SMILEPOT reference

Eugene Vasiliev
Lebedev Physical Institute, Moscow, Russia
email: eugvas@lpi.ru

Version $2.5\beta_1$
February 1, 2015

SMILE is a software for orbital analysis and Schwarzschild modelling, which has, among other features, a flexible framework for representing the gravitational potential, forces and density of arbitrary models of stellar systems. The `smilepot` library contains a subset of its features – the potential solvers and some associated routines. It has a `C` and `Python` interfaces, and bindings to several other N-body software packages – `NEMO`, `AMUSE` and `galpy`. The library interface is described below; for technical details about various potentials that are available in SMILE please refer to `readme.pdf`

General methods

The `C` interface provides functions for constructing an instance of `smilePotential` structure (which is in fact an alias to a `C++` class `smile::CPotential`, but C programs shouldn't care about it), and for evaluating potential, density and forces at a given point. The `Python` interface (`py_smilepot` module) encapsulates these functions into a class `SmilePot` with the same functionality represented by class methods:
`potential(x,y,z) → float` – compute the potential at the given location, specified by a triplet of numbers;
`density(x,y,z) → float` – compute the density at the given location;
`force(x,y,z) → float[3]` – compute the force at the given location (output is a sequence of 3 numbers for x,y,z components of the force);
`force_deriv(x,y,z) → (float[3],float[6])` – compute the force and its derivative (output is a sequence of 3 components of the force, and another sequence of 6 force derivatives – $\partial F_x/\partial x, \partial F_y/\partial y, \partial F_z/\partial z, \partial F_x/\partial y, \partial F_y/\partial z, \partial F_z/\partial x$);
`name() → string` – return the SMILE potential name.
`export(string) → None` – export coefficients of potential expansion to a text file.

In the C interface, functions have the same names prefixed with `smilepot_`, and Python's `float` corresponds to `double` in C.

Initialization

Creating an instance of potential can be done in several different ways. To understand how it works, it is necessary to get a bit deeper into the methods of potential representation in SMILE.

There are several pre-defined potential–density models, such as triaxial Ferrers, and four general-purpose potential expansions that can take either an analytically given density profile, or an N-body snapshot as the input, and compute the coefficients of expansion that are used for calculating potential and its derivatives (including density). These coefficients may be stored in and loaded from a text file, to speed up initialization. Various potential parameters can be provided either as a list of arguments to the initialization routine, or read from an `ini` file. Most of them have reasonable default values. Accordingly, there are several alternative ways of constructing the potential class:

1. Analytic potential models are specified directly with a number of parameters (at the very least, potential type).

2. Potential expansions constructed from an analytic density profile are similar, with the difference that it is necessary to provide both potential (expansion) type and density model.

3. Expansions can also be initialized from an N-body snapshot that is read from a file (several file types are supported by `UNSIO` library – text, NEMO and GADGET files). It is necessary to provide potential type and file name, and optionally parameters controlling the expansion accuracy.

4. Instead of a file, one may provide the positions and masses of particles directly as parameters to the initialization routine: same as above, with the filename replaced by four arrays (x,y,z coordinates and masses).

5. Alternatively, the expansion coefficients computed before and stored in a text file can be read back from this file. In this case, only the file name is required.

6. Finally, one may provide the name of an ini file which contains all other parameters.

In Python, all these possibilities are handled by a single constructor that takes a variable number of arguments of type `key="string"` or `key=number`. The names of these arguments correspond to the names of parameters in an `ini` file (see `readme.pdf`, section 4.1). Python parameter names are case-sensitive and all are given in lowercase, in the ini file and in C interface they are not case-sensitive, and the parameter values are not case-sensitive anywhere. The parameters for Python interface are the following:

- For an analytic potential model (case 1 above): the arguments [default values] are `type="...", [q=1], [p=1], [mass=1], [scalerad=1], [scalerad2=1],`

[gamma=1], [ncoefs_angular=6], where type may be one of the following: "Dehnen", "Ferrers", "Miyamoto-Nagai", "Scale-free", "Scale-free SH", "Logarithmic", "Harmonic". q and p are the axis ratios y/x and z/x, mass is the total mass of the finite-mass models or the normalization factor of infinite-mass (the latter four) models, gamma is the exponent of the Dehnen or Scale-free (power-law) profiles, scalerad is the scale radius (for Dehnen and Ferrers) or core radius (for Logarithmic), Miyamoto–Nagai has two scale radii, commonly denoted as A(scalerad) and B(scalerad2). Scale-free profile comes in two versions – with the exact expressions for the potential (slower) and a spherical-harmonic approximation (with the order of angular expansion given by ncoefs_angular, which must be an even number).

- The four general-purpose potential expansions (cases 2–5) are given by type=:
  - basis-set expansion, coming in two variants – BSE and BSECompact. The first one uses the Zhao(1996) basis set, which is a generalization of the commonly used Hernquist–Ostriker(1992) basis set, and is suitable for infinite-extent, possibly cuspy density models (such as Dehnen). The second one, for cored density models of finite size, involves Bessel functions.
  - Spline is an alternative to BSE in which the angular part of expansion is evaluated in spherical harmonics, as in BSE, but the radial part is represented as a smooth spline-interpolated function of radius for each angular harmonic. It is generally a preferred method over BSE, as it is both faster and more accurate in most cases.
  - CylSpline is a novel method more suitable for highly flattened, possibly non-axisymmetric models (such as barred disc galaxies). It expands the azimuthal dependence of potential in Fourier harmonics, with the coefficients of expansion being specified as two-dimensional spline-interpolated functions in the meridional plane. It is considerably slower than Spline and not designed for cuspy density profiles, but in the case of strongly flattened systems is the only method that is accurate enough without going to prohibitively large number of terms.

There are several parameters that control the accuracy of the expansions.
ncoefs_radial [20] is the number of radial terms in BSE (minus one, so that zero corresponds to a single term), or the number of grid nodes in radial direction for Spline and CylSpline.
ncoefs_angular [6] is the order of angular expansion in spherical harmonics (for BSE/Spline) or Fourier azimuthal harmonics (for CylSpline).
ncoefs_vertical [20] is the grid size in z direction for CylSpline.
alpha [1] is the shape parameter of the BSE set.
rmax [1] is the radial extent of the BSECompact set.
splinermin, splinermax [0] controls the radial extent of spline interpolation nodes for both Spline and CylSpline potentials; zero means auto-adjust based on the behaviour of the density profile and the requested number of terms.

splinezmin, splinezmax [0] controls the vertical extent of interpolation grid for CylSpline; zero means auto-detect.
splinesmoothfactor [1] is the amount of smoothing applied when initializing the Spline potential from an N-body snapshot.

These methods take either an analytic density model (case 2) or an N-body snapshot (cases 3 and 4) as input.
For the case 2, the list of available models include all finite-mass models from the previous section (Dehnen, Ferrers and Miyamoto–Nagai), and additionally the following ones: "Plummer", "PerfectEllipsoid", "Isochrone", "NFW", "Sersic", "ExpDisk". The choice of the model is given by the density="..." parameter, and they may have additional parameters such as [mass=1], [scalerad=1]; Sérsic model has [sersicIndex=4], NFW and ExpDisk have scalerad2 which means the vertical scale for ExpDisk and the cutoff radius for NFW (so that the concentration is given by scalerad2/scalerad).
Additionally, two more general density models are "Ellipsoidal", which provides an arbitrary dependence of enclosed mass on radius with arbitrarily varying shape, and "MGE" (Multi-Gaussian expansion); refer to readme.pdf for their description. These two models are specified by an external text file given in file="..."

In the case 3, the input N-body snapshot is given in the parameter file="...", with density="Nbody". Various assumptions on the symmetry of the model are specified by the symmetry="..." parameter: "Spherical", "Axisymmetric", "Triaxial", "Reflection", "None", with Triaxial being the default value (they may be shortened to one letter). They control which expansion terms to use; for instance, in the triaxial case only even-degree angular terms are computed.

In the case 4, the coordinates and masses of particles that source the potential are given as parameters to the constructor: pointx, pointy, pointz, pointm, which may be either Python lists or NumPy arrays (all should be of equal length). It is not necessary to specify density in this variant, but one may still provide parameters of the expansion (and, of course, its type).
Cases 3 and 4 additionally admit another type of potential, which is not an expansion but deals with point masses directly, by using tree-code approximation. It is specified by potential type="Nbody". In general, its usage is not recommended as it is much slower than expansions and produces a lot of small-scale noise, while not necessarily approximating the "true" continuous potential better that a suitably tuned expansion. It has two additional parameters, softening length treecodeeps [-2] (negative value means proportionality to the local interparticle distance) and tree opening angle treecodetheta [0.5].

Initialization in variant 5 reads pre-computed expansion coefficients from a text file, provided in the parameter file="file.ext". In the INI file, this variant is turned on by setting density="Coefs", but in the Python or C interface it is sufficient to give the file name and not neccessary to specify potential type in this case, as it is determined from the filename extension: coef_bse, coef_bsec, coef_spl, coef_cyl.

- Finally, in variant 6 all potential parameters are provided in an INI file, section [Potential] (see readme.pdf), and the name of this file is passed to the constructor as a single argument file="file.ini". The choice between cases 5 and 6 is determined from file extension.

In short: the required arguments are type in cases 1–4, density in cases 2 and 3, and file in cases 3, 5 and 6.

The C interface has three functions for creating a potential: smilepot_create for cases 5 and 6, smilepot_create_params for cases 1–3, and smilepot_create_from_particles for case 4. The last two take the named parameters as a list of strings "key=value", in the same way as the command-line arguments are passes to the function main(int argc, char** argv) (or parameters read from the [Potential] section of an INI file). Unlike Python constructor, unknown parameters are ignored without warning.

NEMO interface

smilepot may be used as an external potential in various programs from the NEMO toolbox (most importantly, in the N-body code gyrFALCON). It is specified as two command-line parameters: accname=smilepot and accfile=...; for simplicity, only initialization options 5 and 6 are supported (all parameters set in an INI file, or in a potential expansion coefficients file).

AMUSE interface

Within the AMUSE framework, smilepot is available as a "pseudo" stellar-dynamics interface: it can provide external potential and forces in the context of bridge integrator. It is initialized as a community code SmilePot with a constructor that takes the same arguments as in py_smilepot, with the following modifications: (i) dimensional parameters (distances and masses) may have attached units, (ii) unit converter may be provided as the first (unnamed) parameter to the constructor, and (iii) in the case 4, the input points are provided as a single parameter points=... of class amuse.datamodel.Particles. It provides two functions, get_gravity_at_point (returns forces) and get_potential_at_point (returns potential), which accept an array of points.
An example of usage in the bridge code is provided in galactic_center_smilepot.py

galpy interface

smilepot can be used in the Python-based toolbox galpy as galpy.potential.SmilePotential. It is initialized in the same way as py_smilepot.SmilePot (plus the normalize argument from galpy), and has the methods for evaluating various quantities derived from its base class galpy.Potential. Currently it can be used only in Python orbit integrators due to difficulties in matching its interface to the C potential API in galpy.