have coincided with the Hyades cluster in the past.

63. ***Confirming chemical clocks: asteroseismic age dissection of the Milky Way disk(s)*, V. Silva Aguirre, M. Bojsen-Hansen, D. S**
Employs `galpy.orbit` integration in `MWPotential2014` to compute the orbits of a sample of 1989 red giants with spectroscopic and asteroseismic data from the APOKASC catalog, to shed light on the properties of stellar populations defined by age and metallicity.

64. ***The universality of the rapid neutron-capture process revealed by a possible disrupted dwarf galaxy star*, Andrew R. Casey &**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the orbit and its uncertainty of 2MASS J151113.24–213003.0, an extremely metal-poor field star with measureable r-process abundances, and of other similar metal-poor stars. The authors find that all of these stars are on highly eccentric orbits, possibly indicating that they originated in dwarf galaxies.

65. ***The Gaia-ESO Survey: Churning through the Milky Way*, M. R. Hayden, A. Recio-Blanco, P. de Laverny, et al. (2017) *Astro***
Employs `galpy.orbit` integration in `MWPotential2014` to study the orbital characteristics (eccentricity, pericentric radius) of a sample of 2,364 stars observed in the Milky Way as part of the Gaia-ESO survey.

66. ***The Evolution of the Galactic Thick Disk with the LAMOST Survey*, Chengdong Li & Gang Zhao (2017) *Astrophys. J.*, 850,**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the orbital characteristics (eccentricity, maximum height above the plane, angular momentum) of a sample of about 2,000 stars in the thicker-disk component of the Milky Way.

67. ***The Orbit and Origin of the Ultra-faint Dwarf Galaxy Segue 1*, T. K. Fritz, M. Lokken, N. Kallivayalil, A. Wetzel, S. T. Lind**
Employs `galpy.orbit` integration in `MWPotential2014` and a version of this potential with a more massive dark-matter halo to investigate the orbit and origin of the dwarf-spheroidal galaxy Segue 1 using a newly measured proper motion with SDSS and LBC data.

68. ***Prospects for detection of hypervelocity stars with Gaia*, T. Marchetti, O. Contigiani, E. M. Rossi, J. G. Albert, A. G. A. Brov**
Uses `galpy.orbit` integration in a custom Milky-Way-like potential built from `galpy.potential` models to create mock catalogs of hypervelocity stars in the Milky Way for different ejection mechanisms and study the prospects of their detection with *Gaia*.

69. ***The AMBRE project: The thick thin disk and thin thick disk of the Milky Way*, Hayden, M. R., Recio-Blanco, A., de Laverny**
Employs `galpy.orbit` integration in `MWPotential2014` to characterize the orbits of 494 nearby stars analyzed as part of the AMBRE project to learn about their distribution within the Milky Way.

70. ***KELT-21b: A Hot Jupiter Transiting the Rapidly-Rotating Metal-Poor Late-A Primary of a Likely Hierarchical Triple System*,**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the Galactic orbit of KELT-21b, a hot jupiter around a low-metallicity A-type star.

71. ***GalDynPsr: A package to estimate dynamical contributions in the rate of change of the period of radio pulsars*, Dhruv Pathal**
Presents a python package to compute contributions to the GR spin-down of pulsars from the differential galactic acceleration between the Sun and the pulsar. The package uses `MWPotential2014` and `galpy.potential` functions to help compute this.

72. ***Local Stellar Kinematics from RAVE data – VIII. Effects of the Galactic Disc Perturbations on Stellar Orbits of Red Clump S***
Employs `galpy.orbit` integration in `MWPotential2014` and the non-axisymmetric `DehnenBarPotential` and `SteadyLogSpiralPotential` to study the orbits of Milky-Way red-clump stars.

73. ***The VMC survey XXVIII. Improved measurements of the proper motion of the Galactic globular cluster 47 Tucanae*, F. Niede**
Uses `galpy.orbit` integration in `MWPotential2014` to investigate the orbit of the cluster 47 Tuc from a newly measured proper motion, finding that the orbit has an eccentricity of about 0.2 and reaches up to 3.6 kpc above the Galactic midplane.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

R

S

T

U

V

X