AGAMA reference

Eugene Vasiliev
Oxford University
email: eugvas@lpi.ru

November 27, 2016

Contents

1 Overview **3**

2 Structure of the Agama C++ library **4**
2.1 Low-level foundations 4
2.1.1 Math routines 4
2.1.2 Units 9
2.1.3 Coordinates 10
2.1.4 Particles 10
2.1.5 Utilities 11
2.2 Potentials 11
2.2.1 Analytic potentials 12
2.2.2 Multipole expansion 13
2.2.3 Azimuthal harmonic expansion 14
2.2.4 Potential factory 15
2.2.5 Utility functions 19
2.3 Orbit integration and analysis 19
2.4 Action/angle variables 20
2.4.1 Isochrone mapping 20

2.4.2 Spherical potentials . . . 21
2.4.3 Stäckel approximation . . . 21
2.4.4 Torus mapping . . . 22
2.5 Distribution functions . . . 22
2.5.1 Spheroidal components . . . 23
2.5.2 Disky components . . . 23
2.5.3 Nonparametric interpolated models . . . 24
2.5.4 Spherical isotropic models . . . 24
2.6 Galaxy modelling framework . . . 25
2.6.1 Moments of distribution functions . . . 26
2.6.2 Velocity distribution functions . . . 26
2.6.3 Conversion to/from N-body models . . . 27
2.6.4 Iterative self-consistent modelling . . . 28
2.7 Data handling . . . 29
2.7.1 Selection functions . . . 29

3 Interfaces with other languages and frameworks 29
3.1 `Python` interface . . . 29
3.2 `Fortran` interface . . . 30
3.3 AMUSE plugin . . . 30
3.4 GALPY plugin . . . 31
3.5 NEMO plugin . . . 31

4 Tests and example programs 32

A Technical details 34
A.1 Developer's guide . . . 34
A.2 Mathematical methods . . . 42
A.2.1 B-splines . . . 42
A.2.2 Spline interpolation . . . 43
A.2.3 Penalized spline regression . . . 48
A.2.4 Penalized spline density estimate . . . 50
A.3 Potentials . . . 55
A.3.1 Multipole expansion . . . 55
A.3.2 CylSpline expansion . . . 57
A.4 Distribution functions . . . 59
A.4.1 Spherical isotropic DFs . . . 59

1 Overview

Agama (Action-based Galaxy Modelling Architecture) is a software library intended for a broad range of tasks within the field of stellar dynamics. As the name suggests, it is centered around the use of action/angle formalism to describe the structure of stellar systems, but this is only one of its many facets. The library contains a powerful framework for dealing with arbitrary density/potential profiles and distribution functions (analytic, extracted from N-body models, or fitted to the data), a vast collection of general-purpose mathematical routines, and covers many aspects of galaxy dynamics up to the very high-level interface for constructing self-consistent galaxy models. It does not provide any methods for performing N-body simulations, but may assist in their analysis.

The core of the library is written in `C++` and is organized into several modules, which are considered in turn in Section 2:

- Low-level interfaces and generic routines, which are not particularly tied to stellar dynamics: various mathematical tasks, coordinate systems, unit conversion, input/output of particle collections and configuration data, and other utilities.
- Gravitational potential and density interface: the hierarchy of classes representing density and potential models, including two very general and powerful approximations of any user-defined profile, and associated utility functions.
- Routines for numerical computation of orbits and their classification.
- Action/angle interface: classes and routines for conversion between position/velocity and action/angle variables.
- Distribution functions expressed in terms of actions.
- Galaxy modelling framework: computation of moments of distribution functions, interface for creating gravitationally self-consistent multicomponent galaxy models, construction of N-body models and mock data catalogues.
- Data handling interface, selection functions, etc.

A large part of this functionality is available in `Python` through the eponymous extension module. Many high-level tasks are more conveniently expressed in `Python`, e.g., finding best-fit parameters of potential and distribution function describing a set of data points, or constructing self-consistent models with arbitrary combination of components and constraints. A more restricted subset of functionality is provided as plugins to several other stellar-dynamical software packages (Section 3).

The library comes with an extensive collection of test and demonstration programs; some of them are internal tests that check the correctness of various code sections, others are example programs illustrating various applications and usage aspects of the library. There

are both C++ and Python programs, sometimes covering exactly the same topic; a brief review is provided in Section 4.

The main part of this document presents a comprehensive overview of various features of the library and a user's guide. The appendix contains a developer's guide and most technical aspects and mathematical details.

The code can be downloaded from https://github.com/GalacticDynamics-Oxford/Agama.

2 Structure of the Agama C++ library

2.1 Low-level foundations

2.1.1 Math routines

Agama contains an extensive mathematical subsystem covering many basic and advanced tasks. Some of the methods are implemented in external libraries (Gsl, Eigen) and have wrappers in Agama that isolate the details of implementation, so that the back-end may be switched without any changes in the higher-level code; other parts of this subsystem are self-contained developments. All classes and routines in this section belong to the math:: namespace.

Fundamental objects throughout the entire library are functions of one or many variables, vectors and matrices. Any class derived from the IFunction interface should provide a method for computing the value and up to two derivatives of a function of one variable $f(x)$; IFunctionNdim represents the interface for a vector of functions of many variables $\boldsymbol{f}(\boldsymbol{x})$, and IFunctionNdimDeriv additionally provides the Jacobian of this function (the matrix $\partial f_i/\partial x_k$). Many mathematical routines operate on an instance of a class derived from one of these interfaces.

For one-dimensional vectors we use std::vector when a dynamically-sized array is needed; some routines take input arguments of type const double[] or store the output in double[] variables which may be also statically-sized arrays (for instance, allocated on the stack, which is more efficient in tight loops).

For two-dimensional matrices there is a dedicated math::Matrix class, which provides a simple fixed interface to an implementation-dependent structure (either the Eigen matrix type, or a custom-coded flattened array with 2d indexing, if Eigen is not available). Matrices may be dense and sparse; the former provide full read-write access, while the latter are constructed from the list of non-zero elements and provide read-only access. Sparse matrices are implemented in Eigen or, in its absense, in Gsl starting from version 2.0; for older versions we substitute them internally with dense matrices (which, of course, defeats the

purpose of having a separate sparse matrix interface, but at least allows the code to compile without any modifications).

Numerical linear algebra routines in AGAMA are wrappers for either EIGEN (considerably more efficient) or GSL library. There are a few standard BLAS functions (matrix-vector and matrix-matrix multiplication for both dense and sparse matrices) and several matrix decomposition classes (`LUDecomp`, `CholeskyDecomp`, `SVDecomp`) that can be used to solve systems of linear equations $\mathsf{A}\boldsymbol{x} = \boldsymbol{b}$.

LU decomposition of a non-degenerate square matrix A (dense or sparse) into a product of lower and upper triangular matrices is the standard tool for solving full-rank systems of linear equations. Once a decomposition is created, it may be used several times with different r.h.s. vectors $\boldsymbol{b}$.

Cholesky decomposition of a symmetric positive-definite dense matrix $\mathsf{A} = \mathsf{L}\mathsf{L}^T$ serves the same purpose in this more specialized case (being twice more efficient). It is informally known as "taking the square root of a matrix": for instance, a quadratic form $\boldsymbol{x}^T\mathsf{A}\boldsymbol{x}$ may be written as $|\mathsf{L}^T\boldsymbol{x}|^2$ – this is used in the context of dealing with correlated random variables, where A would represent the correlation matrix.

Singular-value decomposition (SVD) represents a generic $M \times N$ matrix (M rows, N columns; here $M \geq N$) as $\mathsf{A} = \mathsf{U}\,\mathrm{diag}(\boldsymbol{S})\,\mathsf{V}^T$, where U is a $M \times N$ orthogonal matrix (i.e., $\mathsf{U}\mathsf{U}^T = \mathsf{I}$), V is a $N \times N$ orthogonal matrix, and the vector $\boldsymbol{S}$ contains singular values, sorted in descending order. In the case of a symmetric positive definite matrix A, SVD is identical to the eigenvalue decomposition, and $\mathsf{U} = \mathsf{V}$. SVD is considerably more costly than the other two decompositions, but it is a more powerful tool that may be applied for solving over-determined and/or rank-deficient linear systems while maintaining numerical stability. If $M > N$, there are more equations than variables, and the solution is obtained in the least-square sense; if the nullspace of the system is non-trivial (i.e., $\mathsf{A}\boldsymbol{x} = \boldsymbol{0}$ for a non-zero $\boldsymbol{x}$), the solution with the lowest possible norm is returned.

Root-finding is handled differently in one or many dimensions. `findRoot` searches for a root of a continuous one-dimensional function $f(x)$ on an interval $[a..b]$, which may be finite or infinite, provided that $f(a)\,f(b) \leq 0$ (i.e., the interval encloses the root). It uses a combination of Brent's method with an optional Hermite interpolation in the case that the function provides derivatives. `findRootNdim` searches for zeros of an N-dimensional function of N variables, which must provide the Jacobian, using a hybrid Newton-type method.

Integration of one-dimensional functions can be performed in several ways. `integrateGL` uses fixed-order Gauss–Legendre quadrature without error estimate. `integrate` uses variable-order Gauss–Kronrod scheme with the order of quadrature doubled each time until it attains the required accuracy or reaches the maximum; it is a good balance between fixed-

order and fully adaptive methods, and is very accurate for smooth analytic functions. `integrateAdaptive` handles more sophisticated integrands, possibly with singularities, using a fully adaptive recursive scheme to reach the required accuracy, but is also more expensive.

Multidimensional integration over an N-dimensional hypercube is performed by the `integrateNdim` routine, which serves as a unified interface to either CUBATURE or CUBA library; the former is actually included into the AGAMA codebase. Both methods are fully adaptive and have similar performance (either one is better on certain classes of functions). The input function may provide $M \geq 1$ values, i.e., several functions may be integrated simultaneously over the same domain.

Sampling from a probability distribution (`sampleNdim`) serves the following task: given a N-dimensional function $f(\boldsymbol{x}) \geq 0$ over a hypercube domain, construct an array of M random sample points $\boldsymbol{x}_k$ such that the density of samples in the neighborhood of any point is proportional to the value of f at that point. Obviously, the function f must have a finite integral over the entire domain, and in fact the integral may be estimated from these samples (however it is not as accurate as the deterministic cubature routines, which are allowed to attribute different weights to each sampled point). This routine uses a multidimensional variant of rejection algorithm with adaptive subdivision of the entire domain into smaller regions, and performing the rejection sampling in each region.

Optimization methods A broad range of tasks may be loosely named "optimization problems", i.e., finding a minimum of a certain function (objective) of one or many variables under certain constraints.

For a function of one variable, there is a straightforward minimization routine `findMin` that can operate on any finite or (semi-)infinite interval $[a..b]$, and finds $\min f(x)$ on this interval (including endpoints); if there are multiple minima, then one of them will be found (not necessarily the global one), depending on the initial guess. The starting point x_0 such that $f(x_0) < f(a), f(x_0) < f(b)$ may be optionally be provided by the caller; in its absense the routine will try to come up with a guess itself. Only the function values are needed by the algorithm.

For a function of N variables $\boldsymbol{x}$, there are several possibilities. If only the values of the function $f(\boldsymbol{x})$ are available, then the Nelder–Mead (simplex, or amoeba) algorithm provided by the routine `findMinNdim` may be used. If the partial derivatives $\partial f/\partial \boldsymbol{x}$ are available, they may be used in a more efficient quasi-Newton BFGS algorithm provided by the routine `findMinNdimDeriv`.

A special case of optimization problem is a non-linear least-square fit: given a function $g(\boldsymbol{x}; \boldsymbol{d})$, where x_i are N parameters that are being optimized, and d_k are M data points, minimize the sum of squared differences between the values of g at these points and target values v_k: $\min f(\boldsymbol{x}) = \sum_{k=1}^{M} [g(\boldsymbol{x}; d_k) - v_k]^2$. This task is solved by the Levenberg–Marquardt

algorithm, which needs the Jacobian matrix of partial derivatives of g w.r.t. its parameters $\boldsymbol{x}$ at each data point d_k. It is provided by the routine `nonlinearMultiFit`. Of course, if the function g is linear w.r.t. its parameters, this reduces to a simpler linear algebra problem, solved by the routine `linearMultiFit`. And if there is only one or two parameters (i.e., a linear regression with or without a constant term), this is solved by the routines `linearFit` and `linearFitZero`.

In the above sequence, more specialized problems require more knowledge about the function, but generally converge faster, although all of them may be recast in terms of a general (unconstrained) minimization problem, as demonstrated in `test_math_core.cpp`. All of them (except the linear regression routines) need a starting point or a N-dimensional neighborhood, but may move away from it in the direction of (one of possible) minima; again there is no guarantee to find the global minimum.

If there are restrictions on the values of $\boldsymbol{x}$ in the form of a matrix A of element-wise linear inequality constraints $\mathsf{A}\boldsymbol{x} \preccurlyeq \boldsymbol{b}$, and if the objective function f is linear or quadratic in the input variables, these cases are handled by the routines `linearOptimizationSolve` and `quadraticOptimizationSolve`. They depend on external libraries (GLPK and/or CVXOPT; the former can only handle linear optimization problems).

Interpolation There are various classes for performing interpolation in one, two or three dimensions. All methods are based on the concept of piecewise-polynomial functions defined by the nodes of a grid $x_0 < x_1 < \cdots < x_{N_x-1}$; in the case of multidimensional interpolation the grid is regular, i.e., aligned with the coordinate lines in each dimension. The advantages of this approach are locality (grid nodes may be more densely placed in the region of interest) and efficiency (the cost of evaluation scales as $\log(N_x)$ – time needed to locate the grid segment containing the point x, plus a constant additional cost to evaluate the interpolating polynomial on this segment).

In one or two dimensions there are linear, cubic and quintic (fifth-order) interpolation schemes. The former two are defined by the values of the interpolant at grid nodes, and the last one additionally requires its (partial) derivatives w.r.t. each coordinate at grid nodes. All these classes compute the function value and up to two (three for quintic splines) derivatives at any point inside the grid; 1d functions are linearly extrapolated outside the grid.

An alternative formulation of the piecewise-polynomial interpolation methods is in terms of B-splines – $N_x + N - 1$ basis functions defined by the grid nodes, which are polynomials of degree N on each of at most $N+1$ consecutive segments of the grid, and are zero otherwise. The case $N = 1$ corresponds to linear interpolation, $N = 3$ – to (clamped) cubic splines[1].

[1] A cubic spline in 1d is defined by $N_x + 2$ parameters: they may be taken to be the values of spline at N_x grid nodes plus two endpoint derivatives, which is called a clamped spline. The more familiar case of a natural cubic spline instead has these two additional parameters defined implicitly, by requiring that the second derivative of the spline is zero at both ends.

The interpolating function is defined as $f(\boldsymbol{x}) = \sum_\alpha A_\alpha B_\alpha(\boldsymbol{x})$, where α is a combined index in all dimensions, A_α are the amplitudes and B_α are the basis functions (in more than one dimension, they are formed as tensor products of 1d B-splines, i.e., $B_{ij}(x,y) = B_i(x)\,B_j(y)$). Again, the evaluation of interpolant only requires $O(\log(N_x)+N)$ operations per dimension to locate the grid segment and compute all N possibly nonzero basis functions.

In three dimensions, this is the only available interpolation method, and it is implemented for $N=1$ and $N=3$. In the latter case, the construction of interpolator from the array of function values at grid nodes involves the solution of a sparse matrix equation to find the amplitudes A_α, so it may be costly if $N_x N_y N_z \gtrsim 10^3$ (but the interpolation itself is still efficient).

Spline interpolation is heavily used throughout the entire Agama library as an efficient and accurate method for approximating various quantities that are expensive to evaluate directly. By performing suitable additional scaling transformations on the argument and/or value of the interpolator, it is possible to achieve an exquisite accuracy (sometimes down to machine precision) with a moderate ($\mathcal{O}(10^2)$) number of nodes covering the region of interest; for one-dimensional splines a linear extrapolation beyond that region often remains quite accurate under a carefully chosen scaling. Quintic splines are employed when it is possible to compute analytically the derivatives (or partial derivatives in the 2d case) of the approximated function at grid nodes during the spline construction in addition to its values – in this case the accuracy of approximation becomes $1-2$ orders of magnitude better than that of a cubic spline. (Of course, computing the derivatives by finite-differencing or from a cubic spline does not achieve the goal).

Penalized spline fitting There are two kinds of tasks that involve the construction of a spline curve from an irregular set of points (as opposed to the values of the curve at grid nodes, as in the previous section).

The first task is to create a smooth least-square approximation $f(x)$ to a set of points $\{x_i, y_i\}$: minimize $\sum_i [y_i - f(x_i)]^2 + \lambda \int [f''(x)]^2\,dx$, where λ is the smoothing parameter controlling the tradeoff between approximation error (the first term) and the curvature penalty (the second term). The solution is given by a cubic spline with grid nodes placed at all input points $\{x_i\}$ [17]; however, it is not practical in the case of a large number of points. Instead, we approximate it with a cubic spline having a much smaller number of grid nodes $\{X_k\}$ specified by the user. The class `SplineApprox` is constructed for the given grid $\{X_k\}$ and x-coordinates of input points; after preparing the ground, it may be used to find the amplitudes of B-splines for any $\{y_i\}$ and λ, and there is a method for automatically choosing the suitable amount of smoothing.

The second task is to determine a density function $P(x)$ from an array of samples $\{x_i\}$, possibly with individual weights $\{w_i\}$. It is also solved with the help of B-splines, this time for $\ln P(x)$, which is represented as a B-spline of degree N defined by user-specified grid

nodes $\{X_k\}$. The routine `splineLogDensity` constructs an approximation for $\ln P$ for the given grid nodes and samples, with adjustable smoothing parameter λ.

Both tasks are presently implemented only for the 1d case, but in the future may be generalized to multidimensional data represented by tensor-product B-splines. More details on the mathematical formulation is given in the Appendix A.2.3.

2.1.2 Units

Handling of units is a surprisingly difficult and error-prone task. AGAMA adopts a somewhat clumsy but consistent approach to unit handling, which mandates a clear separation between internal units inside the library and external units used to import/export the data. This alone is a rather natural idea; what makes it peculiar is that we do not fix our internal units to any particular values. There are three independent physical base units – mass, length, and time, or velocity instead of time. The only convention used throughout the library is that $G = 1$, which is customary for any stellar-dynamical code. This leaves only two independent base units, and we mandate that the results of all calculations should be independent of the choice of base units (up to insignificant roundoff errors at the level $\sim 10^{-4} \div 10^{-6}$ – typical values for root-finder or integration tolerance parameters). This places heavier demand on the implementation – in particular, all dimensional quantities should generally be converted to logarithms before being used in a scale-free context such as finding a root on the interval $[0..\infty)$. But the reward is greater robustness in various applications.

In practice, the `units::` namespace defines *two* separate unit classes. The first is `InternalUnits`, defining the two independent physical scales (taken to be length and time) used as the internal units of the library. Typically, a single instance of this class (let's call it `intUnit`) is created for the entire program. It does not provide any methods – only conversion constants such as `from_xxx` and `to_xxx`, where xxx stands for some physical quantity. For instance, to obtain the value of potential expressed in $(\mathrm{km/s})^2$ at the galactocentric radius of 8 kpc, one needs to write something like

```
double E = myPotential.value(coord::PosCyl( 8 * intUnit.from_Kpc, 0, 0 ));
std::cout << E * pow_2(intUnit.to_kms);
```

The second is `ExternalUnits`, which is used to convert physical quantities between the external datasets and internal variables. External units, of course, do not need to follow the convention $G = 1$, thus they are defined by three fundamental physical scales (length, velocity and mass) plus an instance of `InternalUnits` class that describes the working units of the library. An instance of unit converter is supplied as an argument to all functions that interface with external data: read/write potential and distribution function parameters, N-body snapshots, and any other kinds of data. Thus the dimensional quantities ingested by the library are always in internal units, and are converted back to physical units on output.

When the external data follows the convention $G = 1$ in whatever units, no conversion

is necessary, thus one may provide an ExternalUnits object with a default constructor wherever required (it is usually a default value for this argument); in this case also no InternalUnits need to be defined. The reason for existence of two classes is that neither of them can fulfill both roles: to serve as an arbitrary internal ruler for testing the scale-invariance of calculations, and to have three independent fundamental physical scales (possibly different for various external data sources). In practice, one may create a single global instance of ExternalUnits with a temporary instance of arbitrary InternalUnits as an argument; however, having a separate global instance of the latter class is handy because its conversion constants indicate the direction (to or from physical units).

The Python interface supports the unit conversion internally: the user may set up a global instance of ExternalUnits, and all dimensional quantities passed to the library will be converted to internal library units and then back to physical units on output. Or, if no such conversion has been set up, all data is assumed to follow the convention $G = 1$.

2.1.3 Coordinates

The coords:: namespace contains classes and routines for representing various mathematical objects in several coordinate systems in three-dimensional space.

There are several built-in coordinate systems: Cartesian, Cylindrical, Spherical, and ProlSph – prolate spheroidal. Their names are used as tags in other templated classes and conversion routines; only the last one has an adjustable parameter (interfocal distance).

Templated classes include position, velocity, a combination of the two, an abstract interface IScalarFunction for a scalar function evaluated in a particular coordinate system, gradient and hessian of a scalar function, and coefficients for coordinate transformations from one system to the other. Templated functions convert these objects from one coordinate system to the other: for instance, toPosVelCyl converts the position and velocity from any source coordinate system into cylindrical coordinates; these routines should be called explicitly, to make the code self-documenting. An even more powerful family of functions evalAndConvert take the position in one (output) coordinate system and a scalar function defined in the other (evaluation) system, calls the function with transformed coordinates, and perform the transformation of gradient and hessian back to the output system. The primary use of these routines is in the potential framework (Section 2.2) – each potential defines a method for computing it in the optimal system, and uses the conversion routines to provide the remaining ones. Another use is for transformation of probability distributions, which involve Jacobian matrices of coordinate conversions.

2.1.4 Particles

A particle is an object with phase-space coordinates and mass; the latter is just a single number, and the former may be either just the position or the position and velocity in any coordi-

nate system. Particles are grouped in arrays (templated struct `ParticleArray<ParticleT>`). Particle arrays in different coordinate systems can be implicitly converted to each other, to simplify the calling convention of routines that use one particular kind of coordinate system, but accept all other ones with the same syntax.

Agama provides routines for storing and loading particle arrays in files (`readSnapshot` and `writeSnapshot`), with several file formats available, depending on compilation options. Text files are built-in, and support for Nemo and Gadget binary formats is provided through the Unsio library (optional).

Particle arrays are also used in constructing a potential expansion (`Multipole` or `CylSpline`) from an N-body snapshot, and created by routines from the `galaxymodel` module (Section 2.6), e.g., by sampling from a distribution function.

The particle array type and input/output routines belong to the `particles::` namespace.

2.1.5 Utilities

There are quite a few general-purpose utility functions that do not belong to any other module, and are grouped in the `utils::` namespace. Apart from several routines for string manipulation (e.g., converting between numbers and strings), and logging, there is a self-sufficient mechanism for dealing with configuration files. These files have a standard INI format, i.e., each line contains `name=value`, and parameters belonging to the same subject domain may be grouped in sections, with a preceding line `[section name]`. Values may be strings or numbers, names are case-insensitive, and lines starting with a comment symbol `#` or `;` are ignored.

The class `KeyValueMap` is responsible for a list of values belonging to a single section; this list may be read from an INI file, or created by parsing a single string like `"param1=value1 param2=1.0"`, or from an array of command-line arguments. Various methods return the values converted to a particular type (number, string or boolean) or set/replace values. The class `ConfigFile` operates with a collection of sections, each represented by its own `KeyValueMap`; it can read and write INI files.

2.2 Potentials

Agama provides a versatile collection of density and potential models, including two very general and efficient approximations that can represent almost any well-behaved profile of an isolated stellar system. All classes and routines in this section are located in the `potential::` namespace.

All density models are derived from the `BaseDensity` class, which defines methods for computing the density in three standard coordinate systems (derived classes choose the

most convenient one to implement directly, and the two other ones use coordinate transformations), a function returning the symmetry properties of the model, and two convenience methods for computing mass within a given radius and the total mass (by default they integrate the density over volume, but derived classes may provide a cheaper alternative).

All potential models are derived from the `BasePotential` class, which itself descends from `BaseDensity`. It defines methods for computing the potential, its first derivative (gradient vector) and second derivative (hessian tensor) in three standard coordinate systems. By default, density is computed from the hessian, but derived classes may override this behaviour. Furthermore there are several derived abstract classes serving as bases for potentials that are easier to evaluate in a particular coordinate system (Section 2.1.3): the function `eval()` for this system remains to be implemented in descendant classes, and the other two functions use coordinate and derivative transformations to convert the computed value to the target coordinate system. For instance, a triaxial harmonic potential is easier to evaluate in Cartesian coordinates, while the Stäckel potential is naturally expressed in prolate spheroidal coordinate system.

Any number of density components may be combined into a single `CompositeDensity` class, and similarly for potential components.

2.2.1 Analytic potentials

There are several commonly used models with known expressions for the potential and its derivatives.

Spherical models include the `Plummer`, `Isochrone` and `NFW` (Navarro–Frenk–White) potentials. Moreover there is a wrapper class that turns any user-provided function $\Phi(r)$ with two known derivatives into a form compatible with the potential interface.

Axisymmetric models include the `MiyamotoNagai` and `OblatePerfectEllipsoid` potentials (the latter belongs to a more general class of Stäckel potentials [16], but is the only one implemented at present). There is another type of axisymmetric models that have a dedicated potential class, namely a separable `DiskDensity` profile ($\rho(R,z) = \Sigma(R)h(z)$). A direct evaluation of potential requires 2d numerical quadrature, or 1d in special cases such as the exponential radial profile, which is still too costly. Instead, we use the approach introduced in [20, 15] (GalPot): the potential is split into two parts, `DiskAnsatz` that has an analytic expression for the potential of the strongly flattened component, and the residual part that is represented with the `Multipole` expansion.

Triaxial models include the `Logarithmic`, `Harmonic`, `Dehnen` [13] and `Ferrers` potentials. The first two have infinite extent and are usable only in certain contexts (such as orbit integration), because most routines expect the potential to vanish at infinity. Dehnen models may have any symmetry from spherical to triaxial; in non-spherical cases, the potential and its derivatives are computed using a 1d numerical quadrature [24], so this is rather costly

(and also inaccurate at large distances). A preferred way of using an axisymmetric or triaxial Dehnen model is through the `Multipole` expansion. Ferrers ($n = 2$) models are strictly triaxial, and have analytic expressions for the potential and its derivatives [28]. There is also a `SpheroidDensity` class that describes general triaxial two-power-law ($\alpha\beta\gamma$) density profiles [44] with an optional exponential cutoff. Dehnen, Plummer, Isochrone and NFW profiles are all special cases of this model; however, this class only provides the density profile and not the potential.

2.2.2 Multipole expansion

`Multipole` is a general-purpose potential approximation that delivers highly accurate results for density profiles with axis ratio not very different from unity (say, at most a factor of few). It represents the potential as a sum of spherical-harmonic functions of angles multiplied by arbitrary functions of radius: $\Phi(r,\theta,\phi) = \sum_{l,m} \Phi_{l,m}(r)\, Y_l^m(\theta,\phi)$. The radial dependence of each term is given by a quintic spline, defined by a rather small number of grid nodes ($N_r \sim 10 \div 50$), typically spaced equally in $\log r$ over a range $r_{\max}/r_{\min} \gtrsim 10^6$; the suitable order of angular expansion $l_{\max}$ depends on the shape of the density profile, and is usually $\lesssim 10$.

The potential approximation may be constructed in several ways:

- from another potential (makes sense if the latter is expensive to compute, e.g., a triaxial `Dehnen` model);
- from a smooth density profile, thereby solving the Poisson equation in spherical coordinates;
- from an N-body model (an array of particle coordinates and masses) – in this case a temporary smooth density model is created and used in the same way as in the second scenario;
- by loading a previously computed array of coefficients from a text file.

This type of potential is rather inexpensive to initialize, very efficient to compute, provides an accurate extrapolation to small and large radii beyond the extent of its radial grid, and is the right choice for "spheroidal" density models – from spherical to mildly triaxial, and even beyond (i.e., a model may have a twist in the direction of principal axes, or contain an off-centered odd-m mode).

As a side note, a related class of potential approximations is based on expanding the radial dependence of spherical-harmonic terms $\Phi_{l,m}(r)$ into a sum over functions from a suitable basis set [19, 44]. For several reasons, this approach is less efficient: the choice of the family of basis functions implies certain biases in the approximation, and the need to compute a full set of them (involving rather expensive algebraic operations) at each radius is contrasted with a much faster evaluation of a spline (essentially using only a few adjacent

grid points). [41] demonstrated the superiority of a previous implementation of spline-interpolated spherical-harmonic expansion over the basis-set approach, and `Multipole` is improved even further.

2.2.3 Azimuthal harmonic expansion

`CylSpline`[2] is another general-purpose potential approximation that is more effective for strongly flattened (disky) systems, whether axisymmetric or not. It represents the potential as a sum of Fourier terms in the azimuthal angle (ϕ), with coefficients of each term interpolated via a 2d quintic spline spanning a finite region in the R, z plane. The accuracy of approximation is determined by the number and extent of the grid nodes in R and z (also scaled logarithmically to achieve a high dynamic range) and the order $m_{\max}$ of angular expansion; in the axisymmetric case only one term is used, but generally it may represent any geometry, e.g., spiral arms and a triaxial bar.

This potential may also be constructed in the same four ways as `Multipole`, but the solution of Poisson equation is much more expensive in this case; still, for typical grid sizes of a few dozen in each direction, it takes between a few seconds and minutes on a single CPU core (and is almost ideally parallelized). After initialization, the computation of potential and forces is as efficient as `Multipole`. In many cases, it delivers comparable or better accuracy than the latter, but is not suitable for cuspy density profiles and for extended tails of density at large radii, since it may only represent it over a finite region (the potential and its first derivative is still quite accurately extrapolated outside the grid, but the density is identically zero there). Its main advantage is the ability to handle disky systems which are not suitable for a spherical-harmonic expansion[3].

To summarize, both potential approximations have wide, partially overlapping range of applicability, are equally efficient in evaluation (but not construction), and deliver good accuracy (see Figures 3, 4 in the Appendix, with more technical details given in Section A.3). We note that application of these methods to represent the potential of a galaxy like the Milky Way is computationally more demanding than simple models based e.g. on a combination of Miyamoto–Nagai disks and spherically-symmetric two-power-law profiles, but only moderately (by a factor of 2–3), and allows much greater flexibility and realism (especially if non-axisymmetric features are required).

[2] an improved version of the method presented in [43]

[3] Potential of separable axisymmetric disk density profiles can be efficiently computed using a combination of `DiskAnsatz` and `Multipole` (the GalPot approach), but this applies only to this restricted class of systems, and is comparable to `CylSpline` in both speed and accuracy.

2.2.4 Potential factory

All density and potential classes may be constructed using a universal "factory" interface – several routines `createDensity` and `createPotential` that return new instances of `PtrDensity` or `PtrPotential` according to the provided parameters. The parameters can be supplied in several ways. One is an INI file with one or several components of the potential described in separate sections `[Potential]`, `[Potential2]`, `[Potential disk]`, etc. (all section names should start with "Potential"). Another possibility is to provide a `KeyValueMap` object (Section 2.1.5) corresponding to a single section from an INI file (it may be read from the file, or constructed manually, e.g., from named arguments in the `Python` interface, or from command-line parameters for console programs, or from a single string like `"key1=value1 key2=value2"`). These parameters may describe the potential completely (e.g., if this is one of the known analytical models), or define the parameters of Multipole or CylSpline potential expansions to be constructed from the user-provided density or potential object, or from an array of particles – in the latter case these objects are also passed to the factory routine. Finally, the coefficients of a potential or density expansion may be stored into a text file and subsequently used to load and construct a new object, using `writePotential`/`readPotential` routines.

Here is the list of possible parameters of a single potential or density component for the factory routines (not all of them make sense for all models, but unknown or irrelevant parameters will simply be ignored); see Table 1 for complete information:

- `type` – determines the type of potential used; should be the name of a class derived from `BasePotential` – either an analytic potential or an expansion (Multipole or CylSpline). It is usually required, except if the potential is loaded from a coefficients file – in that case the name of the potential appears in the first line of this text file, so is determined automatically.

- `density` – if `type` is a potential expansion, this parameter determines the density model to be used; should be the name of a class derived from `BaseDensity` (or, by consequence, the name of an analytic potential), except that it cannot be a model with unbound potential (Logarithmic or Harmonic) or another potential expansion.
  There is one exception to the rule that `type` must encode a potential class: it may also contain the names of two density profiles originally used in GalPot – `DiskDensity` or `SpheroidDensity`. All such components are collected first, and used to construct a *single* instance of `Multipole` potential with default parameters, plus zero or more instances of `DiskAnsatz` potentials (according to the number of disk profiles). The source density for this Multipole potential contains all SpheroidDensity and DiskDensity components, plus *negative* contributions of DiskAnsatz potentials (i.e., with inverted sign of their masses). Of course, one may use them also as regular `density` components (e.g.,

type=CylSpline density=DiskDensity, which yields comparable accuracy), but in that case each one would create a separate potential expansion, which is of course not efficient. In order to lift this limitation, one may construct all density components individually, manually combine them into a single composite model, and pass it to the constructor of a potential expansion (this approach is used for self-consistent multicomponent models, Section 2.6.4).

- symmetry – defines the symmetry properties of the density model passed to the potential expansion. All built-in models report this property automatically; this parameter is useful if the input is given by an array of particles, or by a user-defined routine returning the density or potential in Python and Fortran interfaces. It could be either a text string with one of the standard choices ("spherical", "axisymmetric", "triaxial", "none" – only the first letter is used), or a number encoding a more complicated symmetry (see the definitions in coords.h).

- file – the name of a file with potential expansion coefficients, or with an N-body snapshot to be used for creating a potential expansion. In the former case the type of potential expansion is stored in the first line of the file, so the type parameter is not required.

Parameters defining an analytic density or potential model (if type is a potential expansion, they refer to the density argument, otherwise to type); default values are given in brackets:

- mass [1] – total mass of an analytic model.

- scaleRadius [1] – the first (sometimes the only) parameter with the dimension of length that defines the profile.

- scaleRadius2 [1] or scaleHeight [0] – the second such parameter (e.g., for Miyamoto–Nagai or exponential disk models).

- outerCutoffRadius [0] – another length-scale parameter defining the radius of exponential cut-off, used for SpheroidDensity models (0 means no cutoff).

- innerCutoffRadius [0] – similar parameter for DiskDensity that defines the radius of an inner hole.

- surfaceDensity [0] – value of surface density at $R = 0$ for the exponential DiskDensity profile.

- densityNorm [0] – value that defines the volume density at the scale radius for the SpheroidDensity profile. Alternatively, instead of this or the previous parameter, one may provide the total mass of the corresponding model (these two parameters have a priority over mass).

- `alpha` [1] – parameter controlling the steepness of transition between two asymptotic power-law slopes for `SpheroidDensity`.
- `beta` [4] – power-law index of the outer density profile for `SpheroidDensity`; should be > 2 except when there is an outer cutoff, otherwise the potential is unbound.
- `gamma` [1] – power-law index of the inner density profile $\rho \propto r^{-\gamma}$ as $r \to 0$ for `Dehnen` (should be $0 \leq \gamma \leq 2$) or `SpheroidDensity` models (should be $\gamma < 3$).
- `p` or `axisRatioY` [1] – the axis ratio y/x of equidensity surfaces of constant ellipticity for `Dehnen`, `SpheroidDensity` or `Ferrers` models, or the analogous quantity for the `Logarithmic` or `Harmonic` potentials.
- `q` or `axisRatioZ` [1] – the same parameter for z/x.

Parameters defining the potential expansions (default values in brackets are all sensible and only occasionally need to be changed):

- `gridSizeR` [25] – the number of grid nodes in spherical (`Multipole`) or cylindrical (`CylSpline`) radius; in the latter case this includes the 0th node at $R = 0$.
- `gridSizeZ` [25] – same for the grid in z direction in `CylSpline`, including the $z = 0$ node.
- `rmin` [0] – the radius of the innermost nonzero node in the radial grid (for both potential expansions).
- `rmax` [0] – same for the outermost node; zero values mean automatic determination.
- `zmin` [0], `zmax` [0] – same for the vertical grid in `CylSpline`; zero values mean take them from the radial grid.
- `lmax` [6] – the order of `Multipole` expansion in $\cos\theta$; 0 means spherical symmetry.
- `mmax` [lmax] – the order of azimuthal Fourier expansion in ϕ for both `CylSpline` and `Multipole`; 0 means axisymmetry, and $m_{\rm max}$ should be $\leq l_{\rm max}$. Of course, the actual order of expansion in all cases is also determined by the symmetry properties of the input density model – if it reports to be axisymmetric, no $m \neq 0$ terms will be used anyway.
- `smoothing` [1] – the amount of smoothing applied during construction of the `Multipole` potential from an array of particles.

Table 1: Available density and potential models and their parameters

Name	Formula	Parameters
	Density-only models	
`DiskDensity`	$\rho = \Sigma_0 \exp\left(-\frac{R}{R_{\rm disk}} - \frac{R_{\rm cut}}{R}\right) \times \begin{cases} \delta(z) & \text{if } h = 0 \\ \frac{1}{2h}\exp\left(-\left\lvert\frac{z}{h}\right\rvert\right) & \text{if } h > 0 \\ \frac{1}{4\lvert h\rvert}\operatorname{sech}^2\left(\left\lvert\frac{z}{2h}\right\rvert\right) & \text{if } h < 0 \end{cases}$	`surfaceDensity` (Σ_0) or `mass`, `scaleRadius` ($R_{\rm disk}$), `scaleHeight` (h), `innerCutoffRadius` ($R_{\rm cut}$)
`SpheroidDensity`	$\rho = \rho_0 \left(\frac{\tilde{r}}{a}\right)^{-\gamma} \left[1 + \left(\frac{\tilde{r}}{a}\right)^{\alpha}\right]^{\frac{\gamma-\beta}{\alpha}} \times \exp\left[-\left(\frac{\tilde{r}}{r_{\rm cut}}\right)^2\right]$	`densityNorm` (ρ_0) or `mass`, `scaleRadius` (a), `outerCutoffRadius` ($r_{\rm cut}$), `gamma` (γ), `beta` (β), `alpha` (α), `axisRatioY` (p), `axisRatioZ` (q)
	Density/potential models	
`Plummer`	$\Phi = -\frac{M}{\sqrt{a^2+r^2}}$	`mass` (M), `scaleRadius` (a)
`Isochrone`	$\Phi = -\frac{M}{a+\sqrt{r^2+a^2}}$	`mass` (M), `scaleRadius` (a)
`NFW`	$\Phi = -\frac{M}{r}\ln\left(1+\frac{r}{a}\right)$	`mass` (M, note that the total mass is infinite), `scaleRadius` (a)
`MiyamotoNagai`	$\Phi = -\frac{M}{\sqrt{R^2+\left(a+\sqrt{z^2+b^2}\right)^2}}$	`mass` (M), `scaleRadius` (a), `scaleRadius2` (b)
`OblatePerfectEllipsoid`	$\rho = \frac{M}{\pi^2\, q\, a^3}\left[1+\frac{R^2+(z/q)^2}{a^2}\right]^{-2}$	`mass` (M), `scaleRadius` (a), `axisRatioZ` (q)
`Dehnen`	$\rho = \frac{M\,(3-\gamma)}{4\pi\, p\, q\, a^3}\left(\frac{\tilde{r}}{a}\right)^{-\gamma}\left(1+\frac{\tilde{r}}{a}\right)^{\gamma-4}$	`mass` (M), `scaleRadius` (a), `axisRatioY` (p), `axisRatioZ` (q)
`Ferrers`	$\rho = \frac{105\,M}{32\pi\, p\, q\, a^3}\left[1-\left(\frac{\tilde{r}}{a}\right)^2\right]^2$	`mass` (M), `scaleRadius` (a), `axisRatioY` (p), `axisRatioZ` (q)
`Logarithmic`	$\Phi = \sigma^2\,\ln(r_{\rm core}^2 + \tilde{r}^2)$	`mass` (σ), `scaleRadius` ($r_{\rm core}$), `axisRatioY` (p), `axisRatioZ` (q)
`Harmonic`	$\Phi = \frac{1}{2}\Omega^2\,\tilde{r}^2$	`mass` (Ω), `axisRatioY` (p), `axisRatioZ` (q)

$R = \sqrt{x^2+y^2}$ is the cylindrical radius and $\tilde{r} = \sqrt{x^2+(y/p)^2+(z/q)^2}$ is the ellipsoidal radius

These keywords, with some modifications, are also used in potential construction routines in `Python` and `Fortran` interfaces and in the AMUSE and GALPY plugins (Sections 3.1, 3.2, 3.3, 3.4). For instance, `Python` interface allows to provide a user-defined function specifying the density profile in the `density=` argument, or an array of particles in the `particles=` argument.

All dimensional values in the potential factory routines can optionally be specified in physical units and converted into internal units by providing an extra unit conversion parameter (Section 2.1.2). For instance, masses and radii in the INI file may be given in solar masses and parsecs. This conversion also applies during write/read of density or potential coefficients to/from text files. Of course, if all data is given in the same units and follows the convention $G = 1$, no conversion is needed.

2.2.5 Utility functions

There are several frequently used functions that operate on any potential object: conversion between energy E, angular momentum of a circular orbit $L_{\rm circ}$, and radius; epicyclic frequencies κ, ν, Ω as functions of radius[4]; peri- and apocenter radii of a planar orbit with given E, L (in the $z = 0$ plane of an axisymmetric potential), etc. They are implemented as standalone functions (generally using a root-finding routine to solve equations such as $\Phi(r) = E$ for r), and as two interpolator classes that pre-compute these values on a 1d or 2d grid in E or E, L, and provide a faster (but still very accurate) alternative to the standalone functions. These interpolators are used, e.g., in the spherical action finder/mapper class (Section 2.4.2).

2.3 Orbit integration and analysis

Orbits of particles in the smooth time-independent potential are computed using the routine `orbit::integrate` in any of the three standard coordinate systems. It solves the coupled system of ordinary differential equations (ODEs) for time derivatives of position and velocity, using one of the available methods derived from `math::BaseOdeSolver`. The trajectory is stored at regular intervals of time, which is unrelated to the internal timestep of ODE solver (that is, the position/velocity at any time is obtained by interpolation provided by the solver – so-called dense output feature). Currently we have one method available – 8th order Runge–Kutta with adaptive timestep [18]. Other possibilities previously implemented in [41] include 15th order Gauss–Radau scheme [33], 4th order Hermite method [21], and several methods from ODEINT package [1], including Bulirsch–Stoer and various Runge–Kutta schemes. However, in practice all of them have rather similar performance in the appropriate range of tolerance parameters, thus we have only kept one at the moment.

[4] defined as $\kappa^2 \equiv \dfrac{\partial^2\Phi}{\partial R^2} + \dfrac{3}{R}\dfrac{\partial\Phi}{\partial R}$, $\nu^2 \equiv \dfrac{\partial^2\Phi}{\partial z^2}$, $\Omega^2 \equiv \dfrac{1}{R}\dfrac{\partial\Phi}{\partial R} = \left(\dfrac{L_{\rm circ}^2}{R}\right)^2$, evaluated at $z = 0$.

Orbit analysis refers to the determination of orbit class (box, tube, resonant boxlet, etc.) and degree of chaoticity. This is performed using a Fourier transform of position as a function of time and detecting the most prominent "spectral lines"; the ratio between their frequencies is an indicator of orbit type [6, 9], and their rate of change with time is a measure of chaos [40]. A finite-time estimate of Lyapunov exponent is another measure of stochasticity (see [10, 36] for reviews of methods based on variational equations). These methods are implemented in [41], but as the focus of AGAMA in galaxy modelling is shifted from discrete orbits to smooth distribution functions, we have not yet included them in the library.

2.4 Action/angle variables

As the name implies, AGAMA deals with models of stellar system described in terms of action/angle variables. They are defined, e.g., in Section 3.5 of [7].

In a spherical or axisymmetric potential, the most convenient choice for actions is the triplet $\{J_r, J_z, J_\phi\}$, where $J_r \geq 0$ (radial action) describes the motion in cylindrical radius, $J_z \geq 0$ (vertical action) describes the motion in z direction, and $J_\phi \equiv Rv_\phi$ (azimuthal action) is the conserved component L_z of angular momentum (it may have any sign). In a spherical potential, the sum $J_z + |J_\phi|$ is the total angular momentum L.

The `actions::` namespace introduces several concepts: `Actions` and `Angles` are the triplet of action and angle variables, `ActionAngles` is their combination, `Frequencies` is the triplet of frequencies $\boldsymbol{\Omega} \equiv \partial H/\partial \boldsymbol{J}$ (derivatives of Hamiltonian w.r.t. actions). The transformation from $\{\boldsymbol{x}, \boldsymbol{v}\}$ to $\{\boldsymbol{J}, \boldsymbol{\theta}\}$ is provided by action finders, and the inverse transformation – by action mappers. There are several distinct methods discussed later in this section, and they may exist as standalone routines and/or instances of classes derived from the `BaseActionFinder` and `BaseActionMapper` classes. The action finder routines exist in two variants: computing only the actions (the most common usage), or in addition the angles and frequencies (more expensive).

The following sections describe the methods suitable for specific cases of spherical or axisymmetric potentials (see [35] for a review and comparison of various approaches). At present, AGAMA does not contain any methods for action/angle computation in non-axisymmetric potentials, but they may be added in the future within the same general framework.

2.4.1 Isochrone mapping

The spherical isochrone potential, specified by two parameters (mass M and scale radius b) admits analytic expressions for the transformation between $\{\boldsymbol{x}, \boldsymbol{v}\}$ and $\{\boldsymbol{J}, \boldsymbol{\theta}\}$ in both directions. These expression are given, e.g., in Eqs. 3.225–3.241 of [7]. The standalone routines providing these transformations, optionally with partial derivatives of $\{\boldsymbol{x}, \boldsymbol{v}\}$ w.r.t.

$\boldsymbol{J}, M, b$, are located in `actions_isochrone.h`.

2.4.2 Spherical potentials

In a more general case of an arbitrary spherical potential, the radial action is given by

$$J_r = \frac{1}{\pi} \int_{r_{\min}}^{r_{\max}} \sqrt{2[E - \Phi(r)] - L^2/r^2}\, dr,$$

where $r_{\min,\max}(E, L)$ are the roots of the expression under the radical. The standalone routines in `actions_spherical.h` perform the action/angle transformation in both directions, using numerical root-finding and integration functions in each invocation. If one needs to compute actions for many points ($\gtrsim 10^3$) in the same potential, it is more efficient to construct an instance of `ActionFinderSpherical` class that provides high-accuracy interpolation from the pre-computed 2d tables for $r_{\min,\max}(E, L)$ (using the `potential::Interpolator2d` helper class) and $J_r(E, L)$, as well as the reverse mapping between $\boldsymbol{J}$ and $\{\boldsymbol{x}, \boldsymbol{v}\}$.

2.4.3 Stäckel approximation

In a still more general axisymmetric case, the action/angle variables can be exactly computed for a special class of Stäckel potentials, in which the motion is fully integrable and separable in a prolate spheroidal coordinate system. This computation is performed by the standalone routines `actionsAxisymStaeckel` and `actionAnglesAxisymStaeckel` in `actions_staeckel.h`, which operate on an instance of `potential::OblatePerfectEllipsoid` class (the only example of a Stäckel potential in Agama). The procedure consists of several steps: numerically find the extent of oscillations in the meridional plane in both coordinates λ, ν of the prolate spheroidal system; numerically compute the 1d integrals for J_λ, J_ν (which correspond to J_r, J_z); and if necessary, find the frequencies and angles (again by 1d numerical integration).

For the most interesting practical case of a non-Stäckel axisymmetric potential, the actions can only be approximated under the assumption that the motion is integrable and is locally well described by a Stäckel potential. This is the essence of the "Stäckel fudge" approach [2]. In a nutshell, it pretends that the potential *is* of a Stäckel form (without explicitly constructing it), computes the would-be integrals of motion in this presumed potential, and then performs essentially the same steps as the routines for the genuine Stäckel potential. Actions computed in this way are approximate, in the sense that even for a regular (non-chaotic) motion, they are not exactly conserved along the orbit; the variation of $\boldsymbol{J}$ is smallest for nearly-circular orbits close to the equatorial plane, but typically remains $\lesssim 1 - 10\%$ even for rather eccentric orbits that stray far from the plane (note that the method does not provide any error estimate), see Figure 5. However, if the actual orbit is

chaotic or belongs to one of minor resonant families, the variation of estimated actions along the orbit is rather large because the method does not account for resonant motion.

In order to proceed, the Stäckel approximation requires the parameter of the prolate spheroidal coordinate system – the interfocal distance Δ; the accuracy (variation of estimated actions along the orbit) strongly depends on its value. Crucially, we do not need to have a single value of Δ for the entire system, but may use the most suitable value for the given set of integrals of motion (depending on $\{\boldsymbol{x},\boldsymbol{v}\}$). The most accurate result is obtained by averaging the potential derivatives over the region covered by the (numerically computed) orbit. This is of course impractical, and a suitable approximation is provided by the `InterfocalDistanceFinder` class, which constructs an interpolation table for $\Delta(E,L)$ by pre-computing these values for thin (shell) orbits located for each pair of two classical integrals of motion E, L in a 2d grid; the initialization of this table takes $\mathcal{O}(1\text{ s})$. The `ActionFinderAxisymFudge` class encapsulates this Δ-finder and provides methods for computing actions, angles and frequencies for any $\{\boldsymbol{x},\boldsymbol{v}\}$ in the given potential. This is the main workhorse for many higher-level tasks in the Agama library. The computation of a single triplet of actions costs ~ 50 potential evaluations. In the future, we plan to introduce a faster approximation based on 3d interpolation for $\boldsymbol{J}$ as a function of three integrals of motion (one of them itself being approximate).

2.4.4 Torus mapping

The transformation from $\{\boldsymbol{J},\boldsymbol{\theta}\}$ to $\{\boldsymbol{x},\boldsymbol{v}\}$ in an arbitrary axisymmetric potential is performed using the Torus mapping approach [5]. An instance of `ActionMapperTorus` class is constructed for any choice of $\boldsymbol{J}$ and allows to perform this mapping for multiple values of $\boldsymbol{\theta}$; however, the cost of torus construction is rather high, and it may not always succeed (depending on the properties of potential and required accuracy). The code is adapted from the original TM package, with several modifications enabling the use of an arbitrary potential and a more efficient angle mapping approach; however, it does not quite comply to the coding standards adopted in Agama (Section A.1) and in the future will be replaced by a fresh implementation.

2.5 Distribution functions

By Jeans' theorem, a steady-state distribution of stars or other species in a stationary potential may depend only on integrals of motion, taken here to be the actions $\boldsymbol{J}$. The `df::` namespace contains the classes and methods for working with such distribution functions (DFs) formulated in terms of actions. They are derived from the `BaseDistributionFunction` class, which provides a single method for computing the value $f(\boldsymbol{J})$ at the given triplet of actions. All physically valid DFs must have a finite mass $M = (2\pi)^3 \iiint f(\boldsymbol{J})\, d^3J$, by

default computed by numerical integration (the pre-factor comes from a trivial integration over angles). The same DF corresponds to different density profiles in different potentials (Section 2.6.1), but the total mass of the density profile is always the same.

AGAMA provides several DFs suitable for various components of a galaxy, described in the following sections. In addition there is a concept of a multi-component DF: since computing the actions – arguments of the DF – is a non-negligible cost, it is often advantageous to evaluate several DFs at the same set of actions at once. There is also a "DF factory" routine `createDistributionFunction` for constructing various DF classes from a set of named parameters described by a `KeyValueMap` object (Section 2.1.5).

Importantly, the DF formulated in terms of actions does not depend on the potential. However, some models use the concept of epicyclic frequencies to compute the value of $f(\boldsymbol{J})$. These frequencies are represented by a special proxy class `potential::Interpolator`, which is constructed from a given potential, but then serves as an independent entity (essentially an array of arbitrary functions of one variable), so that $f(\boldsymbol{J})$ has the same value in any other potential. This is important in the context of iterative construction of self-consistent models (Section 2.6.4).

2.5.1 Spheroidal components

A suitable choice for DFs of elliptical galaxies, bulges or haloes is the double-power-law model, which is similar to the one presented in [31], with a different notation:

$$f(\boldsymbol{J}) = \frac{M}{(2\pi\,J_0)^3} \left(\frac{h(\boldsymbol{J})}{J_0}\right)^{-\Gamma} \left[1+\left(\frac{g(\boldsymbol{J})}{J_0}\right)^{\eta}\right]^{\frac{\Gamma-\mathrm{B}}{\eta}} \exp\left[-\left(\frac{g(\boldsymbol{J})}{J_{\max}}\right)^2\right], \quad \text{where}$$
$$g(\boldsymbol{J}) \equiv g_r J_r + g_z J_z + (3-g_r-g_z)|J_\phi|, \quad h(\boldsymbol{J}) \equiv h_r J_r + h_z J_z + (3-h_r-h_z)|J_\phi|$$

are linear combinations of actions that control the anisotropy of the model in the outer region (above the break action J_0) and the inner region (below J_0), respectively, the power-law indices $\mathrm{B} > 3$ and $\Gamma < 3$ control the slope of the density profile, η determines the steepness of the transition between the two regimes, and $J_{\max}$ additionally suppresses the DF at large J. This DF roughly corresponds to the $\alpha\beta\gamma$ `SpheroidDensity` model, with the asymptotic power-law indices $\mathrm{B} = 2\beta-3$ and $\Gamma = (6-\gamma)/(4-\gamma)$. The `DoublePowerLawParams` structure contains all parameters of these DF families, and the `DoublePowerLaw` class represents an actual DF model.

2.5.2 Disky components

Stars on nearly-circular (cold) orbits in a disk are often described by a Schwarzschild or Shu DF, which have Maxwellian velocity distribution with different dispersions in each direction. A generalization for warm disks expressed in terms of actions is provided by the

`PseudoIsothermal` class, with many tunable parameters contained in a `PseudoIsothermalParam` structure. The DF for a single population is given by [4]:

$$f(J) = f_r(J_r, J_\phi)\; f_z(J_z, J_\phi)\; f_\phi(J_\phi), \qquad \text{where}$$
$$f_r \equiv \frac{\Omega\,\Sigma}{\pi\,\kappa\,\sigma_r^2}\,\exp\left(-\frac{\kappa\,J_r}{\sigma_r^2}\right), \quad f_z \equiv \frac{\nu}{2\pi\,\sigma_z^2}\,\exp\left(-\frac{\nu\,J_z}{\sigma_z^2}\right), \quad f_\phi \equiv 1+\tanh\left(\frac{J_\phi}{J_{\phi,0}}\right),$$
$$\Sigma(R_c) \equiv \Sigma_0\exp(-R_c/R_{\rm disk}), \quad \sigma_r(R_c) \equiv \sigma_{r,0}\exp(-R_c/R_{\sigma,r}), \quad \sigma_z(R_c) \equiv \sigma_{z,0}\exp(-R_c/R_{\sigma,z}).$$

Here $R_c(J_\phi)$ is the radius of a circular orbit with angular momentum J_ϕ, and other quantities – epicyclic frequencies κ, ν, Ω, surface density Σ and velocity dispersions σ_r, σ_z – are expressed as functions of R_c. However, as stressed above, both epicyclic frequencies and $R_c(J_\phi)$ are merely one-dimensional functions that are once initialized from an actual potential, but no longer need to be related to the potential in which the DF is later used. The DF of this form approximately corresponds to a radially exponential and vertically isothermal density profile in a potential of an exponential disk, whose parameters (surface density Σ_0 and radial scale length $R_{\rm disk}$) match those of the DF. The parameter $J_{\phi,0}$ controls the streaming motion (if $J_\phi \ll J_{\phi,0}$, both clockwise and counter-clockwise orbits are equally populated, and in the opposite case only the orbits with positive J_ϕ are present). A still more general model can describe a composition of populations with different velocity dispersions (see `df_disk.h` for more details).

2.5.3 Nonparametric interpolated models

A general way of representing an arbitrary DF in a three-dimensional action space is through an interpolating spline in suitably scaled coordinates. The formulation in terms of B-splines is also an example of a multi-component DF, where each basis function, formed as a tensor product of one-dimensional B-splines, is a separate component. These DFs are used as building blocks in self-consistent models (work in preparation).

2.5.4 Spherical isotropic models

There is an alternative framework for a special case of a spherically-symmetric potential and an isotropic DF in the velocity space. Such DFs depend only on energy E, which is a certain function of J_r and $L \equiv J_z + |J_\phi|$ (provided, e.g., by the `ActionFinderSpherical` class). To avoid the need to express this function explicitly, and to retain the convenience of action formalism, we instead use the phase volume h as the argument of the DF. It is defined as

the volume of phase space enclosed by the given energy hypersurface:

$$h(E) \equiv \iiint d^3x \iiint d^3v\, H\Big[E - \big(\Phi(|\boldsymbol{x}|) + |\boldsymbol{v}|^2/2\big)\Big], \quad \text{where } H \text{ is the step function,}$$
$$= \int_0^{r_{\max}(E)} 4\pi\, r^2\, dr \int_0^{v_{\max}(E,r)} 4\pi\, v^2\, dv = \frac{16\pi^2}{3} \int_0^{r_{\max}(E)} r^2 \Big[2\big(E - \Phi(r)\big)\Big]^{3/2} dr.$$

The advantages of using h instead of E are that the total mass of the model is simply $M = \int_0^\infty f(h)\, dh$, that the same DF may be used in different potentials, etc. The bi-directional correspondence between E and h is provided by a helper class `PhaseVolume`, constructed for a given potential. The derivative $dh(E)/dE \equiv g(E)$ is called the density of states ([7], eq. 4.56), and is given by

$$g(E) \equiv 16\pi^2 \int_0^{r_{\max}(E)} r^2 \sqrt{2\big(E - \Phi(r)\big)}\, dr = 4\pi^2\, L_{\rm circ}^2(E)\, T_{\rm rad}(E).$$

Any non-negative function of one variable (the phase volume h) may serve as an isotropic distribution function in a spherically-symmetric potential, provided that it satisfies the condition that $\int_0^\infty f(h)\, dh$ is finite. One possible way of computing such a DF is through the Eddington inversion formula for any density profile in any given potential (not necessarily related), implemented in the routine `makeEddingtonDF`. The other is to construct an approximating DF from an array of particles sampled from it, using the log-density estimation approach (Section A.2.4), provided by the routine `fitSphericalDF`. More information on these models is given in section A.4.1.

These one-dimensional DFs may also be put on equal grounds with other action-based DFs, using a proxy class `PseudoIsotropic`. It provides the mapping $\boldsymbol{J} \to E \to h$ via `ActionFinderSpherical` and `PhaseVolume` classes, both constructed in a given potential; the value h is then used as the argument to an arbitrary function $f(h)$ provided by the user. Similarly to the case of disky DFs (Section 2.5.2), the potential is only needed to construct the intermediate mappings between actions and the arguments of the DF; the resulting object is then viewed as a function of actions only, and could be used in any other potential. This is especially convenient for iterative construction of multicomponent self-consistent models (Section 2.6.4): the DFs of spheroidal components (bulge, halo) may be obtained using the Eddington inversion formula in the initial spherically-symmetric approximation of the total potential, and then expressed as functions of actions $\boldsymbol{J}$. In subsequent iterations, the potential is no longer spherical, but the density profiles of these components, obtained by integration of their DFs over velocity, are nevertheless quite close to the initial ones.

2.6 Galaxy modelling framework

This module (namespace `galaxymodel::`) broadly encompasses all tasks that involve both a DF and a potential, and additionally an action finder constructed for the given potential

and used for transforming $\{\boldsymbol{x},\boldsymbol{v}\}$ to $\boldsymbol{J}$. As stressed previously, using $\boldsymbol{J}$ as the argument of f has the advantage that the DF may be used with an arbitrary potenial without any modifications (because the possible range of actions does not depend on the potential, unlike, e.g., the possible range of energy).

2.6.1 Moments of distribution functions

The most basic task is the computation of DF moments (density, velocity dispersion, etc.), defined as

$$\rho(\boldsymbol{x}) \equiv \iiint d^3v\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big),$$
$$\overline{\boldsymbol{v}} \equiv \rho^{-1} \iiint d^3v\, \boldsymbol{v}\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big),$$
$$\overline{v_{ij}} \equiv \rho^{-1} \iiint d^3v\, v_{ij}\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big).$$

The routine `computeMoments` calculates any combination of these quantities at the given point $\boldsymbol{x}$ by numerically integrating f over $\boldsymbol{v}$; the DF may be single- or multi-component. This is not a cheap operation, as the integration requires $\gtrsim 10^3$ evaluation of DF and hence calls to the action finder; the computation of density is the major cost in self-consistent modelling (Section 2.6.4).

The routine `computeProjectedMoments` calculates the surface density and the line-of-sight velocity dispersion at the given cylindrical radius R (currently for axisymmetric systems only) – this involves an additional integration over z:

$$\Sigma(R) \equiv \int_{z=-\infty}^{\infty} dz\, \rho(R,z), \qquad \sigma_{\rm los} \equiv \frac{1}{\Sigma(R)} \int_{z=-\infty}^{\infty} dz\, \rho(R,z)\, v_z^2.$$

2.6.2 Velocity distribution functions

Instead of just a few DF moments at a given point, one may consider one-dimensional velocity distribution functions (VDFs):

$$\mathfrak{f}(\boldsymbol{x}; v_1) \equiv \frac{1}{\rho(\boldsymbol{x})} \iint dv_2\, dv_3\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big),$$
$$\mathfrak{f}_{\rm proj}(x_1,x_2; v_k) \equiv \frac{1}{\Sigma(x_1,x_2)} \int dx_3\, \rho(x_1,x_2,x_3)\, \mathfrak{f}(x_1,x_2,x_3; v_k).$$

Currently this is only implemented in cylindrical coordinates: $\boldsymbol{x}=\{R,z,\phi\}, \boldsymbol{v}=\{v_R,v_z,v_\phi\}$. VDFs in each dimension are represented as B-splines of degree N: $\mathfrak{f}(\boldsymbol{x},v_k) = \sum_{\alpha=1}^{N_{\rm basis}} A_\alpha B_\alpha(v_k)$,

where B_α are defined by the nodes of the grid in velocity space (provided by the user, but typically covering the entire available range of velocity with ~ 100 points). To compute the coefficients of expansion A_α, we follow the standard approach by integrating the DF weighted with each basis function to obtain $f(\dots, v_k)\, B_\alpha(v_k)\, dv_k$, and solving the resulting linear system (Section A.2.2). Thus all three VDFs are computed at once in the course of a single 3-dimensional integration (or 4-dimensional for projected VDFs), which is, however, rather expensive (typically $\sim 10^6$ function evaluations). The simplest case $N = 0$ corresponds to a familiar velocity histogram, but a more accurate one is given by $N = 1$ (linear interpolation) or $N = 3$ (cubic spline); note that in the latter case, the interpolated $\mathfrak{f}(v)$ may attain negative values, but on average better approximates the true VDF. The VDFs or projected VDFs are computed by the routine `computeVelocityDistribution<N>`.

2.6.3 Conversion to/from *N*-body models

As the DF is a probability distribution function (PDF), it can be sampled with a large number of points to create an N-body model of the system. There are two possible ways of doing this:

- Draw samples of actions from $f(\boldsymbol{J})$, used as a three-dimensional PDF. Then create (possibly several) $\{\boldsymbol{x}, \boldsymbol{v}\}$ points for each value of actions with a random choice of angles, using the torus mapping approach (Section 2.4.4). This is performed by the routine `generateActionSamples`.
- Draw samples directly from the six-dimensional $\{\boldsymbol{x}, \boldsymbol{v}\}$ space, evaluating $f\big(\boldsymbol{J}(\{\boldsymbol{x}, \boldsymbol{v}\})\big)$ with the help of an action finder. This is performed by the routine `generatePosVelSamples`.

Both approaches should in principle deliver an equivalent discrete representation of the model, but may have a different cost; generally, the second one is preferred.

There is also a related task for sampling just the density profile $\rho(\boldsymbol{x})$ with particles, without assigning any velocity to them; this may be used to visualize the density model, and is performed by the routine `generateDensitySamples` (of course, it does not use any action finder). All these tasks employ the adaptive multidimensional rejection method implemented in `math::sampleNdim`.

The inverse procedure for constructing a DF from a given N-body model is less well defined. In the case of a spherical isotropic system (Section 2.5.4), the one-dimensional function of phase volume $f(h)$ is estimated non-parametrically with the penalized density fitting method and represented as a spline in scaled coordinate (the routine `fitSphericalDF`). In principle this may be generalized for the case of a three-dimensional $f(\boldsymbol{J})$, but this has not been implemented yet. The alternative is to fit a parametric DF to the array of actions, computed for the N-body particles in the given potential (of course, a suitable self-consistent

`Multipole` or `CylSpline` potential itself may also be constructed from the same N-body model). This approach is demonstrated by one of the example programs (Section 4).

2.6.4 Iterative self-consistent modelling

As explained above, the same DF gives rise to a different density profile in each potential. A natural question is whether there always exists a unique potential-density pair ρ, Φ such that $\rho(\boldsymbol{x}) = \int d^3v\, f\big(\boldsymbol{J}(\boldsymbol{x}, \boldsymbol{v}\ |\Phi)\big)$ corresponds to $\Phi(\boldsymbol{x})$ via the Poisson equation, with the mapping $\{\boldsymbol{x}, \boldsymbol{v}\} \implies \boldsymbol{J}$ constructed for the same potential. While we are not aware of a strict mathematical proof, in most practical cases the answer is positive, and such potential may be constructed by the iterative self-consistent modelling approach [3, 29]. In a more general formulation, one may have several DF components $f_c(\boldsymbol{J}), c = 1..N_{\rm comp}$ and optionally several additional (external) density or potential components. The procedure consists of several steps, which use various pieces of machinery described previously:

1. Create a plausible initial guess for the total potential $\Phi(\boldsymbol{x})$.
2. Construct the action finder for this potential (Section 2.4).
3. Compute the density profiles $\rho_c(\boldsymbol{x})$ of all components with DFs (Section 2.6.1).
4. Calculate the updated potential by solving the Poisson equation $\nabla^2\Phi = 4\pi\sum_c \rho_c$ for the combined density of all components (plus any external density or potential components, which are called static since they are not updated throughout the iterative procedure), using one or both general-purpose potential expansions (Sections 2.2.2, 2.2.3).
5. If desired, add new components or replace a static component with a DF-based one.
6. Repeat from step 2, until the potential changes negligibly between iterations. This typically requires $\mathcal{O}(10)$ steps.

The only non-trivial aspect of this procedure is to choose whether the density of a given component is better described as spheroidal (not strongly flattened, possibly with a central cusp or an extended envelope) or disky (possibly strongly flattened, but with a finite-density core and a finite extent, or at least sharply declining at large radii). In the first case it will contribute to the potential represented by the `Multipole` expansion, and in the second – by the `CylSpline` expansion. This applies to both DF-based and static density components; in addition there may be static components with already known potentials, which will be added directly to the total potential. Importantly, all disky components will be represented by a single `CylSpline` object, and similarly all spheroidal components by a single `Multipole` object. The density of each DF-based component is first computed on a suitable grid of $\mathcal{O}(10^2 - 10^3)$ points, and a corresponding density interpolator (`DensitySphericalHarmonic` – Section A.3.1, or `DensityAzimuthalHarmonic` – Section A.3.2) is created that will be used

in solving the Poisson equation. Presently this method is restricted to axisymmetric models, due to the lack of more general action finders.

This approach is implemented with the help of several classes derived from `BaseComponent`, the `SelfConsistentModel` structure which binds together the array of components, the potential, the action finder, and the parameters of potential expansions, and finally the routine `doIteration`, all defined in `galaxymodel_selfconsistent.h`. All these concepts are also available in the `Python` wrapper (Section 3.1), and a complete annotated example illustrating the entire workflow is presented both in the `C++` and `Python` variants.

2.7 Data handling

2.7.1 Selection functions

3 Interfaces with other languages and frameworks

3.1 `Python` interface

The `Python` interface provides a large subset of Agama functionality expressed as Python classes and routines. Presently, this includes:

- A few mathematical tasks such as multidimensional integration and sampling, and penalized spline fitting.
- Unit handling.
- Potential and density classes.
- Orbit integration.
- Action finders (both classes and standalone routines).
- Distribution functions.
- Galaxy modelling framework: computation of DF moments, drawing samples from density profiles and DFs, iterative self-consistent modelling.

The shared library `agama.so` can be used directly as a `Python` extension module, by writing `import agama` in the `.py` file. The `Density`, `Potential`, `DistributionFunction` classes serve as universal proxies to the underlying hierarchy of `C++` classes, and their constructors take a variety of named arguments covering all possible variants (as usual, writing `help(agama.Whatever)` brings a complete description of the class or routine and its arguments). Additionally, density and DF object may also be represented by an arbitrary user-defined Python function – this can be used in all contexts where a corresponding interface is needed, e.g., in constructing a potential expansion from a density profile, or in computing DF moments, which greatly increases the flexibility of the `Python` interface. Most routines

or methods that operate on individual points in `C++` (such as action finders or potentials) can accept `numpy` arrays in `Python`, which again leads to a more concise code with nearly the same efficiency as a pure `C++` implementation. There are several example programs demonstrating the usage of various aspects of the library (some of them have exact `C++` equivalents).

3.2 `Fortran` interface

The `Fortran` interface is much more limited compared to the `Python` interface, and provides access to the potential solvers only.

One may create a potential in several ways:

1. Load the parameters from an INI file (one or several potential components).
2. Pass the parameters for one component directly as a single string argument.
3. Provide a `Fortran` routine that returns a density at a given point, and use it to create a potential approximation with the parameters provided in a text string.
4. Provide a `Fortran` routine that returns potential and force at a given point, and create a potential approximation for it in the same way as above (this is useful if the original routine is expensive).

Once the potential is constructed, the routines that compute the potential, force and its derivatives (including density) at any point can be called from the `Fortran` code. No unit conversion is performed (i.e., $G = 1$ is implied). There is an example program showing all these modes of operation.

3.3 Amuse plugin

AMUSE [30] is a heterogeneous framework for performing and analyzing N-body simulations using a uniform approach to a variety of third-party codes. The core of the framework and the user scripts are written in `Python`, while the community modules are written in various programming languages and interact with each other using a standartized interface.

AGAMA may be used to provide an external potential to any N-body simulation running within AMUSE. The plugin interface allows to construct a potential using either any of the built-in models, or a potential approximation constructed from an array of point masses provided from the AMUSE script. This potential presents a `GravityFieldInterface` allowing it to be used as a part of the `Bridge` coupling scheme in the simulation. For instance, one may study the evolution of a globular cluster that orbits a parent galaxy, by following the internal dynamics of stars in the cluster with an N-body code, while the galaxy is represented by a static potential using this plugin. An example script is provided in the AMUSE plugin folder.

3.4 Galpy plugin

GALPY [8] is a `Python`-based framework for galaxy modelling, similar in scope to AGAMA. It includes a collection of gravitational potentials, routines for orbit integration, action finders, distribution functions and more. The potential solvers and action finders from AGAMA may be seamlessly integrated into the GALPY framework with the help of a thin compatibility layer on top of the standard AGAMA `Python` interface. This layer introduces a GALPY-compatible potential object that can be constructed in the same way as the `agama.Potential` class in `Python`, but provides the methods suitable for use in GALPY. Orbit integration and action finders can only be used with AGAMA potentials, but they produce results in the form compatible with GALPY.

An example, comparing the native GALPY action finder with that from AGAMA, is provided in the file `pytests/test_actions_galpy.py`. Overall, the potential approximations and action finders in AGAMA are more versatile, accurate and computationally efficient, while GALPY provides a convenient plotting interface.

3.5 Nemo plugin

NEMO [39] is a collection of programs for performing and analyzing N-body simulations, which use common data exchange format and UNIX-style pipeline approach to chain together several processing steps. The centerpiece of this framework is the N-body simulation code GYRFALCON [14]. It computes the gravitational force between particles using the fast multipole method, and can optionally include an external potential.

The NEMO plugin allows to use any AGAMA potential as an external potential in GYRFALCON and other NEMO programs (in a similar context as the AMUSE plugin). The potential may be specified either as a file with coefficients (for potential expansions), or more generally, as an INI file with parameters of possibly several components defined in groups `[Potential1]`, `[Potential whatever]`, ...

To build the plugin, one needs to have NEMO installed (obviously) and the environment variable $NEMO defined; then `make nemo` will compile the plugin and place it in `$NEMO/obj/acc` folder, where it can be found by NEMO programs. For instance, this adds an extra potential in a GYRFALCON simulation:

```
$ gyrfalcON infile outfile accname=agama accfile=mypot.ini [accpars=1.0] ...
```

where the last optional argument specifies the pattern speed Ω (frequency of rotation of the potential figure about z axis). All units in the INI or coefs file here should follow the convention $G = 1$.

4 Tests and example programs

The Agama library itself is indeed just a "library", not a "program", but it comes with a number of example programs and internal tests. The latter ones are intended to ensure the consistency of results as the development goes on, so that new or improved features do not break any existing code. All `test_***.cpp` programs are intended to run reasonably quickly and display either a PASS or FAIL message; they also illustrate some aspects of the code, or check the accuracy of various approximations on realistic data. Example programs, on the other hand, are more targeted towards the library users and demonstrate how to perform various tasks. Some of them are described below [not everything is ready yet].

example_actions_nbody shows how to determine the actions for particles from an N-body snapshot taken from a simulation of a disk+halo system. It first reads the snapshot and constructs two potential approximations – `Multipole` for the halo component and `CylSpline` for the disk component – from the particles themselves. Then it computes the actions for each particle and writes them to another file. This program exists both in `C++` and `Python` variants that perform the same task.

example_df_fit shows how to find the parameters of a DF belonging to a particular family from a collection of points drawn from this DF in a known potential. It first computes the actions for these points, and then uses the multidimensional minimization routine `findMinNdim` to locate the parameters which maximize the likelihood of the DF given the data. The `Python` equivalent of this program additionally determines the confidence intervals on these parameters by running a MCMC algorithm starting around the best-fit parameters.

A more elaborate example in `Python` determines simultaneously the parameters of the spherically-symmetric potential and the DF that together describe the mock data points drawn from a certain (non-self-consistent) DF but with incomplete data (only the line-of-sight velocity and the projected distance from the origin). The goal is to determine the properties of the potential, treating the DF as nuisance parameters; it also uses the MCMC algorithm to determine uncertainties. This example is inspired by one of the Gaia Challenge assignments [32].

example_self_consistent_model illustrates various steps of the workflow for creating multicomponent self-consistent galaxy models determined by DFs. It begins with a static density profile of the disk and a DF-based halo component, and performs a few iterations to find the potential generated by the halo. It then replaces the disk component with a DF-based one (the expressions for $f(\boldsymbol{J})$ use epicyclic frequencies computed in the approximate potential delivered by the first stage), and performs a few more iterations, updating the density of both disk and halo components. Finally, it creates an N-body realization of

the composite system by sampling particles from both DFs in the converged potential. It also demonstrates the use of INI files for keeping parameters of the model. This example is provided in equivalent `C++` and `Python` versions.

example_fokker_planck is a `C++` program for computing the evolution of a spherical isotropic stellar system driven by two-body relaxation, using the formalism presented in Section A.4.1.

example_galpy is a `Python` program showing the use of AGAMA plugin for GALPY to construct a potential, integrate orbits, and compare the accuracy of action finders between the two libraries.

example_fortran demonstrates how to create and use AGAMA potentials in `Fortran`, both for built-in density or potential models, or for user-defined `Fortran` functions that provide the density or potential.

A Technical details

A.1 Developer's guide

Any large piece of software needs to follow a number of generic programming rules, which are well-known standards in commercial software development, but unfortunately are insufficiently widespread in the scientific community. Here we outline the most important guidelines adopted in the development of Agama. Some of them are `C++`-specific [26, 38], others are more general [22, 23]. As a practical matter, we do not use any of `C++11` features, except smart pointers, to keep compatibility with older compilers.

Code readability is extremely important in long-term projects developed and used by several persons. All public classes, types and routines in Agama are documented in-code, using the Doxygen syntax for comments that can be parsed and used to automatically generate a collection of HTML pages. These comments mostly describe the intent of each class and function and the meaning of each argument or variable, at least in their public interface – in other words, a programmer's reference to the library. Sometimes a more technical description is also provided in these comments, but generally it is more likely to be presented in this document rather than in the code (with the inevitable risk of desynchronizing as the code development progresses...)

Modularity is an essential approach for keeping the overall complexity at a reasonable level. What this means in practice is that each unit of the code (a class or a function) should be responsible for a single well-defined task and provide a minimal and clean interface to it, isolating all internal details. The calling code should make no assumptions about the implementation of the task that this unit of code is promised to deliver. On many occasions, there are several interchangeable back-ends for the same interface – this naturally applies to all class hierarchies descending from a base abstract class such as `BasePotential`, but also to the choice of back-end third-party libraries dictated by compilation options, with a single wrapper interface to all alternatives implementations.

Another facet of modularity is loose coupling, that is, instead of a single large objects that manages many aspects of its internal state, it is better to create a number of smaller objects with minimal necessary interaction. For instance, composition (when one class has another class as a member variable) is preferred over inheritance (when the class has full access to the parent class's private members), as it reduces the strength of coupling.

Programming paradigm throughout the library is a mixture of object-oriented and procedural, gently spiced with template metaprogramming.

Generally, when there is a need to provide a common interface to a variety of implementations, the choice between compile-time (templates) and run-time (virtual functions) polymorphism is dictated by the following considerations.
Templates are more efficient because the actual code path is hardwired at the compilation time, which allows for more optimizations and diagnoses more possible errors already at this stage. On the other hand, it is applicable when the actual workflow is syntactically the same, or the number of possible variants is known in advance – for instance, conversion between all built-in coordinate systems (Section 2.1.3) is hard-coded in the library. Each function that uses a templated argument produces a separate compiled fragment; therefore it is impossible for the user to extend built-in library functions with a new variety of template parameter.
Abstract classes (or, rather, "interfaces") providing virtual functions that are fleshed out in descendant classes offer more flexibility, at the expense of a small overhead (negligible in all but the tighest loops) and impossibility to securely prevent some errors. This is the only way to provide a fully extensible mechanism for supplying a user-defined object (e.g., a mathematical function implementing a `IFunction` interface) into a pre-compiled library function such as `findRoot`.

The boundary between object-oriented and procedural paradigms is less well-defined. There are several possible ways of coupling the code and the data:

1. data fields are encapsulated as private members of a class, and all operations are provided through public methods of that class;
2. a collection of assorted variables is kept in a structure or array, and there is a standalone function performing some operation on this data;
3. a standalone function takes an instance of a class and performs some operation using public member functions of this class;
4. a class contains a pointer, reference or a copy of another class or structure, and its member functions follow either of the two previous patterns.

The first approach is used for most classes that provide nontrivial functionality and can be treated as immutable objects, or at least objects with full control on their internal state. If the data needs to be modified, it is usually kept in a structure with public member fields and no methods, so that it may be accessed by non-member functions (which need to check the correctness of data on each call). The third approach is used mostly for classes that are derived from an abstract base class that declares only virtual methods; since any non-trivial operation on this class only uses this public interface, it does not need to be a part of the class itself, thus loosening the coupling strength. That's why we have many non-member functions operating on `BasePotential` descendants. Finally, the fourth scenario is the preferred way of creating layered and weakly coupled design.

Naming conventions are quite straightforward: class names start with a capital letter and variable names or function arguments – with a lowercase, constants are in all capital with underscores, and other names are in CamelCase without underscores. Longer and more descriptive names are preferred – as a matter of fact, we read the code much more than write, so it's better to aid reading than to spare a few keystrokes in writing.

We use several namespaces, roughly corresponding to the overall structure of the library as described in Section 2: this improves readability of the code and helps to avoid naming collisions, e.g., there could be two different `Isochrone` classes – as a potential and as a concept in stellar evolution, living in separate namespaces. A feature of `C++` called "argument-dependent lookup" allows to omit the namespace prefix if it can be deduced from the function arguments: for instance, if `pot` is an instance of class derived from `potential::BasePotential`, we may call ~~`potential::`~~`writePotential(fileName, pot)` without the prefix. This doesn't apply to name resolution of classes and templates, and to functions which operate on builtin types (e.g., in the `math::` namespace). We also do not use the `auto` keyword which is only available in `C++11`.

When several different quantities need to be grouped together, we use `struct` with all public members and no methods (except possibly a constructor and a couple of trivial convenience functions). If something has an internal state that needs to be maintained consistently, and provides a nontrivial behaviour, this should be a `class` with private member variables. We prefer to have named fields in structures rather than arrays, e.g., a position in any coordinate system is specified by three numbers, but they are not just a `double pos[3]` – rather, each case has its own dedicated type such as `struct PosCyl { double R,z,phi; };` (Section 2.1.3). This eliminates ambiguity in ordering the fields (e.g., what is the 3rd coordinate in a cylindrical system – z or ϕ? different codes may use different conventions, but naming is unique) and makes impossible to accidentally mis-use a variable of an incorrect type which has the same length.

Immutability of objects is a very powerful paradigm that leads to simpler design and greater robustness of programs. We allow only "primitive" variables – builtin types, `struct`s with all public member fields, or arrays (including vectors and matrices) – to change their content. Almost all instances of non-trivial classes are read-only: once created, they may not be changed anymore; if any modification is needed, a new object should be constructed. All nontrivial work in setting up the internal state is done in the constructor, and all member functions are marked as `const`. This convention is a strong constraint that allows to pass around complex objects between different parts of the code and be sure that they always do the same thing, and that there are no side effects from calling a method of a class. This also simplifies the design of parallel programs: if an object needs a temporary workspace for some function to operate, it should not be allocated as a private variable in the class, but rather as a temporary variable on the stack in each function; thus concurrent calls

to the same routine from different threads do not interfere, because each one has its own temporary variable, and only share constant member variables of the class instance. There are, of course, some exceptions, for instance, in classes that manage the input/output, or string collections (`utils::KeyValueMap`).

Another aspect of the same rule is `const` correctness of the entire code. Instances of read-only classes may safely be declared as `const` variables, and all input arguments for functions are also marked as `const` (see below). These rules improve readability and allow many safety checks to be performed at compile time – an incorrect usage scenario will not even compile, rather than produce an unexpected error at runtime.

Memory management is a non-trivial issue in `C` and a source of innumerable bugs in poorly written software. Fortunately, `C++` has very powerful features that almost eliminate these problems, if followed consistently. The key element is the automatic management of object lifetimes for all classes or structures. Namely, if a variable of some class is created in a block of code, the destructor of this class is guaranteed to be called when the variable goes out of scope – whether it occurs in a normal code path or after an exception has occurred (see below). Thus if some memory allocation for a member variable was done in the constructor, it should be freed in the destructor and not in any other place. And of course the rule applies recursively, i.e., if a member variable of a class is a complex structure itself, its destructor is called automatically from the destructor of this class, and then the destructor of the parent class (if it exists) is invoked. In practice, it is almost never necessary to deal with these issues explicitly – by using standard containers such as `string`s and `vector`s instead of `char*` or `double*` arrays, one transfers the hassle of dynamic memory management entirely to the standard library classes.

The picture gets more complicated if we have objects that represent a hierarchy of descendants of an abstract base class, and need to create and pass around instances of the derived types without knowing their actual type. In this case the object must be created dynamically, and a correct destructor will be automatically called when an object is deleted – but the problem is that a raw pointer to a dynamically-allocated object is not an object and must be manually deallocated before it goes out of scope, which is precisely what we want to avoid. The solution is simple – instead of raw pointers, use "smart pointers", which are proxy objects that manage the resource themselves. There are several kinds of smart pointers, but we generally use only one – `shared_ptr`. Its main feature is automatic reference counting: if we dynamically create an object derived from `BasePotential` and wrap the pointer into `PtrPotential` (which is defined as `shared_ptr<const BasePotential>`), we may keep multiple copies of this shared pointer in different routines and objects, and the underlying potential object will stay alive as long as it is used in at least one place, and will be automatically deallocated once all shared pointers go out of scope and are destroyed. If a new value is assigned to the same shared pointer, the reference counter for the old object

is also decreased, and it is deallocated if necessary. Thus we never need to care about the lifetime of our dynamically created objects; this semantics is similar to `Python`. Of course, if we know the actual type of potential that we only need locally, we may create it on the stack without dynamical allocation; most routines only need a (`const`) reference to a potential object – does not matter whether it is an automatic local variable or a dereferenced pointer.

Finally, it's better to avoid dynamical memory allocation (including creation of `std::vector`s) in routines that are expected to be called frequently (such as `BasePotential::eval()`). All temporary variables should be created on the stack; if the size of an array is not known at compile time (e.g., it depends on the parameters of potential), we either reserve an array of maximum permitted size, or use `alloca()` routine which creates a variable-length array on the stack.

Calling conventions refer to the way of passing and returning data between various parts of the code. Arguments of a function can be input, output, or both. All input arguments are either passed by value (for simple built-in types) or as a `const` reference (for more complex structures or classes); if an argument may be empty, then it is passed as a const pointer which may take the `NULL` value. Output and input/output arguments are passes as non-`const` references to existing objects. Thus the function signature unambiguously defines what is input and what is not, but does not indicate whether a mixed-intent argument must have any meaningful value on input – this should be explained in the DOXYGEN comment accompanying the definition. Unfortunately there is no indication of the direction of arguments at the point where the function is called.

Usually the input arguments come first, followed by output arguments, except the cases of input arguments with default values, which must remain at the end of the list. (Unfortunately, `C++` does not have named arguments, which would be more descriptive, but we encourage their use in the `Python` interface). When the function outputs a single entity (even if it is a complex object), it is usually a return type, not an output argument; in most contexts, there is no extra cost because temporary objects are not created (copy elision and return-value optimization rules). However, extra output information may be stored in output arguments (sometimes optional, i.e., they may be `NULL`, indicating that this extra information is not required by the caller). When the return value is not a copyable type (e.g., if a function creates a new instance of a class derived from an abstract base class), then it is returned as a smart pointer.

These conventions apply to ordinary non-member functions and class methods; for constructors they are somewhat different. If we create an object A which has a link to another object B, it usually should not be just a reference or a raw pointer – because the lifetime of B may be shorter than the newly created A. In these cases, B is either copied by value (like a `std::vector`), or else it should be provided as a shared pointer to the actual object, and a copy of this pointer is stored in A, increasing its reference counter. This ensures that the

actual instance of B continues to exist as long as it is used anywhere, and is automatically destroyed when it is no longer needed. Thus, if a class constructor takes a shared pointer as an argument, this indicates that a copy of this pointer will be kept in the class instance during its lifetime; if it takes a reference, then it is only used within the constructor but not any longer. This rule also has exceptions – several wrapper classes used as proxy object for type conversion. For instance, when a certain routine (e.g., `math::integrate`) expects an argument of `const math::IFunction&` type to perform some calculations on it without storing the object anywhere else, this argument could be a temporary instance of a wrapper class (e.g., `potential::DensityWrapper`) taking a reference to a `const potential::BaseDensity&` object in the constructor. In other words, instances of these wrapper classes should only be created as unnamed temporary objects passed as an argument to another function, but not as stack- or heap-allocated variables – even local ones.

Numerical issues – efficiency and accuracy – are taken very seriously throughout the code. Floating-point operations often require great care in re-arranging expressions in a way that avoids catastrophic cancellation errors. A classic example is the formula for the roots of a quadratic equation: $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/(2a)$. In the case of $ac \ll b^2$, one of the two roots is a difference between two very close numbers, thus it may suffer from the loss of precision. Another mathematically equivalent expression is $2c/(-b \mp \sqrt{b^2 - 4ac})$, and a numerically robust approach is to use both expressions – each one for the root that has two numbers of the same sign *added*, not subtracted. Going one step further, if the coefficients a, b, c are themselves obtained from other expressions, it may be necessary to reformulate them in such a way as to avoid subtraction under the radical, etc. These details are necessary to ensure robust behaviour in all special and limiting cases; a good example are coordinate conversion routines.

Efficiency is also a prime goal: this includes a careful consideration of algorithmic complexity and minimization of computational effort. Some mathematically equivalent functions have rather different computational cost: for instance, generating two Gaussian random numbers with the Box–Muller algorithm is a few times faster than using the inverse error function; finding a given quantile (e.g., median) of an array can be done in $\mathcal{O}(N)$ operations without sorting the entire array (which costs $\mathcal{O}(N \log N)$ operations); and computing potential and three components of force simultaneously is faster than doing it separately. For numerical integration, a suitable coordinate transformation may dramatically improve the accuracy – or reduce the number of function calls in the adaptive integration routine; often a fixed-order Gauss–Legendre integration is enough in a particular case, with the degree of quadrature selected by extensive numerical experiments.

A few words about `INFINITY` and `NAN` values. Infinities are valid floating-point numbers and are quite useful in some contexts where a really large number is required (for instance, as the endpoint of the root-finder interval); they propagate correctly through most expressions

and some functions (e.g., `exp`, `log`, comparison operators). The infamous `NAN` is a different story: it usually[5] indicates an incorrect result of some operation, and is infectious – it propagates through all floating-point operations, including comparison operators (that is, `a>b` and `a<b` are *both* false if `b` is `NAN`; however, `!(a<b)` and `b!=b` are true in this case). This feature is useful to pass the indication of an error to the upper-level routines, but it does not allow to tag the origin of the offensive operation. This brings us to the next topic:

Error handling is an indispensable part of any software. Whenever something goes wrong, this must be reported to the upper-level code – unless there is a safe fallback value that may be returned without compromising the integrity of calculation. The standard approach in `C++` is the mechanism of exceptions. They propagate through the entire call stack, until handled in a `catch` statement – or terminate the program if no handler was found. They also carry upward any user-defined diagnostic information (e.g., a string with error description), and most importantly, they ensure a correct disposal of all temporary objects created in intermediate routines (of course, if these are real objects with destructors, not just raw pointers to dynamically-allocated memory – which are bad anyway). Thus a routine does not need to care about a possible exception occurring at a lower level, if it cannot handle it in a meaningful way – it should simply let it propagate upwards. Exceptions should be used to check that the input parameters are correct and consistent with the internal state of an object, or perhaps to signal an un-implemented special case. Within a constructor of a class, they are the only available mechanism for error handling – since a constructor cannot return a value, and storing an error code as a class member variable doesn't make sense, because the object is not usable anyway. Instead, if a constructor fails, the object is immediately and correctly destroyed.

On the other hand, if a certain condition is never expected to occur, this may be expressed as an `assert` statement – which terminates the program unconditionally if violated, and this clearly indicates some internal inconsistency in the code, e.g., a memory corruption. It also promotes a self-documenting code – all assumptions on input parameters and (preferrably) results of calculation (pre- and post-conditions) are clearly visible. This mechanism should only be used within a single unit of code (e.g., a class), which has a full control on its internal state; if a function is part of public interface, it may not assume that the passed arguments are valid and should check them, but in the case of incorrect values should raise an exception rather than terminate the entire program.

Finally, it should be noted that exceptions do incur some run-time penalty if triggered, so they should not be used just to inform about something that may routinely occur, e.g., in a function that searches for a substring and does not find it. Sometimes propagating a `NAN`

[5] In some functions, `NAN` it is used as a special, usually a default value of an input argument, indicating something like "value is unknown".

is a cheaper alternative, used, for instance, in action finders if the energy is positive (does not correspond to bound motion).

Diagnostic output is a separate issue from error handling, and is handled by a dedicated printout routine `utils::msg` that may be used with different levels of verbosity and write the messages to console or a log file. Its behaviour is controlled at runtime by the environment variables `LOGLEVEL` (ranging from 0 to 3; default 0 means print only necessary messages, 1 adds some non-critical warnings, 2 prints ordinary debugging information, 3 dumps even more debugging information) and `LOGFILE` (if set, redirects output to the given file, otherwise it is printed to `stderr`). The library's default handler may be reassigned to a user-provided function.

Parallelization in Agama is using the `OpenMP` model, which is nearly transparent for the developer and user. Only a few operations that are supposed to occur in a serial context have internal loops parallelized: this includes the construction of `Multipole` and `CylSpline` potentials from density profiles or from `ParticleArray`s, and sampling from a multidimensional probability density. The former situation occurs, for instance, in the context self-consistent modelling (Section 2.6.4), when the evaluation of density at each point is a costly operation involving multidimensional integration of distribution function over velocities and thousands of calls to an action finder, and the values of density at different points are collected in parallel. Other typical operations, such as computation of potential or action for many points simultaneously, should be paralellized in the caller code itself. Almost all classes and functions provided by the library can be used from multiple threads simultaneously, because they operate with read-only or thread-local data (exceptions from this rule are linear and quadratic optimization routines, which are not thread-safe, but hardly would need to be called in parallel); we do not have any mutex locks in the library routines. For instance, in the `Python` interface, a single call to the potential or action finder may provide an array of points to work with, and the loop is internally parallelized in the `C++` extension module.

An important thing to keep in mind is that an exception that may occur in a parallel section should be handled in the same section, otherwise the program immediately aborts. Thus in such loops it is customary to provide a general handler that stores the error text, and then re-throws an exception when the loop is finished. Also, the `Python` interface provides a way to supply a user-defined `Python` callback function to some of the routines implemented in `C++`, but the standard `Python` interpreter has a global locking mechanism preventing its simultaneous usage from multiple threads. Therefore, when such callback functions are used with `C++` routines, this temporarily disables `OpenMP` parallelization. (An alternative that is not yet implemented is to allow this callback function to handle a vectorized input instead of one point at a time).

A.2 Mathematical methods

A.2.1 B-splines

We often need to represent approximately an arbitrary function of one variable, defined on a finite or infinite interval $[a, b]$. This is conventionally achieved by defining the inner product of two functions $f(x), g(x)$:

$$\langle f, g \rangle \equiv \int_a^b f(x)\, g(x)\, W(x) dx \;, \tag{1}$$

where $W(x)$ is the weight function, and introducing a complete set of basis functions $B_i(x)$, so that any sufficiently well-behaved function $f(x)$ can be approximated by a weighted sum with a finite number of terms M to any desired accuracy:

$$\tilde{f}^{(M)}(x) = \sum_{i=1}^{M} A_i\, B_i(x) \;. \tag{2a}$$

The coefficients of expansion A_i satisfy the linear system of equations

$$C_j \equiv \int_a^b \tilde{f}^{(M)}(x)\, B_j(x)\, W(x)\, dx = \sum_{i=1}^{M} A_i\, \mathsf{B}_{ij} \;, \tag{2b}$$

$$\mathsf{B}_{ij} \equiv \int B_i(x)\, B_j(x)\, W(x)\, dx \;. \tag{2c}$$

Classical basis sets usually are orthonormal, i.e., $\mathsf{B}_{ij} = \delta_{ij}$, and addition of each subsequent term does not change existing expansion coefficients (e.g., Fourier series, orthogonal polynomials). Of course, it is possible to construct an orthogonal set by employing the Gram–Schmidt procedure for any sequence of independent basis functions. However, it is not always necessary, as long as we can solve efficiently the linear system (2b) to find A_i from C_j.

The B-spline set of basis function is defined by a grid of points (knots) on the interval $[a, b]$: $a = k_1 < k_2 < \cdots < k_K = b$. Each basis function is a piecewise polynomial of degree $N \geq 0$ that is non-zero on at most $N + 1$ consecutive segments of the grid (or fewer at the edges of the grid). Specifically, it is a polynomial inside each grid segment, and its $N - 1$-th derivative is continuous at each knot (except the endpoints a, b, but including all interior knots). The total number of basis functions is $M = K + N - 1$. These functions are defined through the following recurion (de Boor's algorithm):

$$B_j^{[0]}(x) \equiv \begin{cases} 1 & \text{if } k_j \leq x \leq k_{j+1} \\ 0 & \text{otherwise} \end{cases} \;, \tag{3a}$$

$$B_j^{[N]}(x) \equiv B_j^{[N-1]}(x)\, \frac{x - k_j}{k_{j+N} - k_j} + B_{j+1}^{[N-1]}(x)\, \frac{k_{j+N+1} - x}{k_{j+N+1} - k_{j+1}} \;. \tag{3b}$$

B-splines have the following convenient properties:

- At any grid segment, at most $N+1$ basis functions are nonzero. This makes the computation of interpolant (2a) very efficient – the grid segment enclosing the point x is located in $\mathcal{O}(\log M)$ operations, and the computation of all N possibly non-zero basis functions takes $\mathcal{O}(N^2)$ operations (with N typically ranging from 0 to 3), instead of $\mathcal{O}(M)$ as for traditional basis sets.
- The basis is not orthogonal, but the matrix B_{ij} (2c) is block-diagonal with bandwidth $2N-1$, thus the coefficients of decomposition A_i are obtained from C_j in $\mathcal{O}(NM)$ operations. (The weight function $W(x)$ is unity).
- Although the number and degree of basis functions must be fixed in advance before computing any decompositions, we may use the freedom to put the knots at the most suitable locations to improve the accuracy of approximation.
- The basis functions are non-negative, thus to ensure that $f(x) \geq 0$, it is sufficient to have $A_i \geq 0$.
- The sum of all basis functions is always 1 at any x.

The case $N=0$ corresponds to a histogram (piecewise-constant function), $N=1$ – to a piecewise-linear interpolator, and $N=3$ – to a cubic spline with clamped boundary conditions (it is defined by K nodes but has $K+2$ independent components, unlike the more familiar natural cubic spline, in which the extra two degrees of freedom are used to make the second derivative zero at endpoints), see Figure 1 (left panel). In more than one dimension, the basis set is formed by tensor products of one-dimensional B-splines in each coordinate.

B-splines are well suited for constructing the approximating function with a relatively small number of terms from a possibly large array of points (essentially replacing the integral in (2b) by a discrete sum, see Sections A.2.3 and A.2.4). However, their construction from the array of function values at grid nodes (especially in more than one dimension), as is typical for spline interpolation, and the evaluation of the interpolant, is less efficient than conventional splines. Thus we consider the alternatives in the next section.

A.2.2 Spline interpolation

The task of interpolation in 1, 2 and 3 dimensions is performed by several kinds of splines: linear interpolators (not particularly exciting), cubic splines and quintic splines (the latter – only in 1d and 2d). The degree of spline (1, 3 or 5) refers to the degree of the piecewise polynomial at each grid segment (in more than one dimension, along each axis). However, the continuity conditions at grid nodes may be different from B-splines (in which the function has $N-1$ continuous derivatives at all interior nodes).

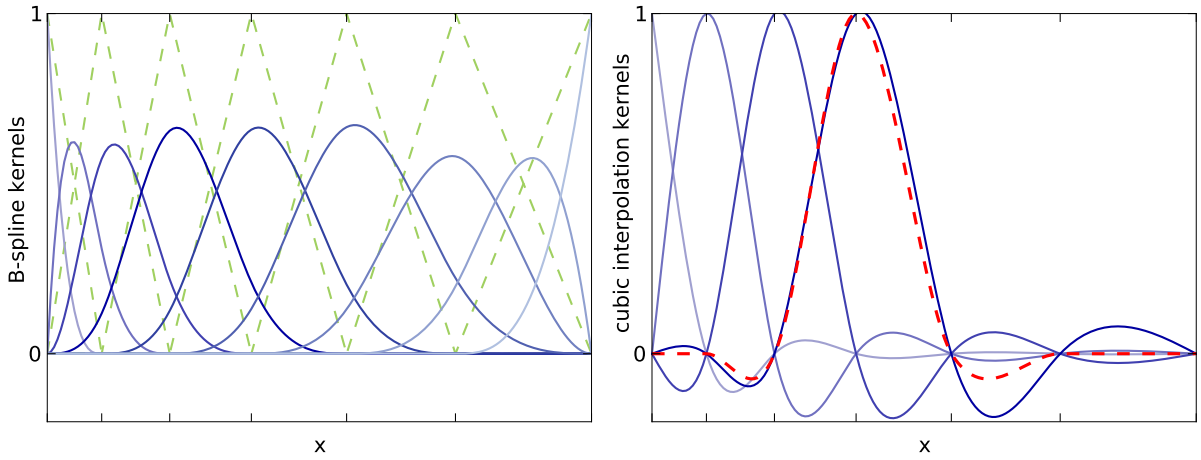

Figure 1: Various interpolation kernels. To obtain the value of interpolated function, one sums up the values of all interpolating kernels with appropriate weight coefficients.
Left panel: B-splines of degree $N = 1$ (dashed) and $N = 3$ (solid lines), defined by the nodes of a non-uniform grid (marked by x-axis ticks). The former are piecewise-linear and non-zero on at most two segments, and the latter are piecewise-cubic, with two continuous derivatives, and nonzero on at most four consecutive segments. The sum of all B-spline functions at any point x is unity, and they are always non-negative; thus the cubic kernels never reach unity (except the endpoint ones), because at any point more than one of them is positive. The total number of functions is $K + N - 1$, where K is the number of grid nodes.
Right panel: interpolation kernels of a natural cubic spline defined by the same grid (solid lines). This type of spline is constructed from the array of function values at grid nodes, thus each kernel reaches unity at the corresponding grid point, and the number of kernels is K (not all of them are shown). Consequently, they must attain negative values for their sum to be unity at any x. Moreover, since the weights of kernels are computed by solving a global linear system of equations, each kernel spans the entire grid, although its amplitude rapidly decreases away from its central node. For comparison, a cubic interpolating Catmull–Rom kernel (dashed curve) is nonzero only on four adjacent grid segments, although it also attains negative values on the two outermost segments.

Let us first consider the 1d case with $K \geq 2$ grid points ($K-1$ segments).

Interpolation by piecewise-cubic polynomials requires 4 coefficients for each segment, which are conveniently taken to be the values and first derivatives of the function at two adjacent nodes: $f(x_i), f(x_{i+1}), f'(x_i), f'(x_{i+1})$ – this is called the Hermite interpolation. Thus the total number of coefficients is $2K$; the function and its derivative is continuous across segments, but the second derivative may change discontinuously.

What if we are only given the function values $f(x_i)$, but not the derivatives? One may come up with a plausible approximation for derivatives at each node, and then use the Hermite interpolation on each segment – this will yield a continuously differentiable curve no matter what the values of $f'(x_i)$ are. Of course, we want it not only to be smooth, but also to approximate the true function accurately, and this requires a judicious assignment of derivatives. One possibility is to use finite-differences to estimate $f'(x_i)$ as $[f(x_{i+1}) - f(x_{i-1})]/[x_{i+1} - x_{i-1}]$, with a suitable modification for boundary points, or a generalization for unequally-spaced grids. This is called the Catmull–Rom spline, and is frequently used in resampling (especially in more than one dimension), due to its locality: the value of interpolated function on each interval $x_i \leq x \leq x_{i+1}$ depends on four nearby values – $f(x_{i-1}), f(x_i), f(x_{i+1}), f(x_{i+2})$. Another possibility is the familiar cubic spline, in which the first derivatives are computed from the requirement that the *second* derivatives are continuous at each interior node (i.e. $K-1$ points). This results in a tridiagonal linear equation system, relating $f'(x_i)$ to $f'(x_{i-1})$ and $f'(x_{i+1})$[6] Two additional boundary conditions are required to close the equation system; most commonly, these are $f''(x_1) = f''(x_K) = 0$ (so-called natural cubic splines), but alternatively, one may specify the first derivatives at the endpoints (clamped cubic spline). Thus the derivatives, and consequently the interpolated curve, depend on the values of f at all grid nodes, not just the adjacent ones; since $f'(x_i)$ are expressed as a linear function of $f(x_1) \dots f(x_K)$, we may consider the result of interpolation as a smoothing kernel defined by the grid nodes and linearly depending on all input values $f(x_i)$ (see Figure 1, right panel, for comparison of Catmull–Rom and natural cubic spline interpolation kernels).

Thus natural cubic splines are defined by K function values at grid points and are a subset of all cubic splines with K grid points (parametrized by $K+2$ numbers); the latter, in turn, are a subset of a wider class of piecewise-cubic Hermite interpolators (which are fully specified by $2K$ coefficients). The set of all cubic splines is also equivalent to B-splines of degree 3 over the same grid. The evaluation of cubic splines and optionally their derivatives is more efficient than the B-splines, and the array of B-spline amplitudes may be directly

[6] Usually this is expressed as a relation between second derivatives at grid nodes, and the spline function is defined in terms of $f(x_i)$ and $f''(x_i), i = 1..K$. Since all cubic splines are a subset of Hermite piecewise-cubic polynomial interpolators, we may equivalently parametrize them by the values and *first* derivatives at grid nodes, and this automatically ensures that the second derivative is continuous because the first derivative was computed from this condition.

used to initialize an equivalent clamped cubic spline.

If, on the other hand, one can independently compute both $f(x_i)$ and $f'(x_i)$ at all grid nodes, and use these $2K$ numbers to construct a piecewise-cubic Hermite interpolator, this should generally improve the accuracy of approximation (even though will decrease its *smoothness* compared to the case of cubic splines). In practice, within the class of piecewise-cubic polynomials the improvement is not dramatic. However, one may instead go to higher order and use piecewise-quintic polynomials, specified by 6 coefficients on each segment, which are again conveniently taken to be the values and first two derivatives of the function at two adjacent nodes – a natural extension of cubic Hermite interpolation. The resulting curve will be twice continuously differentiable for any choice of $f''(x_i)$, but the accuracy of approximation will be good only if these second derivatives are assigned carefully. In close analogy to cubic splines, one may compute them from the requirement that the 3rd derivative is continuous at all interior grid nodes, augmented with two boundary conditions; the natural choice for them is $f''''(x_1) = f''''(x_K) = 0$, because in a degenerate case $K = 2$ they simply lead to a cubic Hermite interpolator.

$$\frac{1}{3}f'''_i = 20\frac{f_{i+1}-f_i}{(x_{i+1}-x_i)^3} - \frac{12f'_i+8f'_{i+1}}{(x_{i+1}-x_i)^2} - \frac{3f''_i-f''_{i+1}}{x_{i+1}-x_i} = \tag{4a}$$

$$= 20\frac{f_i-f_{i-1}}{(x_i-x_{i-1})^3} - \frac{12f'_i+8f'_{i-1}}{(x_i-x_{i-1})^2} + \frac{3f''_i-f''_{i-1}}{x_i-x_{i-1}} \quad \text{for } 2 \le i \le K-1,$$

$$0 = 30\frac{f_2-f_1}{(x_2-x_1)^3} - \frac{16f'_1+14f'_2}{(x_2-x_1)^2} - \frac{3f''_1-2f''_2}{x_2-x_1} \quad \text{for } i = 1, \text{ and similarly for } i = K. \tag{4b}$$

This tridiagonal system results in a quintic spline, which provides 5th degree piecewise-polynomial interpolation with three continuous derivatives. It is *not* equivalent to a 5th degree B-spline – the latter would have 4 continuous derivatives and is fully specified by $K + 4$ coefficients, whereas a quintic spline is specified by $2K + 2$ numbers (the values and first derivatives at all nodes, plus two endpoint conditions). The accuracy of approximation with a quintic spline is far better than anything achievable with a cubic interpolation – but only in the case when one may compute the function derivatives independently and accurately enough (i.e., using a cubic spline to find the derivatives results in a quintic spline which is almost equivalent to a cubic one in terms of accuracy).

Let's move to the multidimensional case, where the interpolation is provided on a separable grid with piecewise-polynomial functions of degree N (per dimension) on each cell: $\sum_{p=0}^{N}\sum_{q=0}^{N}\ldots C_{pq\ldots}x_1^p\,x_2^q\ldots$ A straightforward way of performing a multidimensional interpolation is to employ a sequence of 1d interpolation operations for each dimension separately. The 1d interpolation function is defined by $N+1$ coefficients – the values and certain derivatives of the polynomial at two adjacent nodes in the d-th coordinate. These coefficients are obtained from $(N-1)$-dimensional interpolation along other coordinates in each of these two nodes.

To illustrate this, consider the case of piecewise-cubic Hermite interpolation, specified by the value and the first derivative in each dimension. For a 2d point $\{x, y\}$, we first locate the indices of grid nodes $\{i, j\}$ in each dimension that enclose the point: $x_i \le x \le x_{i+1},\ y_j \le y \le y_{j+1}$. Then we perform four 1d interpolation operations in x to find $f(x, y_j),\ f(x, y_{j+1}),\ f'_y(x, y_j),\ f'_y(x, y_{j+1})$, using 16 coefficients at 4 corners of the cell (i.e., $f(x_i, y_j),\ f(x_{i+1}, y_j),\ f'_x(x_i, y_j),\ f'_x(x_{i+1}, y_j)$ for the first value, etc.). Finally the last interpolation in y produces the required value. The same result would be obtained, had we reversed the order of coordinates. If we also want partial derivatives in both dimensions, then we need to compute higher derivatives on the first stage (i.e., from the first four coefficients we get not only $f(x, y_j)$, but also $f'_x(x, y_j)$ and $f''_{xx}(x, y_j)$), and then use extra 1d interpolation in y per each output derivative on the second stage (e.g., $f''_{xx}(x, y)$ is obtained from $f''_{xx}(x, y_j),\ f''_{xx}(x, y_{j+1}),\ f'''_{xxy}(x, y_j),\ f'''_{xxy}(x, y_{j+1})$, and the latter one is computed on the first stage from $f'_y(x_i, y_{j+1}),\ f'_y(x_{i+1}, y_{j+1}),\ f''_{xy}(x_i, y_{j+1}),\ f''_{xy}(x_{i+1}, y_{j+1})$).

Thus we see that for a cubic Hermite interpolation we need the values and derivatives in each direction at all grid nodes: $f(x_i, y_j),\ f'_x(x_i, y_j),\ f'_y(x_i, y_j),\ f''_{xy}(x_i, y_j)$. How do we find them, given only the function values at grid nodes? It turns out that the concept of cubic splines can naturally be extended to the multidimensional case, using the following multi-stage procedure. At the beginning, the first derivatives f'_x, f'_y in each dimension are found from the condition that the corresponding *second* derivatives f''_{xx}, f''_{yy} are continuous at grid nodes. Then the mixed second derivative $f''_{xy}(x_i, y_j)$ is computed by establishing a cubic spline for $f'_y(x, y_j)$ at each j and taking its derivative in x. The elegance of this approach is in its symmetry: the same value for f''_{xy} is also produced by constructing a cubic spline for $f'_x(x_i, y)$ at each x_i and taking the derivative in y. To understand why, recall that the natural cubic spline for $g(x)$ is a linear function of the input values $g_i \equiv g(x_i)$ defined by the array of grid nodes $\{x_i\}$. Equivalently, it is described by the matrix X that transforms the array of input values to the array of spline derivatives at grid nodes: $g'_i \equiv g'(x_i) = \sum_{k=1}^{K_x} X_{ik}\, g_k$. The complete 2d matrix f'_x of x-derivatives $f'_{x;\,ij} \equiv f'_x(x_i, y_j)$ is then given by $\mathsf{f}'_x = \mathsf{X}\,\mathsf{f}$, where the elements of matrix f are $f_{ij} \equiv f(x_i, y_j)$. On the other hand, the interpolation in the y direction is provided by the matrix Y such that for any array of values $h_j \equiv h(y_j)$, the y-derivatives are given by $h'_j \equiv h'(y_j) = \sum_{k=1}^{K_y} Y_{jk}\, g_k$. The complete 2d matrix f'_y of y-derivatives $f'_{y;\,ij} \equiv f'_y(x_i, y_j)$ is given by $\mathsf{f}'_y = (\mathsf{Y}\,\mathsf{f}^T)^T = \mathsf{f}\,\mathsf{Y}^T$. Now if we compute the mixed second derivatives f''_{xy} by constructing the y-spline from f'_x, this results in $\mathsf{f}''_{xy} = \mathsf{f}'_x\,\mathsf{Y}^T = (\mathsf{X}\,\mathsf{f})\,\mathsf{Y}^T$, whereas computing them from the f'_y results in $\mathsf{X}\,\mathsf{f}'_y = \mathsf{X}\,(\mathsf{f}\,\mathsf{Y}^T)$ – which are identical matrices.

The same scheme works in three dimensions: now we need eight coefficients at each grid node $\{x_i, y_j, z_k\}$ – $f, f'_x, f'_y, f'_z, f''_{xy}, f''_{xz}, f''_{yz}, f'''_{xyz}$, which are all found in three steps, using just the values of f and continuity conditions on higher derivatives. The evaluation of $f(x, y, z)$ also proceeds in three stages: first a total of 64 coefficients (8 numbers at 8 cell corners) are fed into 16 Hermite cubic interpolation operations in x, then the resulting 16 numbers

are used in 4 interpolations in y, and finally one interpolation in z. As before, the outcome does not depend on the order. This should also work in higher dimensions, although would clearly become more clumsy.

We now consider a more complicated case of 2d quintic interpolation. Similarly to the cubic Hermite case, we may interpolate with a piecewise-quintic polynomial in each direction using the following 9 quantities stored at each node: $f, f'_x, f''_{xx}, f'_y, f''_{xy}, f'''_{xxy}, f''_{yy}, f'''_{xyy}, f''''_{xxyy}$, first performing 6 interpolations in y to compute f, f'_x, f''_{xx} at two nodes of the x-grid, and then the final interpolation in x, or vice versa. By a similar argument, if we were provided with four matrices $\mathsf{f}, \mathsf{f}'_x, \mathsf{f}'_y, \mathsf{f}''_{xy}$, we could construct 1d quintic splines in each direction and use them to initialize the remaining five derivatives at each node from the conditions of continuity of still higher derivatives; again this will not result in a conflicting assignment of the mixed fourth derivative f''''_{xxyy}. Unfortunately, in practice we only have three matrices to work with – the function and its two partial derivatives at each node, but no mixed second derivative. We were not able to come up with an elegant method in this case, although the following kludge seems to deliver satisfactory results. We first construct the 1d quintic splines for $f(x, y_j)$ and $f(x_i, y)$ at each node, and use them to compute the second derivatives (f''_{xx} and f''_{yy}, correspondingly). To compute the mixed derivatives, we construct natural cubic splines in y from the values of f'_x and f''_{xx}, and similarly in x from the values of f'_y and f''_{yy}, and then differentiate them. The resulting values do not agree with each other, so we take either the average of the two estimates, or the one that is expected to be more accurate: for instance, if $i = 1$, we would compute $f''_{xy,ij}$ by differentiating the spline for $f'_x(x_i, y)$ by y, because the alternative variant (differentiating the spline for $f'_y(x, y_j)$ by x) does not provide a good estimate at the endpoint of its x-grid. In this way, all remaining derivatives are assigned.

A.2.3 Penalized spline regression

Suppose we have $N_{\rm data}$ points $\{x_i, y_i\}$, and we need to find a smooth function $y = f(x)$ that approximates the data in the least-square sense, but does not fluctuate too much – in other words, minimize the functional

$$\mathcal{Q} \equiv \sum_{i=1}^{N_{\rm data}} [y_i - f(x_i)]^2 + \lambda \int [f''(x)]^2 \, dx. \tag{5}$$

Here $\lambda \geq 0$ is the smoothing parameter that controls the tradeoff between approximation error and wiggliness of the function [17]; its choice is discussed below.

We represent $f(x)$ as a sum of B-spline basis functions $B_k^{[N]}(x)$, defined by the grid of $N_{\rm grid}$ knots, with adjustable amplitudes A_k:

$$f(x) \equiv \sum_{k=1}^{N_{\rm basis}} A_k \, B_k(x) \ , \quad N_{\rm basis} = N_{\rm grid} + N - 1, \text{ and we use cubic B-splines } (N = 3). \tag{6}$$

Let the matrix B with $N_{\rm data}$ rows and $N_{\rm basis}$ columns contain the values of basis functions at data points: $B_{ik} = B_k(x_i)$. Due to the locality of B-splines, this matrix is sparse – each row contains at most $N+1$ nonzero elements. The grid in x does not need to encompass all data points – the function $f(x)$ is linearly extrapolated outside the grid boundaries. Define the "roughness matrix" R containing the integrals of products of second derivatives of basis functions: $R_{kl} \equiv \int B_k''(x)\, B_l''(x)\, dx$. This matrix is also sparse (band-diagonal) and symmetric. The minimum of $\mathcal{Q} \equiv (\boldsymbol{y} - \mathsf{B}\boldsymbol{A})^T(\boldsymbol{y} - \mathsf{B}\boldsymbol{A}) + \lambda \boldsymbol{A}^T\mathsf{R}\boldsymbol{A}$ (5) is then obtained by solving for $\partial\mathcal{Q}/\partial\boldsymbol{A} = 0$:

$$\left(\mathsf{B}^T\mathsf{B} + \lambda\mathsf{R}\right)\boldsymbol{A} = \mathsf{B}^T\boldsymbol{y}. \tag{7}$$

Note that the size of this linear system is only $N_{\rm basis} \times N_{\rm basis}$, possibly much smaller than the number of data points $N_{\rm data}$. If one needs to solve the system for several values of λ and/or different vectors $\boldsymbol{y}$, there is an efficient algorithm for this [34]:

1. Compute the Cholesky decomposition of the matrix $\mathsf{B}^T\mathsf{B}$, representing it as $\mathsf{L}\mathsf{L}^T$, where L is a lower triangular matrix with size $N_{\rm basis}$. To avoid problems when $\mathsf{B}^T\mathsf{B}$ is singular (which occurs when some grid segments contain no points), we add a small multiple of R before computing the decomposition.
   Then compute the singular-value decomposition of the matrix symmetric positive definite matrix $\mathsf{L}^{-1}\mathsf{R}\mathsf{L}^{-T}$, representing it as $\mathsf{U}\,\mathrm{diag}(\boldsymbol{S})\,\mathsf{U}^T$, where U is a square orthogonal matrix (i.e., $\mathsf{U}\mathsf{U}^T = \mathsf{I}$) with size $N_{\rm basis}$, and $\boldsymbol{S}$ is the vector of singular values.
   Now the matrix in the l.h.s. of (7) can be written as
   $\mathsf{B}^T\mathsf{B} + \lambda\mathsf{R} = \mathsf{L}\mathsf{L}^T + \mathsf{L}\mathsf{L}^{-1}\mathsf{R}\mathsf{L}^{-T}\mathsf{L}^T = \mathsf{L}\mathsf{U}\mathsf{U}^T\mathsf{L}^T + \mathsf{L}\,\mathrm{diag}(\boldsymbol{S})\,\mathsf{U}^T\mathsf{L}^T$.
   Finally, compute a matrix $\mathsf{M} \equiv \mathsf{L}^{-T}\mathsf{U}$.

2. For any vector of $\boldsymbol{y}$ values, pre-compute $\boldsymbol{p} \equiv \mathsf{B}^T\boldsymbol{y}$ and $\boldsymbol{q} \equiv \mathsf{M}^T\boldsymbol{p}$ (vectors of length $N_{\rm basis}$).

3. Now for any choice of λ, the solution is given by

$$\boldsymbol{A} = \mathsf{M}\left[\mathsf{I} + \lambda\,\mathrm{diag}(\boldsymbol{S})\right]^{-1}\boldsymbol{q}, \tag{8}$$

   i.e., involves only a multiplication of a vector by inverse elements of a diagonal matrix and a single general matrix-vector multiplication.

The residual sum of squares – first term in (5) – is given by

$$\mathrm{RSS} \equiv |\boldsymbol{y} - \mathsf{B}\boldsymbol{A}|^2 = |\boldsymbol{y}|^2 - 2\boldsymbol{A}^T\boldsymbol{p} + |\mathsf{L}^T\boldsymbol{A}|^2. \tag{9}$$

In case of non-zero smoothing, the effective number of free parameters is lower than the number of basis functions, and is given by the number of equivalent degrees of freedom:

$$\mathrm{EDF} \equiv \mathrm{tr}[\mathsf{I} + \lambda\,\mathrm{diag}(\boldsymbol{S})]^{-1} = \sum_{k=1}^{N_{\rm basis}} \frac{1}{1+\lambda S_k}\,, \tag{10}$$

and it varies from N_{basis} for $\lambda = 0$ to 2 for $\lambda \to \infty$ (which corresponds to a two-parameter linear least-square fit). The amount of smoothing thus may be specified by EDF, which has a more direct interpretation than λ. The optimal choice of smoothing parameter is given by minimization of Akaike information criterion:

$$\mathrm{AIC} \equiv \ln(\mathrm{RSS}) + \frac{2\,\mathrm{EDF}}{N_{\mathrm{data}} - \mathrm{EDF} - 1}. \tag{11}$$

Often one may wish to apply a somewhat stronger smoothing than the one given by minimizing AIC, for instance, by allowing it to be larger than the minimum value by a specified amount $\Delta\mathrm{AIC} \sim \mathcal{O}(1)$. In both cases, the corresponding value of λ is obtained by standard one-dimensional minimization or root-finding routines.

A.2.4 Penalized spline density estimate

Let $P(x) > 0$ be a density function defined on the entire real axis, a semi-infinite interval $[x_{\min}, +\infty)$ or $(-\infty, x_{\max}]$, or a finite interval $[x_{\min}, x_{\max}]$. Let $\{x_i, w_i\}$ be an array of N_{data} samples drawn from this distribution, where x_i are their coordinates, and $w_i \geq 0$ are weights. We follow the convention that $\int P(x)\,dx$ over its domain is equal to $M \equiv \sum_i w_i$ (not necessarily unity).

We estimate $P(x)$ using a B-spline approximation to $\ln P$ constructed for a grid of N_{grid} nodes $\{X_j\}$, that is,

$$\ln P(x;\, \boldsymbol{A}) = \sum_k A_k\, B_k(x) - \ln G_0 + \ln M \equiv Q(x;\boldsymbol{A}) - \ln G_0(\boldsymbol{A}) + \ln M, \tag{12}$$

where A_k are the amplitudes – free parameters that are adjusted during the fit,
$B_k(x)$ are basis functions (B-splines of degree N defined by grid nodes, $N_{\mathrm{basis}} = N_{\mathrm{grid}} + N - 1$),
$Q(x;\boldsymbol{A}) \equiv \sum_k A_k\, B_k(x)$ is the weighted sum of basis function,
and $G_0(\boldsymbol{A}) \equiv \int \exp[Q(x;\boldsymbol{A})]\,dx$ is the normalization constant determined from the condition that $\int P(x)\,dx = M$. There is a gauge freedom in the choice of amplitudes A_k: if we add a constant to $Q(x;\boldsymbol{A})$, it would not have any effect on $\ln\mathcal{L}$ because this shift will be counterbalanced by G_0. We eliminate this freedom by fixing the amplitude of the last basis function to zero ($A_{N_{\mathrm{basis}}} = 0$), thus retaining $N_{\mathrm{ampl}} \equiv N_{\mathrm{basis}} - 1$ free parameters.

As in the case of penalized spline regression, we first compute the matrix B of weighted basis-function values at each input point: $B_{ik} \equiv w_i\, B_k(x_i)$. This matrix is large ($N_{\mathrm{data}}$ rows, N_{ampl} columns) but sparse, and is further transformed into two vectors of length N_{ampl} and a square matrix of the same size: $\boldsymbol{V} \equiv \mathsf{B}^T \mathbf{1}_{N_{\mathrm{data}}}$ (i.e., $V_k = \sum_i w_i\, B_k(x_i)$), $\boldsymbol{W} \equiv \mathsf{B}^T \boldsymbol{w}$, $\mathsf{U} \equiv \mathsf{B}^T\,\mathsf{B}$. In the remaining steps of the procedure, only vectors and matrices of size N_{ampl} rather than N_{data} are involved, which allows to deal efficiently even with very large arrays of samples described by a moderate number of parameters (such as fitting the density profile of an N-body model with $\mathcal{O}(10)$ grid points).

The total penalized likelihood of the model given the vector of amplitudes $\boldsymbol{A}$ is

$$\begin{aligned}\ln \mathcal{L} \;&\equiv\; \ln \mathcal{L}_{\rm data} - \lambda \mathcal{R}(\boldsymbol{A}) \;\equiv\; \sum_{i=1}^{N_{\rm data}} w_i \ln P(x_i;\, \boldsymbol{A}) - \lambda \int \left[\ln P''(x_i)\right]^2 \, dx \\ &= \sum_{i=1}^{N_{\rm data}} w_i \left(\sum_{k=1}^{N_{\rm ampl}} A_k B_k(x_i) - \ln G_0(\boldsymbol{A}) + \ln M \right) - \lambda \sum_{k=1}^{N_{\rm ampl}} \sum_{l=1}^{N_{\rm ampl}} A_k \, A_l \, R_{kl} \\ &= \left[\boldsymbol{V}^T \boldsymbol{A} - M \, \ln G_0(\boldsymbol{A}) + M \, \ln M \right] - \lambda \, \boldsymbol{A}^T \, \mathsf{R} \, \boldsymbol{A}, \end{aligned} \tag{13}$$

where $\lambda\mathcal{R}$ is the roughness penalty term, and the matrix $R_{kl} \equiv \int B_k''(x)\, B_l''(x)\, dx$ is also pre-computed at the beginning of the procedure[7]. The smoothing parameter λ controls the tradeoff between the likelihood of the data and the wiggliness of the estimated density; its choice is discussed below.

Unlike the penalized spline regression problem, in which the amplitudes are obtained from a linear equation, the problem of penalized spline density estimation is nonlinear because of the normalization factor $G_0(\boldsymbol{A})$. The amplitudes $\boldsymbol{A}$ that minimize $-\ln \mathcal{L}$ (13) are found by solving the system of equations $\partial \ln \mathcal{L} / \partial A_k = 0$ iteratively, using a multidimensional Newton method with explicit expressions for the gradient and hessian:

$$-\frac{\partial \ln \mathcal{L}}{\partial A_k} = -V_k + M \frac{\partial \ln G_0}{\partial A_k} + 2\lambda \sum_l R_{kl} \, A_l \;, \tag{14a}$$

$$-\frac{\partial^2 \ln \mathcal{L}}{\partial A_k \, \partial A_l} = M \frac{\partial^2 \ln G_0}{\partial A_k \, \partial A_l} + 2\lambda R_{kl} \;\equiv H_{kl} \;, \tag{14b}$$

$$\text{where } G_0 \equiv \int \exp[Q(x;\boldsymbol{A})]\, dx \;, \quad Q(x;\boldsymbol{A}) \equiv \sum_k A_k \, B_k(x) \;,$$

$$\frac{\partial \ln G_0}{\partial A_k} = \frac{\int B_k(x) \, \exp[Q(x;\boldsymbol{A})]\, dx}{G_0} \;,$$

$$\frac{\partial^2 \ln G_0}{\partial A_k \, \partial A_l} = \frac{\int B_k(x) \, B_l(x) \, \exp[Q(x;\boldsymbol{A})]\, dx}{G_0} - \frac{\partial \ln G_0}{\partial A_k} \, \frac{\partial \ln G_0}{\partial A_l} \;.$$

The choice of smoothing parameter λ may be done by cross-validation: for each sample i, we compute its likelihood using best-fit parameters $\boldsymbol{A}^{(i)}$ calculated for all samples except this one, and then sum these values over all samples.

$$\ln \mathcal{L}_{\rm CV}(\lambda) \equiv \sum_{i=1}^{N_{\rm data}} w_i \, \ln P(x_i;\, \boldsymbol{A}^{(i)}) = \sum_{i=1}^{N_{\rm data}} w_i \left(\sum_{k=1}^{N_{\rm basis}} A_k^{(i)} B_k(x_i) - \ln G_0(\boldsymbol{A}^{(i)}) \right) + M \ln M. \tag{15}$$

[7]It may be advantageous to use third derivatives here [37], in which case the solution in the limit of infinite smoothing ($\mathcal{R} \to 0$) corresponds to a Gaussian density profile.

Of course, it would be prohibitively expensive to compute the best-fit amplitudes $\boldsymbol{A}^{(i)}$ separately for each omitted point; instead, we express them as small perturbations of $\boldsymbol{A}$, by demanding that the l.h.s. of (14a) is zero for each i at the corresponding $\boldsymbol{A}^{(i)}$:

$$0 = -V_k + w_i\, B_k(x_i) + M\frac{\partial \ln G_0}{\partial A_k} + M\frac{\partial^2 \ln G_0}{\partial A_k\, \partial A_l}(A_l^{(i)} - A_l) + 2\lambda \sum_l R_{kl}\, A_l^{(i)} \ ,$$

$$\delta A_l^{(i)} \equiv A_l^{(i)} - A_l = -\left[M\frac{\partial^2 \ln G_0}{\partial A_k\, \partial A_l} + 2\lambda R_{kl}\right]^{-1} w_i\, B_k(x_i) \ , \ \text{or} \ \ \delta\boldsymbol{A} = -\mathsf{H}^{-1}\mathsf{B}^T.$$

Here the gradient and hessian of G_0 are taken at the overall best-fit amplitudes $\boldsymbol{A}$ for the entire sample, computed for the given value of λ. The matrix $\delta\boldsymbol{A}$ with $N_{\rm ampl}$ rows and $N_{\rm data}$ columns needs not be computed explicitly each time. Finally, the cross-validation score (15) is expressed as

$$\ln \mathcal{L}_{\rm CV}(\lambda) = \ln \mathcal{L}_{\rm data} - \mathrm{tr}(\mathsf{H}^{-1}\mathsf{U}) + \frac{\partial \ln G_0(\boldsymbol{A})}{d\boldsymbol{A}}\, \mathsf{H}^{-1}\, \boldsymbol{W}. \tag{16}$$

Here $\ln \mathcal{L}_{\rm data}$ is the expression in brackets in (13). The optimal value of $\lambda > 0$ that maximizes the cross-validation score is found by a simple one-dimensional search. We first assign a reasonable initial guess for amplitudes (approximating the density as a Gaussian with the mean and dispersion computed from input samples). At each step, the multidimensional nonlinear root-finder routine is invoked to find best-fit amplitudes $\boldsymbol{A}$ for the given λ, starting from the current initial guess. If it was successful and $\ln \mathcal{L}_{\rm CV}(\lambda)$ is higher than the current best estimate, the initial guess is replaced with the best-fit amplitudes: this not only speeds up the root-finder, but also improves the convergence. The range of λ is progressively narrowed until the maximum has been located with sufficient accuracy, at which point we return the last successful array of $\boldsymbol{A}$.

Figure 2 illustrates the application of linear (non-smoothed) and cubic spline with optimal smoothing to a test problem. In this case the grid spacing was deliberately too dense for the given number of samples, so that some grid segments do not contain any samples, but nevertheless the penalized density estimate comes out rather close to the true one. This regime is not very stable, though, and for normal operation the grid should be assigned in such a way that each segment contains at least a few samples – this ensures that even the un-smoothed estimate is mathematically well defined.

Maximization of cross-validation score is considered to be "optimal" smoothing; however, in some cases the inferred $\ln P(x)$ may still be too wiggly. An alternative approach is to estimate the expected scatter in $\ln \mathcal{L}_{\rm data}$ for a sample of finite size $N_{\rm data}$. In the case of

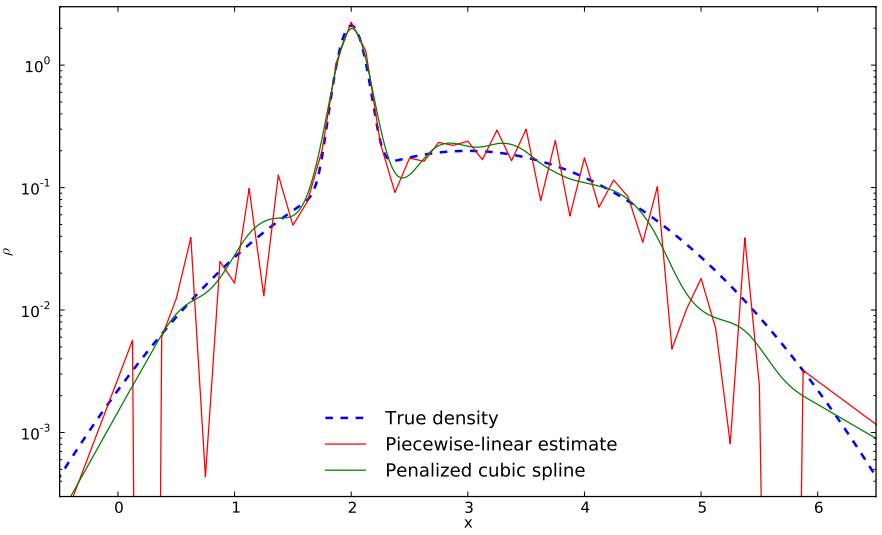

Figure 2: Penalized spline estimate of density from sample points. Here $N_{\mathrm{data}} = 1000$ were drawn from the original density profile (shown in dashed blue) described by a sum of two gaussians, with dispersions equal to 0.1 and 1. We reconstruct the logarithm of density using a linear ($N = 1$) and cubic ($N = 3$) B-splines with 50 nodes uniformly placed on the interval [0..6], so that it is linearly extrapolated beyond the extent of this grid. The linear B-spline estimate (shown in red) is rather wiggly, because the grid spacing is intentionally made too fine for the given number of samples – some elements do not contain any samples. The non-penalized cubic B-spline (not shown) is very close to the linear one, and also close to a standard Gaussian kernel density estimate with the same bandwidth as the grid spacing (also not shown). By contrast, the penalized cubic B-spline with the smoothing parameter determined automatically in order to maximize the cross-validation likelihood (shown in blue) is much closer to the true density.

uniform-weight samples and zero smoothing, the mean and dispersion in $\ln\mathcal{L}$ are

$$\left\langle \ln\mathcal{L} \right\rangle = \int P(x)\, \ln P(x)\, dx = M\, \left[\frac{G_1}{G_0} + \ln M - \ln G_0 \right], \tag{17}$$

$$\left\langle \left(\ln\mathcal{L} - \langle \ln\mathcal{L} \rangle \right)^2 \right\rangle = N_{\rm data}^{-1} \left(M \int P(x)\, [\ln P(x)]^2\, dx - \left\langle \ln\mathcal{L} \right\rangle^2 \right) = M^2 N_{\rm data}^{-1} \left[\frac{G_2}{G_0} - \left(\frac{G_1}{G_0} \right)^2 \right],$$

$$\text{where} \quad G_n \equiv \int \left[Q(x) \right]^n \exp\left[Q(x) \right] dx\ .$$

We first determine the best-fit $\boldsymbol{A}$ for the optimal value of $\lambda_{\rm opt}$ and compute the expected r.m.s. scatter $\delta\ln\mathcal{L} \equiv \sqrt{\langle (\ln\mathcal{L} - \langle\ln\mathcal{L}\rangle)^2 \rangle}$ from the above equation; then we search for λ such that $\ln\mathcal{L}_{\rm data}(\lambda) = \ln\mathcal{L}_{\rm data}(\lambda_{\rm opt}) - \kappa\,\delta\ln\mathcal{L}$, where $\kappa \sim 1$ is a tunable parameter. The resulting density is less fluctuating, but the asymptotic behaviour near or beyond grid boundaries, where the number of samples is low, may be somewhat biased as a result of more aggressive smoothing.

A.3 Potentials

A.3.1 Multipole expansion

The potential in the multipole expansion approach is represented as a sum of individual spherical-harmonic terms with coefficients being arbitrary functions of radius:

$$\Phi(r,\theta,\phi) = \sum_{l=0}^{l_{\rm max}} \sum_{m=-m_0(l)}^{m_0(l)} \Phi_{l,m}(r)\, \sqrt{4\pi}\tilde{P}_l^m(\cos\theta)\ \mathrm{trig}\, m\phi, \tag{18}$$

$$\mathrm{trig}\, m\phi \equiv \begin{cases} 1 & , \quad m = 0 \\ \sqrt{2}\,\cos m\phi & , \quad m > 0 \\ \sqrt{2}\,\sin |m|\phi & , \quad m < 0 \end{cases}$$

Here $\tilde{P}_l^m(x) \equiv \sqrt{\frac{2l+1}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}\, P_l^{|m|}(x)$ are normalized associated Legendre polynomials, $l_{\rm max}$ is the order of expansion in meridional angle θ, and $m_0(l) = \min(l, m_{\rm max})$, where $m_{\rm max} \le l_{\rm max}$ is the order of expansion in azimuthal angle ϕ (they do not need to coincide, e.g., if the model is considerably flattened but only weakly triaxial, then $m_{\rm max} = 2$ may be sufficient, while $l_{\rm max}$ may be set to 6 or 8). The normalization is chosen so that for a spherically-symmetric potential, $\Phi_{0,0}(r) = \Phi(r)$, and that for each l, the sum of squared coefficients over all m is invariant under rotations of coordinate system.

In the `Multipole` class, individual terms are approximated as scaled spline functions in scaled radius: $\Phi_{l,m}(r) = A(r)\, S_{l,m}(\ln r)$, where $A(r) \equiv (r^2 + r_0^2)^{-1/2}$ is a scaling factor

that allows to reach higher accuracy at large radii (where the actual potential is $\propto r^{-1}$, so the scaled term $S_{0,0}(r)$ tends to a constant), and $S_{l,m}$ are quintic splines in log-radius. These functions are defined by the values and first derivatives at nodes of a radial grid; usually this grid would be linear in log-radius, with a constant ratio between consecutive radii $f \equiv r_{k+1}/r_k$.

If the minimum/maximum grid radii are not provided, they are assigned automatically using the following approach. First the half-mass radius $r_{1/2}$ of the density profile is determined (this obviously can be done only for models with a finite total mass). Then a suitable grid spacing factor f is assigned. Finally, we set $r_{\rm max/min} = r_{1/2}\, f^{\pm N_R/2}$. As N_R gets larger, both the dynamical range $D \equiv r_{\rm max}/r_{\rm min}$ is increased, and the resolution gets better (nodes are spaced more densely); e.g., for $N_R = 20$, these are $D \sim 10^6$ and $f \sim 2$.

To compute the potential and its derivatives at a given point, one needs to sum the contributions of each harmonic term. For systems with certain symmetries, many of these terms are identically zero, and this is taken into account thereby reducing the amount of computation. By convention, negative m correspond to sine terms and positive – to cosine; if a triaxial model is aligned with the principal axes, all sine terms must be zero; symmetry w.r.t. reflection about one of the principal planes also zeroes down some terms, and axisymmetry retains only $m = 0$ terms. All possible combinations of symmetries are encoded in the `coords::SymmetryType` class, and each one corresponds to a certain combination of non-trivial spherical-harmonic terms (`math::SphHarmIndices`). For instance, a model of a disk galaxy with two spiral arms is symmetric w.r.t. z-reflection (change of sign of z coordinate) and xy-reflection (change of sign of both x and y simultaneously), and this retains only terms with even l and even m (both positive and negative).

At each nontrivial m, we may need to compute up to $l_{\rm max} - |m|$ 1d interpolating splines in r multiplied with Legendre polynomials in $\cos\theta$. This may be replaced with a single evaluation of a 2d interpolation spline in $\ln r, \theta$ plane (in fact a suitably scaled analog of θ is used to avoid singularities along z axis), which was pre-computed during potential initialization – this is more efficient for $l_{\rm max} > 2$.

Extrapolation to small and large radii (beyond the extent of the grid) is performed using the assumption of a power-law behaviour of individual multipole components: $\Phi_{l,m}(r) = U_{l,m}\, r^{s_{l,m}} + W_{l,m}\, r^v$, where $v \equiv l$ or $-1-l$ for the inward or outward extrapolation, correspondingly. The term with r^v represents the "principal" component with a zero Laplacian, while r^s corresponds to a power-law density profile $\rho_{l,m} \propto r^{s-2}$, and is typically much smaller in magnitude. This allows to describe very accurately the asymptotic behaviour of potential beyond the extent of the grid, if the coefficients U, W and s can be determined reliably. In order to do so, we use the value and derivative of each harmonic coefficient at the first or the last grid node, plus its value at the adjacent node, to obtain a system of 3 equations for these variables. Thus the value and derivative of each term are continuous at the boundaries.

A `Multipole` potential may be constructed either from an existing potential object (in

which case it simply computes a spherical-harmonic transform of the original potential at radial grid nodes), or from a density profile (thereby solving the Poisson equation).

$$\Phi_{l,m}(r) = -\frac{4\pi}{2l+1}\left[r^{-l-1}\int_0^r \rho_{l,m}(r')\,r'^{\,l+2}\,dr' + r^l\int_r^\infty \rho_{l,m}(r')\,r'^{\,1-l}\,dr'\right], \tag{19}$$

$$\rho_{l,m}(r) \equiv \frac{1}{\sqrt{4\pi}}\int_{-1}^{1} d\cos\theta\,\tilde{P}_l^m(\cos\theta)\int_0^{2\pi} d\phi\ \mathrm{trig}\,m\phi\,\rho(r,\theta,\phi). \tag{20}$$

A separate class `DensitySphericalHarmonic` serves to approximate any density profile with its spherical-harmonic expansion, with coefficients being cubic splines in $\ln r$. Similarly to the Multipole potential class, we extrapolate the profile to small or large radii using power-law asymptotes, with slopes deduces from the values of the $l=0$ coefficient at two inner- or outermost grid points. This class is mainly used in self-consistent modelling (Section 2.6.4) to provide a computationally cheap way of evaluating the density at any point in space, once it is initialized by computing the costly integrals over distribution function at a small number of points (grid nodes in radius and nodes of Gauss–Legendre quadrature rule in $\cos\theta$). This interpolated density is then used to construct the Multipole potential: the solution of Poisson equation requires integration of harmonic terms in radius using a more densely spaced internal grid, and the values of these terms are easily evaluated from the density interpolator. Note that this process involves two forward and one reverse spherical-harmonic transformation (first time during the construction of density interpolator, then the reverse transformation to obtain the interpolated values at the required spatial points, and then again in the Multipole potential). However, since the spherical-harmonic transformation is invertible (reproduces the source density at this special set of points to machine precision), this double work does not add to error, and incurs negligible overhead.

`DensitySphericalHarmonic` may also be constructed from an array of particles, and then used to create the `Multipole` potential in a usual way. To do so, we first compute the spherical-harmonic expansion coefficients at each particle's radius:

$$\rho_{l,m;i} \equiv m_i\,\sqrt{4\pi}\,\tilde{P}_l^m(\cos\theta_i)\ \mathrm{trig}\,m\phi_i.$$

Then the $l=0$ coefficients (which contain just particle masses) are used to determine the spherically-symmetric part of the density profile. We use penalized spline log-density fit (Section A.2.4) to estimate the logarithm of an auxiliary quantity $P(\ln r) \equiv dM(<r)/d\ln r$ from the array of point masses and log-radii; the actual density is $\rho_{0,0}(r) = P(\ln r)/(4\pi\,r^3)$. Finally, we create smoothing splines (Section A.2.3) for all non-trivial $\rho_{l,m}(\ln r)$ terms. This temporary density model is used to construct the `Multipole` potential from an N-body model – even though the Poisson equation (19,20) can be solved directly by summing over particles (an approach used in [41]), this results in a noisier and less accurate potential than the intermediate smoothed density can provide.

A.3.2 CylSpline expansion

The `CylSpline` potential is represented as a sum of azimuthal Fourier harmonics in ϕ, with coefficients of each term intepolated on a 2d grid in R, z plane with suitable scaling. Namely, both R and z coordinates are transformed to $\tilde{R} \equiv \ln(1 + R/R_0), \tilde{z} \equiv \ln(1 + z/R_0)$, where R_0 is a characteristic radius. The amplitude of each interpolated term is additionally multiplied by a factor $\sqrt{R^2 + z^2 + R_0^2}$ to obtain the actual potential; this serves the same purpose as in `Multipole` – improving the accuracy of interpolation at large radii, where the $m = 0$ harmonic of the potential tends to $-M/r$, so that the scaled interpolant tends to a constant. We use either 2d quintic splines or 2d cubic splines to construct the interpolator, depending on whether the partial derivatives of potential by R and z are available. Normally, if the potential is constructed from a smooth density profile or from a known potential, it is advantageous to use 5th order interpolation to improve accuracy, even though this increases the computational cost of construction (but not of evaluation of the potential). On the other hand, in the case of a potential constructed from an array of particles, estimates of derivatives are too noisy and in fact deteriorate the quality of approximation.

Unlike the `Multipole` potential, which can handle a power-law asymptotic behaviour of density both at small and large radii, `CylSpline` is more restricted – since the grid covers the origin, it can only represent a model with finite density at $r = 0$. Extrapolation to large radii (beyond the extent of the rectangular grid in R, z) is performed using a similar approach to `Multipole`, but keeping only the principal spherical-harmonic terms Wr^{-l-1} with zero Laplacian, i.e., corresponds to a zero density outside the grid. The coefficients for a few low-order multipoles are determined from a least-square fit to the values of potential at the outer boundary of the grid; thus the potential values inside and outside the boundary are not exactly the same, but still are quite close – the relative error in potential and force in the extrapolated regime is typically $\lesssim 10^{-3}$ (see Figures 3, 4).

Since the grid spacing is near-uniform at small and near-exponential at large R, z, the dynamical range of `CylSpline` is also very broad. If the values of first/last grid nodes are not specified, they are determined automatically using the same approach as for `Multipole`. Typically, $20 - 25$ grid nodes are enough to span a range from 0 to $\gtrsim 10^3\, r_{\rm half-mass}$. Symmetries of the model are taken into account in the choice of non-trivial azimuthal Fourier terms (in the case of axisymmetry, only $m = 0$ term is retained; for triaxial models only even $m \geq 0$ are used, etc.); and for models with z-reflection symmetry, coefficients are computed and stored only for the $z \geq 0$ half-space.

The main advantage of `CylSpline` is in its ability to efficiently represent even very flattened density profiles, which are not suitable for `Multipole` expansion. When `CylSpline` approximation is constructed from another potential, this boils down to taking the Fourier transform in ϕ of potential and forces of the original potential at the nodes of 2d grid in R, z plane. When it is constructed from a density profile, this involves the solution of Poisson

equation in cylindrical coordinates, which is performed in two steps. First, a Fourier transform of the source model is created (if it was neither axisymmetric nor a `DensityCylGrid` class, see below). Next, for each m-th harmonic ρ_m, the potential is computed at each node R, z of the 2d grid using the following approach [11]:

$$\Phi_m(R,z) = -\int_{-\infty}^{+\infty} dz' \int_0^{\infty} dR'\, 2\pi R'\, \rho_m(R',z')\, \Xi_m(R,z,R',z')\ , \tag{21}$$

$$\Xi_m \equiv \int_0^{\infty} dk\, J_m(kR)\, J_m(kR')\, \exp(-k|z-z'|)\ , \quad \text{which evaluates to} \tag{22}$$

$$\Xi_m = \frac{1}{\pi\sqrt{RR'}}\, Q_{m-1/2}\left(\frac{R^2+R'^2+(z-z')^2}{2RR'}\right) \quad \text{if } R>0, R'>0,$$

$$\Xi_m = \frac{1}{\sqrt{R^2+R'^2+(z-z')^2}} \quad \text{if } R=0 \text{ or } R'=0, \text{ and } m=0, \text{ otherwise } 0.$$

Here Q is the Legendre function of the second kind, which is computed using a Padé approximation for $m \le 12$ or Gauss' hypergeometric function otherwise (more expensive). For an array of particles,

$$\Phi_m(R,z) = -\sum_k m_k\, \Xi_m(R,z,R_k,z_k)\ \mathrm{trig}\, m\phi_k. \tag{23}$$

The computation of `CylSpline` coefficients is much more expensive than that of `Multipole`, because at each of $\mathcal{O}(N_R \times N_z \times m_{\max})$ nodes we need to evaluate a 2d integral in (21) or a sum over all particles in (23). On a typical workstation, this may take from from a few seconds to a few minutes, depending on the resolution and the number of CPU cores. Nevertheless, this is a one-time cost; once the coefficients are calculated, the evaluation of both `Multipole` and `CylSpline` potentials is very fast – the cost depends very weakly on the number of grid nodes, and is proportional to the number of azimuthal-harmonic terms ($m_{\max}$, but not $l_{\max}$ in the case of Multipole).

Figure 3 demonstrates that the accuracy of both approximations is fairly good (relative error in force $\lesssim 10^{-3}$) with default settings ($N_R = 25, l_{\max} = m_{\max} = 6$) and improves with resolution. For the case of initialization from an array of particles, discreteness noise is the main limiting factor. Figure 4 illustrates that each of the two potential expansions has its weak points: `Multipole` is not suitable for strongly flattened systems and `CylSpline` performs poorly in systems with density cusps; but for most density profiles at least one of them should deliver a good accuracy.

A separate class `DensityCylGrid` serves the same task as `DensitySphericalHarmonic`: provides an interpolated density model that is initialized from the values of source density at nodes of a 2d grid in R, z plane (for an axisymmetric model) or 3d grid in R, z, ϕ (in general). The density is represented as a Fourier expansion in ϕ, with each term being a 2d cubic spline

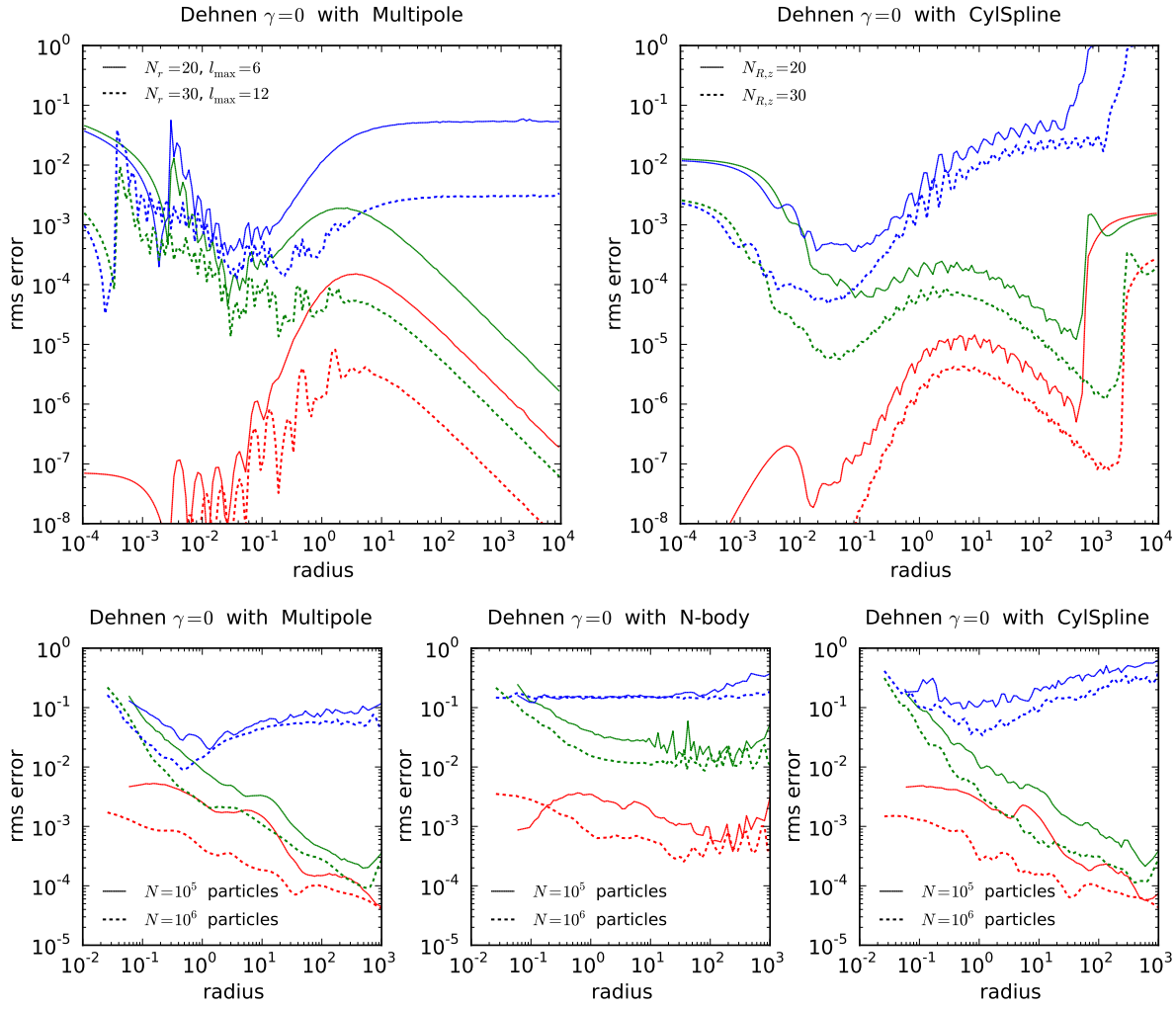

Figure 3: Accuracy of potential approximations in the case of initialization from a smooth density profile (top panels) and from an array of N particles (bottom panels). In both cases we compare the potential (red), force (green) and density (blue) computed using the potential expansions (left: Multipole, right: CylSpline) with the "exact" values for a triaxial $\gamma = 0$ Dehnen profile ($x : y : z = 1 : 0.8 : 0.5$), obtained by numerical integration, and plot the relative errors as functions of radius. In the top panels we vary the order of spherical-harmonic expansion and the number of grid nodes. Both potential approximations deliver fairly high accuracy, which increases with resolution. In the bottom panels we additionally show these quantities computed with a conventional N-body approach (direct-summation and SPH density estimate). Here the error is dominated by noise in computing the potential from discrete samples, and not by the approximation accuracy (it is almost independent of the grid parameters, but decreases with N). Notably, both smooth potential approximations are closer to the true potential than the N-body estimate.

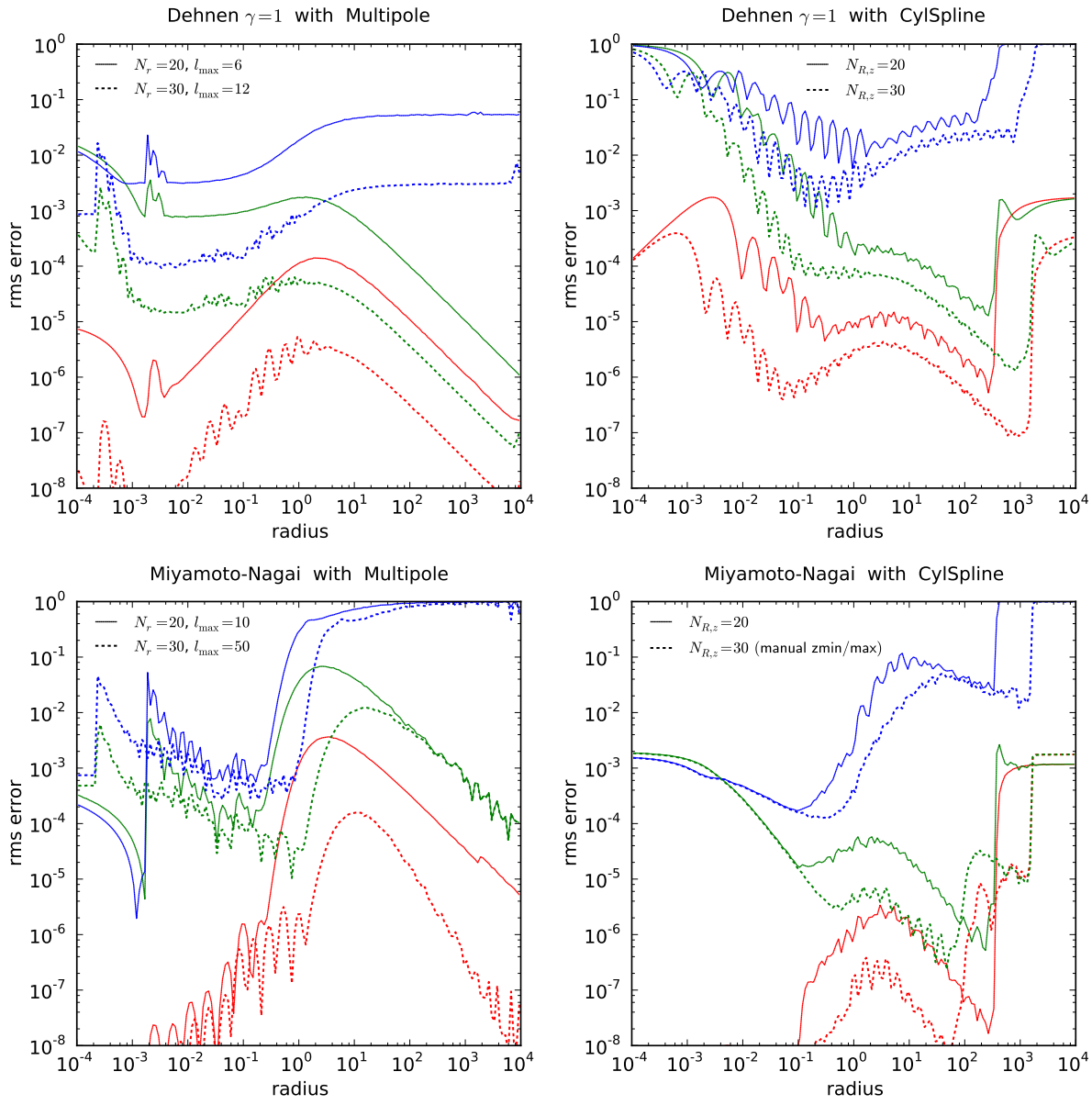

Figure 4: Accuracy of potential approximations for different types of density profiles. As in the previous figure, we plot the relative errors in potential (red), force (green) and density (blue) for Multipole (left) and CylSpline (right) potential expansions. Top panels are for a triaxial $\gamma = 1$ Dehnen model, and bottom – for a Miyamoto–Nagai disk. In the former case, CylSpline cannot efficiently deal with cuspy density profiles, while Multipole is able to deliver accurate results even for a $\gamma = 2$ cusp without any difficulty. On the other hand, in the latter case the strongly flattened density model is poorly represented by the spherical-harmonic expansion even with $l_{\max} = 50$, whereas CylSpline performs well, especially with manually assigned grid boundaries.

in $\tilde{R}, \tilde{z}$ coordinates (scaled in the same way as in `CylSpline`). Interpolated density is zero outside the grid. This class serves as a counterpart to `DensitySphericalHarmonic` in the context of DF-based self-consistent models for disk-like components: the values of density at grid nodes are computed by (expensive) integration of DF over velocities, and density in the entire space, necessary for computing the potential, is given by the interpolator.

A.4 Distribution functions

A.4.1 Spherical isotropic DFs

The correspondence between energy E and phase volume h in the given potential is provided by the class `PhaseVolume`. Phase volume $h(E)$ and its derivative (density of states) $g(E) \equiv dh(E)/dE$ are defined as

$$h(E) = \frac{16\pi^2}{3} \int_0^{r_{\max}(E)} r^2\, v^3(E,r)\, dr = 4\pi^2 \int_0^{L^2_{\rm circ}(E)} J_r(E,L)\, dL^2 = \int_{\Phi(0)}^{E} g(E')\, dE', \quad (24a)$$

$$g(E) = 16\pi^2 \int_0^{r_{\max}(E)} r^2\, v(E,r)\, dr = 4\pi^2\, L^2_{\rm circ}(E)\, T_{\rm rad}(E), \quad (24b)$$

where $v = \sqrt{2(E-\Phi(r))}$ is the velocity, $L_{\rm circ}(E)$ is the angular momentum of a circular orbit with energy E, and $T_{\rm rad}(E) = 2\int_0^{r_{\max}(E)} dr/v$ is the period of a radial orbit with this energy. In other words, phase volume is literally the volume of phase space enclosed by the energy hypersurface.

The bi-directional correspondence between E and h is given by two 1d quintic splines (with derivative at each node given by g) in scaled coordinates. Namely, we use $\ln h$ as one coordinate, and the scaled energy $\mathscr{E} \equiv \ln[1/\Phi(0) - 1/E]$ as the other one (both when the potential has a finite value $\Phi(0)$ at origin, or when it tends to $-\infty$. The purpose of this scaling is twofold. First, in the case of a finite $\Phi(0)$, any quantity that depends on E directly is poorly resolved as $E \to \Phi(0)$ because of finite floating-point precision: e.g., if $\Phi(0) = -1$, and $E = -1 + 10^{-8}$ (corresponding to the radius as large as 10^{-4} in a constant-density core), we only have half of the mantissa available to represent the variation of E. By performing this scaling, we "unfold" the range of $\mathscr{E}$ down to $-\infty$ with full precision. Second, this scaling converts a power-law asymptotic behaviour of $h(\Phi(r))$ at small and large radii into a linear dependence between $\ln h$ and $\mathscr{E}$, suitable for extrapolation. Namely, as $E \to 0$ and $\Phi \propto -1/r$ (which is true for any well-behaved potential), $h(E) \propto (-E)^{-3/2}$ and $g(E) \propto (-E)^{-5/2}$. At small radii, if the density behaves as $\rho \propto r^{-\gamma}$ and the corresponding potential – as $\Phi \propto r^{2-\gamma}$, then $h(E) \propto [E-\Phi(0)]^{(12-3\gamma)/(4-2\gamma)}$ in the case $\gamma < 2$ (when $\Phi(0)$ is finite), or $h(E) \propto (-E)^{(12-3\gamma)/(4-2\gamma)}$ if $2 \le \gamma \le 3$ (including the case of the Kepler potential, $\gamma = 3$); in both cases, $g(h) \propto h^{(8-\gamma)/(12-3\gamma)}$. The interpolation in scaled coordinates typically attains

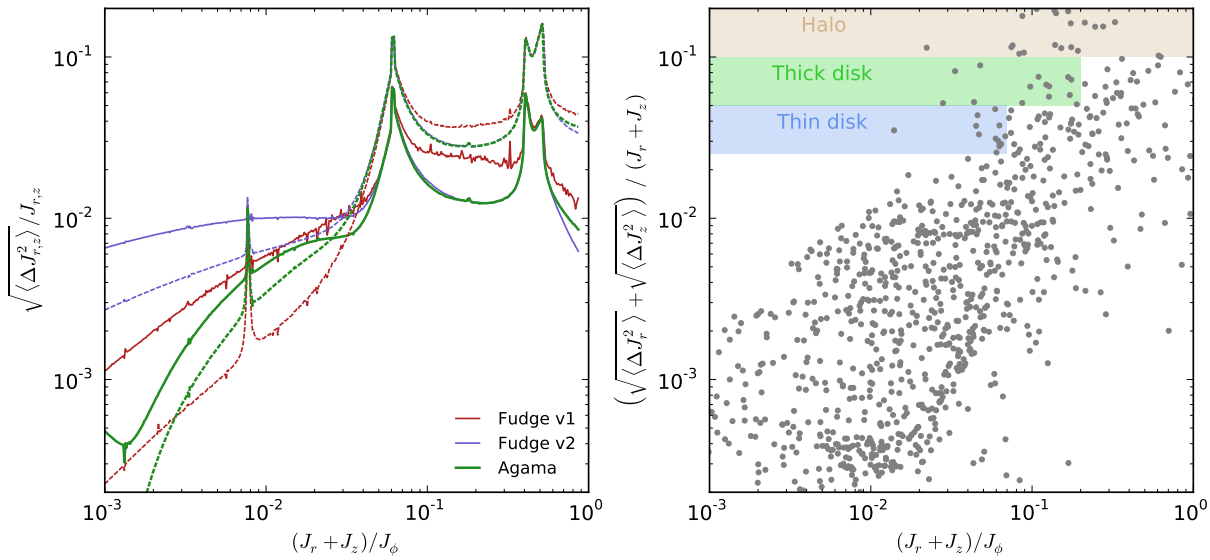

Figure 5: Accuracy of actions determined by the Stäckel approximation method.
Actions are computed at 10^3 points on a numerically computed orbit, and the error is estimated as the r.m.s. variation of actions (which in reality should stay constant for each orbit). The potential used this experiment is the same as in [35], consists of several disky and spheroidal components, and is intended to represent the Milky Way.
Left panel shows the relative error in radial (J_r, solid lines) and vertical (J_z, dashed lines) actions as a function of $(J_r + J_z)/J_\phi$, which serves as a proxy for orbit eccentricity. It is computed for a sequence of orbits launched at solar radius with the same tangential velocity (equal to the local circular speed), and varying meridional velocity (always directed at $\sim 40^\circ$ to the disk plane, so that $J_r/J_z \simeq 2-3$). The initial conditions are equivalent to those used in Fig. 3 of [35], and the red and blue curves are the same as in that plot, except that the vertical axis shows the relative, not absolute error. These two curves show the error in the Stäckel fudge as implemented in two variants in that paper; the thicker green curve is the implementation from Agama, which generally yields equal or better accuracy and is $\sim 2\times$ faster, requiring correspondingly fewer potential evaluations. The strong peaks are located at resonances, where the Stäckel approximation breaks down.
The right panel shows the relative error in the sum of two actions as a function of the same proxy for eccentricity, computed for 10^3 orbits drawn from a distribution function describing the stellar components of this galaxy model (thin and thick disks and the stellar halo); they span the range of radii from 1 to 100 kpc. It illustrates that the relative errors are typically $\lesssim 1\%$ for thin-disk orbits and $\lesssim 10\%$ for other orbits, although again they may be larger at resonances.

a level of accuracy better than 10^{-9} over the range of h covered by the spline, and $\sim 10^{-5}$ in the extrapolated regime (if the potential indeed has a power-law asymptotic behaviour).

Any non-negative function $f(h)$ may serve as a spherical isotropic DF. One possible representation is provided by the `math::LogLogSpline` class – an interpolating spline in doubly-logarithmically scaled coordinates (i.e., $\ln f(\ln h)$ is a cubic spline and is extrapolated linearly to small and large h). Such DFs are constructed, e.g., by routines `makeEddingtonDF` and `fitSphericalDF` in the `galaxymodel_spherical.h` module.

The main application of these DFs is for simulating the effect of two-body relaxation, used in the Monte Carlo code Raga [42]. We assume that test stars are moving in the background of fields stars with distribution function $f(h)$, and both test and field stars have the same mass. There are two possible descriptions of relaxation phenomena: either locally, as a perturbation to the velocity $v \equiv \sqrt{2(E-\Phi(r))}$ of the test star at the given position r, or, in the orbit-averaged approach, as a perturbation to the star's energy E averaged over its radial motion. In both cases, the rate of change of the given quantity per unit time is denoted by $\langle \dots \rangle$.

The local (position-dependent) drift and diffusion coefficients in velocity are given by

$$v\langle \Delta v_\parallel \rangle = -2\Gamma\, J_{1/2}\ , \tag{25a}$$

$$\langle \Delta v_\parallel^2 \rangle = \tfrac{2}{3}\Gamma\, \left(I_0 + J_{3/2}\right), \tag{25b}$$

$$\langle \Delta v_\perp^2 \rangle = \tfrac{2}{3}\Gamma\, \left(2I_0 + 3J_{1/2} - J_{3/2}\right), \quad \text{where} \tag{25c}$$

$$I_0(E) \equiv \int_E^0 f(E')\,dE' = \int_{h(E)}^\infty \frac{f(h')}{g(h')}\,dh', \tag{25d}$$

$$J_{n/2}(E,\Phi) \equiv \int_{\Phi(r)}^E f(E') \left(\frac{E'-\Phi}{E-\Phi}\right)^{n/2} dE' = \int_{h(E)}^\infty \frac{f(h')}{g(h')} \left(\frac{E'(h')-\Phi}{E-\Phi}\right)^{n/2} dh'. \tag{25e}$$

Orbit-averaged energy drift and diffusion coefficients are given by

$$\langle \Delta E \rangle_{\rm av} = \Gamma\, \left[I_0 - K_g/g\right], \tag{26a}$$

$$\langle \Delta E^2 \rangle_{\rm av} = 2\Gamma\, \left[I_0\, h + K_h\right]/g, \tag{26b}$$

$$K_g(E) \equiv \int_{\Phi(0)}^E f(E')\, g(E')\, dE' = \int_0^{h(E)} f(h')\, dh', \tag{26c}$$

$$K_h(E) \equiv \int_{\Phi(0)}^E f(E')\, h(E')\, dE' = \int_0^{h(E)} \frac{f(h')\, h'}{g(h')}\, dh'. \tag{26d}$$

In these expressions, $\Gamma \equiv 16\pi^2 G^2 M_{\rm total} \times (N_\star^{-1} \ln\Lambda)$, where the term in brackets is the amplitude of relaxation term for the given number of stars $N_\star$ representing the stellar system ($\Lambda \sim N$ is the Coulomb logarithm). We note that $K_g(E)$ is the mass of stars with energies

less than E (and thus $M_{\rm total} = K_g(0)$), and $K_h(E)$ is their kinetic energy (up to a factor 3/2).

Of course, an efficient evaluation of diffusion coefficients again requires interpolation from a pre-computed table. From the above expressions it is clear that I_0, K_g and K_h can be very accurately approximated by quintic splines in h, log-scaled in both coordinates and linearly extrapolated (provided that $f(h)$ also has power-law asymptotic behaviour at large and small h). Moreover, $J_0(E,\Phi) = I_0(\Phi) - I_0(E)$, and $J_{n/2}(E,\Phi) \lesssim J_0$ thanks to the weighting factor (the ratio of velocities of field and test stars to the power of n). Indeed, for $E \to \Phi$, $J_{n/2} \to 1/(n+1)$. We interpolate the ratio $J_{n/2}/J_0$ as a function of $\ln h(\Phi)$ and $\ln h(E) - \ln h(\Phi)$ on a 2d grid covering a very broad range of h; the accuracy of this cubic spline interpolation is $\sim 10^{-4}..10^{-6}$, and it is extrapolated as a constant outside the definition region (while this is a good asymptotic approximation for large h, there is no easy way of delivering a reasonably correct extrapolation to small $h(\Phi)$ – fortunately, the volume of this region is negligible in practice).

Another method for studying the evolution of stellar distribution driven by the two-body relaxation is the Fokker–Planck (FP) equation for $f(h,t)$ coupled with the 1d Poisson equation for $\Phi(r,t)$. The latter provides the potential corresponding to the density profile which is obtained by integrating the DF over velocity: $\rho(r,t) = \int f(h,t) d^3v$. This system of two PDEs – parabolic for the DF and elliptic for the potential – is solved using interleaved steps: first the orbit-averaged drift and diffusion coefficients entering the FP equation are obtained from (26), then the DF is evolved for some interval of time in a fixed potential, then the density is recomputed and the potential is updated through the Poisson equation. Traditionally, the DF is expressed as a function of energy, but this has a disadvantage when it comes to solving the Poisson equation: as the potential changes, the DF should be kept fixed as a function of phase volume, not energy (e.g., [12]). Thus the formulation entirely in terms of $f(h)$ is preferrable, and does not introduce any additional complications. It is convenient to write down the FP equation in the flux-conservative form:

$$\frac{\partial f(h,t)}{\partial(\Gamma t)} = \frac{\partial}{\partial h}\left[D_{hh}\frac{\partial f(h,t)}{\partial h} + D_h f(h,t)\right], \tag{27}$$

$$D_{hh} = g(h)\left[h\, I_0(h) + K_h(h)\right], \qquad D_h = K_g(h). \tag{28}$$

To achieve high dynamical range, h is further replaced by $\ln h$, with a trivial modification of the above expressions.

The Fokker–Planck solver, dubbed PhaseFlow, has several ingredients:

- The distribution function $f(h,t)$ for the evolving population of stars, represented by its values on a grid in $\ln h$. Values of $f(h)$ for an arbitrary argument are obtained from a cubic spline interpolator for $\ln f$ as a function of $\ln h$.

- The potential $\Phi(r)$ corresponding to the density profile $\rho(r)$ of the evolving population, plus optionally an external component (e.g., a central point mass). The 1d Poisson equation is solved by the `Multipole` class (of course, a monopole is a particular case of a multipole).

- The density $\rho(\Phi(r)) = \int_\Phi^0 f\big(h(E)\big)\, 4\pi\sqrt{2(E-\Phi)}\, dE$ is obtained from the DF *in the given potential*, i.e., to update the density entering the Poisson equation, we use the current values of $f(h)$ and the previous potential.

- The drift and diffusion coefficients D_h, D_{hh} are computed using the `SphericalModel` class which combines the DF $f(h)$ with a potential $\Phi(r)$ and a mapping $h \leftrightarrow E$ (`PhaseVolume`) constructed for the given Φ.

- The FP equation (27) in the discretized form is solved with the Chang–Cooper scheme (e.g., [27]) which has excellent energy conservation properties. Typically several FP steps (which are cheap – just a solution of a tridiagonal system) are taken between each update of the potential and the diffusion coefficients (the Poisson step).

References

[1] Ahnert K., Mulansky M., 2011, AIP Conf. Proc. 1389, 1586

[2] Binney J., 2012, MNRAS, 426, 1324

[3] Binney J., 2014, MNRAS, 440, 787

[4] Binney J., McMillan P., 2011, MNRAS, 413, 1889

[5] Binney J., McMillan P., 2016, MNRAS, 456, 1982

[6] Binney J., Spergel D., 1984, MNRAS, 206, 159

[7] Binney J., Tremaine S., 2008, *Galactic Dynamics*, Princeton Univ. press

[8] Bovy J., 2015, ApJS, 216, 29

[9] Carpintero D., Aguilar L., 1998, MNRAS, 298, 1

[10] Carpintero D., Maffione N., Darriba L., 2014, Astronomy & computing, 5, 19.

[11] Cohl H., Tohline J., 1999, ApJ, 527, 86

[12] Cohn H., 1980, ApJ, 242, 765

[13] Dehnen W., 1993, MNRAS, 265, 250

[14] Dehnen W., 2000, ApJL, 536, L39

[15] Dehnen W., Binney J., 1998, MNRAS, 294, 429

[16] de Zeeuw T., 1985, MNRAS, 216, 273

[17] Green P., Silverman B., 1994, *Nonparametric regression and generalized linear models*, Chapman&Hall, London

[18] Hairer E., Nørsett S., Wanner G., 1993, *Solving ordinary differential equations*, Springer-Verlag

[19] Hernquist L., Ostriker J., 1992, ApJ, 386, 375

[20] Kuijken K., Dubinski J., 1995, MNRAS, 277, 1341

[21] Makino J., Aarseth S., 1992, PASJ, 44, 141

[22] Martin R., 2008, *Clean code*, Prentice Hall

[23] McConnell S., 2004, *Code complete*, Microsoft press

[24] Merritt D., Fridman T., 1996, ApJ, 460, 136

[25] Merritt D., Tremblay B., 1994, AJ, 108, 514

[26] Meyers S., 2005, *Effective C++*, Addison–Wesley

[27] Park B., Petrosian V., 1996, ApJS, 103, 255

[28] Pfenniger D., 1984, A&A, 134, 373

[29] Piffl T., Penoyre Z., Binney J., 2015, MNRAS, 451, 639

[30] Portegies Zwart S., McMillan S., van Elteren E., Pelupessy I., de Vries N., 2013, Comput. Phys. Commun., 184, 3, 456

[31] Posti L., Binney J., Nipoti C., Ciotti L., 2015, MNRAS, 447, 3060

[32] Read J., et al., in prep. (Gaia Challenge)

[33] Rein H., Spiegel D., 2015, MNRAS, 446, 1424

[34] Ruppert D., Wand M.P., Carroll R.J., 2003, *Semiparametric regression*, Cambridge Univ. press

[35] Sanders J., Binney J., 2016, MNRAS, 457, 2017

[36] Skokos Ch., 2010, LNP, 790, 63

[37] Silverman B., 1982, Annals of statistics, 10, 795.

[38] Sutter H., Alexandrescu A., 2004, *C++ coding standards*, Addison–Wesley

[39] Teuben P., 1995, in Shaw R. A., Payne H. E., Hayes J. J. E., eds, ASP Conf. Ser. 77, *Astronomical data analysis software and systems IV*, p.398, San Francisco

[40] Valluri M., Merritt D., 1998, ApJ, 506, 686

[41] Vasiliev E., 2013, MNRAS, 434, 3174

[42] Vasiliev E., 2015, MNRAS, 446, 3150

[43] Vasiliev E., Athanassoula E., 2015, MNRAS, 450, 2842

[44] Zhao H.-S., 1996, MNRAS, 278, 488