AGAMA reference

Eugene Vasiliev
email: eugvas@protonmail.com

September 27, 2024

Contents

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 provides tools for analyzing N-body simulations, serves as a base for the Monte Carlo stellar-dynamical code RAGA [72], the Fokker–Planck code PHASEFLOW [73], and the Schwarzschild modelling code FORSTAND [76] (in turn, derived from the earlier code SMILE [71, 75]).

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, demonstration programs and ready-to-use tools; 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, and several programs that actually perform some useful tasks are also included in the distribution. 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 science paper describing the code is [74].

The code can be downloaded from `http://agama.software`.

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 instances of classes 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`, `QRDecomp`, `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.

QR decomposition represents a generic $M \times N$ matrix (M rows, N columns) as $\mathsf{A} = \mathsf{Q}\,\mathsf{R}$, where Q is a $M \times M$ orthogonal matrix (i.e., $\mathsf{Q}\mathsf{Q}^T = \mathsf{I}$) and R is a $M \times N$ upper triangular matrix (i.e., elements below the main diagonal are zero). It can be used, e.g., for orthogonalizing a set of basis vectors (with $M = N$); the previous two decompositions are more efficient for solving a non-degenerate square linear system.

Singular-value decomposition (SVD) represents a generic $M \times N$ matrix ($M \geq N$) as $\mathsf{A} = \mathsf{U}\,\mathrm{diag}(\boldsymbol{S})\,\mathsf{V}^T$, where U is a $M \times N$ orthogonal matrix, 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 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 [30]; 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 (a more detailed description is given in Section A.2.7).

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 rectangular, i.e., aligned with the coordinate lines in each dimension. The advantages of this approach are locality (the function value depends only on the adjacent grid points), adaptivity (grid nodes need not be uniformly spaced and may be concentrated 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).

There are linear, cubic and quintic (fifth-order) interpolation schemes in one, two and three dimensions (quintic – only in 1d and 2d). 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 three (1d) or two (2d/3d) 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]. The interpolating function is defined as $f(\boldsymbol{x}) = \sum_\alpha A_\alpha B_\alpha(\boldsymbol{x})$, where α is a combined index

[1] A general 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.

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 $O(\log(N_x)+N^2)$ operations per dimension to locate the grid segment and compute all N possibly nonzero basis functions using a N-step recursion relation. This formulation is more suitable for constructing approximating splines from a large number of scattered points (see next paragraph), and the resulting B-splines may be subsequently converted to more efficient linear or cubic interpolators. This approach is currently implemented in 1 and 3 dimensions.

B-splines can also be used as basis functions in finite-element methods: any sufficiently smooth function can be approximated by a linear combination of B-splines on the given interval, and hence represented as a vector of expansion coefficients. Various mathematical operations on the original functions (sum, product, convolution) can then be translated into linear algebra operations on these vectors. The 1d finite-element approach is used in the Fokker–Planck code PhaseFlow, which is included in the library, and in a few other auxiliary tasks (e.g., solution of Jeans equations).

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 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 (usually logarithmic). 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). In addition, all 1d interpolators (cubic/quintic splines and B-splines) provide methods for computing analytic integrals, convolutions with arbitrary kernels, and determination of roots and extrema. Mathematical foundations of splines are described in more detail in the Appendix (sections A.2.2 and A.2.3).

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 $\sum_i [y_i - f(x_i)]^2 + \lambda \int [f''(x)]^2\,\mathrm{d}x$, 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 $\{x_i\}$ [29]; 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 are given in the Appendix (sections A.2.4 and A.2.5).

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 $G = 1$. In the future, we might adopt an alternative unit handling approach that would be seamlessly integrated with the units subsystem of the ASTROPY library [3].

2.1.3 Coordinates

The `coord::` 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: `Car`tesian, `Cyl`indrical, `Sph`erical, 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 (focal 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. In the future, we may add other coordinate systems (e.g., heliocentric) into the same framework.

Some routines (e.g., in the `galaxymodel::` namespace, Sections 2.6.1, 2.6.4) can use an "observed" coordinate system XYZ that is arbitrarily oriented with respect to the "intrinsic" coordinate system xyz of the model, parametrized by three Euler rotation angles (see Section A.3 for details and illustrations).

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 coordinate 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`, `BasisSet` 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 a prolate spheroidal coordinate system.

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

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`, `NFW` (Navarro–Frenk–White) potentials, and a generalized `King` (lowered isothermal) model which is specified by its distribution function $f(E)$, as given by Equation 1 in [28]. 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. A point mass (Kepler) potential is obtained by constructing a `Plummer` potential with zero scale radius.

Axisymmetric models include the `MiyamotoNagai` and `OblatePerfectEllipsoid` potentials (the latter belongs to a more general class of Stäckel potentials [26], but is the only one implemented at present). There is another type of axisymmetric models that have a dedicated potential class, namely a separable `Disk` profile with $\rho(R,z) = \Sigma(R)\,h(z)$. 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 GalPot approach introduced in [37, 25]: 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` [22] 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. Ferrers ($n = 2$) models are strictly triaxial, and have analytic expressions for the potential and its derivatives [47]. 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 [42], 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 constructed from a `Spheroid` density profile. This class describes general triaxial two-power-law ($\alpha\beta\gamma$) density profiles[2] [79] with an optional exponential cutoff. Many well-known models are special cases of this profile: Dehnen, Plummer, Isochrone, NFW, Gaussian, Einasto, Prugniel–Simien. This class only provides the density profile and not the potential, so it needs to be used in conjunction with the `Multipole` potential solver. Some models are defined in terms of the radial profile of the surface density, from which the 3d density may be reconstructed by deprojection. These include the two-power-law `Nuker` model and the exponential `Sersic` model; both provide only the density but not potential, and can also be flattened or triaxial (note that the deprojected spherical density profile is constructed first, and then it is compressed/stretched along y and z axes). Generalized spherical `King` models (with an adjustable strength of the outer cutoff, as in [28]) provide both the density and the potential. A time-dependent potential of two point masses orbiting each other is represented by the `KeplerBinary` class, and a spatially uniform but time-dependent acceleration field is provided by the `UniformAcceleration` class.

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 ($N_r \sim 20 \div 50$), 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:

[2] α here corresponds to $1/\alpha$ in the original paper: higher values of α produce sharper transitions between inner and outer asymptotic slopes.

- 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).

A related `BasisSet` potential approximation 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 [32, 79]. 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). For analytic density profiles, `Multipole` usually provides much higher accuracy than `BasisSet` for a comparable (or lower) computational cost. When initialized from an N-body snapshot, the accuracy of both expansions is limited by discreteness noise in the coefficients, typically saturating at $l_{\max} \sim 6-8$ and twice as many radial grid points [59].

`DensitySphericalHarmonic` is a class derived from `BaseDensity`, which uses the same mechanism (spherical-harmonic expansion in angles with coefficients given by spline functions in radius) to represent an arbitrary density profile, without an associated potential. It is used as an intermediate step in constructing a `Multipole` potential from an N-body snapshot, or as an internal representation for some spherically symmetric density profiles such as `King` (in this case, with $l_{\max} = 0$).

2.2.3 Azimuthal harmonic expansion

`CylSpline`[3] 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

[3] an improved version of the method presented in [75]

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[4].

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 9, 10 in the Appendix, with more technical details given in Section A.4). 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).

A related class `DensityAzimuthalHarmonic` is used to represent an arbitrary density profile as a Fourier expansion in ϕ, with each term being a 2d cubic spline in R, z. This class is typically not constructed directly, but together with `DensitySphericalHarmonic`, serves as an interpolator for the density profiles in the iterative self-consistent modelling procedure (Section 2.6.3). All potential and density expansions can be stored to and loaded from text files, as described in the next section.

2.2.4 Potential factory

All density and potential classes may be constructed using a universal "factory" interface – several routines that return new instances of `PtrDensity` or `PtrPotential` according to the provided parameters. The parameters can be supplied in several ways. The routines `readDensity` and `readPotential` read them from a text file (referred to as INI file thereafter) containing one or several components of the potential described in separate sections `[Potential]`, `[Potential2]`, `[Potential disk]`, etc. (all section names should start with "Potential") for `readPotential`, or similarly a file with one or more density components listed in sections `[Density]`, `[Density1]`, etc. for `readDensity`. These sections may contain coefficients of density or potential expansion previously written by `writeDensity` /

[4]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.

Table 1: Static density and potential models and their parameters

Name	Formula	Parameters
	Density-only models	
`Disk`	$\rho = \Sigma_0 \exp\left(-\left[\frac{R}{R_\mathrm{d}}\right]^{\frac{1}{n}} - \frac{R_\mathrm{cut}}{R}\right) \times \begin{cases} \delta(z) & \text{if } h = 0 \\ \frac{1}{2h}\exp\left(-\left\lvert\frac{z}{h}\right\rvert\right) & h > 0 \\ \frac{1}{4\lvert h\rvert}\operatorname{sech}^2\left(\left\lvert\frac{z}{2h}\right\rvert\right) & h < 0 \end{cases}$	`surfaceDensity` (Σ_0) or `mass`, `scaleRadius` (R_d), `scaleHeight` (h), `innerCutoffRadius` (R_cut), `sersicIndex` (n)
`Spheroid`	$\rho = \rho_0 \left(\frac{\tilde{r}}{a}\right)^{-\gamma}\left[1+\left(\frac{\tilde{r}}{a}\right)^{\alpha}\right]^{\frac{\gamma-\beta}{\alpha}} \times \exp\left[-\left(\frac{\tilde{r}}{r_\mathrm{cut}}\right)^{\xi}\right]$	`densityNorm` (ρ_0) or `mass`, `alpha` (α), `beta` (β), `gamma` (γ), `scaleRadius` (a), `axisRatioY` (p), `axisRatioZ` (q), `outerCutoffRadius` (r_cut), `cutoffStrength` (ξ)
`Nuker`	deprojection of $\Sigma = \Sigma_0 \left(\frac{R}{a}\right)^{-\gamma}\left[\frac{1}{2}+\frac{1}{2}\left(\frac{R}{a}\right)^{\alpha}\right]^{\frac{\gamma-\beta}{\alpha}} \times \exp\left[-\left(\frac{R}{r_\mathrm{cut}}\right)^{\xi}\right]$	`surfaceDensity` (Σ_0) or `mass`, `alpha` (α), `beta` (β), `gamma` (γ), `scaleRadius` (a), `axisRatioY` (p), `axisRatioZ` (q), `outerCutoffRadius` (r_cut), `cutoffStrength` (ξ)
`Sersic`	deprojection of $\Sigma = \Sigma_0 \exp\left[-b_n\,(R/a)^{1/n}\right]$	`surfaceDensity` (Σ_0) or `mass`, `scaleRadius` (a), `sersicIndex` (n), `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 is the mass enclosed in $\sim 5.3a$, the total mass is ∞), `scaleRadius` (a)
`MiyamotoNagai`	$\Phi = -\frac{M}{\sqrt{R^2+\left(a+\sqrt{z^2+b^2}\right)^2}}$	`mass` (M), `scaleRadius` (a), `scaleRadius2` or `scaleHeight` (b)
`PerfectEllipsoid`	$\rho = \frac{M}{\pi^2\, q\, a^3}\left[1+\frac{R^2+(z/q)^2}{a^2}\right]^{-2}$	`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), `gamma` (γ), `axisRatioY` (p), `axisRatioZ` (q), `scaleRadius` (a)
`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)
`King`	specified by $f(E)$, see text	`mass`, `scaleRadius` (r_c), `W0`, `trunc` (g)
`Logarithmic`	$\Phi = \frac{1}{2}v_0^2\,\ln(r_\mathrm{core}^2+\tilde{r}^2)$	`v0` (v_0), `scaleRadius` (r_core), `axisRatioY` (p), `axisRatioZ` (q)
`Harmonic`	$\Phi = \frac{1}{2}\Omega^2\,\tilde{r}^2$	`Omega` (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

Table 2: Time-dependent potential models and their parameters

Name	Formula	Parameters
`KeplerBinary`	two moving point masses	`mass`, `binary_q`, `binary_sma`, `binary_ecc`, `binary_phase`
`UniformAcceleration`	$\Phi = -\boldsymbol{a}(t)\,\boldsymbol{x}$	`file`
`Evolving`	piecewise-constant or piecewise-linear sequence of other potentials	(see Section 2.2.5)

Table 3: Density and potential expansion types

Name (density)	Name (potential)	Parameters
—	`BasisSet`	`nmax`, `eta`, `r0`, `lmax`, `mmax`, `symmetry`, `fixOrder`
`DensitySphericalHarmonic`	`Multipole`	`gridSizeR`, `rmin`, `rmax`, `lmax`, `mmax`, `symmetry`, `fixOrder`
`DensityAzimuthalHarmonic`	`CylSpline`	`gridSizeR`, `gridSizeZ`, `Rmin`, `Rmax`, `zmin`, `zmax`, `mmax`, `symmetry`, `fixOrder`

Table 4: Density and potential modifiers

Name	Parameter	Description (see Section 2.2.5)	Result
`Shifted`	`center`	3 numbers or a file with $\boldsymbol{x}_0(t)$	$\Phi(\boldsymbol{x}-\boldsymbol{x}_0)$
`Tilted`	`orientation`	three Euler angles $\alpha,\beta,\gamma \Rightarrow$ rotation matrix R	$\Phi(\mathsf{R}_{\alpha\beta\gamma}\,\boldsymbol{x})$
`Rotating`	`rotation`	a single value or a file with $\psi(t)$	$\Phi(\mathsf{R}_{\psi 00}\,\boldsymbol{x})$
`Scaled`	`scale`	two values or a file with $A(t), S(t)$	$A\,S^{-1}\,\Phi(\boldsymbol{x}/S)$

Table 5: Symmetry types and their implications

Name	Invariant transformations	Sph.-harm. coefs identically zero
`None`	—	—
`Reflection`	$\{x,y,z\} \to \{-x,-y,-z\}$ (twofold discrete symmetry)	odd l
`Bisymmetric`	same or $z \to -z$, also implies $\{x,y\} \to \{-x,-y\}$ (fourfold discrete symmetry, e.g., a two-arm spiral)	same + odd m
`Triaxial`	same or $x \to -x$ or $y \to -y$ (eightfold discrete symmetry, e.g., a bar)	same + negative m
`Axisymmetric`	same or rotation about z axis by any angle (continuous symmetry in ϕ)	same + any $m \neq 0$
`Spherical`	same or rotation about origin by any angle (continuous symmetry in both θ and ϕ)	same + any $l \neq 0$

writePotential routines, or references to other INI files with yet other parameter sets, etc. Alternatively, the routine createDensity and several overloaded routines createPotential take 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, BasisSet 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 routines.

Below follows 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 a static analytic potential listed in the first column of Table 1, or a time-dependent potential from Table 2, or an expansion listed in the second column of Table 3, or a modifier listed in Table 4. It is usually required, unless this section contains a file parameter referring to another INI file with potential parameters.

- 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 from Table 1, except unbound potentials – Logarithmic or Harmonic).
  There is one exception to the rule that type must encode a potential class: it may also contain the names of the density profiles originally used in GalPot – Disk, Spheroid, Nuker or Sersic. 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 Spheroid, Nuker, Sérsic and Disk 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=Disk, 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 CompositeDensity model, and pass it to the constructor of a potential expansion (this approach is used for self-consistent multicomponent models, Section 2.6.3).

- 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 needed 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 from Table 5 (only the first letter is used), or a number encoding a more complicated symmetry (see the definitions in `coord.h`).

- `file` can serves several purposes. It may refer to another INI file with one or more sections describing density or potential parameters, which may also contain `Multipole`, `BasisSet` or `CylSpline` potential expansion coefficients (if used with `readPotential`), or likewise `DensitySphericalHarmonic` / `DensityCylindricalHarmonic` coefficients (if used with `readDensity`) previously written by `writePotential` / `writeDensity` routines. In this case the `type` parameter should not be provided.
  Alternatively, it may point to an N-body snapshot file used to create such an expansion (in this case the `type` of expansion needs to be specified, possibly with some other parameters).
  Finally, for the `UniformAcceleration` potential type, this file contains the time-dependent acceleration field, and should have 4 columns – time (monotonically increasing) and three acceleration components, which will be interpolated in time as regularized cubic splines (see Figure 3) and linearly extrapolated beyond the endpoints. One may provide the same 2d array directly as a text string in the `file` argument, serialized as follows: `[[t1,ax1,ay1,az1],[t2,ax2,ay2,az2],...]` – when called from `Python`, this argument may contain a `numpy` array, which is automatically converted into a string in this format and then parsed inside the `C++` code.

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[5].
- `scaleRadius` [1] – the first (sometimes the only) parameter with the dimension of length that defines the profile.
- `scaleHeight` [1] or `scaleRadius2` – the second such parameter (e.g., for Miyamoto–Nagai or exponential disk models).
- `outerCutoffRadius` [∞] – another length-scale parameter defining the radius of exponential truncation, used for `Spheroid` or `Nuker` models (∞ means no cutoff).
- `innerCutoffRadius` [0] – similar parameter for `Disk` that defines the radius of an inner hole.

[5]Except the `NFW` profile, in which the total mass is formally infinite, and the parameter M refers to the mass within $\sim 5.3a$, where a is the scale radius. It is related to the so-called virial mass $M_{\rm vir}$ and so-called concentration c by $M_{\rm vir} = M\,[\ln(1+c) - c/(c+1)]$. An NFW profile sharply cut at the virial radius is equivalent to setting `outerCutoffRadius` to the virial radius ($a\,c$) and `cutoffStrength` to a very large value (however, this may lead to numerical artifacts).

- surfaceDensity [0] – normalization of surface density (its value at $R = 0$ for the exponential Disk or Sersic profiles (*not* at the half-light radius!), or at $R =$scaleRadius for the Nuker profile).

- densityNorm [0] – value that defines the volume density at the scale radius for the Spheroid 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), but this can't be done for infinite-mass models, so the density normalization remains the only option.

- alpha [1] – parameter controlling the steepness of transition between two asymptotic power-law slopes for Spheroid or Nuker.

- beta [4] – power-law index of the outer density profile for Spheroid or Nuker; 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 $0 \le \gamma \le 2$) or Spheroid ($\gamma < 3$) models, or $\Sigma \propto R^{-\gamma}$ for Nuker model ($0 \le \gamma < 2$).

- cutoffStrength [2] – parameter ξ controlling the steepness of the exponential cutoff in Spheroid or Nuker models: $\rho \propto \exp\left[-(r/r_{\rm cut})^{\xi}\right]$. It can also be used to create the Einasto profile, in which $\rho(r) \propto \exp\left[-c_n\,(r/r_{\rm half})^{1/n}\right]$, where $c_n \approx 3n - 1/3 + \frac{0.0075}{n-0.05}$ is the root of $2\Gamma(3n, c_n) = \Gamma(3n)$ and n is the Einasto index: in this case set $\gamma = \beta = 0$, $\xi = 1/n$, $r_{\rm cut} = r_{\rm half}/c_n^n$ and densityNorm $= 3M/[4\pi\, r_{\rm cut}^3\, \Gamma(3n+1)]$.

- sersicIndex – shape parameter n of the Sersic profile (larger values correspond to a models with steeper inner and shallower outer profiles, default is the de Vaucouleur's value of 4), or the same parameter for the Disk profile (default is 1 corresponding to the exponential disk). Please note that the meaning of scaleRadius is *not the same* for the two cases: it corresponds to the projected half-light radius for the Sersic profile, but differs from it by a constant factor that depends on n and $b_n(n)$ (see the expressions in Table 1; $b_n \approx 2n - 1/3$ is computed automatically) for the Disk profile. The projected density of the Disk profile matches the Sérsic profile (after appropriate rescaling of length) only in the face-on orientation, and the flattening is also specified differently (q=z/x for the Sersic profile and scaleHeight for the Disk profile).

- p or axisRatioY [1] – the axis ratio y/x of equidensity surfaces of constant ellipticity for Dehnen, Spheroid, Nuker, Sersic or Ferrers models, or the analogous quantity for the Logarithmic or Harmonic potentials.

- q or axisRatioZ [1] – the axis ratio z/x, same list of models plus PerfectEllipsoid.

- `W0` – dimensionless potential depth of generalized `King` (lowered isothermal) models: $W_0 = [\Phi(r_t) - \Phi(0)]/\sigma^2$; larger values correspond to more extended envelopes (larger ratio between the outer truncation radius r_t and the scale radius). In the above expression, the velocity dispersion σ is not an independent parameter: the model in dimensionless units is specified by W_0 and the truncation strength parameter g; the potential, the truncation radius, and the total mass in dimensionless units are all determined by integrating a second-order ODE, and then the length and mass units are rescaled to match the given total mass M and the scale radius (also called King radius or core radius).

- `trunc` [1] – truncation strength parameter of lowered isothermal models (denoted by g in [28]); should be between 0 and 3.5 (0 corresponds to Woolley, 1 – to King, 2 – to Wilson models), larger values result in softer density fall-off near the truncation radius.

- `Omega` [1] – frequency of oscillation in the `Harmonic` potential.

- `v0` [1] – asymptotic circular velocity for the `Logarithmic` potential.

- `binary_sma` [0] – semimajor axis a for the `KeplerBinary` potential. This model represents a time-dependent potential of two point masses orbiting each other in the $x-y$ plane, with the center of mass specified by the `center` parameter. $a = 0$ means a single point mass (the same effect is produced by a `Plummer` model with `scaleRadius`=0).

- `binary_q` [0] – mass ratio q of the `KeplerBinary` potential (0 means a single massive object, otherwise the masses of the two components are $m_1 = m/(1+q),\ m_2 = qm_0$).

- `binary_ecc` [0] – orbital eccentricity e of the `KeplerBinary` potential ($0 \le e \le 1$).

- `binary_phase` [0] – orbital phase ϕ_0 of the `KeplerBinary` at time $t = 0$. The positions of two point masses at time t are given by $x_1 = a\,\frac{q}{1+q}\,(\cos\eta - e),\ y_1 = a\,\frac{q}{1+q}\,\sqrt{1-e^2}\,\sin\eta$ for the first one and $x_2 = -x_1/q,\ y_2 = -y_1/q$ for the second one, where the eccentric anomaly $\eta(t)$ is the solution of Kepler's equation: $\eta - e\,\sin\eta = \Omega t + \phi_0$, and $\Omega \equiv \sqrt{m/a^3}$ is the orbital frequency.

Parameters defining the density or 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`, `BasisSet` and `DensitySphericalHarmonic`) or cylindrical (`CylSpline` and `DensityAzimuthalHarmonic`) radius; in the latter case this includes the 0th node at $R = 0$.

- `gridSizeZ` [25] – same for the grid in z direction in `CylSpline` and `DensityAzimuthalHarmonic`, including the $z = 0$ node.

- `rmin` [0] – the radius of the innermost nonzero node in the radial grid (for all expansion classes); zero means automatic determination.

- `rmax` [0] – same for the outermost node; zero values mean automatic determination.

- `zmin` [0], `zmax` [0] – same for the vertical grid in `CylSpline`/`DensityAzimuthalHarmonic`; zero values mean take them from the radial grid. Note that the grid auto-setup mechanism is currently less optimal in `CylSpline` than in `Multipole`, so a sensibly chosen manual grid extent may be beneficial for accuracy.

- `lmax` [6] – the order of `Multipole`, `BasisSet` and `DensitySphericalHarmonic` expansion in $\cos\theta$; 0 means spherical symmetry.

- `mmax` [lmax] – the order of azimuthal Fourier expansion in ϕ for all classes; 0 means axisymmetry, and $m_{\max}$ should be $\le l_{\max}$. 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. Moreover, if all terms in the computed expansion beyond a certain order are zero, the actual values of $l_{\max}$ and $m_{\max}$ can be smaller than the requested ones. Note that for `CylSpline`, values of $m_{\max} > 12$ significantly increase the cost of construction of the potential from a density profile (though not of its evaluation, which is roughly proportional to $m_{\max}+1$ in any case).

- `fixOrder` [false] – whether to restrict the number of integration points in angles θ and ϕ to the minimum necessitated by the requested expansion order $l_{\max}$, $m_{\max}$. A spherical-harmonic transformation of a band-limited input function needs $l_{\max}/2+1$ points in θ (or twice as many if the input is not z-reflection-symmetric), and a Fourier transformation needs $m_{\max}+1$ points in ϕ (or twice as many for non-y-reflection-symmetric inputs). However, the routines typically use more than this minimum number, because the input is rarely band-limited (e.g., when `mmax=0`, the expansion will be axisymmetric, but it still needs to integrate the input model over ϕ to produce a correct result). By default (when `fixOrder=false`) the internally constructed expansions have an order $\mathsf{max}(12, \{l/m\}_{\max}+6)$ and query the input models at the corresponding number of angular points, then are truncated to the requested output order. On the other hand, when the input density is expensive to compute (e.g., in the context of DF-based self-consistent models), one may limit this internal expansion order to exactly the output order, thus having a more explicit control on the number of input density evaluations.

- `smoothing` [1] – the amount of smoothing applied to the non-spherical harmonics during the construction of the `Multipole` potential from an array of particles.

- `nmax` [12] – the order of radial expansion in `BasisSet` potential.

- `eta` [1] – parameter controlling the shape of basis functions in the Zhao basis set [79]. The zeroth-order function is a double-power-law (`Spheroid`) profile with $\alpha = 1/\eta$, $\beta = 3 + 1/\eta$ and $\gamma = 2 - 1/\eta$; the default value $\eta = 1$ corresponds to the widely used Hernquist–Ostriker basis set [32], although values up to 2 and even higher may provide more accurate results for cuspy models.

- `r0` – scale radius of basis functions. If not provided, it is set to the half-mass radius of the density profile (unless the latter has infinite mass, in which case one needs to specify `r0` explicitly), and this choice is close to optimal for the approximation accuracy.

These keywords, with some modifications, are also used in potential construction routines in `Python` and `Fortran` interfaces and in the AMUSE, GALPY and GALA plugins (Sections 3.1, 3.2, 3.6, 3.4, 3.5). For instance, `Python` interface allows to provide a user-defined function specifying the density or potential profile in the `density=` or `potential=` 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.

The coefficients of a density or a potential expansion can be stored to a text file by `writeDensity` / `writePotential` routines (which in fact refer to the same routine), and subsequently loaded back by the `readDensity` / `readPotential` routines. The `write***` routine, in fact, accepts any density/potential class, including composite and modified models, but it can only write the parameters and coefficients of expansion models, and simply stores the name of any other model without additional parameters or modifiers (rather than throwing an error) – note that these other models may not be correctly loaded back unless you manually edit the file and add the missing properties. Its main purpose is indeed to store the non-parametric (expansion) models. The storage format is compatible with the INI file – in fact, the density/potential model, or each component of a composite model, is written to a separate `[Density***]` or `[Potential***]` section, with the `type` parameter specifying the name of the expansion model, followed by the parameters of the grids and expansion orders, and then the coefficients themselves after a line containing a single word `Coefficients`. When loading a density or a potential from an INI file, these coefficients are then used to reconstruct the appropriate expansion (note that they do not follow the INI format of `key=value` parameters, but are simply appended after all such parameters at the end of each INI section). All components of a composite density or potential object are stored in a single file, one after another.

2.2.5 Modifiers and time-dependent density/potential types

All potential and density classes provide functions for evaluating them at an arbitrary moment of time (0 by default), although almost all built-in models are time-independent (except `KeplerBinary` and `UniformAcceleration`). However, there are two ways in which even a static density or potential can be made time-dependent:

- By constructing an `Evolving` potential from an INI file (or a section in such a file) which has the following format: a line with `type=Evolving`, an optional line `interpLinear=[true/false]`, optional modifier parameters discussed below, followed by a line with a single word `Timestamps`, and the remaining lines in this section containing a table with two columns – timestamps and names of corresponding INI files with potential parameters. The actual potential at the given time t is either taken from the nearest timestamp, or linearly interpolated between the two potentials associated with timestamps $t_1 \le t \le t_2$, if the parameter `interpLinear` is set to `true` (note that this is twice more expensive than taking the nearest one). Of course, the individual INI files may contain coefficients of potential expansions, or specify composite or time-dependent potentials of arbitrary complexity.

- By applying one or more "modifiers" from Table 4. The modifiers are not named explicitly, but are constructed whenever a corresponding parameter appears in the section of an INI file or in a `KeyValueMap` object passed to the factory routines, and their effects are described below: $\Phi(\boldsymbol{x},\,t)$ refers to the original potential and $\widetilde{\Phi}(\boldsymbol{x},\,t)$ – to the modified one; all these modifiers can be applied to density objects as well.
  `center` displaces the potential center by a time-dependent vector $\boldsymbol{x}_0(t)$: $\widetilde{\Phi}(\boldsymbol{x},\,t) = \Phi\big(\boldsymbol{x} - \boldsymbol{x}_0(t),\,t\big)$. A `Shifted` potential or density automatically degrades `symmetry` to `None`.
  `orientation` changes the orientation of its principal axes. The transformation between the original and the modified coordinate system is effected by a rotation matrix R specified by a triplet of Euler angles α, β, γ[6]; see Section A.3 for a definition of R and illustration of the two coordinate systems. Lowercase letters x, y, z denote the coordinates supplied to the modified potential $\widetilde{\Phi}$, and uppercase X, Y, Z are the rotated coordinates fed to the original (underlying) potential Φ; in other words, $\widetilde{\Phi}(\boldsymbol{x},\,t) = \Phi\big(\mathsf{R}\,\boldsymbol{x},\,t\big)$. A `Tilted` density/potential has at most a `Reflection` symmetry.
  `rotation` makes the potential figure rotate about the z axis by an angle $\psi(t)$ that is an arbitrary function of time. The transformation from the modified to the underlying coordinate systems is again given by a (simpler) rotation matrix, in which ψ is the first Euler angle, and the other two angles are zero. For instance, if $\psi(t) = \Omega\,t$, the underlying potential steadily rotates anticlockwise with an angular frequency Ω. The symmetry of a `Rotating` model is at most `Bisymmetric` (except when the model is

[6] no relation to the parameters of a `Spheroid` potential! These three angles are not named explicitly, but rather given as a space- or comma-separated string, e.g., `orientation=1,2,3`.

already spherical, in which case the rotation is meaningless anyway).
`scale` varies the mass normalization $A(t)$ and spatial scale $S(t)$ with time: $\widetilde{\Phi}(\boldsymbol{x},\,t) = A(t)\,S^{-1}(t)\,\Phi\big(S(t)\,\boldsymbol{x},\,t\big)$. The factor S^{-1} ensures that when changing the spatial scale S, the total mass remains the same (unless, of course, adjusted by A). A `Scaled` model retains its original symmetry.
The parameters of these transformations can be constant or time-dependent, except `orientation`, whose triplet of Euler angles is kept fixed. For the other three modifiers, if the corresponding parameter is a single (for `rotation`), two (for `scale`), or three (for `center`) space- or comma-separated numbers, it is fixed in time, otherwise it is interpreted as a 2d table describing the time-dependent variation of this parameter, or the name of a text file containing such a table. A text file should contain timestamps in the first column, and the value(s) of the parameter in the remaining columns, and will be converted into a regularized cubic spline (linearly extrapolated beyond the endpoints of the specified time interval). To extrapolate as a constant, make the next-to-last point identical to the last point. The file may contain not only values but also time derivatives of the parameter (e.g., have 7 columns for `center`: $t, x, y, z, \dot{x}, \dot{y}, \dot{z}$), in which case a Hermite spline will be constructed (it is extrapolated with a slope that is explicitly set by the derivative at the endpoint). Instead of a text file, the same table can be provided directly in the corresponding parameter, serialized as follows (example given for the rotation modifier): `[[t1,a1],[t2,a2],[t3,a3]]`. This is most useful when constructing the potential from the `Python` interface and providing a nested list or a 2d `numpy` array directly. For example, a common case of a constant rotation frequency Ω with an initial phase ϕ_0 may be specified as `rotation=[[0,phi0],[1,phi0+Omega]]`.
If a section of the INI file or a `KeyValueMap` object prescribes multiple modifiers, they will be applied in the following, most natural order (regardless of their order of appearance in the INI file): the underlying potential is modulated in amplitude and size, then made rotating about its z axis, then its principal axes are tilted w.r.t. the external inertial reference frame (in which it will be evaluated), and finally the origin of the potential is shifted. For instance, one can make the potential spin about an arbitrary axis, not just z, by providing both `rotation` and `orientation`. Evaluating this multiply-modified potential or its derivatives unfolds the chain of modifiers in the reverse order to their creation, i.e., the input point is shifted, tilted, rotated and scaled, then fed into the underlying potential, and the result is propagated back through this sequence of transformations.
When constructing a multicomponent density/potential from an entire INI file or a vector of `KeyValueMap`s, the routines `readPotential` / `createPotential` will group the components sharing the same modifier parameters into separate "bunches" of elementary or composite potentials, which will then be wrapped into the corresponding modifier classes, thereby making the potential evaluation more efficient. The `readDensity` routine that constructs a possibly composite density from an INI file will apply modifiers to each component separately without attempting to group them.

On the other hand, one can add modifiers to an already existing density or potential object in the same way as creating a density/potential expansion. In this case, the original density/potential instance is provided to the factory routine `createDensity` / `createPotential` together with a `KeyValueMap` containing the parameters of modifiers to be added. This makes possible to apply multiple modifiers in a different order from the default one, by calling the factory routine several times and "dressing up" the model one layer at a time.

The time-dependent potentials are fully supported by the orbit integration routine, but not by the rest of the library (i.e., all utility functions, sampling from the density profile, construction of action finders, etc., evaluate the potentials at the default time 0).

2.2.6 Utility functions

Methods of the `BaseDensity` class include `enclosedMass` (compute the mass enclosed within a given spherical radius) and `totalMass`, which both use 3d integration by default, but may be reimplemented more efficiently by derived classes. `potential_base.h` contain several utility functions that operate on any potential object: determination of the radius that encloses a given mass; projection of density or force along arbitrary lines of sight specified by Euler angles (Section A.3). It also provides wrapper classes `Sphericalized`/`Axisymmetrized` for both density or potential inputs, which perform on-the-fly symmetrization by averaging over angles (unless the input model already has the desired symmetry level).

`potential_utils.h` provides routines for conversion between energy E, angular momentum of a circular orbit $L_{\rm circ}$, and radius; epicyclic frequencies κ, ν, Ω as functions of radius[7]; peri- and apocenter radii of an orbit with given E, L in the $z = 0$ plane, 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 and axisymmetric action finder/mapper classes. The standalone functions accept potentials of any symmetry, but produce an exact result only if the potential is axisymmetric; otherwise the input potential is axisymmetrized on the fly, which usually gives a meaningful result that one is interested in (e.g., a "typical" orbital period). The interpolators, on the other hand, refuse to work with a non-axisymmetric input potential.

2.3 Orbit integration and analysis

Orbits of particles in a [possibly time-dependent] potential are computed using the class `orbit::OrbitIntegrator`, specifically its method `run`, in any of the three standard coordinate systems, plus optionally a rotating reference frame (see Section A.5 for details). Note

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

that in the of a rotating reference frame (with angular frequency Ω directed along z axis), the velocity (both in the initial conditions and in the output trajectory) is still specified in an inertial frame that is instantaneously aligned with the rotating frame at the corresponding moment of time (i.e., has the same value independently of the pattern speed). For instance, an orbit trapped into a 1:1 corotation resonance with a bar would have a fixed position in the rotating frame, but a nonzero azimuthal velocity. On the other hand, if a `Rotating` modifier is applied to the potential itself and the orbit integration is performed in the inertial reference frame, then the trajectory is also stored in the inertial frame. This setup is more general since the angle of rotation may vary arbitrarily (not just linearly with time, as in the case of a constant angular frequency), but extra steps would be needed to convert the trajectory into the instantenously corotating frame.

There are various tasks that can be performed during orbit integration, using classes derived from `orbit::BaseRuntimeFnc`. The simplest one (`orbit::RuntimeTrajectory`) is the recording of the trajectory either at every timestep of the ODE solver, or at regular intervals of time, which are unrelated to the internal solver timestep (that is, the position and velocity at any time are obtained by interpolation provided by the solver – so-called dense output feature). More complicated tasks involve storage of some other kind of information, e.g., in the context of Schwarzschild modelling, or in some cases, even modifying the orbit itself (random perturbations mimicking the effect of two-body relaxation in the Monte Carlo code Raga).

The class `orbit::RuntimeVariational` handles the integration of the variational equation, which contains the second derivatives of the potential and describes the evolution of deviation vectors (infinitesimally small perturbations to the orbital initial conditions). In other words, this provides the time-dependent Jacobian $\mathsf{J}(t) = \partial \boldsymbol{w}(t) / \partial \boldsymbol{w}(t_0)$, where $\boldsymbol{w} \equiv \{\boldsymbol{x}, \boldsymbol{v}\}$ is the 6d phase-space vector, making the orbit computation differentiable at the expense of a moderate increase in cost (between 25% and 200%).

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 [10, 16], and their rate of change with time is a measure of chaos [69]. These methods were implemented in [71], 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.

A finite-time estimate of Lyapunov exponent λ is another measure of stochasticity, which also uses the variational equation (see [17, 61] for reviews). The `orbit::RuntimeVariational` task can integrate a single deviation vector or a full set of them, and in either case records the evolution of the magnitude of the largest vector $|\boldsymbol{w}|$. For a regular orbit, it grows at most linearly with time, while for a chaotic orbit it eventually starts to grow exponentially. Once the orbit integration is finished, this class reports the slope of $\ln(|\boldsymbol{w}|)$ as a function of time, normalized to the characteristic orbital time (so that orbits at different energies can be more directly compared); if no exponential growth has been detected, it returns $\lambda = 0$.

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 [11].

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. Actions are only defined for a bound orbit – if the energy is positive, they will be reported as `NAN` (except L_z which can always be computed).

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 can output any combination of actions, angles and frequencies, skipping the computation of unneeded quantities.

The following sections describe the methods suitable for specific cases of spherical or axisymmetric potentials (see [58] 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. The file `action_factory.h` provides driver routines for computing actions and creating action finder/mapper instances, which automatically choose the appropriate implementation among the ones described below, depending on the potential.

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 [11]. The standalone routines `evalIsochrone`/`mapIsochrone` providing these transformations, and the corresponding wrapper class `ActionFinderIsochrone`, 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

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

where $r_{\rm min,max}(E,L)$ are the roots of the expression under the radical. The standalone routines `evalSpherical`/`mapSpherical` 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 $r_{\rm min,max}(E,L)$ (using the helper class `potential::Interpolator2d`) and $J_r(E,L)$, the inverse mapping $E(J_r,L)$ also provided via an interpolation table, and the complete inverse mapping $\{\boldsymbol{J},\boldsymbol{\theta}\} \Rightarrow \{\boldsymbol{x},\boldsymbol{v}\}$.

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 routine `evalAxisymStaeckel` in `actions_staeckel.h`, which operates 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 [6]. 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). 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 focal distance Δ; the accuracy (variation of estimated actions along the orbit) strongly depends on its value. Importantly, 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 `ActionFinderAxisymFudge` class precomputes a table of best-fit values of Δ as a function of E, L_z (this takes a couple of seconds) and uses interpolation to obtain a suitable value at any point, which is then fed into the standalone routine `evalAxisymFudge` that compute the actions, angles and/or frequencies.

This is the main workhorse for many higher-level tasks in the AGAMA library.

A variation of this approach is to pre-compute the actions J_r, J_z as functions of three integrals of motion (one of them being approximate) on a suitable grid, and then use a 3d interpolation to obtain the values of actions at any point. The construction of such interpolation table takes another couple of seconds, and the evaluation of actions through interpolation is $\sim 10\times$ faster than using the Stäckel approximation directly. However, the accuracy of this approach is somewhat worse (not because of interpolation, but due to the approximate nature of the third integral); nevertheless, it is still sufficient in many contexts. It is implemented in the same `ActionFinderAxisymFudge` class with the parameter `interpolate=true`.

More technical details are provided in Section A.6.1.

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 [9]. A single torus is constructed for a given triplet of actions $\boldsymbol{J}$, which takes some time and might not always succeed (depending on the properties of the potential and the required accuracy); the subsequent mapping for any values of angles $\boldsymbol{\theta}$ is relatively fast. The class `ActionMapperTorus` implements the torus construction "on-the-fly", with each unique triplet of actions producing a new torus, which is then cached and reused in subsequent calls that use the same $\boldsymbol{J}$ but different $\boldsymbol{\theta}$. The torus code is adapted from the original TORUSMAPPER 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 might 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 $M = (2\pi)^3 \iiint f(\boldsymbol{J})\, \mathrm{d}^3 J$, computed by numerical integration (the pre-factor comes from a trivial integration over angles) and returned by the `totalMass` method of the DF instance. 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); the choice of model is set by `type=...`, and model-specific parameters are described in the following sections.

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 2.6.3).

All built-in DF classes provide analytic derivatives $\partial f / \partial \boldsymbol{J}$.

2.5.1 Disky components

There are two classes of disk DFs in Agama: the first, `QuasiIsothermal`, expresses the DF in terms of auxiliary functions that are related to a particular potential, while the second, `Exponential`, is written in an entirely self-contained form. We describe them in turn.

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 [23] expressed in terms of actions [8] is provided by the `QuasiIsothermal` class, which represents the following DF:

$$
\begin{aligned}
f(\boldsymbol{J}) &= \frac{\Omega}{2\pi^2\,\kappa^2}\,\frac{\Sigma_0}{1-q_{J_\phi}}\,\exp_q\left(-\frac{R_\mathrm{c}}{R_\mathrm{disk}},\ -q_{J_\phi}\right) \\
&\times \frac{\kappa}{(1-q_{J_r})\,\tilde\sigma_r^2}\exp_q\left(-\frac{K(\boldsymbol{J})}{\tilde\sigma_r^2},\ -q_{J_r}\right) \times \frac{\nu}{(1-q_{J_z})\,\tilde\sigma_z^2}\exp_q\left(-\frac{\nu\,J_z}{\tilde\sigma_z^2},\ -q_{J_z}\right), \\
K(\boldsymbol{J}) &\equiv \begin{cases} \kappa\,J_r & \text{if } J_\phi \ge 0, \\ \kappa\,J_r - 2\Omega\,J_\phi & \text{if } J_\phi < 0, \end{cases} \\
R_\mathrm{c} &= R_\mathrm{c}(\hat J), \qquad \hat J \equiv \sqrt{\tilde J^2 + J_\mathrm{min}^2}, \quad \tilde J \equiv |J_\phi| + k_r J_r + k_z J_z, \\
\tilde\sigma_r^2(R_\mathrm{c}) &\equiv \sigma_{r,0}^2\exp(-2R_\mathrm{c}/R_{\sigma,r}) + \sigma_\mathrm{min}^2, \\
\tilde\sigma_z^2(R_\mathrm{c}) &\equiv \sigma_{z,0}^2\exp(-2R_\mathrm{c}/R_{\sigma,z}) + \sigma_\mathrm{min}^2 \quad \text{or} \quad \tilde\sigma_z^2(R_\mathrm{c}) \equiv 2\,h_\mathrm{disk}^2\,\nu^2(R_\mathrm{c}) + \sigma_\mathrm{min}^2, \\
\exp_q(x,\,q) &\equiv \begin{cases} \exp(x) & \text{if } q = 0, \\ (1+qx)^{1/q} & \text{if } -1 < q < 0. \end{cases}
\end{aligned}
$$

In these expressions, $R_\mathrm{c}(L_z)$ is the radius of a circular orbit as a function of the z-component of angular momentum, but evaluated at an argument $\hat J(\boldsymbol{J})$. The epicyclic frequencies κ, ν, Ω and the two quantities with a dimension of squared velocity $\tilde\sigma_r^2, \tilde\sigma_z^2$ are, in turn, evaluated at a radius R_c. The function $\exp_q(-x,-q)$ is a generalization of the usual exponent $\exp(-x)$, which is identical to it when $q = 0$, but falls off more slowly when

$0 < q < 1$. To construct such a DF, one needs to provide an instance of potential, which is used to initalize the function $R_c(L_z)$ and the epicyclic frequencies κ, ν, Ω as functions of radius. Other parameters listed in the `QuasiIsothermalParam` structure are:

- `Sigma0` is the overall normalization of the surface density profile Σ_0; alternatively, one may provide the total `mass`, from which the normalization will be computed automatically.

- `coefJr` [1], `coefJz` [0.25] are the dimensionless coefficients $k_r, k_z \sim \mathcal{O}(1)$ in the linear combination of the actions $\tilde{J} \equiv |J_\phi| + k_r J_r + k_z J_z$. This $\tilde{J}$ is then summed in quadrature with $J_{\rm min}$ to form $\hat{J}$, which is used as the argument of R_c instead of the angular momentum. The reason for this is that $\tilde{J}$ better corresponds to the average radius of a given orbit than J_ϕ alone: for instance, a star with $J_\phi \approx 0$ does not in reality stay close to origin if the other two actions are large.

- `Jmin` [0] is the lower limit $J_{\rm min}$ imposed on the argument of the circular radius function $R_c(\hat{J})$. It is introduced to prevent a pathological behaviour of DF in the case of a cuspy potential, when epicyclic frequencies tend to infinity as $R_c \to 0$.

- `Rdisk` sets the scale length of the disk $R_{\rm disk}$.

- `Hdisk` if provided, determines the scale height of the disk $h_{\rm disk}$ and the vertical velocity dispersion, which are related through the vertical epicyclic frequency ν.

- Alternatively, `sigmaz0` and `Rsigmaz` can be used instead of `Hdisk` to make the vertical velocity dispersion profile close to an exponential function of radius, with central value $\sigma_{z,0}$ and radial scale length $R_{\sigma,z}$; this choice has been used historically for quasi-isothermal DFs [8, 13], but it does not generally produce constant-scaleheight disk profiles.

- `sigmar0` and `Rsigmar` control the radial velocity dispersion profile, which is nearly exponental with central value $\sigma_{r,0}$ and radial scale $R_{\sigma,r}$. Typically $R_{\sigma,r}, R_{\sigma,z} \sim 2\,R_{\rm disk}$.

- `sigmamin` [0] is the minimum value of velocity dispersion $\sigma_{\rm min}$, added in quadrature to both $\tilde{\sigma}_r$ and $\tilde{\sigma}_z$; it is introduced to avoid the pathological situation when the velocity dispersion drop so rapidly with radius that the value of DF at $J_r = J_z = 0$ actually increases indefinitely at large J_ϕ. A reasonable lower limit could be $(0.02\,..\,0.05)\sigma_0$.

- `qJphi` [0] is responsible for the shape of the radial profile of the surface density: the default value of zero creates nearly-exponential profiles, whereas a positive value makes it less steeply declining with radius.

- `qJr` [0], `qJz` [0] play a similar role for the velocity distributions in: they should be approximately isothermal when these parameters are zero, and have fat tails when they are positive.

As stressed above, both epicyclic frequencies and $R_c(\hat{J})$ 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. If these two potentials are close enough, then this DF produces density profiles and velocity dispersions that approximately correspond to the exponential disks (at least when the $q_{...}$ coefficients are zero): the surface density $\Sigma(R)$ is close to exponentially declining with central value Σ_0 and radial scale length $R_{\rm disk}$, the vertical profile is roughly isothermal with scale height $h_{\rm disk}$, and the radial velocity dispersion is similar to $\tilde{\sigma}_r$, although the actual profiles are somewhat different from the tilded functions.

If the potential has a nearly flat rotation curve with circular velocity $v_\circ$, then $R_c(\tilde{J}) \approx \tilde{J}/v_\circ$, and epicyclic frequencies are $\propto v_\circ^2/\tilde{J}$. This motivates the introduction of another family of disk-like DFs, which has a similar functional form, but does not itself contain any reference to the potential, simplifying the construction of self-consistent models (Section 2.6.3). It is represented by the `Exponential` class:

$$f(\boldsymbol{J}) = \frac{M}{(2\pi)^3}\,\frac{J_{\rm d}}{J_{\phi,0}^2}\,\exp_q\!\left(-\frac{J_{\rm d}}{J_{\phi,0}},\ -q_{J_\phi}\right)$$
$$\times\,\frac{J_{\rm v}}{J_{r,0}^2}\exp_q\!\left(-\frac{J_{\rm v}\,K(\boldsymbol{J})}{J_{r,0}^2},\ -q_{J_r}\right)\times\frac{J_{\rm v}}{J_{z,0}^2}\exp_q\!\left(-\frac{J_{\rm v}\,J_z}{J_{z,0}^2},\ -q_{J_z}\right)$$
$$K(\boldsymbol{J}) \equiv \begin{cases} J_r & \text{if } J_\phi \geq 0, \\ J_r - J_\phi & \text{if } J_\phi < 0, \end{cases} \qquad J_{\rm d} \equiv \sqrt{\tilde{J}^2 + J_{\rm d,0}^2}, \quad J_{\rm v} \equiv \sqrt{\tilde{J}^2 + J_{\rm v,0}^2}$$

Its parameters listed in the `ExponentialParam` structure are:

- `norm` is the overall scaling with the dimension of mass (M); alternatively, one may provide the total `mass`, from which the normalization will be computed automatically.
- `Jphi0` determines the scale length of the disk: $J_{\phi,0} \sim R_{\rm disk} v_\circ$.
- `Jz0` controls the scale height and vertical velocity dispersion: $J_{z,0} \sim h_{\rm disk} v_\circ \sim R\,\sigma_z(R)$.
- `Jr0` plays a similar role for the radial velocity dispersion and the extent of radial excursion ΔR of a typical orbit: $J_{r,0} \sim \Delta R\, v_\circ \sim R\,\sigma_r(R)$.
- `coefJr` [1], `coefJz` [0.25] are the coefficients k_r, k_z in the linear combination of actions $\tilde{J} \equiv |J_\phi| + k_r J_r + k_z J_z$.
- `addJden` [0] and `addJvel` [0] are the lower limits $J_{\rm d,0}$ and $J_{\rm v,0}$ for the arguments of the q-exponential functions, which are introduced to tweak the behaviour of density and velocity dispersion profiles, correspondingly, in the limit of small actions. Their role is similar to $J_{\rm min}$ for the `QuasiIsothermal` model: since the rotation curves of realistic potentials cannot be flat all the way down to the center, the unmodified DF would produce too low density and too high velocity dispersions at small radii, which is compensated by these additional parameters. Their values should typically be of order $J_{\rm d,0} \sim J_{\rm r,0}$ and $J_{\rm v,0} \sim J_{\phi,0}$.

- `qJr`, `qJz`, `qJphi` play the same role (creating fat tails) as in the `QuasiIsothermal` model.

2.5.2 Spheroidal components

A suitable choice for DFs of elliptical galaxies, bulges or haloes is the `DoublePowerLaw` model, which is similar to the ones presented in [7, 52], with a different notation:

$$f(\boldsymbol{J}) = \frac{M}{(2\pi\,J_0)^3}\left[1+\left(\frac{J_0}{h(\boldsymbol{J})}\right)^{\eta}\right]^{\Gamma/\eta}\left[1+\left(\frac{g(\boldsymbol{J})}{J_0}\right)^{\eta}\right]^{-\mathrm{B}/\eta}$$
$$\times\left[1-\beta\left(\frac{J_{\rm core}}{h(\boldsymbol{J})}\right)+\left(\frac{J_{\rm core}}{h(\boldsymbol{J})}\right)^2\right]^{-\Gamma/2}\exp\left[-\left(\frac{g(\boldsymbol{J})}{J_{\rm cutoff}}\right)^{\zeta}\right]\left(1+\varkappa\tanh\frac{J_\phi}{J_{\phi,0}}\right),$$
$$g(\boldsymbol{J}) \equiv g_r J_r + g_z J_z\ + (3-g_r-g_z)\,|J_\phi|,$$
$$h(\boldsymbol{J}) \equiv h_r J_r + h_z J_z + (3-h_r-h_z)|J_\phi|.$$

The parameters of this model are listed in the structure `DoublePowerLawParams` and have the following meaning (default values in brackets):

- `norm` is the overall normalization M with the dimension of mass; the actual total mass differs from M by a numerical factor ranging from 1 to a few tens, which depends on other parameters of the model. Alternatively, one may provide the total `mass`, from which the normalization will be computed automatically.

- `J0` is the characteristic action J_0 corresponding to the break in the double-power-law profile; it sets the overall scale of the model.

- `slopeIn` is the power-law index Γ in the inner part of the model, below the characteristic action; must be < 3.

- `slopeOut` is the power-law index B in the outer part of the model; must be > 3.

- `steepness` [1] is the parameter η controlling the steepness of the transition between the two asymptotic regimes.

- `coefJrIn` [1], `coefJzIn` [1] are the coefficients h_r, h_z in the linear combination of actions, controlling the flattening and anisotropy in the inner part of the model; the third coefficient is implied to be $3-h_r-h_z$, and all three must be non-negative.

- `coefJrOut` [1], `coefJzOut` [1] are the similar coefficients g_r, g_z for the outer part of the model.

- `Jcore` [0] introduces an optional central core, forcing the DF to have a finite central value even when the inner slope Γ is nonzero [20]; in this case, the auxiliary coefficient β is assigned automatically from the condition that the total normalization of the DF remains the same (although this is only true when $J_{\rm core} \ll \min(J_0, J_{\rm cutoff})$ or when the coefficients $g_r = h_r, g_z = h_z$, otherwise it still changes somewhat).
- `Jcutoff` [0] additionally suppresses the DF at large actions (beyond $J_{\rm cut}$), 0 means disable.
- `cutoffStrength` [2] sets the steepness ζ of this exponential cutoff at large J.
- `rotFrac` [0] controls the amount of streaming motion by setting the odd-J_ϕ part of DF to be $\varkappa$ times the even-J_ϕ part; $\varkappa = 0$ disables rotation and $\varkappa = \pm 1$ correspond to models with maximum rotation.
- `Jphi0` [0] sets the extent $J_{\phi,0}$ the central core with suppressed rotation.

This DF roughly corresponds to the $\alpha\beta\gamma$ `Spheroid` model, with the asymptotic power-law indices $\mathrm{B} = 2\beta - 3$ and $\Gamma = (6-\gamma)/(4-\gamma)$ in the self-consistent case. An example program `example_doublepowerlaw.cpp` helps to find the values of these parameters that best correspond to the given spherical isotropic model.

2.5.3 Spherical DFs constructed from a density profile

The previously described types of DFs were defined in terms of an analytic functions of actions, and the density profiles that they generate in a particular potential (Section 2.6.1) are not available in a closed form. The alternative approach is to start from a given density profile and a potential, and determine a DF that generates this density profile, using some sort of inversion formula. So far this approach has been mostly used in spherical systems, with the most well-known case being the Eddington inversion formula for a spherical isotropic DF. We implement a more general version of this formula [21], which produces a DF with the following velocity anisotropy profile:

$$\beta(r) \equiv 1 - \frac{\sigma_t^2}{2\sigma_r^2} = \frac{\beta_0 + (r/r_a)^2}{1 + (r/r_a)^2}.$$

β_0 is the limiting value of anisotropy in the center, and if $r_a < \infty$, the anisotropy coefficient tends to 1 at large r (Osipkov–Merritt profile), otherwise stays equal to β_0 everywhere. The usual isotropic case is obtained by setting $\beta_0 = 0, r_a = \infty$.

The DF, expressed in terms of energy E and angular momentum L, has the following form:

$$f(E, L) = \hat{f}(Q)\; L^{-2\beta_0}, \quad Q \equiv E + L^2/(2r_a^2).$$

The function $\hat{f}(Q)$ is computed numerically for the given choice of parameters, density and potential, as explained in Section A.7.1), and represented in an interpolated form on a suitable grid in Q. For some combinations of parameters, this produces a DF which is negative in some range of Q, in particular, when the central slopes of the density profile $\gamma \equiv -d\ln\rho/d\ln r$, the potential $\delta \equiv -d\ln(-\Phi)/d\ln r$, and the coefficient β_0 violate the so-called slope–anisotropy theorem [2]: $\gamma \geq 2\beta + (1/2-\beta)\delta$. For the self-gravitating case, $\delta = 0$ if the potential is finite at origin, or $\delta = \beta - 2$ otherwise. If the computed DF is negative, it is replaced by zero.

This type of DF has the following parameters:

- `beta0` [0] is the central value of velocity anisotropy, must be in the range $-0.5 \leq \beta_0 < 1$.
- `r_a` [∞] is the anisotropy radius, must be positive (in particular, infinity means a constant-anisotropy model).

In addition, one needs to provide one-dimensional functions representing the radial profile of the density and the potential (if they are taken from the same model, only the latter is needed). Any spherically-symmetric instances of `Density` and `Potential` classes can be used.

The DF is traditionally expressed in terms of E, L, and in this form can only be used in the same potential Φ as it was constructed in. However, it can be put on equal grounds with other action-based DFs, which are manifestly independent of potential, using the following approach. An instance of `ActionFinderSpherical` class, constructed for the original potential Φ, is attached to the instance of a `QuasiSpherical` DF. To compute the value of DF at the given triplet of actions $\boldsymbol{J}$, we map them to E, L using this original spherical action finder, and then feed these values to the DF. Crucially, in this form the actions $\boldsymbol{J}(\boldsymbol{x}, \boldsymbol{v} \mid \tilde{\Phi})$ may be computed in any other potential $\tilde{\Phi}$ (not necessarily spherical), not just the original Φ. This corresponds to the DF being adiabatically transformed from the original potential to the new one, without changing its dependence on actions. This is especially convenient for iterative construction of multicomponent self-consistent models (Section 2.6.3): the DFs of spheroidal components (bulge, halo) may be obtained using the Eddington inversion formula or its anisotropic generalization in the initial spherically-symmetric approximation of the total potential, and then expressed as functions of actions $\boldsymbol{J}$. The resulting DF is then viewed as a function of actions only. In subsequent iterations, the potential is no longer spherical, but the density and anisotropy profiles of these components, obtained by integration of their DFs over velocity, are nevertheless quite close to the initial ones.

2.5.4 Spherical isotropic models

In a special case of spherical isotropic models, the DF has the form $f(E)$. There is an alternative formulation which makes it invariant with respect to the potential, retaining the convenience of action formalism without the need to compute actions explicitly. Namely, we

use the phase volume h instead of E as the argument of the DF. It is defined as the volume of phase space enclosed by the given energy hypersurface:

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

The advantages of using h instead of E are that the total mass of the model is simply $M = \int_0^\infty f(h)\,\mathrm{d}h$, 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 $\mathrm{d}h(E)/\mathrm{d}E \equiv g(E)$ is called the density of states ([11], eq. 4.56), and is given by

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

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 $\int_0^\infty f(h)\,\mathrm{d}h$ 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 `createSphericalIsotropicDF`. The other is to construct an approximating DF from an array of particles sampled from it, using the log-density estimation approach (Section A.2.5), provided by the routine `fitSphericalIsotropicDF`. More information on these models is given in section A.7.2.

These one-dimensional DFs may also be put on equal grounds with other action-based DFs, using a proxy class `QuasiSphericalIsotropic`. 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 2.5.1), 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. When the original DF is constructed using the Eddington inversion formula, it is easier to use the `QuasiSpherical` class from the previous section directly, avoiding the additional transformation $E \leftrightarrow h$.

2.6 Galaxy modelling framework

This module, residing in the 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).

Various routines described below work with instances of the `GalaxyModel` class, which is a simple aggregate of a potential, action finder, DF, and a fourth concept – selection function (SF). The SF is a function of the 6d position/velocity space $S(\boldsymbol{x},\boldsymbol{v})$ with values ranging from 0 to 1, which is multiplied by the value of the DF at the corresponding point in action space $\boldsymbol{J}$. By providing a non-trivial SF to these routines, one may limit the volume of phase space over which the integration is carried. A simple example of a SF is provided by the `SelectionFunctionSpatial` class: $S(\boldsymbol{x},\boldsymbol{v}) = \exp\big[-(|\boldsymbol{x}-\boldsymbol{x}_0|/R_0)^\xi\big]$, where $\boldsymbol{x}_0$ is a fixed point in the coordinate space, R_0 is the cutoff radius, and ξ is the cutoff steepness (0 implies no cutoff, ∞ – a sudden transition from $S=1$ inside the sphere of radius R_0 to $S=0$ outside this radius, and anything in between – to a more gradual decay of the SF).

2.6.1 Moments of distribution functions

Routines in this group make a distinction between the "intrinsic" coordinate system xyz of the model and the "observed" coordinate system XYZ, whose orientation with respect to xyz is parametrized by three Euler angles α,β,γ, or equivalently by an orthogonal rotation matrix R (Section A.3). The position $\boldsymbol{X}$, velocity $\boldsymbol{V}$ or other vectors in the observed system are related to $\boldsymbol{x}$, $\boldsymbol{v}$ in the intrinsic system by $\boldsymbol{X}=\mathsf{R}\,\boldsymbol{x}$, etc. When all three angles are zero, the two systems coincide; generally, β is the most useful angle, specifying the inclination. The Z axis in the observed coordinate systems is the line of sight, and some of the routines come in two variants – computing the intrinsic quantities at the given point $\{X,Y,Z\}$ or projected quantities at $\{X,Y\}$ integrated by Z. Naturally, adding another dimension of integration increases the cost by an order of magnitude.

When the DF consists of several components, and one needs the moments of each DF component separately, it is more efficient to compute them in a single operation, since the most expensive task – conversion from $\boldsymbol{x},\boldsymbol{v}$ to $\boldsymbol{J}$ – is shared between all components. All three routines described below have an additional argument `separate`, which provides this option if set to `true`.

DF moments (density, velocity dispersion, etc.) are defined as the DF integrated over the velocity, weighted by the SF and various combinations of velocity components:

$$\rho(\boldsymbol{X}) \equiv \iiint \mathrm{d}^3V\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big)\, S(\boldsymbol{x},\boldsymbol{v}),$$
$$\overline{\boldsymbol{V}} \equiv \rho^{-1} \iiint \mathrm{d}^3V\, \boldsymbol{V}\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big)\, S(\boldsymbol{x},\boldsymbol{v}),$$
$$\mathsf{V} \equiv \overline{V_iV_j} \equiv \rho^{-1} \iiint \mathrm{d}^3V\, V_iV_j\, f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big)\, S(\boldsymbol{x},\boldsymbol{v}).$$

The routine `computeMoments` calculates any combination of these quantities at the given point $\boldsymbol{X}$ by numerically integrating f over $\boldsymbol{v}$ and transforming the result to the observed

frame; 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 2.6.3). The first moment of velocity may have only one non-zero component ($\overline{v_\phi}$) in the intrinsic coordinate system, but possibly all three components of $\overline{\boldsymbol{V}}$ can be nonzero in the observed system. Likewise, the symmetric tensor of second velocity moments V can be diagonalized (and in axisymmetric systems one of its principal axes is aligned with the ϕ coordinate, while the other two lie in the meridional plane), but all six components of V are reported and can be non-zero for a general orientation of the observed system. The first and second velocity moments at a fixed 3d point in the two coordinate systems are related by the rotation matrix R: $\boldsymbol{V} = \mathsf{R}\,\boldsymbol{v}$, $\mathsf{V} = \mathsf{R}\,\mathsf{v}\,\mathsf{R}^T$.

A variant of this routine computes the projected moments (surface density Σ, mean sky-plane velocities $\overline{v_X}, \overline{v_Y}$, mean line-of-sight velocity $\overline{v_Z}$ and the full tensor of second moments) by integrating the corresponding intrinsic moments along the line of sight Z:

$$\Sigma(X,Y) \equiv \int_{Z=-\infty}^{\infty} \mathrm{d}Z\,\rho(X,Y,Z), \qquad \overline{V_Z} \equiv \frac{1}{\Sigma(X,Y)} \int_{Z=-\infty}^{\infty} \mathrm{d}Z\,\rho(X,Y,Z)\,V_Z, \quad \text{etc.}$$

These moments cannot be easily transformed to another rotated frame $X'Y'Z'$, since the direction of integration axis also changes.

Velocity distribution functions offer a more detailed description of model kinematics. The one-dimensional VDFs in each of the three orthogonal directions V_k ($k \in \{X,Y,Z\}$ are the axes of the observed coordinate system) are defined as the DF×SF integrated over the other two velocity axes, and optionally along Z for the projected variant:

$$\mathfrak{f}(\boldsymbol{X};\,V_1) \equiv \frac{1}{\rho(\boldsymbol{X})} \iint \mathrm{d}V_2\,\mathrm{d}V_3\,f\big(\boldsymbol{J}(\boldsymbol{x},\boldsymbol{v})\big)\,S(\boldsymbol{x},\boldsymbol{v}),$$
$$\mathfrak{f}_{\mathrm{proj}}(X,Y;\,V_k) \equiv \frac{1}{\Sigma(X,Y)} \int \mathrm{d}Z\,\rho(X,Y,Z)\,\mathfrak{f}(X,Y,Z;\,V_k).$$

VDFs in each dimension (intrinsic or projected) are represented as B-splines of degree N: $\mathfrak{f}^{(k)}(V_k) = \sum_{j=1}^{N_{\mathrm{basis}}} A_j^{(k)} B_j^{(k)}(V_k)$, where the basis functions $B_j^{(k)}$ are defined by the nodes of the grid in velocity space. The grid(s) may be provided by the user (separately for each of the three velocity axes) or constructed automatically to cover the entire available range of velocity from $-v_{\mathrm{esc}}$ to $v_{\mathrm{esc}} \equiv \sqrt{-2\Phi(\boldsymbol{x})}$ with a specified number of points, typically ~ 50. The VDFs are normalized so that $\int_{-v_{\mathrm{esc}}}^{v_{\mathrm{esc}}} \mathfrak{f}(V_k)\,dV_k = 1$, and the density ρ or the surface density Σ is computed as a by-product. To determine the coefficients of expansion $A_j^{(k)}$, we follow the finite-element approach by integrating the DF weighted with each basis function to obtain $f(\ldots,V_k)\,B_j^{(k)}(V_k)\,dV_k$, and solving the resulting linear system (Section A.2.1). 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 constructed by the routine `computeVelocityDistribution<N>`, $N = 0, 1, 3$ and returned as three sets of B-spline coefficients $A_j^{(k)}$, $k \in \{X, Y, Z\}$; in the recommended case $N = 3$ these can be converted into a conventional cubic spline interpolator (Section A.2.3).

Projected DF provides another mechanism for integrating the DF over velocity and along the line of sight, which is most useful when some of the velocity components in the observed coordinate system have known values $\boldsymbol{V}_{\rm obs}$ with some measurement uncertainties $\epsilon_{\boldsymbol{V}}$:

$$f_{\rm proj}(X, Y, \boldsymbol{V}_{\rm obs}, \epsilon_{\boldsymbol{V}}) \equiv \int_{Z=-\infty}^{\infty} \mathrm{d}Z \iiint \mathrm{d}^3V \prod_{k=1}^{3} \mathcal{E}(V_k;\, V_{{\rm obs},k}, \epsilon_{V,k})\; f\big(\boldsymbol{J}(\boldsymbol{x}, \boldsymbol{v})\big)\, S(\boldsymbol{x}, \boldsymbol{v}),$$

$$\text{where} \quad \mathcal{E}(V_k;\, V_{{\rm obs},k}, \epsilon_{V,k}) \equiv \begin{cases} \dfrac{1}{\sqrt{2\pi}\,\epsilon_{V,k}} \exp\left[-\dfrac{1}{2}\dfrac{(V_k - V_{{\rm obs},k})^2}{\epsilon_{V,k}^2}\right] & \text{when } \epsilon_{V,k} \text{ is finite,} \\ 1 & \text{when } \epsilon_{V,k} = \infty \end{cases}$$

In other words, for every velocity component that has a measured value $V_{{\rm obs},k}$ with a finite uncertainty $\epsilon_{V,k}$, the DF is convolved with a Gaussian of the appropriate width (or not convolved at all if the uncertainty is zero), while for an infinite uncertainty (in absence of any measurement), the DF is integrated over the entire available velocity range. The case of infinite uncertainty is essentially the limiting case of a large uncertainty $\epsilon_{V,k} \gg v_{\rm esc}$ (the argument of the exponent is nearly constant over the entire available velocity range), but the different normalization convention ensures that the result does not vanish. The routine `computeProjectedDF` is most useful for evaluating the likelihood of a model against a set of discrete kinematic tracers (e.g., stars with measured line-of-sight velocities and/or proper motions in any combination), regardless of whether the measurement uncertainties are negligible, small or large compared to the characteristic velocity range of the model. The only limitation is that the uncertainties of the two velocity components in the image plane ϵ_{V_X}, ϵ_{V_Y} should both be either finite or infinite (which is driven by computational convenience as well as practical considerations – the two components of the proper motion are simultaneously known or not).

The various quantities computed by these three routines satisfy the following relations, which should hold up to integration errors ($\lesssim 10^{-3}$):

- The surface density $\Sigma(X, Y)$ and the 3d density $\rho(\boldsymbol{X})$ are reported by projected and non-projected variants of `computeMoments` and `computeVelocityDistribution`, respectively. Of course, the latter case is much more expensive, and the density comes out as a by-product of normalizing the VDFs. Moreover, `computeProjectedDF` with all three velocity uncertainties $\epsilon_{V,k}$ set to infinity is also equivalent to $\Sigma(X, Y)$, with a

computational cost comparable to `computeMoments`, but using different internal scaling transformations of the integrand (whereas moments and VDF use velocity coordinates aligned with the intrinsic system of the model, projected DF operates with the observed velocity components and may be less accurate if the velocity ellipsoid of the model is very anisotropic and misaligned with the observed coordinates).

- Mean velocity $\overline{\boldsymbol{V}}$ and diagonal elements of the tensor of second velocity moments V can be obtained by integrating the VDF multiplied by the corresponding power of velocity, e.g., $\overline{V_k^2} = \int_{-v_{\rm esc}}^{v_{\rm esc}} \mathfrak{f}(V_k)\, V_k^2\, dV_k$, both in the projected and non-projected variants. Non-diagonal elements of V cannot be recovered from the VDF (which also implies that VDFs computed in different orientations of the observed coordinate systems cannot be easily transformed into each other).

- The value of the interpolated projected VDF $\mathfrak{f}^{(k)}_{\rm proj}(X, Y;\, V_{{\rm obs},k})$ multiplied by the surface density $\Sigma(X, Y)$ is equivalent to the projected DF computed for the given value of the k-th velocity component $V_{{\rm obs},k}$ (with zero uncertainty $\epsilon_{V,k}$) and infinite uncertaintes for the remaining two components. Constructing the entire VDF is significantly more expensive than computing the projected DF for a single value of velocity, but is more efficient than repeated calls to `computeProjectedDF` if one needs to evaluate the VDF many times for different values of $V_{{\rm obs},k}$. The spline-interpolated VDF can also be cheaply and accurately convolved with a Gaussian function, producing an equivalent result to a projected DF with a finite uncertainty $\epsilon_{V,k}$.

There are analogous routines for spherical isotropic DFs of the form $f(h)$, defined in `galaxymodel_spherical.h` and used by the `mkspherical` tool (Section 4).

2.6.2 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 `sampleActions`.
- Draw samples directly from the six-dimensional $\{\boldsymbol{x}, \boldsymbol{v}\}$ space, evaluating the product $f\big(\boldsymbol{J}(\boldsymbol{x}, \boldsymbol{v})\big)\, S(\boldsymbol{x}, \boldsymbol{v})$ with the help of an action finder. This is performed by the routine `samplePosVel`.

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. It also has a

separate, more efficient implementation for spherical isotropic DFs $f(h)$ in spherical potentials $\Phi(r)$ (without a SF); it is used by the `mkspherical` tool (Section 4).

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 `sampleDensity` (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).

2.6.3 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 $\rho(\boldsymbol{x}) = \int \mathrm{d}^3v\, f\big(\boldsymbol{J}(\boldsymbol{x}, \boldsymbol{v}\ |\Phi)\big)$ 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 [7, 48]. 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 such as a central supermassive black hole (SMBH), 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.

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.

There are several important things to keep in mind when adapting these examples to other systems:

- Each component (whether DF-based or a static density profile) needs to be assigned to either spheroidal or disk-like subsets. The former include profiles that are not strongly concentrated towards the equatorial plane, can have a central density cusp or an extended envelope, and will be represented by a `Multipole` expansion. The latter may be strongly flattened, but need to have a finite central density and a finite extent, or at least sharply declining density at large radii, and will be represented by a `CylSpline` 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, then extended to the entire space with the help of a corresponding density interpolator (`DensitySphericalHarmonic` – Section A.4.1, or `DensityAzimuthalHarmonic` – Section A.4.2), and the latter is then used in solving the Poisson equation. In addition, there may be static components with already known potentials (e.g., a Plummer potential with a very small scale radius representing the central SMBH), which will be added directly to the total potential.

- The parameters of the grids used to construct the intermediate density interpolators and the final potential expansions should be selected with care (there are no automatically assigned parameters here!). The computational cost is mainly determined by the resolution of the density grid (which is typically different for each component). For spheroidal components, these include the inner/outer boundaries and the number of points in a logarithmic grid in radius, plus the order of the angular expansion $l_{\max}$. For disky components, one needs to specify the minimum grid segments and the overall extent in R and z directions, as well as the number of nodes (spaced linearly at small radii, gradially transitioning towards logarithmic spacing). In particular, the innermost segments should be comparable (perhaps twice smaller) than the relevant spatial scale (e.g. the size of the density core, if present, or the vertical scale height of the disk), but not much smaller, whereas the outer radius should enclose almost all of the total mass of each component (say, up to ~ 10 scale radii). The grids for the global `Multipole` and `CylSpline` potential solvers are specified separately, and should typically have a larger spatial extent and somewhat denser spacing. Failure to assign suitable values for grid parameters may result in obscure errors during model iterations (e.g., "non-monotonic potentials" and similar numerical artifacts).

- The final model will always end up close to equilibrium, given enough iterations, but it may end up looking rather different from the initial guess for the potential, if the parameters of the potential and the DF are not in agreement. For disk components, the scale length, scale height, and central surface density should be synchronized between the `Disk` density profile and the `QuasiIsothermal` DF; the velocity dispersion parameters have no counterpart in the density profile, but if the dispersion is too high, the density generated by the DF in the central parts may be reduced compared to the initial density profile. For spheroidal components, the easiest way to ensure agreement is to use `QuasiSpherical` DF constructed from the given density profile; alternatively, when using `DoublePowerLaw` DFs, one may need to adjust the initial density profile retrospectively after examining the final one. This synchronization is only relevant when there are `QuasiIsothermal` DF components, which need a potential for initialization, and if this potential is very different from the final self-consistent one, the properties of the model may change in rather unpredictable ways.

Despite these caveats, the approach is fairly general and powerful (though restricted to axisymmetric models, due to the lack of more general action finders), and the resulting DF-based models can be converted to N-body models with arbitrarily large number of particles, or used to compute observable quantities (moments, velocity distributions, etc.) with great accuracy. A typical disk-bulge-halo model takes a few CPU minutes per iteration for reasonable choices of grid parameters, and is efficiently parallelized over multiple CPU cores.

2.6.4 Schwarzschild orbit-superposition modelling

Schwarzschild modelling approach [60] is an alternative method for creating self-consistent models, in which the DF is determined numerically as a weighted superposition of individual building blocks. In the original approach, these blocks are numerically computed orbits (hence the alternative name "orbit-superposition method"), which are essentially δ-functions in the space of integrals of motion, but a more general definition might use finite-size building blocks, or bunches of individual orbits. In what follows, we use the original formulation.

There are several ingredients in these models:

- The total gravitational potential $\Phi(\boldsymbol{x})$ used to integrate the orbits.
- One or more `Target` objects, which define various kinds of constraints: 3d density distribution, intrinsic (3d) or projected (line-of-sight) kinematics, etc. The required values of these constraints are denoted as $U_n^{(t)}$, with the index $t = 1..N_{\rm tar}$ enumerating `Target`s, and $n = 1..N_{{\rm cons},t}$ – constraints of each target.
- The orbit library (collection of $N_{\rm orb}$ orbits and their weights w_i in the model), and associated arrays $u_{i,n}^{(t)}$ of contributions of i-th orbit to n-th constraint of t-th target.

The modelling workflow is split into several steps:

1. Initialize the potential Φ and $N_{\rm tar}$ `Target` objects.
2. Create initial conditions (IC) for the orbit library.
3. Integrate orbits while recording the arrays $u_{i,n}^{(t)}$.
4. Determine the orbit weights that satisfy the constraints.
5. (Optional) create an N-body representation of the model.

The potential for Schwarzschild models needs to be specified in advance, in contrast to the DF-based iterative self-consistent models (Section 2.6.3). Often it will be computed from the deprojected surface density profile, e.g., parametrized by a Multi-Gaussian Expansion (MGE); there are several `Python` routines for manipulating these parametrizations.

Target objects provide an abstract interface for discretizing both the data and the model into an array of constraints. There are 3 types (5 variants) of 3d density discretization schemes, described in the Appendix A.8, one target (4 variants) for representing the spherically averaged 3d kinematic profiles, and one target (4 variants) for recording the line-of-sight velocity distributions (LOSVD). All these schemes use B-splines (Section A.2.2) for defining the discretization elements, and variants of the same scheme differ by the order of the B-spline basis. A `Target` object does not contain any data itself, it only provides the methods for computing discretized representations of various other entities and storing them in external arrays. For instance, a density target acting on a `Density` model produces the array of masses associated with each discretization element (in the simplest case, the mass contained in each cell of the 3d density grid), while a LOSVD target acting on a `Density` model computes the integrals of the PSF-convolved surface density profile over each spatial region (aperture) on the sky plane. The same LOSVD target applied to a `GalaxyModel` object computes the PSF-convolved LOSVDs produced by the combination of the DF and the potential in each aperture. Any target applied to an N-body snapshot computes the relevant quantities $U_n^{(t)}$ from the array of particles, weighted by their masses. And finally, any target can be attached to the orbit integrator to construct the discretized representaion of the i-th orbit $u_{i,n}^{(t)}$ (this is conceptually similar to recording the orbit as a collection of points sampled from the trajectory, with weights proportional to the time intervals between adjacent points, and then applying the target to this N-body snapshot, although in practice it is implemented on-the-fly, without actually storing the trajectory). The LOSVD target is used to constrain the model by observed kinematics, but this involves an additional step to convert the internal B-spline representation of the datacube into observable quantities (for details, see Appendix A.8).

The IC for the orbit library may be generated by one of the complementary approaches for constructing dynamical models. This is achieved by first sampling positions from the actual 3d density profile of the galaxy or one of its components, then assigning velocities drawn from a suitable DF or from a Jeans model. For spheroidal systems or galaxy components, the Eddington inversion or its anisotropic generalization (Section 2.5.3) provide a suitable DF, while for strongly flattened and rotating disk components (including bars), velocities may be

drawn from a Gaussian distribution with the dispersions computed from the axisymmetric anisotropic Jeans equations. In either case, the resulting IC are not necessarily in equilibrium, but merely provide a convenient starting point for the orbit-based modelling. Moreover, one may stack together several sets of IC created with different parameters (e.g., to provide a denser sampling of orbits at high binding energies near the galactic center).

The orbit weights $w_i \geq 0$ that satisfy the constraints are determined from the system of linear equations

$$\sum_{i=1}^{N_{\rm orb}} w_i \, u_{i,n}^{(t)} = U_n^{(t)}, \quad t = 1..N_{\rm tar}, \; n = 1..N_{{\rm cons},t}.$$

In practice, the constraints may not be satisfied exactly, especially if they come from noisy observations. In this case, the best solution is obtained by minimizing the L_2-norm of the residual in the above equation system, weighted by the observational uncertainties $\epsilon_{U_n^{(t)}}$. Additionally, one may employ some sort of regularization to make the solution more well-behaved. In practice, this is achieved by adding a regularization term proportional to the sum of squared orbit weights to the objective function to be minimized, which encourages a more uniform distribution of orbit weights in the solution. The objective function is thus written as

$$\mathcal{Q} \equiv \sum_{t=1}^{N_{\rm tar}} \sum_{n=1}^{N_{{\rm cons},t}} \left(\frac{\sum_{i=1}^{N_{\rm orb}} w_i \, u_{i,n}^{(t)} - U_n^{(t)}}{\epsilon_{U_n^{(t)}}} \right)^2 + \lambda \sum_{i=1}^{N_{\rm orb}} \left(\frac{w_i}{w_{i,0}} \right)^2 .$$

Here λ is the regularization coefficient, $w_{i,0}$ are the priors on orbit weights (in the simplest case, uniform). If some constraints need to be satisfied exactly, the corresponding uncertainties should be set to zero; these constraints will not contribute to $\mathcal{Q}$. The minimization of the objective function, subject to the non-negativity constraints $w_i \geq 0$, is performed by the quadratic optimization solver CVXOPT.

In the case of constructing models constrained by observed LOSVD kinematics, one may use the same orbit library multiple times to represent systems with different mass normalizations Υ, rescaling the velocities by $\sqrt{\Upsilon}$ as described in the Appendix A.8).

Finally, the orbit-superposition model can be converted into an N-body representation, e.g., to test its stability or provide initial conditions for a simulation. In order to do this, one needs to record the trajectories during orbit integration along with the other arrays $u_{i,n}^{(t)}$, and then select a certain number of points from each orbit's trajectory in proportion to its weight. In the case that the number of points recorded during orbit integration was insufficient to represent an orbit with a particularly high weight, this orbit needs to be re-integrated while storing the points more frequently. There is a `Python` routine `sampleOrbitLibrary` that performs this task in a transparent way.

All computationally heavy operations of this approach are implemented in the `C++` library, but the top-level modelling workflow is more conveniently expressed in `Python`.

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, penalized spline fitting and density estimate, linear/quadratic optimization solvers.
- 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 and Schwarzschild orbit-superposition modelling.

The shared library `agama.so` can be used directly as a `Python` extension module[8]. Both `Python` 2.6–2.7 and `Python` 3.x are supported, but the library must be compiled separately for either version (in case of version mismatch, you get an obscure error like "`dynamic module does not define init function`"). In addition, there are a few auxiliary routines written in `Python` (they reside in `py/pygama.py`). To simplify the usage, the package initialization (`__init__.py`) imports everything from both `agama.so` and `py/pygama.py` into a single namespace, which is what you get by writing `import agama` in your `Python` script[9].

The `Density`, `Potential`, `DistributionFunction`, `SelectionFunction`, `Target` 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 (including those requiring a more complicated setup in the `C++` code). Additionally, density, potential, DF and SF objects may also be represented by arbitrary user-defined Python functions – 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

[8] for the most common implementation of `Python` interpreter, named `CPython` – not to be confused with `Cython`, which is a distinct compiled language. The `Python` interface layer in AGAMA relies solely on the `Python` C API, and does not use any third-party libraries such as `swig` or `boost::python`.

[9] note that if the file `agama.so` is found in the current directory, it will have a priority over the `Python` modules stored in `site-packages` or other standard locations specified by `PYTHONPATH`. Thus in this case `import agama` will only load the `C++` extension module but not the auxiliary `Python` routines, possibly leading to obscure complaints about missing attributes of the module.

again leads to a more concise code with nearly the same efficiency as a pure C++ implementation. Moreover, operations on such arrays are internally OpenMP-parallelized in the Python extension module, except if they include callbacks to user-defined Python functions[10]. One may adjust the number of OpenMP threads by calling `agama.setNumThreads(N)`, either for the entire script, or in a context manager to temporarily change it – most usefully, to avoid unnecessary thread-switching overheads when using user-defined callback functions:

```
with agama.setNumThreads(1):  do_something_with_user_callbacks()
```

Below follows a brief overview of the classes and routines provided by the Python interface. As usual, writing `help(agama.Whatever)` brings up a complete description of the class or routine and its arguments. Moreover, there are several test and example programs demonstrating various aspects of usage of the Python interface to AGAMA; some of them have exact C++ equivalents.

Density class instances can be constructed with the following syntax:

```
d = agama.Density(type="...", mass=..., scaleRadius=..., otherParameters=...)
```

using the parameters listed in Section 2.2.4; argument names are case-insensitive, as are the names of density models. Note that the list of parameters relevant for a specific model depends on the `type` argument, but no error will be issued if other parameters are provided (they will simply be ignored). Alternatively, to combine several density objects `d1`, `d2`, etc., one may list them as the unnamed arguments of the constructor (these could be proper Density objects, dictionaries with density parameters, or user-defined functions providing the Density interface, see below):

```
comp = agama.Density(d1, dict(type="Plummer", mass=42), d2)
```

More idiomatically, one can "add" two or more `Density` or `Potential` instances to create a composite object:

```
comp2 = comp + d3 + d4   # will end up having 5 components
```

Elements of such composite density objects can be accessed by index or iterated over:

```
for i in range(len(comp)):  print(comp[i])
for d in comp:  print(d)
```

Note that an elementary density or potential has zero length and thus cannot be indexed, but a density/potential wrapped into a modifier is represented as a length-one object, and its 0th element gives access to the underlying non-modified object (modifiers can be chained as well). Modifiers can be attached at the time of creation of the density or potential object, or later (in this case, the original object remains unmodified); the parameters of time-dependent modifiers can be provided in the form of arrays, with one line per timestamp (in the example below, creating a density moving on a straight line with velocity v_x, v_y, v_z):

```
shifted_d = agama.Density(type="Plummer", center=[1,2,3])
moving_d = agama.Density(d, center=[[0,0,0,0],[1,vx,vy,vz]])
```

[10] this restriction is due to the global interpreter lock (GIL) mechanism in CPython, precluding its simultaneous access from multiple threads.

```
print(moving_d[0] == d)  # True
```

Another possibility is to construct a spherically-symmetric density model from a cumulative mass profile – a 2d array with two columns: r, $M(<r)$. The following example corresponds to a $\gamma = 1$ Dehnen (aka Hernquist) profile:

```
r = numpy.logspace(-3,3)
M = (r / (r+1))**2
d = agama.Density(cumulMass=numpy.column_stack((r, M)) )
```

In this case the `Density` object is internally represented by a `DensitySphericalHarmonic` `C++` class. Both this class and `DensityAzimuthalHarmonic` are also utilized in the iterative self-consistent modelling framework (Section 2.6.3); such density expansions can be written out to a text file and loaded back by

```
d.export("hernquist_model.ini")
d = agama.Density("hernquist_model.ini")
```

The `Density` class provides only a couple of methods: first of all, the computation of the density itself – the argument is a single point or a $N \times 3$ array of N points (the `Python` interface always deals with cartesian coordinates to avoid confusion):

```
print(d.density(1,0,0))
print(d.density([[1,0,0],[2,3,4]]))
```

To compute a time-dependent density at an arbitrary moment of time, provide an extra `t=...` argument to the `density` method (note that all other methods still evaluate it only at time 0). `t` could be a single number or an array of the same length as points, in which case each the density at each point is evaluated at its own time.

`projectedDensity(points [, alpha, beta, gamma])` computes the surface density (integral of density along the line of sight) at a 2d point (or an array of points) in the image plane, using an arbitrarily rotated coordinate system, whose orientation is specified by Euler angles α, β, γ (Section A.3). The angles may be the same for all points or provided individually for each point.

Another useful method is `totalMass`, with an obvious role; the mass may well be infinite, e.g., for `NFW` or `Logarithmic` models. Likewise, `enclosedMass(radius)` estimates the mass enclosed within the given `radius` (a single value or an array of radii).

The `principalAxes` method determines the shape and orientation of the density profile.

Finally, the density profile may be sampled with particles, producing two arrays – coordinates ($N \times 3$) and masses (N):

```
pos, mass = d.sample(10000)
```

The density framework can be augmented with user-defined `Python` functions that return the density at a given point (or, rather, an array of points). Such a function must be a free function (not a class method) or a lambda expression, accepting one argument, which should be treated as a $N \times 3$ array of points (even when $N = 1$). This function would typically be

called from the C++ code with more than one input point, to reduce overhead from switching between C++ and Python. The following two equivalent examples define a spherical Plummer model:

```
fnc1 = lambda x: 3/(4*numpy.pi) * (1 + numpy.sum(x**2, axis=1))**-2.5
def fnc2(x): return 3/(4*numpy.pi) * (1 + numpy.sum(x**2, axis=1))**-2.5
```

One may create an instance of Density class wrapping a user-defined Python callback function. For simple operations such as evaluation of 3d and projected density, it is sufficient to pass the user-defined function as a single unnamed argument to the constructor of Density class; however, for other operations (computation of total mass, creation of density or potential expansions, etc.), the symmetry of the model needs to be specified explicitly (there is no default value!), because the cost of these operations is significantly reduced in typical cases of at least triaxial symmetry:

```
print(agama.Density(fnc1).projectedDensity(1,2))
print(agama.Density(fnc2, symmetry="s").totalMass())
```

Alternatively, one may directly pass the Python callback function, or even a tuple consisting of several Density objects, dictionaries with their parameters, or user-defined callback functions, in all contexts where a Density object is expected, such as the Potential and DistributionFunction constructors, without creating a wrapper instance of Density class:

```
pot1 = agama.Potential(type="Multipole", density=(fnc1, d), symmetry="s")
```

In case that the density is expensive but needs to be evaluated many times (e.g., to compute surface density), it may be approximated with either DensitySphericalHarmonic or DensityAzimuthalHarmonic interpolators (their applicability is similar to Multipole or CylSpline potentials, respectively):

```
dsh = agama.Density(type="DensitySphericalHarmonic", density=fnc1,
    symmetry="s", lmax=0, gridSizeR=25, rmin=0.01, rmax=100)
```

This is most useful to represent a density generated by a DF – the same idea is used in the iterative self-consistent modelling framework (Section 2.6.3), and a pure-Python reimplementation of this workflow is illustrated in one of the test programs.

Remember that the callback function must return finite values for any input point used by the code – pay special attention to possible indeterminacies, e.g., at $R = 0$! (If the density is singular at origin, it may still be used in the Multipole potential or DensitySphericalHarmonic density, but not in CylSpline / DensityAzimuthalHarmonic).

Potential class is inherited from Density and provides access to the entire hierarchy of C++ potentials. It can be constructed in a variety of ways:

```
p = agama.Potential(type="...", mass=..., otherParameters=...)
```

i.e., in the same way as a Density object, using named arguments listed in Section 2.2.4;

```
p = agama.Potential("MWPotential2014.ini")
```

reads the potential parameters from an INI file, where each section [Potential1], [Potential disk], etc., corresponds to a single component (see below for special rules about the con-

struction of composite potentials). Such an INI file, or any of its sections, may contain potential expansion coefficients previously stored by the `export` method.

One may also introduce a custom potential model by defining a function returning the value of the potential at the provided 2d array of points (similarly to a user-defined density model):
```
def fncPlummerPot(x):  return -(1 + numpy.sum(x**2, axis=1))**-0.5
```
Again, this function may be wrapped into an instance of `Potential` class, thus providing a variety of other methods such as `force` and `density` (which are evaluated via finite differences), or used in orbit integration. However, this is rather inefficient, and for most practical purposes, it is advisable to create a `Multipole` or `CylSpline` potential approximation for the custom potential (see example `p5` below or a more elaborate illustration).

If `type="Multipole"` or `"CylSpline"`, then a potential expansion is constructed from the given density or potential profile in one of the following mutually exclusive ways. First, one may specify the input density for the potential expansion, either as the name of a built-in density model (e.g., `density="Spheroid"`), or an instance of a `Density` class, or a user-defined function providing the Density interface as described above (in this case the symmetry of the input model needs to be specified explicitly[11]), or a tuple of several such instances. Second, one may provide the input potential rather than density; this again can be either an instance of a `Potential` class (this may be useful e.g. for creating sphericalized or axisymmetrized versions of an existing potential), or another user-defined function that returns the potential at a given 2d array of points (with the same calling convention as a density function; in this case one should explicitly specify its symmetry)[12]. Third, this could

[11]When constructing a `CylSpline` potential from a non-axisymmetric density profile, the latter is first used to create an internal `DensityAzimuthalHarmonic` expansion, which is used in the subsequent solution of the Poisson equation for each m term (integrating the density over R, z as described in Section A.4.2). On the other hand, if the input density is axisymmetric, it is used directly. For built-in density models, this is usually more efficient, avoiding the creation of the intermediate interpolator, but for a user-defined Python function the outcome is opposite, since evaluating the density directly is much more costly than evaluating the intermediate interpolator. The number of calls to the user-defined density function is much lower when constructing the interpolator that during the solution of the Poisson equation, so it is advisable to trick the code into creating the interpolator, by specifying `symmetry="t"`, or even better, by manually constructing a `DensityAzimuthalHarmonic` approximation of the Python function and providing it as input to the `CylSpline` potential.

[12]The relation between `lmax`, `mmax` and `symmetry` may be explained as follows. When the input density or potential is one of built-in models (even complex multicomponent ones with modifiers), its symmetry is known to the code and any provided value is ignored. However, when the input is an N-body snapshot or a user-defined model, the symmetry should be assigned explicitly. In the example `p3`, the input density is a sum of two models, the user-defined function `fnc1` and another `Multipole` potential `p1`, and `symmetry="s"` is applied only to the user-defined density function, while the built-in model automatically reports its axisymmetry, also making the new `Multipole` potential axisymmetric with a default order of angular expansion. Now, if `lmax=0` for a `Multipole` or a `BasisSet` potential, it will be necessarily spherically symmetric, but in order to create this sphericalized potential, we still need to know the symmetry of the input model, because it significantly affects the cost of construction (e.g., axisymmetric input profiles will only be queried in the $x-z$ plane, and triaxial – only in one quadrant of the 3d space). Likewise, setting `mmax=0` for any of the

be an N-body snapshot provided as a tuple of two arrays (coordinates and masses), or a file containing such snapshot:

```
p1 = agama.Potential(type="Multipole", density="Plummer", axisRatioZ=0.5)
p2 = agama.Potential(type="Multipole", density=agama.Density(type="Plummer"))
p3 = agama.Potential(type="Multipole", density=(fnc1, p1), symmetry="s")
p4 = agama.Potential(type="Multipole", potential=agama.Potential(type="Plummer"))
p5 = agama.Potential(type="Multipole", potential=fncPlummerPot, symmetry="s",
    gridSizeR=15, rmin=0.01, rmax=100)  # may explicitly specify the grid parameters
p6 = agama.Potential(type="Multipole", particles=(pos,mass), symmetry="t")
p7 = agama.Potential(type="Multipole", file="nbody_snapshot.txt", symmetry="n")
```

The construction of potential expansions (especially `CylSpline`) from user-defined `Python` density models may be quite expensive. In order to save computational effort by reducing the number of points at which the density function is evaluated, one may restrict the order of internal harmonic expansion to the minimum necessitated by the output order (parameter `fixOrder=true`, see its description in Section 2.2.4), and/or create an intermediate density interpolator (`DensitySphericalHarmonic` or `DensityAzimuthalHarmonic`).

A composite potential can also be created from several other `Potential` objects:

```
p8 = agama.Potential(p1, p2, p3)
```

Like a composite `Density`, elements of such composite potential can be accessed by index or iterated over. Another way of creating a composite potential is to provide a list of `dict` instances containing the parameters for each potential to the constructor:

```
disk_par = dict(type="Disk", mass=5, scaleRadius=3, scaleHeight=0.4)
bulge_par= dict(type="Sersic", mass=1, scaleRadius=1, axisRatioZ=0.6)
halo_par = dict(type="Spheroid", densityNorm=0.01, scaleRadius=20,
    axisRatioZ=0.7, gamma=1, beta=3, alpha=1)  # oblate NFW profile
potgal   = agama.Potential(disk_par, bulge_par, halo_par)
```

This is equivalent to providing all these parameters in separate sections of an INI file and then constructing the potential from this file.

If we examine the potential created in the last line,

```
print(potgal)  # CompositePotential{ DiskAnsatz, Multipole }
```

it becomes apparent that some rearrangement took place behind the stage. Indeed, in the case when the potential is constructed from several sets of parameters (but *not* from several existing potential instances), the code attempts to optimize the efficiency by using the GalPot approach. In this example, the `Disk` density profile was split into two parts – the `DiskAnsatz` potential class and the residual density profile; other spheroidal density compo-

potential or density expansions makes them axisymmetric, but the input models will be queried in different ways depending on their reported symmetry, significantly reducing the cost of construction in most common cases of partial (e.g., triaxial) symmetry of inputs. In short: `symmetry` describes the property of input model when it is not known to the code, while `lmax` and `mmax` specify the properties of the resulting expansion.

nents (`Sersic` and `Spheroid`) were combined with this residual profile, and used to initialize a single instance of `Multipole` potential. This is advantageous if one needs to evaluate the potential many times (e.g., in action computation), but makes it difficult to examine the contribution of each mass component separately. In order to do so, we may instead create another potential used only for visualization:

```
potvis = agama.Potential(agama.Potential(disk_par),
    agama.Potential(bulge_par), agama.Potential(halo_par))
```

An instance of `Potential` class provides the same methods as the `Density` class (and may be used in all places where a density instance is needed), plus the following ones:

`potential(points [, t=t0])` evaluates the potential at one or several input points (which should be either 3 numbers, or a 1d list/array of length 3, or a 2d array of size $N \times 3$) at time `t0` (0 by default; may be a single number or an array of the same length as points).

`force(points [, t=t0])` computes the acceleration (force per unit mass) at time `t0`, i.e., *minus* the derivative of potential, returning 3 numbers or an $N \times 3$ array.

`forceDeriv(points [, t=t0])` computes the forces and force derivatives ($-\partial\Phi/\partial x_i\,\partial x_j$ in the following order: xx, yy, zz, xy, yz, zx) at time `t0`:

```
r = numpy.linspace(0,20)
points = numpy.column_stack((r, r*0, r*0))   # a N × 3 array
force,deriv = potgal.forceDeriv(points)   # ⇒ N × 3 and N × 6 arrays
kappa = numpy.sqrt(-deriv[:,0] - 3*force[:,0]/r)   # radial epicyclic frequency κ
nu    = numpy.sqrt(-deriv[:,2])   # vertical epicyclic frequency ν
```

Plotting the rotation curve of the above constructed potential and all its components:

```
plt.plot(r, numpy.sqrt(-r*potvis.force(points)[:,0]))
for pot in potvis:  plt.plot(r, numpy.sqrt(-r*pot.force(points)[:,0]))
```

`projectedEval(points [, pot, acc, der, alpha, beta, gamma])` performs the same task as `projectedDensity` (integration along the line of sight), but for the potential, acceleration and its derivatives, as requested by the boolean arguments `pot`, `acc`, `der`. The input arguments are the 2d point (or an array of points) X, Y in the image plane, whose orientation with respect to the intrinsic model coordinates is specified by Euler angles α, β, γ (Section A.3), and the output consists of one, two or three arrays, depending on the requested quantities to be computed. If `pot=True`, the first array contains the integral(s) $\int_{-\infty}^{\infty} dZ\ [\Phi(X,Y,Z) - \Phi(0,0,Z)]$; the second term is introduced to compensate the logarithmic divergence of the integral at large $|Z|$. If `acc=True`, the next array contains the integrals of the X, Y components of acceleration (the Z component integrates out to zero). Finally, if `der=True`, the last array contains the integrals of $-\partial^2\Phi/\partial X^2$, $-\partial^2\Phi/\partial Y^2$, $\partial^2\Phi/\partial X\partial Y$; the remaining components are also identically zero and are not reported.

`Tcirc(E)` computes the characteristic time (period of a circular orbit with the given energy); for convenience it may also be called with an $N \times 6$ input array of position/velocity coordi-

nates: `Tcirc(points)`.

`Rcirc(E=...)` or `Rcirc(L=...)` return the radius of a circular orbit in the equatorial plane corresponding to the given energy or z-component of the angular momentum.

`Rmax(E)` returns the maximum radius accessible with the given energy, i.e., the root of $\Phi(R_{\max}) = E$.

`Rperiapo(E, L)` returns the pericenter and apocenter radii for an orbit with the given energy and angular momentum; for convenience, it may also be called with an $N \times 6$ input array of position/velocity coordinates, and of course, with a $N \times 2$ array of E, L values for multiple input points (all methods can accept vectorized input). This and the previous three functions operate with orbits in the equatorial plane and give exact results for spherical or axisymmetric potentials; otherwise the result refers to the axisymmetrized version of the potential and is therefore approximate.

`export(filename)` stores the potential into an INI file, which can be used later to read it back (by passing the filename as the parameter to the constructor). This operation only makes sense for the potential expansions (`Multipole` or `CylSpline`) without any modifiers; other potentials just store their name but nothing else (and hence are unusable or even incorrectly loaded back!) – as explained above, due to the rearrangements of constituent parts during the construction of a multicomponent potential, it is not always possible to reconstruct the original parameters from the existing objects. Note that all components of a composite potential are stored in a single file, each one in its own section `[Potential]`, `[Potential1]`, etc.

ActionFinder is the `Python` class constructed for the given potential:
`af = agama.ActionFinder(pot, [interp=True|False])`
where the optional second argument chooses between the interpolated (faster, less accurate) and non-interpolated version of Stäckel fudge. If the potential is an `Isochrone` or an arbitrary spherical potential, the underlying `C++` implementation will use the specialized `ActionFinderIsochrone` or `ActionFinderSpherical` classes, otherwise the `ActionFinderAxisymFudge` class (interpolated or not).
This class has only one method `__call__` (i.e., can be applied as a function), computing any combination of actions, angles and/or frequencies, for the given 6d position/velocity point or an $N \times 6$ array of points:

```
act = af(points)   # 0th column is J_r, 1st – J_z, 2nd – J_φ
act, ang, freq = af(points, angles=True, frequencies=True)
```

There is also a standalone routine `actions` performing the same task, which may be used without constructing an instance of action finder:

```
agama.actions(points, potential [, fd=focal_distance]
    [, actions=True] [, angles=False] [, frequencies=False])
```

for a non-spherical potential one needs to provide the focal distance Δ (the `ActionFinder`

class retrieves it from an internally constructed interpolation table).

ActionMapper class performs the inverse operation – transform from actions/angles to position/velocity coordinates. Depending on the potential, it will use either the Isochrone, a generic spherical mapping, or Torus machinery (Section 2.4.4) for axisymmetric potentials.

```
am = agama.ActionMapper(pot)
```

When applied to one or more sextets of actions and angles, it returns the corresponding $\boldsymbol{x}, \boldsymbol{v}$, and optionally frequencies in a separate array:

```
xv, freq = am([[J_r1, J_z1, J_phi1, theta_r1, theta_z1, theta_phi1],
               [J_r2, J_z2, J_phi2, theta_r2, theta_z2, theta_phi2]],
              frequencies=True)
```

Note that in the case of Torus mapping, each unique triplet of actions will trigger the construction of a new torus (a non-negligible computation cost), whereas mapping many angles for a single value of actions is relatively cheap. Previously constructed tori are cached for subsequent calls. The Torus mapping is not thread-safe and thus not OpenMP-parallelized; Spherical and Isochrone mapping are free of these limitations.

DistributionFunction class provides the `Python` interface to the hierarchy of `C++` DF classes. It is constructed either from a list of keyword arguments,

```
df = agama.DistributionFunction(type="...", [mass=..., other params])
```

where `type` may be one of the following: `DoublePowerLaw`, `QuasiIsothermal`, `Exponential`, `QuasiSpherical`, and the other parameters are specific to each type of DF (Section 2.5), or from a list of existing `DistributionFunction` objects, creating a composite DF:

```
dfcomp = agama.DistributionFunction(df1, df2, myfnc)
```

Similarly to the `Density` class, one may provide a custom `Python` function `myfnc` which returns the DF value at the given triplet of actions $\{J_r, J_z, J_\phi\}$ (again the calling convention is to process a 2d array of such triplets, even if with one row).

The `QuasiIsothermal` DF (Section 2.5.1) additionally needs an instance of `Potential` to initialize the auxiliary functions $R_c(J)$, $\kappa, \nu, \Omega, \sigma$.

The `QuasiSpherical` DF (Section 2.5.3) is constructed from the provided instances of `Density` and `Potential`, using the generalized Eddington inversion formula to create $f(E, L)$ and then a spherical action finder to convert it to an action-based form.

The DF object applied to a triplet of actions or an array of shape $N \times 3$ returns the values of the DF (summed together if the DF is composite). If called with an optional argument `der=True`, it additionally returns derivatives of the DF w.r.t. actions.

SelectionFunction class provides an example of a position-dependent selection function (SF) for the `GalaxyModel` class: $S(\boldsymbol{x}, \boldsymbol{v}) = \exp\big[-(|\boldsymbol{x}-\boldsymbol{x}_0|/R_0)^\xi\big]$. This function has 3 parameters: the fiducial point $\boldsymbol{x}_0$, cutoff radius R_0, and optional cutoff steepness ξ (by

default $\xi = \infty$, meaning a sharp transition between $S = 1$ at distances smaller than R_0 to $S = 0$ at larger distances, but one can make it more gradual). In addition, an arbitrary user-defined Python function can be used instead of an instance of `SelectionFunction` class. The built-in class

```
sf = agama.SelectionFunction(point=(x0,y0,z0), radius=r0, steepness=xi))
```

is mathematically equivalent to

```
sf = lambda x:  numpy.exp(
    -((x[:,0]-x0)**2 + (x[:,1]-y0)**2 + (x[:,2]-z0)**2) / r0**2)**(xi/2))
```

but of course, the latter is less computationally efficient due to extra costs in using a Python callback function from the C++ code.

GalaxyModel is the combination of a potential, an action finder (which can be constructed internally for the given potential, or provided to the constructor if it was already created), a DF, and optionally a SF:

```
gm = agama.GalaxyModel(pot, df [, af][, sf=sf])
```

This class provides methods for computing DF moments and drawing position/velocity samples from it. These operations are rather expensive, and if the input consists of several points, the computation is internally parallelized using `OpenMP` (except when the DF or SF are user-defined Python functions). In all examples below, the input may contain a single point (with the dimension D depending on the function), or an $N \times D$ array of several points. The input point and the output quantities are given in the "observed" coordinate system XYZ, which may be arbitrarily oriented with respect to the "intrinsic" coordinate system xyz of the model, as specified by the three Euler rotation angles `alpha`, `beta`, `gamma` (Section A.3). When all three angles are zero (default), both coordinate systems coincide. The inclination angle `beta` is the most useful of these parameters.

```
dens, meanvel, vel2 = gm.moments(points, dens=True, vel=True, vel2=True, ...)
```

computes the density ρ, mean velocity $\overline{V}$, and six second moments of the velocity $\overline{V_i V_j}$ at the provided point(s) (Section 2.6.1); one may choose which of these quantities are needed, eliminating unnecessary computations (by default only `dens` and `vel2` are computed). This method comes in two variants: the first version computes the moments at the given 3d point with Cartesian coordinates $\{X, Y, Z\}$ in the observed coordinate system, and the second one computes the projected moments at the given 2d point $\{X, Y\}$, additionally integrating along the line of sight Z. Thus `points` should be either a triplet/pair of numbers for a single point, or an $N \times 3$ / $N \times 2$ array of points. The velocity moments also refer to the observed coordinate system.

```
fproj = gm.projectedDF(points, ...)
```

computes the projected DF integrated along the line of sight Z and optionally over some or all of the velocity components, weighted with Gaussian measurement uncertainties. A single `point` is specified by 8 numbers: X, Y coordinates, three velocity components V_X, V_Y, V_Z, and three corresponding velocity uncertainties $\epsilon_{\boldsymbol{V}}$, all in the observed coordinate system.

The uncertainties can range from zero to infinity inclusive: a zero uncertainty means no integration along this velocity axis, an infinite uncertainty means the integration over the entire available range of velocity, and a finite uncertainty invokes a convolution with a Gaussian, as explained in the corresponding section.

`fVX, fVY, fVZ = gm.vdf(points [, gridv], ...)`
`fVX, fVY, fVZ, dens = gm.vdf(points, dens=True [, gridv=...], ...)`
constructs normalized 1d velocity distribution functions at the given point(s), represented by cubic spline interpolators. One may compute either the full VDFs at the given 3d point(s), or the projected VDFs additionally integrated over the Z axis; the choice again depends on the shape of the `points` array (3 or 2 coordinates per point, respectively). If the optional argument `dens=True` is provided, the output additionally contains the 3d density ρ or the surface density Σ. The optional argument `gridv` specifies the grid in velocity space used to represent the spline (if given as an array) or the size of this grid (if given as an integer). If not provided, the default is to cover the range $\pm v_{\rm escape}$ with 50 points. The grid needs not be too dense – a 3rd degree interpolating spline provides a substantially higher level of detail than an equivalently spaced histogram. One may plot the velocity distribution as a smooth function on a denser grid:

```
v_esc = numpy.sqrt(-2 * pot.potential(point))
gridv = numpy.linspace(-v_esc, v_esc, 200)
plt.plot(gridv, fVX(gridv))
```

`mass = gm.totalMass()`
computes the total mass of the DF inside the spatial region delineated by the SF. If the latter is trivial, the result should be identical to `df.totalMass` up to integation errors, but is naturally much more expensive to compute, because it involves integration over the 6d phase space with actions computed at each point, rather than integration over the 3d action space. Note that if the SF is very localized, the integration routine may fail to find the region where the SF is nonzero and will incorrectly return a zero result; more generally, the accuracy may be rather poor in case of sharp selection boundaries.

All four of the above methods additionally accept an optional argument `separate=True`, which produces separate output for each component of a composite DF – this is more efficient than computing them individually. In this case, output arrays have one extra dimension with length equal to the number of DF components (even when it is 1).

`posvel, mass = gm.sample(N)`
draws N equal-mass samples from the DF multiplied by the SF (Section 2.6.2); the result is an $N \times 6$ array of position/velocity points and a 1d array of masses (the particle masses approximately equal `gm.totalMass()/N`, but here the total mass is computed by the sampling routine). This routine is used for constructing an N-body representation of the DF-based model. Note that if N is too small (say, $\lesssim 10^3$), the sampling routine might not be able to explore the entire volume 6d phase space in an unbiased way, and the resulting snapshot

may miss some parts of it (especially if the DF multiplied by SF is strongly concentrated in a small volume of the phase space); the remedy is to sample a larger number of points and retain a fraction of them.

One may also sample positions from a given density profile, and then assign velocities using either the spherical anisotropic DF (Section 2.5.3) or axisymmetric Jeans equations. This is currently achieved by the `sample` method of a `Density` object, which additionally takes an instance of the `Potential` and optionally the parameters of Jeans equations. Be cautioned that the API will be changed in the future, to harmonize the usage conventions with those used for action-based `GalaxyModel`s.

SelfConsistentModel is the driver class for performing iterative self-consistent modelling (SCM; Section 2.6.3). It is initialized with a list of arguments determining the parameters of the two potential expansions (`Multipole` and `CylSpline`) that are constructed in the process, and optionally a list of components and/or an initial potential. To run the model, one needs to add one or more components; the list of components and other properties of the model may be changed between iterations.

```
scm = agama.SelfConsistentModel(minSph=1e-3, rmaxSph=1e3, sizeRadialSph=40,
    lmaxAngularSph=0, components=[comp])
```

Component class is a single component of this model; it could either represent a static density or potential profile, or provide a DF which will contribute to the density used to compute either of the two potential expansions.

```
comp1 = agama.Component(df=df, density=initdens, disklike=False, **params)
```
creates the component with a spheroidal DF and an initial guess for the density profile (the latter is not needed if the entire SCM has an initial guess for the potential). The extra `params` depend on the `disklike`'ness of the component and specify the size and extent of spatial grid for recomputing its density.

```
comp2 = agama.Component(density=agama.Density(type="disk", mass=0.1,
    scaleRadius=1, scaleHeight=0.05), disklike=True)    # a static gas disk
comp3 = agama.Component(potential=agama.Potential(type="Plummer", mass=0.01,
    scaleRadius=0.001)    # a softened central supermassive black hole
```
creates static components with a given density or potential and without a DF, which remain unchanged in the course of iterative modelling (so do not need extra parameters for a density grid).

```
scm.components = [comp1, comp2]
scm.components.append(comp3)
```
modifies the list of components in the SCM.

```
scm.iterate()
```
performs one iteration of the modelling procedure, recomputing the density of all components with DFs and then reinitializing the total potential.

`comp1.density`
(a read-only property) contains the most recently updated density of this component (it would give `None` for `comp3`, which instead has a `potential` property).

`comp1.df`
(a read-only property) returns the DF associated with the component, or `None` for a static component.

`scm.potential`
is the instance of the total potential, which may be combined with the DF of each component into a `GalaxyModel` object, and used to compute other DF moments or construct an N-body model:

```
posvel, mass = agama.GalaxyModel(scm.potential, comp1.df).sample(10000)
```

The `Component` and `SelfConsistentModel` classes are further illustrated by several example scripts.

Target class represents one of several possible targets in Schwarzschild models. It is initialized by providing `type="..."` and other parameters depending on the target type (see Section A.8). This object can be used in two contexts: either as an argument for the `orbit` routine, collecting the contribution of each orbit to each constraint in the target, or as a function applied to a `Density` or `GalaxyModel` object or an N-body snapshot, returning the array of constraints computed from this object.

Orbit integration is performed by the following routine:

```
result = agama.orbit(potential=pot, ic=posvel, time=int_time, ...)
```

here `posvel` contains initial conditions for one or several orbits (a $N \times 6$ array), total integration `time` for each orbit may be different, but typically is a multiple of the dynamical time returned by the `Tcirc` method of `Potential` (e.g., `100*pot.Tcirc(posvel)`), and dots indicate additional parameters (at least some must be provided to produce a result):
`Omega` specifies the pattern speed of a rotating coordinate system (note that in this case, the velocity is still given in an inertial frame that is instantaneously aligned with the rotating frame, as explained in Section 2.3).
`trajsize` and `dtype` together control the format in which the trajectory is represented, as detailed below.
`timestart` is the initial moment of time (default 0), and similarly to the total `time`, may be either a single value or an array of separate values for each orbit. The final time of each orbit is `time`+`timestart`.
`accuracy` (default 10^{-8}) is the relative accuracy of the ODE integrator.
`der=True` turns on the computation of deviation vectors (derivatives of the trajectory w.r.t. the initial conditions).
`lyapunov=True` additionally estimates the Lyapunov exponent for each orbit (an indicator of chaos) using the fastest growing deviation vector.

`verbose=False` disables the display of progress indicator that is normally shown when more than one orbit are computed.
`method` specifies the ODE integration method; the default `'dop853'` is the 8th order Runge–Kutta method, while the 4th order `'hermite'` method may be more efficient in the regime of low accuracy ($\gtrsim 10^{-4}$).
`targets=...` optionally lists `Target` objects for Schwarzschild modelling.

The `result` returned by this routine is one or a tuple of several items, depending on the requested output. For each `Target`, a $N \times K$ array is produced with the contribution of each orbit to each of K constraints in the given target. When the output trajectory is requested by providing the argument `trajsize`, this adds another item in the output tuple, as detailed below. When `der=True`, the deviation vectors are stored in another array in the output tuple. Finally, when `lyapunov=True`, yet another array of length N is appended to the output tuple, which contains the estimates of Lyapunov exponents for each orbit. If the output consists of a single item (typically, trajectory), it is returned directly instead of a length-1 tuple.

Trajectories can be provided in one of the two alternative formats: as arrays of timestamps and phase-space points, or as interpolator objects.
In the first case, each orbit k produces two arrays: timestamps (1d array of length M_k, which may be different for each orbit) and corresponding position/velocity points (2d array of size $M_k \times 6$ when `dtype` is `float32` or `float64`, or combining position and velocity into real and imaginary parts of a complex number when `dtype` is `complex64` or `complex128`, so that the array has shape $M_k \times 3$). The points can be recorded after every timestep of the ODE integrator (typically meaning a denser sampling near the pericenter) – this regime is triggered by setting `trajsize`=0 (this is not the same as not setting it at all, in which case the trajectory is not recorded). Alternatively, one may request the output points to be regularly spaced in time at intervals `time`/(`trajsize`-1), by setting `trajsize` $= M_k \geq 1$. If `trajsize`=1, only the final point is stored, otherwise both the initial and the final point are stored in the first and the last array elements. If `time`< 0, the integration is carried backward in time, and the array of times is monotonically decreasing (in any case, it goes from the initial moment `timestart` to `timestart+time`, whatever their signs are). In case of $N > 1$ orbits, the output item is a 2d array of shape $N \times 2$, with its elements being arrays themselves (1d arrays of timestamps in the first column and 2d arrays of trajectories in the second); since the lengths of these arrays may differ between orbits, the overall result may not be possible to represent as a regular 3d array. Although the orbit integration is always performed in double precision (`C++ double`, equivalent to `Python float`), by default the output format is single precision (`numpy.float32`) to save memory (with 7 digits of precision usually being enough).
In the second case, the output trajectory is provided in the form of an interpolator – an instance of a special class `agama.Orbit`, which can only be returned by the `orbit` routine, and cannot be created manually. This class is little more than a thin wrapper for three quin-

tic spline interpolators (implemented by the `Spline` class) for each Cartesian component of position and velocity as functions of time. Although it can be constructed from points regularly spaced in time (if `trajsize`≥1), it makes more sense to use the points recorded at every timestep of the ODE integrator (`trajsize`=0), and this is the default setting that does not need to be specified (unlike the array output format), i.e., it is sufficient to set `dtype=object`. The `orbit` routine produces one interpolator object per orbit, i.e., a 1d array of length $N > 1$ or a single object in the case of one orbit. When used as a sequence indexed by the `[]` operator, this object provides the timestamps at which the orbit was recorded, and when used as a function of one variable (time – a single number or an array), it provides the interpolated trajectory at the given time(s) with a typical accuracy $\sim 10^{-6}$ for position and $\sim 10^{-5}$ for velocity, which is usually sufficient. This dual interface makes possible a rather amusing syntax, when the actually recorded trajectory is retrieved by applying an `Orbit` object to itself.

Examples of usage (assuming that `posvel` is an array of $N > 1$ initial conditions):

```
result1 = agama.orbit(potential=pot, ic=posvel, time=int_time, trajsize=1000)
result2 = agama.orbit(potential=pot, ic=posvel, time=int_time, dtype=object)
```

Plot the time evolution of z coordinate of each orbit:

```
for times,trj in result1:  plt.plot(times, trj[:,2])    # regularly spaced
for trj in result2:  plot(trj, trj(trj)[:,2])    # stored at every ODE timestep
for trj in result2:
    times = numpy.linspace(trj[0], trj[-1], 1000)
    plot(times, trj(times)[:,2])    # again regularly spaced
```

Plot the meridional ($R-z$) cross-section of each orbit:

```
for trj in result1[:,1]:  plt.plot((trj[:,0]**2 + trj[:,1]**2)**0.5, trj[:,2])
```

When the computation of deviation vectors is requested by the argument `der=True`, they are provided in the same format as the trajectory itself. There are six deviation vectors per orbit, so if `dtype=object`, they are stored as six `Orbit` objects, otherwise as six arrays of shape $M_k \times 3$ or 6. A matrix formed by column-stacking the deviation vectors at a given point on the orbit represents the Jacobian $\mathsf{J}(t)$ of mapping between the initial conditions $\boldsymbol{w}^{(0)}$ and the current point $\boldsymbol{w}(t)$: $J_{ij} = \partial w_i(t) / \partial w_j^{(0)}$:

```
delta = numpy.random.normal(size=6)*1e-8  # small offset in initial conditions
(time,orbit0), der = agama.orbit(potential=pot, ic=posvel[0], time=int_time,
    trajsize=trajsize, dtype=float, der=True)  # original orbit and dev.vectors
(time,orbit1) = agama.orbit(potential=pot, ic=posvel[0]+delta, time=int_time,
    trajsize=trajsize, dtype=float)  # slightly perturbed orbit
jac = numpy.dstack(der)  # shape: (trajsize,6,6); jac[i] is the Jacobian matrix at time[i]
linear_dif = numpy.einsum('ijk,k->ij', jac, delta)  # same as jac.dot(delta)
actual_dif = orbit1-orbit0
print(numpy.max(abs(actual_dif)))  # max difference between nearby orbits
```

```
print(numpy.max(abs(actual_dif-linear_dif)))  # error in linear prediction should be much smaller than the actual difference, as long as both stay small (no chaotic divergence)
```

sampleOrbitLibrary routine constructs an N-body snapshot from the orbit library of a Schwarzschild model, in which each orbit has a weight assigned by the `solveOpt` routine (see Section A.8 for details):

```
matrix, traj = agama.orbit(potential=pot, ic=posvel, time=int_time,
    dtype=object, targets=[target])
weights = agama.solveOpt(matrix, rhs)
nbody = 1000000
status, result = agama.sampleOrbitLibrary(nbody, traj, weights)
```

The above syntax stores the trajectories in the form of orbit interpolators, so that any number of sampling points can be drawn from each orbit (this number is proportional to its weight in the model, so that the particle mass is the same for all orbits). On the other hand, when the trajectories are recorded as regularly spaced arrays with a certain length `trajsize`, it may not be sufficient to sample orbits with particularly high weights. In this case, the function fails (returns `status=False`), and `result` is a tuple containing the list of orbit indices which did not have enough points in previously recorded trajectories, and corresponding required numbers of samples for each orbit in this list. Then one should rerun the `orbit` routine with these parameters:

```
if not status:
    indices, trajsizes = result
    traj[indices] = agama.orbit(potential=pot, ic=posvel[indices],
        time=int_time[indices], trajsize=trajsizes)
    status, result = agama.sampleOrbitLibrary(nbody, traj, weights)
```

In case of success (`status=True`), `result` contains a tuple of two arrays: `nbody`×6 coordinates/velocities and `nbody` masses of particles in the N-body snapshot. If an optional argument `returnIndices` is set to True, `result` contains three arrays, the last one providing the indices of orbits from which particles were drawn.

N-body snapshot handling is very rudimentary; the routines `agama.writeSnapshot(filename, particles[, format])` and `agama.readSnapshot(filename)` can deal with text files (7 columns – x, y, z, vx, vy, vz, m), and optionally NEMO or GADGET snapshots if the library was compiled with their support. Here `particles` is a tuple of two arrays: $N \times 6$ position/velocity points and N masses; the same convention is used to pass around snapshots in the rest of the `Python` extension (e.g., in potential and sampling routines). A more powerful framework for dealing with N-body snapshots is provided, e.g., by the PYNBODY library [49].

Unit handling is optional: if nothing is specified explicitly, the library operates with the natural N-body units ($G = 1$). However, the user may set up a unit system with three

independent basic dimensional units (mass and any two out of three other units: length, velocity and time), and all dimensional quantities in Python will be expressed in these basic units and converted into natural units internally within the library:

`agama.setUnits(mass=1, length=1, velocity=1)`

instructs the library to work with the commonly used Galactic units: $1\,M_\odot$, 1 kpc, 1 km/s, with the derived unit of time being 0.98 Gyr (this is *not* the same as using no units at all, because $G = 4.3 \times 10^{-6}$ in these units; the numerical value of the gravitational constant is stored as `agama.G`, but explicitly assigning a new value to this variable is not allowed – one should call `setUnits`). Importantly, this setup needs to be performed at the beginning of work, otherwise the values returned by the previously constructed classes and methods (e.g., potentials) would be incorrectly scaled (a warning will be issued if you call `setUnits` after creating instances of other classes). If the ASTROPY framework [3] is installed, one may provide instances of `astropy.Quantity` or `astropy.Unit` as arguments for `agama.setUnits`. The function `agama.getUnits` returns the current units as a dictionary with four items: `length`, `velocity`, `time` and `mass`, all expressed in the same base units as the arguments of `setUnits`, namely: kpc, km/s, Myr and $M_\odot$. Again, if ASTROPY is installed *and imported*, the returned dictionary will contain `astropy.Quantity` instances rather than plain numbers. At the moment there is no way to explicitly attach the units to the dimensional quantities passed to or returned from the library: for instance, `posvel` would be still a plain `numpy` array, with the implied convention that the first three columns are expressed in the length unit (e.g., 1 kpc) and the second three columns – in velocity units (e.g., km/s), but it carries no attributes containing this information. This is a deliberate design choice, as ASTROPY unit conversion incurs a significant overhead cost for small amounts of data.

Coordinate transformation routines provide a simple mechanism for converting between Cartesian and celestial coordinates, or between two celestial reference frames. These conversions are a subset of those provided by ASTROPY, but are faster especially for small-to-medium size inputs, because they operate directly on `numpy` arrays and work only with a specific choice of units, bypassing the overheads of the comprehensive unit conversion subsystem of the ASTROPY framework:
`makeRotationMatrix` constructs a 3×3 orthogonal rotation matrix R parametrized by Euler angles (Section A.3); the coordinates in the rotated frame $\boldsymbol{x}'$ are related to the original coordinates $\boldsymbol{x}$ via $\boldsymbol{x}' = \mathsf{R}\boldsymbol{x}$;
`makeCelestialRotationMatrix` constructs a rotation matrix serving the same purpose, but parametrized by a different triplet of angles;
`transformCelestialCoords` converts the celestial coordinates (longitude/latitude), and optionally proper motions and their dispersion tensor between two celestial reference frames;
`getCelestialCoords` and `getCartesianCoords` perform the bidirectional transformation between celestial coordinates, proper motions, line-of-sight distance and velocity to/from Cartesian coordinates/velocities centered on the observer;
`getGalacticFromGalactocentric` and `getGalactocentricFromGalactic` perform a simi-

lar transformation, but with additional positional and velocity offsets of the observer from the origin of the Galactocentric Cartesian reference frame.

Mathematical methods provided by the `Python` extension module include:
`Spline` class providing one-dimensional cubic, quintic or B-spline interpolation, optionally computing up to three derivatives or an analytic convolution with a Gaussian kernel or another spline function, and providing methods for analytic root-finding and determination of extrema;
`splineApprox` and `splineLogDensity` routines for constructing a penalized cubic spline approximation or density estimate from discrete samples (Section 2.1.1);
`integrateNdim` routine for multidimensional integration (implemented by the `cubature` library, which is included in the `C++` code);
`sampleNdim` routine for sampling from a user-defined multidimensional function;
`solveOpt` routine for solving linear or quadratic optimization problems (used in the context of Schwarzschild modelling);
`nonuniformGrid` and `symmetricGrid` routines for constructing one- and two-sided semi-exponentially-spaced arrays;
`bsplineInterp`, `bsplineMatrix`, `bsplineIntegrals` routines for dealing with B-splines (the first one can be more efficiently replaced by the `Spline` class);
`ghInterp`, `ghMoments` routines for dealing with Gauss–Hermite moments (see Section A.8 for examples of usage on the last two groups);
`randomSeed` routine for setting the seed of the internal random number generator used, e.g., in various sampling methods.

Colormaps augment the rich collection of color schemes included in `matplotlib` with several custom-designed maps, which are better-balanced analogues of `hsv`, `RdBu`, `jet`, `rainbow`, `gist_earth` and `inferno` maps (Figure 1). They can be accessed from any `matplotlib` function under the names `circle`, `bluered`, `breeze`, `mist`, `earth` and `hell`, respectively.

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 `C` interface

There is also a minimalistic `C` interface, with similar functionality as the `Fortran` interface (see the header file `interface_c.h` for details). One may question why it is needed at all, given that the core of the library is written in `C++`, which is "not too different" from pure `C`. However, most of the functionality is implemented in terms of objects, which are not directly available to the `C` code. This interface provides routines for constructing density or potential instances (represented by opaque pointers) from a given array of parameters or from an INI file, evaluating the density, potential and its derivatives, and evaluating actions, angles and frequencies, either directly for the given potential and phase-space point, or mediated by an action finder. The `C` interface also serves as the basis for the `Julia` interface, which is currently in the development stage.

3.4 Interoperability with Galpy

Galpy [13] 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 computation, distribution functions and more. The interface between the two libraries is provided by the "chimera" wrapper class `agama.GalpyPotential`, which is a subclass of both `agama.Potential` and `galpy.potential.Potential`, and thus can act as a regular potential in each of the two frameworks. Internally, it can represent either a native Agama potential or a native Galpy potential – the choice is determined by the arguments supplied to the constructor. If one passes an instance of a Galpy potential, or a list of such instances (which is the approached used to represent a composite potential in Galpy), then the wrapper object is built around the Galpy potential; the routines from Agama access it in the same way as any other user-defined potential model (see example) – in particular, the derivatives are computed by finite differences. In all other cases, the arguments to the constructor are interpreted in the same way as for the `agama.Potential` class, and initialize a native Agama potential object, which behaves in exactly the same way as an instance of `agama.Potential` when used with the classes and routines from Agama, but also provides the interface methods for Galpy. The same functionality (e.g., computation of acceleration) is thus provided by two similar methods: the Agama-native method `force(x)` outputs three components of force per unit mass for a single point or an array of points in Cartesian coordinates, while the Galpy-native `Rforce(R,z[,phi])` outputs the component of acceleration along the cylindrical radius for a single point or an array of points specified in cylindrical coordinates.

An instance of the wrapper class can be used in all relevant contexts in both frameworks, but of course, the efficiency may vary dramatically. In the case of a wrapped Agama-native potential, it is identical to a builtin `Potential` object for Agama routines (e.g., orbit integration, action computation), and is at the level of any user-defined potential for Galpy routines. In the opposite case of a wrapped Galpy-native potential, routines from Agama (e.g. orbit integration) would be quite slow both because of a repeated transfer of control flow between `Python` and `C++`, and because the potential derivatives are computed by finite differences, requiring 13 or 21 evaluations per point. At the same time, routines from Galpy would treat it in the same way as a regular potential implemented in `Python` (i.e., can not use C-accelerated variants even if the underlying potential has a C implementation). Nevertheless, for many purposes, e.g. plotting the isopotential surfaces, the performance penalty can be tolerated. For computationally heavy tasks such as integration of a large number of orbits or action computation for many points, it is preferrable to use Agama-native potentials. Even if there is no direct counterpart of the user potential among the builtin classes, it is often possible to construct an accurate approximation for it in term of either `Multipole` or `CylSpline` expansions, which then run at native speed.

An example, comparing the native Galpy orbit integrator and action finder with those from Agama, is provided in the file `py/example_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 Interoperability with Gala

Gala [53] is another `Python`/`Cython`-based framework for galaxy modelling, similar in scope to Agama. It includes a collection of gravitational potentials, routines for orbit integration, action computation, and is well integrated with Astropy unit and coordinate subsystems. The interaction between Agama and Gala is organized in a similar way as with Galpy, namely, through a "chimera" wrapper class `agama.GalaPotential`, which is a subclass of both `agama.Potential` and `gala.potential.PotentialBase`, and provides both interfaces regardless of the internal representation. It can be initialized either from a native Gala potential, in which case the routines from Agama will access it in the same way as a custom `Python` function representing a potential model (in particular, derivatives are computed by finite differences), or in any of the possible ways to initialize a native Agama potential (e.g., providing a list of named arguments, or a filename, or another instance of `agama.Potential`), in which case it behaves identically to the wrapped object in all contexts within Agama. In the second case, one may also provide an additional argument `units=...` with the same meaning as used within Gala. If initialized with units, or when encapsulating a Gala potential with units, the methods inherited from `gala.potential.PotentialBase` (`density`, `energy`, `gradient`) will automatically convert units in input arguments and output values, but the methods from `agama.Potential` do not perform such conversions (note that due to name clash, the Agama-native method for computing density is renamed to `agamadensity`).

Performance-wise, the same considerations apply here as for the GALPY interface: namely, a wrapper object hosting a native AGAMA potential behaves identically to it in AGAMA, but a wrapper object hosting a native GALA potential will be only accessed from that library via its `Python` interface, and hence built-in GALA classes written in `Cython` will suffer a performance penalty. An example of usage is provided in `py/example_gala.py`.

3.6 Amuse plugin

AMUSE [51] 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. One may 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. This application is illustrated in the example script `py/example_amuse.py`.

Moreover, the stellar-dynamical simulation code RAGA (described in a separate document `readme_raga.pdf`) can be used from within AMUSE, as well as a standalone program (see below).

The AMUSE plugin is built automatically if the environment variable `$AMUSE_DIR` is defined, and the interface is provided by the class `amuse.community.agama.interface.Agama`.

3.7 Nemo plugin

NEMO [68] 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 [24]. 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). It supports all features of the potential interface, including composite, time-dependent or off-centered models, plus a rotating reference frame. All parameters are provided in a single INI file (which, as usual, may include references to other INI files), whose name is given as the command-line argument `accfile=...` In the simplest case, it contains a single section `[Potential]` with `type=...` specifying the potential type, plus other parameters as needed. Such a file may contain coefficients of a `Multipole` or a `CylSpline` potential, and in more complicated

cases may contain several sections [Potential1], [Potential whatever], ...(any name starting with Potential is interpreted as a separate component, and as always, parameter names and values are case-insensitive). Individual potential components may have [possibly time-dependent] offsets from origin, or represent time-dependent potentials (such as Evolving or UniformAcceleration), see Section 2.2.4 for a full list of features. An additional NEMO-specific feature is a possibility of providing a pattern speed Ω for the potential (frequency of rotation of the potential figure about z axis) as a separate command-line argument accpars=...

The shared library agama.so itself provides the NEMO interface without any extra compilation options (it does not depend on any NEMO header files). If the environment variable $NEMOOBJ is defined during the compilation, the shared library will be automatically copied into the folder $NEMOOBJ/acc/, where it could 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 a pattern speed $\Omega = 1.0$. All units in the INI or coefs file here should follow the convention $G = 1$.

As an example, consider the evolution of an N-body model of a Plummer sphere of mass $m = 0.1$ and radius $a = 0.1$ (satellite), which moves on a nearly-circular orbit inside a larger Plummer sphere of mass $M = 1$ and radius $A = 1$ (host galaxy), represented by a smooth external potential. First create the satellite:

```
mkplum out=sat.nemo nbody=10000 r_s=0.1 mass=0.1
```

Put it on an orbit with radius $R = 1$ and initial velocity approximately equal to the circular velocity of the external potential at this radius $V = 0.6$:

```
snapshift sat.nemo sat_shift.nemo rshift=-1,0,0 vshift=0,-0.6,0
```

Create an INI file (pot.ini) with the external potential:

```
[Potential]
type=Plummer
mass=1
scaleRadius=1
```

Now run a simulation for a few orbital periods:

```
gyrfalcON sat_shift.nemo sat_out1.nemo eps=0.02 kmax=5 step=0.25 tstop=50 \
    accname=agama accfile=pot.ini
```

The satellite nicely orbits around the origin, leaving a small tidal tail.

We can model the same situation in the reference frame that is centered not on the host galaxy, but rather on the satellite [approximately]. Namely, the potential of the host galaxy now will be moving on a circular orbit around origin of the coordinate system associated with the simulation, in which the satellite was initially at rest and centered at origin. We need

to create two additional files describing the time-dependent offset of the external potential's center and the acceleration associated with the non-inertial reference frame:

```
import numpy
t = numpy.linspace(0,50,201)   # grid in time
w = 0.6   # rotation frequency ω = V/R
x,y,z = numpy.cos(w*t), numpy.sin(w*t), t*0   # orbit coordinates
numpy.savetxt("center.txt", numpy.column_stack((t, x, y, z)))
ax,ay,az = -w**2*x, -w**2*y, 0*z   # components of centrifugal acceleration
numpy.savetxt("accel.txt", numpy.column_stack((t, ax, ay, az)))
```

And add the following lines to `pot.ini` (the first line refers to the previous potential section, remaining ones add a second section):

```
center=center.txt
[Potential 1]
type=UniformAcceleration
file=accel.txt
```

Now run another simulation (with the un-shifted initial conditions for the satellite):

```
gyrfalcON sat.nemo sat_out2.nemo eps=0.02 kmax=5 step=0.25 tstop=50 \
    accname=agama accfile=pot.ini
```

The outcome should be similar, but expressed in a different reference frame. We can convert it back to the host-centered frame by running the following chain of operations:

```
snapprint sat_out2.nemo t,x,y,z | \
awk '{print $2-cos(0.6*$1),$3-sin(0.6*$1),$4;}' | \
tabtos - sat_out3.nemo nbody=10000 block1=x,y,z
```

In general, the non-inertial acceleration needs not correspond to the second derivative of the offset of the potential center – one could imagine running the simulation in an inertial frame (hence zero acceleration), but forcing the external potential to dance around in some unpredictable way. Finally, the INI file may contain multiple sections, each one could have its own fixed or time-dependent offset (center), or represent an `Evolving` sequence of potentials... The possibilities are boundless!
A more elaborate illustration is provided in the script **example_nbody_simulation.py**

3.8 Arepo and Gadget4 plugins

Arepo [65] and Gadget4 [66] are two hydrodynamical and gravitational N-body simulation codes sharing a common ancestry. Agama can be used to provide an external gravitational potential for both codes in a similar way as for gyrfalcON. However, the design of these codes does not support "plugins" in the traditional sense (external modules that can be loaded dynamically); instead, one needs to introduce some modifications of the source code

and recompile them.

In both cases, the potential is specified in the INI file `agama_potential.ini` (the name is hardcoded in the source code), and its use is turned on by a macro `EXTERNALGRAVITY_AGAMA` in the file `config.sh` during the compilation. It can be used instead of or in addition to the self-gravity in the simulation. One limitation is that the simulation must be using units in which $G = 1$ (specified by the `GravityConstantInternal` parameter), because there is no mechanism for passing dimensional units to the potential constructor in the `C` interface (which is used for the coupling). The external potential is unlikely to be useful for cosmological simulations, and only works in physical (not comoving) coordinates. In case of hydrodynamical simulations in the moving-mesh code Arepo, the simulation box extends from zero to some `BoxSize`, so is not centered on zero; one can shift the Agama potential by half box length, using the parameter `center=...` in the INI file.

A `Python` script `example_nbody_simulation.py` demonstrates the use of Agama to provide an external potential for N-body simulations conducted with gyrfalcON, Arepo or Gadget4; the latter two codes are patched before compilation.

4 Tests and example programs

The Agama framework 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` and `test_***.py` 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. Finally, there are several tools built on top of the library that perform some useful tasks themselves. Some of them are described below.

tutorial_potential is a `Jupyter` notebook illustrating various features of density, potential and orbit integration framework, starting from the basic concepts and progressing to the discussion of composite potentials, user-defined profiles, modifiers, and orbit integration in the rotating frame.

tutorial_streams is a `Jupyter` notebook demonstrating several methods for creating stellar streams from tidally disrupting satellites (see also **example_tidal_stream** below).

example_galpy is a `Python` program illustrating the use of `GalpyPotential` class, which provides a common interface for native potentials from both Agama and Galpy. This example compares the equivalent methods for potential evaluation, orbit integration and action computation from both libraries.

example_gala is a `Python` program illustrating the `GalaPotential` class, which provides a common interface for potentials from AGAMA and GALA, and the equivalent methods for orbit integration.

example_basis_set is a `Python` program comparing the `BasisSet` potential expansion with equivalent classes from GALPY and GALA, which both implement a special case of it (the Hernquist–Ostriker basis set). The coefficients of expansion can be converted from the formats used in these libraries to the input file for the native and more computationally efficient `BasisSet` potential.

example_amuse illustrates the use of AGAMA to provide an external potential in an N-body simulation performed within the AMUSE framework.

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.

example_time_dependent_potential is a `Python` script illustrating various ways of constructing time-dependent potentials (a star-planet system) and integrating test-particle orbits.

example_lmc_mw_interaction is a more elaborate illustration of time-dependent potentials, adapted from [77]. Given two extended massive galaxies (Milky Way and the Large Magellanic Cloud), we first compute their past trajectories in the common center-of-mass frame. Then we create a composite total potential in the reference frame associated with the Milky Way center, which contains the Milky Way itself, the moving LMC, and the acceleration caused by the non-inertial frame. Finally, we compute trajectories of a large number of tracer stars in the Milky Way halo in this evolving potential, and display the perturbations in their spatial and kinematic distribution caused by the LMC.

example_spiral is a `Python` script illustrating the use of a user-defined potential model (a three-arm spiral) for orbit integration: the `Python` function defining the potential can be used directly, but it is rather inefficient, and the preferred way is to construct a `CylSpline` approximation for it. See also **test_user_profiles.py** for other examples of providing user-defined `Python` functions representing density, distribution function and selection function concepts to various routines within the library.

example_mw_bar_potential is a `Python` script that constructs a smooth analytic density profile [64] and the associated `CylSpline` potential approximating the made-to-measure model of the Milky Way bar [50]. **example_mw_potential_hunter24** provides an updated variant of this potential [33] with the same bar, but different disk and halo models, which have been fitted to the recent observational constraints.

example_tidal_stream is a `Python` program demonstrating two approaches for simulating a sinking and tidally disrupting satellite galaxy. In the first approach, it is modelled with the N-body code GYRFALCON (using a `Python` interface to it, PYFALCON), and in the second one (restricted N-body simulation), as a collection of particles orbiting in a smooth potential of the satellite, which itself moves in the host galaxy and is periodically updated to account for the mass loss (this method is similar to the RAGA code discussed below). In both cases, the host galaxy is represented by another analytic potential and the dynamical friction on the satellite is imposed manually.

example_nbody_simulation is a `Python` script for launching an N-body simulation of a star cluster (or a satellite galaxy) in the Milky Way-like potential of the host system, provided by AGAMA as an external potential to one of the following codes: GYRFALCON, AREPO or GADGET4, as well as a restricted N-body simulation similar to the previous script. It creates initial conditions (a simple Plummer model), downloads and compiles the simulation code (if necessary), launches it, and shows the result. Of course, these simulations can be run without a driver `Python` script, by feeding in appropriate initial conditions and parameter files directly to the codes – the files produced by this script could serve as templates.

example_poincare is an interactive `Python` program illustrating the Poincaré surface of section and orbits in the meridional plane of axisymmetric potentials; it demonstrates the use of spline-interpolated orbits in combination with analytic determination of roots of spline functions.

example_deprojection is an interactive `Python` plotting script illustrating the projection and deprojection of triaxial ellipsoidal bodies viewed at different orientations.

example_smoothing_spline is a `Python` program showing the use of penalized smoothing splines for fitting a curve to noisy data (`splineApprox`) and for estimating 1d probability distributions from discrete samples (`splineLogDensity`).

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_torus is a `Python` script illustrating the use of the `ActionMapper` class to construct an orbit, comparing it to a numerically integrated trajectory.

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 `Python` program `gc_runfit` 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 program is designed to work with the mock data from the Gaia Challenge test suite [54], but can be easily adapted to other situations.

example_doublepowerlaw performs a related task: given a spherically-symmetric density and potential profiles (not necessarily related through Poisson equation), numerically construct a `QuasiSpherical` DF and approximate it with a `DoublePowerLaw` DF which has an analytic form (see [35] for a similar approach).

example_vdf_fit_bspline demonstrates the use of B-splines to construct an arbitrary non-negative probability distribution (in this case, the velocity distribution function – VDF) from a discrete sample of velocity measurements, while deconvolving it from the observational uncertainties.

example_adiabatic_contraction presents a `Python` function for computing the effect of contraction of dark matter halo due to baryonic potential, implemented in two variants: (a) the 'true' adiabatic contraction with action-based DF, or (b) an approximate prescription based on enclosed mass profiles.

example_self_consistent_model illustrates various steps of the workflow for creating multicomponent self-consistent galaxy models determined by DFs. It begins with initial guesses for the density profiles of all components, and computes the total potential, which is used to construct a `QuasiIsothermal` DF for the disk; the `DoublePowerLaw` DFs of the halo and the bulge do not need a potential. Then it performs several iterations, updating the density of both disk and halo components and recomputing the total potential. 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.

There are a couple of closely related programs (the first one is `C++`, the rest are `Python`):
example_self_consistent_model_mw is a very similar program that creates a Milky Way model fitted to *Gaia* DR2 kinematic data [12], which uses non-standard (user-defined) DF classes for the disk and the halo components;
example_self_consistent_model3 performs the same task for a slightly different three-component galactic model fully specified by DFs and plots several physical quantities in the

model (velocity dispersion profiles, LOSVDs, etc.);
example_self_consistent_model_simple is a stripped-down version showing only the bare minimum of steps in the context of a spherical model;
example_self_consistent_model_flattened is a slightly more complicated variant that creates a flattened rotating Sérsic model;
example_mw_nsd constructs a DF-based self-consistent model of the Milky Way nuclear stellar disk [63].
test_self_consistent_model illustrates the iterative self-consistent modelling workflow by reimplementing the `Component` and `SelfConsistentModel` classes in pure `Python` and testing their equivalence to the `C++` classes. The same approach can be used in other contexts besides self-consistent modelling, as a means of producing the density profile corresponding to the given DF – essentially, computing the zeroth moment of the DF at a small number of points (~ 100) and then interpolating it into the entire 3d space as `DensitySphericalHarmonic` or `DensityCylindricalHarmonic`. This could be useful if one needs to evaluate it at many points – for instance, to compute the integrals of density over some finite volume (equivalent to a spatial selection function) or the surface density.

example_schwarzschild_triaxial and **example_schwarzschild_flattened_rotating** are two `Python` scripts illustrating the basic steps in Schwarzschild orbit-superposition modelling. The first one creates a self-consistent triaxial model with a Dehnen density profile, and then converts it into an N-body snapshot. The second one creates a flattened rotating Sérsic model with a central black hole, again converts it into an N-body snapshot, and plots various diagnostic information.

schwarzschild is a more general `Python` program for constructing multicomponent Schwarzschild models and converting them into N-body models. It reads all model parameters from an `ini` file (an example of a three-component axisymmetric disk galaxy model is provided in `data/schwarzschild_axisym.ini`). This program can be used in the "theoretical" context, when the goal is to construct a single equilibrium model with given parameters, not to fit a model to some observational data – that job is performed by the following program.

example_forstand is a `Python` program illustrating various aspects of observationally-constrained Schwarzschild modelling. It creates mock datasets from N-body models, runs a grid of models, and displays results in an interactive plot. This program can serve as a template for user scripts adapted to particular observational datasets.

measureshape is a `Python` program providing a function `getaxes` (which can be used in other scripts) for measuring the shape and orientation of triaxial ellipsoidal N-body snapshots, using the moment of inertia tensor (method E1 in [78]). For analytic density models, the same functionality is provided by the `principalAxes` method of the `Density` class.

mkspherical is a `C++` program for creating and analyzing spherical isotropic models. These models are defined by a potential $\Phi(r)$ and a distribution function $f(E)$, or rather, $f(h)$, where $h(E)$ is the phase volume (Section 2.5.4). These models may or may not be self-consistent, i.e., the potential may be generated by the density profile corresponding to $f(h)$, but also contain other external components (e.g., a central massive black hole). This tool can be used in two distinct modes: (a) creating a spherical model with prescribed properties (given by a built-in density profile, or interpolated from a user-provided table of enclosed mass within a range of radii), using the Eddington inversion formula to compute the distribution function, or (b) analyzing an N-body snapshot (constructing smooth spline approximations to the spherical potential, density and isotropic distribution function). The output consists of a text table with several variables (Φ, ρ, f, etc.) as functions of radius or energy, and/or an N-body realization of the model, which may then serve as the initial conditions in a simulation. Note that in the case that a given density and potential cannot be reproduced by a non-negative isotropic DF (e.g., a constant-density core in the Kepler potential), no error is emitted, and instead the DF is replaced with zeros whenever the computed value is negative.

phaseflow is a `C++` program for computing the dynamical evolution of a spherical isotropic stellar system driven by two-body relaxation [73]. It solves a coupled system of Fokker–Planck and Poisson equations for the joint evolution of $\Phi(r), \rho(r)$ and $f(h)$ discretized on a grid, using the formalism presented in Section A.7.2. It reads all input parameters from an `ini` file; a couple of examples are given in `data/phaseflow_corecollapse.ini` and `data/phaseflow_bahcallwolfcusp.ini`

raga is a Monte Carlo stellar-dynamical code for simulating the evolution of non-spherical stellar systems [72]. It represents the system as a collection of particles that move in the smooth potential, represented as a `Multipole` expansion with coefficients being regularly recomputed from the particles themselves, and can include several additional dynamical processes: explicitly simulated two-body relaxation, loss-cone effects (capture of particles by a massive black hole), and interaction between a binary massive black hole and the stellar system. The code is described in a separate document (`readme_raga.pdf`), and an example input file is provided in `data/raga.ini`. The AMUSE interface to RAGA has extra features compared to the standalone program, namely the possibility of adding particles or changing the stellar masses and radii during the simulation.

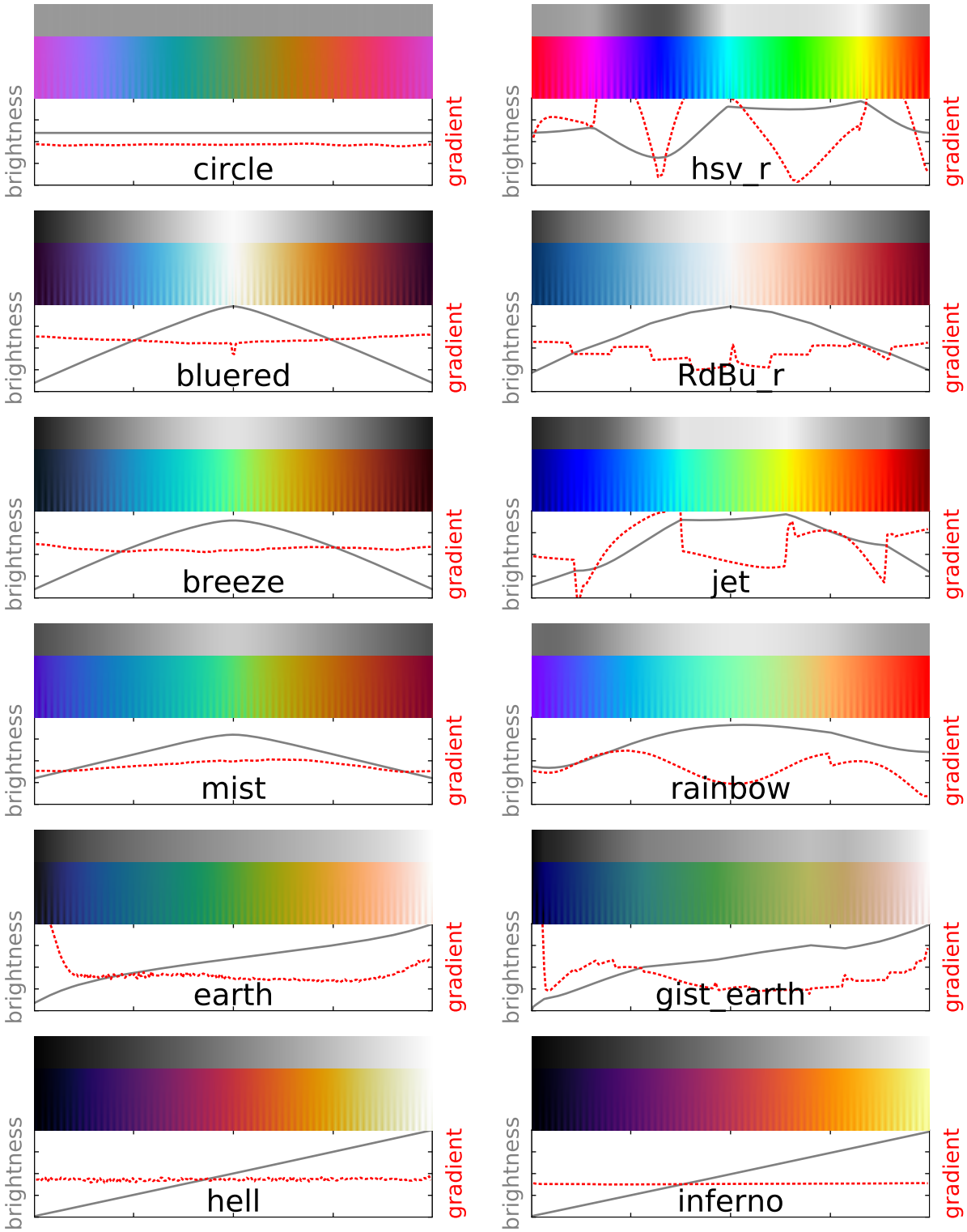

Figure 1: Custom colormaps in AGAMA (left) compared to the ones from MATPLOTLIB.

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 [44, 67], others are more general [39, 40]. 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 object 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 `SelfConsistentModel` struct and assocated routines follow this pattern. 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, 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, string collections (`utils::KeyValueMap`), and "runtime functions" that perform some data collection tasks during orbit integration (even in this case, sometimes the data is stored in an external non-member variable).

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 some pre-defined 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. Consider, for instance, two one-dimensional integrals $I_a \equiv \int_{-1}^{1} \sqrt{1-x^2}\,\mathrm{d}x$, $I_b \equiv \int_{-1}^{1} 1/\sqrt{1-x^2}\,\mathrm{d}x$; their analytical values are respectively $\pi/2$ and π. Both integrands have integrable endpoint singularities (even though the first one tends to zero as $x \to \pm 1$, it is not analytic at these points), and a naïve N-point Gauss–Legendre quadrature rule has poor convergence: relative errors are 0.003 (0.0005) with $N = 5$ (10) for I_a, and 0.1 (0.05) for I_b. However, if we apply transformation of the interval $[-1..1]$ onto itself, stretching it near the endpoints, the results are far better: for a substitution $x = (3 - y^2)y/2$, the errors in 5- or 10-point rule are $\sim 10^{-6}$ or 10^{-12} for both functions, while another similar substitution $x = \sin(y\,\pi/2)$ would even make the second integral exact. Such transformations, in exactly the same context, are used, e.g., in computing the actions and frequencies.

Multidimensional integration methods employ adaptive refinement of the integration domain, and in principle should be able to cope with strongly varying or non-analytic functions, again at the expence of accuracy (for a fixed upper limit on the number of function evaluations). However, there is another reason for applying non-trivial hand-crafted scaling

transformations: if the function in question is very strongly localized in some small region of the entire domain, the adaptive integration routine may simply miss some part of the domain where none of the initially considered points yielded non-negligible function values, and hence this part was not refined at all. Such situation may arise, e.g., when computing the density $\rho(\boldsymbol{x}) = \int f(\boldsymbol{x},\boldsymbol{v})\,\mathrm{d}^3v$ from a DF of a cold disk, which is very narrowly peaked at $v_r=0, v_z=0, v_\phi=v_{\rm circ}(r)$; hence we employ a sophisticated transformation that stretches the region around $|v|=v_{\rm circ}$, facilitating the task of "hitting the right point".

Dimensional quantities are usually converted to logarithms before applying any scaling transformations, to ensure a (nearly-)scale-invariance. Consider, e.g., $\int_0^\infty f(x)\,\mathrm{d}x$ with a naïve transformation of the interval onto [0..1]: $x=y/(1-y)$. If the function $f(x)$ peaks between $x=10^6$ and 10^7 (not uncommon when dealing with astronomical scales), no reasonable adaptive quadrature rule would be able to detect this peak crammed into a tiny interval $1-10^{-6} < y < 1-10^{-7}$! If, on the other hand, we employ a two-stage transformation, $x=\exp(y)$, $y=1/(1-z)-1/z$, then the peak lies between $0.93<z<0.94$, which is far more likely to be found and handled correctly. Same considerations apply to root-finding and minimization routines; in these cases the added benefit is increased precision. If a root in the naïvely scaled variable is at $1-10^{-10}$, it has only 5 significant digits; if our relative tolerance in root-finder was 10^{-12}, the un-scaled variable would carry a relative error of 10^{-2}! By contrast, in the two-stage transformation the root will be around 0.96 and we only lose one or two digits of precision.

Consistency and reproducibility of floating-point (FP) calculations is a painful issue, as anyone seriously working with them can attest. Theoretically, with no compiler optimizations and running in a single thread, one should be able to get identical results on all architectures conforming to the IEEE 754 FP standard (that is, almost all current ones). However, once we wish the program to be even mildly efficiently optimized, the results can differ between compilers and even between running the same executable on different processors. Adding the multi-threading computations surely makes things even less predictable. For instance, computing a sum of array elements in parallel will yield different results depending on the number of threads, because the sub-ranges summed by different threads will be different, and FP addition is not commutative. When dynamic load-balancing is allowed in runtime, these sub-ranges may be different between runs even with the same number of threads. These issues notwithstanding, a reasonable effort has been invested to keep as much reproducibility as possible when running on the same machine with the same number of threads, and keep the differences at the level of FP precision when running with different number of threads. However, even though the roundoff errors in lower-level functions stay at the machine precision, various high-level operations (such as construction of `CylSpline` potential from density or computation of DF moments, both involving multidimensional integration) have much less strict tolerances, typically at the level 10^{-6} to 10^{-4}, and eventually the results obtained on different machines or even with different compilation options could differ by that much.

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[13] 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 signalling, because a constructor cannot return a value, and storing an error code as a class member variable doesn't make sense, since 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` is a cheaper alternative, used, for instance, in action finders if the energy is positive (does

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

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 on screen and to text files) 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`, `BasisSet` and `CylSpline` potentials from `ParticleArrays`, initialization of interpolation tables in `ActionFinder***` constructors, sampling from a multidimensional probability density, and a couple of other routines marked by a `"\note OpenMP-parallelized loop"` doxygen comment. Other typical operations, such as computation of potential or action for many points simultaneously, should be paralellized in the caller code itself, if needed. 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 (GIL) preventing its simultaneous usage from multiple threads. Therefore, when such callback functions are used with `C++` routines, they introduce a barrier (acquiring GIL in a thread that is currently executing Python code and releasing it upon exiting from the user function), effectively negating the effect of `OpenMP` parallelization altogether. However, these user-defined functions are invoked with a vector of input points that may be processed in a single call (e.g. using `numpy` array operations), instead of one point at a time, thus reducing the cost of transferring the control flow between `C++` and `Python`.

Vectorization is a complementary concept to parallelization, and means calling various routines with names such as `evalmany***` with a vector of input points rather than one point at a time. Such routine may perform a single-threaded or an `OpenMP`-parallelized

loop over the points, or pass the entire array of points (or any intermediate array derived from these points) onward to another vectorized routine. It is used whenever possible: for instance, N-dimensional integration routines call the integrand with an array of a few dozen points at a time, N-dimensional sampling operates on blocks of 10^3 points, `galaxymodel::computeMoments` and similar functions collect the values of DF and SF for a bunch of points at a time while performing N-dimensional integrations, `evalmany***` methods of composite density or potential classes call the corresponding methods of underlying components, etc. Constructors of density and potential expansions collect the values of input density or potential instances in a single call for all points used in integration. Naturally, this extends to the user-defined functions in Python – they are usually invoked with vectorized input.

A good illustration of vectorization coupled with parallelization is the self-consistent modelling workflow (Section 2.6.3). The `Component***::update` methods create spherical- or azimuthal-harmonic density expansions from the intermediate class `DensityFromDF`. The constructors of density expansions collect the values of this intermediate density class in one vectorized call to its `evalmanyDensityCyl` method, which in turn, performs an `OpenMP`-parallelized loop over input points, calling `computeMoments` for one point from each thread simultaneously. The latter function, in turn, involves integration over velocity, and the integrand is called in a vectorized manner, but is not additionally parallelized (this would make no sense).

A.2 Mathematical methods[14]

A.2.1 Basis-set approximation of functions

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 $f(x), g(x)$:

$$\langle f, g \rangle \equiv \int_a^b f(x)\, g(x)\, \mathrm{d}x\,, \tag{1}$$

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:

$$\tilde{f}^{(M)}(x) = \sum_{j=1}^{M} A_j\, B_j(x)\,. \tag{2}$$

The coefficients of this expansion (amplitudes) A_j are determined from the requirement that the inner product $\mathcal{P}_i$ of the original function f and each of the basis elements B_i is the

[14] In this chapter, we use ***boldface*** for column-vectors and Sans-serif font for matrices, while keeping the ordinary *cursive* script for writing element-wise expressions.

same as the inner product of the approximated function $\tilde{f}^{(M)}$ and the same basis element (Galerkin projection):

$$\mathcal{P}_i\{f\} \equiv \langle f, B_i \rangle = \tag{3a}$$

$$\mathcal{P}_i\{\tilde{f}^{(M)}\} \equiv \langle \tilde{f}^{(M)}, B_i \rangle = \int_a^b \tilde{f}^{(M)}(x)\, B_i(x)\, \mathrm{d}x = \sum_{j=1}^{M} G_{ij}\, A_j\,, \tag{3b}$$

where G is the Gram matrix of inner products of basis functions:

$$G_{ij} \equiv \langle B_i, B_j \rangle = \int_a^b B_i(x)\, B_j(x)\, \mathrm{d}x\,. \tag{4}$$

Classical basis sets are usually orthonormal, i.e., $G_{ij} = \delta_{ij}$, 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 $\mathsf{G}\,\boldsymbol{A} = \boldsymbol{\mathcal{P}}$ (3) to find A_j from $\mathcal{P}_i$ (this is the case for the B-spline basis set discussed in the Section A.2.2).

The computation of projection integrals $\langle f, B_i \rangle$ ideally should be performed with a method that gives an exact result if the function f is itself a basis element (or, consequently, a weighted sum of these elements, such as $\tilde{f}$); equivalently, the Gram matrix (4) needs to be computed exactly. This implies the use of the trapezoidal rule with equidistant points for the Fourier basis, the Gauss–Legendre rule for a basis of Legendre polynomials (spherical harmonics), the Gauss–Hermite rule for the eponymous basis set (Section A.2.6), or again the Gauss–Legendre rule *separately on each grid segment* for a B-spline basis (Section A.2.2).

We now discuss several commonly encountered tasks and the techniques for solving them with the aid of basis sets.

Estimation of the probability distribution function $f(x)$ from discrete samples $\{x_n\}_{n=1}^N$ drawn from this function can be formulated in the framework of basis functions as follows. The discrete realization of the probability density is $\hat{f}(x) \equiv \frac{1}{N}\sum_{n=1}^N \delta(x - x_n)$, and we identify it with the smooth approximation $\tilde{f}$ expressed in terms of a weighted sum of basis functions (2). According to (3), the amplitudes $\boldsymbol{A}$ of this expansion satisfy the linear equation system

$$\sum_{j=1}^{M} G_{ij}\, A_j = \mathcal{P}_i\{\hat{f}\} = \frac{1}{N}\sum_{n=1}^{N} B_i(x_n)\,. \tag{5}$$

This expression is trivially generalized to the case of unequal-weight samples.
The drawback of this formulation is that the approximated density $\tilde{f}$ is not guaranteed to be non-negative, unlike the original function f. An alternative approach, discussed in

Section A.2.5, is to represent the *logarithm* of f as a basis-set approximation, ensuring the non-negativity property; however, it is a non-linear operation, unlike the simpler approach introduced above. Of course, there exist various other methods for estimating the density from discrete samples, for instance, using the kernel density estimation (KDE). However, the latter is fundamentally a smoothing operation, producing the estimate of the convolution of f with the kernel, rather than of the original function f. [illustration?]

Linear operators acting on the function f can be represented as linear transformations of the vector of amplitudes of $\tilde{f}$: $A'_k = T_{kj} A_j$. The matrix T may correspond to an exact representation of the transformed function $\tilde{f}'(x) \equiv T\{\tilde{f}\}(x)$, possibly in terms of a different basis set B'_k, or to its approximation constructed in the same way as for the original function f (by Galerkin projection of the transformed function $\tilde{f}$ onto the basis).

An example of the first kind is the differentiation or integration of $\tilde{f}$, represented by its expansion in terms of a similarly transformed set of basis functions. For instance, in the case of Fourier or Chebyshev series, the transformed basis functions can be exactly represented by the same or a related basis set (increasing the degree M in the case of integration). For a M-th degree B-spline basis set described in the next section, the integrals or derivatives of basis functions are also B-splines of degree $M+1$ or $M-1$, correspondingly, defined by the same grid. This feature is used in the finite-element analysis to represent differential equations in a discretized form, and solve the equivalent linear algebra equations.

An example of the second kind is a convolution operation $\tilde{f} * K \equiv \int_a^b \tilde{f}(y)\, K(x-y)\, \mathrm{d}y$. If $\tilde{f}$ is represented by the vector of amplitudes $\boldsymbol{A}$, and the convolved function $\breve{f} \equiv \tilde{f} * K$ is represented by the vector of amplitudes $\breve{\boldsymbol{A}}$, the relation between these two vectors is found by applying the projection operator $\mathcal{P}_i$ to $\breve{f}$:

$$\mathcal{P}_i\{\breve{f}\} = \sum_j K_{ij} A_j, \qquad K_{ij} \equiv \int_a^b \mathrm{d}x \int_a^b \mathrm{d}y\; B_i(x)\, B_j(y)\, K(x-y)\,. \tag{6}$$

Hence, according to the general rule, $\breve{\boldsymbol{A}} = \mathsf{G}^{-1}\, \mathsf{K}\, \boldsymbol{A}$.

Change of basis: if one needs to express the function $\tilde{f}(x) = \sum_j A_j\, B_j(x)$ in terms of a different basis set as $\sum_m \mathcal{A}_m\, \mathcal{B}_m(x)$, the amplitudes of this expansion are given by $\boldsymbol{\mathcal{A}} = \mathcal{G}^{-1}\, \mathsf{H}\, \boldsymbol{A}$, where $H_{mj} \equiv \langle \mathcal{B}_m, B_j \rangle$, and $\mathcal{G}$ is the Gram matrix of the basis set $\mathcal{B}_m$. This is, in general, a lossy operation (one may call it a *reinterpolation* onto the new basis), in a sense that constructing an approximation for a function f directly in terms of the new basis is not equivalent to the approximation for f in terms of the old basis and then a further approximation of this $\tilde{f}$ in terms of the new basis. Depending on the properties of both the function and the basis, the extra error may be negligible or not.

A.2.2 B-splines

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 recursion (de Boor's algorithm):

$$B_j^{[0]}(x) \equiv \begin{cases} 1 & \text{if } k_j \le x \le k_{j+1} \\ 0 & \text{otherwise} \end{cases} , \tag{7a}$$

$$B_j^{[N]}(x) \equiv B_j^{[N-1]}(x)\, \frac{x - k_j}{k_{j+N} - k_j} + B_{j+1}^{[N-1]}(x)\, \frac{k_{j+N+1} - x}{k_{j+N+1} - k_{j+1}} . \tag{7b}$$

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 (2) 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 G (4) is block-diagonal with bandwidth N, thus the coefficients of decomposition A_j are obtained from $\mathcal{P}_i$ in $\mathcal{O}(N^2 M)$ operations.

- 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 $f(x) \ge 0$, it is sufficient to have $A_i \ge 0$.

- 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). It is possible to construct an equivalent representation of a natural cubic spline in terms of a modified $N = 3$ B-spline basis set, in which the first three basis functions B_0, B_1, B_2 are replaced with two linear combinations that have zero second derivative: $\tilde{B}_1 \equiv B_0 + \frac{x_2 - x_0}{x_1 + x_2 - 2x_0} B_1$, $\tilde{B}_2 \equiv B_2 + \frac{x_1 - x_0}{x_1 + x_2 - 2x_0} B_1$, and similarly for the last three basis functions, see Figure 2 (left panel).

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 (3) by a discrete sum, see Sections A.2.4 and A.2.5). On the other hand, if one needs to construct an interpolating function passing through the given set of points (the standard

interpolation problem), B-splines are less convenient, and the evaluation of the interpolant is also less efficient than for the "classical" textbook splines discussed in the next section.

A.2.3 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).

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 $f'(x_{i+1})$[15] 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 $f(x_1) \ldots f(x_K)$, we may consider the result of interpolation

[15] 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. Furthermore, the computed derivatives may subsequently be modified by the regularization filter to improve the behaviour around sharp discontinuities (see below).

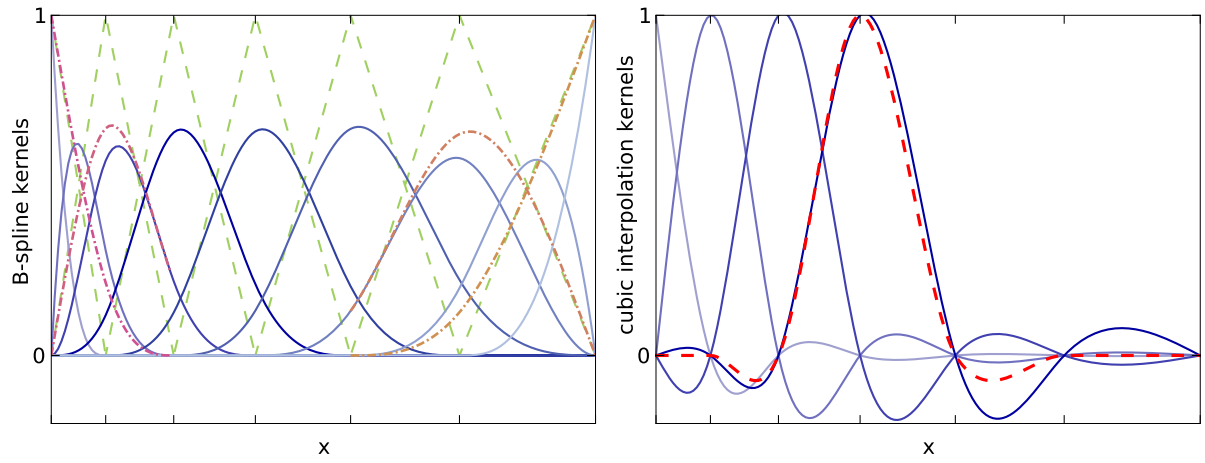

Figure 2: Various interpolation kernels. To obtain the value of interpolated function, one sums up the values of all interpolating kernels with appropriate weight coefficients.
Left panel: B-splines of degree $N = 1$ (dashed) and $N = 3$ (solid lines), defined by the nodes of a non-uniform grid (marked 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. Dash-dotted lines show modified cubic B-spline basis functions that have natural boundary conditions (zero second derivative at endpoints); they are obtained by replacing three leftmost $N = 3$ B-splines with two new functions (essentially distributing the middle one between the other two), and similarly for the three rightmost B-splines.
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.

as a smoothing kernel defined by the grid nodes and linearly depending on all input values $f(x_i)$ (see Figure 2, right panel, for a 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 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.

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

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

$$0 = 30\frac{f_2-f_1}{(x_2-x_1)^3} - \frac{16f_1'+14f_2'}{(x_2-x_1)^2} - \frac{3f_1''-2f_2''}{x_2-x_1} \quad \text{for } i=1, \text{ and similarly for } i=K. \tag{8b}$$

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).

A common problem with high-order interpolation arises when there are sharp discontinuities in the input data, which lead to overshooting and nasty oscillations in the interpolated curve. To cope with these cases, there is a possibility of applying a regularizing (monotonicity-preserving) filter for a natural cubic spline [34]. As usual, the first derivatives at each node are computed from the standard tridiagonal system under the requirement that the second derivative is continuous. Then the filter examines the behaviour of the cubic interpolation polynomial at each segment and determines whether it is monotonic or not. In the latter case, and if the input data values were monotonic, it adjusts the first derivative, so that the interpolated curve also becomes monotonic. This does not preclude it from having a local maximum or minimum on a segment adjacent to a local extremum of input data points, but sometimes may also soften the amplitude of this maximum. Figure 3 illustrates various aspects of the regularization filter. The downside of this filter is that it converts a twice continuously differentiable curve into a generic piecewise-cubic Hermite interpolator with only one continuous derivative, but it typically does so only in the cases when the more smooth curve is actually a worse approximation of the function. One should also keep in mind that this procedure breaks the linearity of spline construction (i.e. a sum of two filtered splines may not be equal to a filtered spline of a summed input points). Thus the regularization filter does not apply by default, but is useful in particular for log-scaled interpolators ($\ln f(\ln x)$ is represented as a cubic spline), where an occasional very small positive value of an input point results in a large jump in the log-scaled curve, which then overshoots and oscillates on nearby segments, aggravated by the inverse exponential scaling.

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

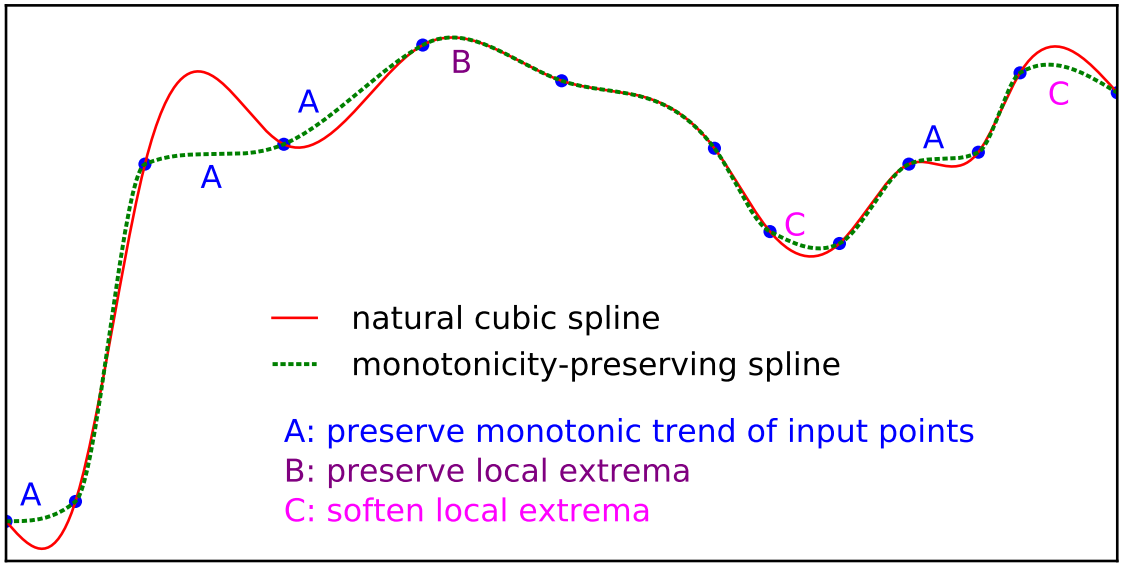

Figure 3: Monotonicity-preserving spline (dotted line) compared to the natural cubic spline (solid line). This demonstrates several aspects of regularization filter: it preserves a monotonic trend of input points, avoiding spurious bumps (left part of the plot), does nothing if the spline is smooth enough (center), and preserves the information about local minima/maxima but may soften them somewhat (right).

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 are provided with four matrices $\mathsf{f}, \mathsf{f}'_x, \mathsf{f}'_y, \mathsf{f}''_{xy}$, we can 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 often have only 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 equally 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.2.4 Penalized spline regression

Suppose we have N_{data} points $\{\boldsymbol{x}, \boldsymbol{y}\}$ with weights $\boldsymbol{w}$, and we need to find a smooth function $y = f(x)$ that approximates the data in the weighted least-square sense, but does not

fluctuate too much – in other words, minimize the functional

$$\mathcal{Q} \equiv \sum_{i=1}^{N_{\rm data}} w_i \; [y_i - f(x_i)]^2 + \lambda \int [f''(x)]^2 \; {\rm d}x. \tag{9}$$

Here $\lambda \geq 0$ is the smoothing parameter that controls the tradeoff between approximation error and wiggliness of the function [29]; its choice is discussed below.

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

$$f(x) \equiv \sum_{k=1}^{N_{\rm basis}} A_k \, B_k(x). \tag{10}$$

We use the basis set of modified cubic ($N = 3$) B-splines with natural boundary conditions, so that the number of basis functions $N_{\rm basis}$ is equal to the number of grid knots. Note that the value of interpolated function $f(x)$ at a grid point x_k is *not* equal to the amplitude of the corresponding basis function A_k, but rather to a linear combination of three adjacent amplitudes A_{k-1}, A_k, A_{k+1}, see Figure 2, left panel; however, for convenience we represent the result in terms of the array of function values $f(x_k)$, which may be used to initialize a natural cubic spline in the usual way.

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 consecutive 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: $R_{kl} \equiv \int B_k''(x) \, B_l''(x) \, {\rm d}x$. This matrix is also sparse (band-diagonal) and symmetric. The functional $\mathcal{Q}$ to be minimized (9) may be rewritten as

$$\mathcal{Q} \equiv (\boldsymbol{y} - \mathsf{B}\boldsymbol{A})^T \, \mathsf{W} \, (\boldsymbol{y} - \mathsf{B}\boldsymbol{A}) + \lambda \, \boldsymbol{A}^T \, \mathsf{R} \, \boldsymbol{A}, \qquad \mathsf{W} \equiv {\rm diag}(\boldsymbol{w}), \tag{11}$$

and its minimum is obtained by solving the normal equations ${\rm d}\mathcal{Q}/{\rm d}\boldsymbol{A} = 0$ for the amplitudes $\boldsymbol{A}$:

$$\left(\mathsf{B}^T \mathsf{W} \, \mathsf{B} + \lambda \mathsf{R}\right) \boldsymbol{A} = \mathsf{B}^T \mathsf{W} \, \boldsymbol{y}. \tag{12}$$

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 $\boldsymbol{y}$ (with the same coordinates $\boldsymbol{x}$ and weights $\boldsymbol{w}$), there is an efficient algorithm for this [56]:

1. Compute the Cholesky decomposition of the matrix $\mathsf{B}^T \mathsf{W} \, \mathsf{B}$, representing it as $\mathsf{L}\mathsf{L}^T$, where L is a lower triangular matrix with size $N_{\rm basis}$. To avoid problems when $\mathsf{B}^T \mathsf{W} \, \mathsf{B}$

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 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_{basis}, and $\boldsymbol{S}$ is the vector of singular values.
Now the matrix in the l.h.s. of (12) can be written as
$\mathsf{B}^T\mathsf{W}\,\mathsf{B} + \lambda\mathsf{R} = \mathsf{L}\mathsf{L}^T + \mathsf{L}\mathsf{L}^{-1}\mathsf{R}\mathsf{L}^{-T}\mathsf{L}^T = \mathsf{L}\mathsf{U}\mathsf{U}^T\mathsf{L}^T + \mathsf{L}\,\mathrm{diag}(\boldsymbol{S})\,\mathsf{U}^T\mathsf{L}^T$.
Finally, compute a matrix $\mathsf{M} \equiv \mathsf{L}^{-T}\mathsf{U}$.

2. For any vector of $\boldsymbol{y}$ values, pre-compute $\boldsymbol{p} \equiv \mathsf{B}^T\mathsf{W}\,\boldsymbol{y}$ and $\boldsymbol{q} \equiv \mathsf{M}^T\boldsymbol{p}$ (vectors of length N_{basis}), and the weighted sum of squared y-values $V \equiv \boldsymbol{y}^T\mathsf{W}\,\boldsymbol{y} = \sum_{i=1}^{N_{\mathrm{data}}} w_i y_i^2$.

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

$$\boldsymbol{A} = \mathsf{M}\,[\mathsf{I} + \lambda\,\mathrm{diag}(\boldsymbol{S})]^{-1}\,\boldsymbol{q}, \tag{13}$$

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 – the first term in (11) – is given by

$$\mathrm{RSS} \equiv (\boldsymbol{y} - \mathsf{B}\boldsymbol{A})^T\,\mathsf{W}\,(\boldsymbol{y} - \mathsf{B}\boldsymbol{A})^T = V - 2\boldsymbol{A}^T\boldsymbol{p} + |\mathsf{L}^T\boldsymbol{A}|^2. \tag{14}$$

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:

$$\mathrm{EDF} \equiv \mathrm{tr}[\mathsf{I} + \lambda\,\mathrm{diag}(\boldsymbol{S})]^{-1} = \sum_{k=1}^{N_{\mathrm{basis}}} \frac{1}{1+\lambda S_k}\ . \tag{15}$$

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 determined by minimization of the generalized cross-validation score:

$$\mathrm{GCV} \equiv \frac{N_{\mathrm{data}}\ \mathrm{RSS}}{(N_{\mathrm{data}} - \mathrm{EDF})^2}. \tag{16}$$

Often one may wish to apply a somewhat stronger smoothing than the one given by minimizing GCV, for instance, by allowing ln(GCV) to be larger than the minimum value by a specified amount $\Delta\ln(\mathrm{GCV}) \sim \mathcal{O}(1)$. In both cases, the corresponding value of λ is obtained by standard one-dimensional minimization or root-finding routines. Figure 4 illustrates the method.

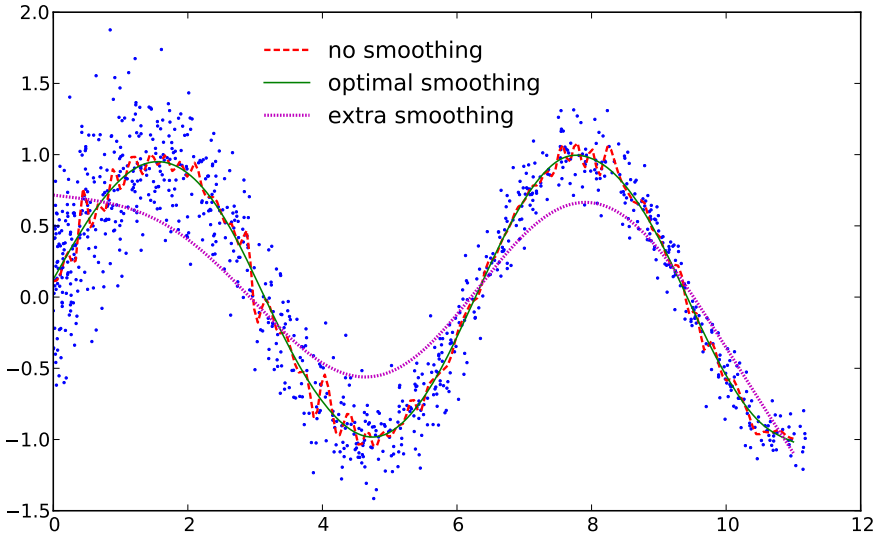

Figure 4: Penalized spline fit to noisy data. $N_{\rm data} = 1000$ points follow a sine curve with a random noise in y-coordinate. A non-penalized spline fit with 100 nodes (red dashed line) is overfitting the noise, whereas an optimally-smoothed spline with the same number of points (solid green line) recovers the original trend very well; of course, we could use a far smaller number of nodes ($\lesssim 10$) and still get a decent fit, but the optimal amount of smoothing prevents overfitting even if the node spacing is too dense. Finally, dotted magenta line illustrates the effect of oversmoothing.

A.2.5 Penalized spline density estimate

Let $P(x) > 0$ be a density function defined on the entire real axis, a semi-infinite interval $[x_{\rm min}, +\infty)$ or $(-\infty, x_{\rm max}]$, or a finite interval $[x_{\rm min}, x_{\rm max}]$. Let $\{x_i, w_i\}$ be an array of $N_{\rm data}$ samples drawn from this distribution, where x_i are their coordinates, and $w_i \geq 0$ are weights. We follow the convention that $\int P(x)\,{\rm d}x$ 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 $N_{\rm grid}$ nodes $\{X_k\}$ [45]. We implemented two variants of basis set: linear B-splines and modified cubic B-splines with natural boundary conditions; in both cases the number of basis function $N_{\rm basis}$ is equal to the number of grid nodes. The estimated log-density is thus

$$\ln P(x;\, \boldsymbol{A}) = \sum_{k=1}^{N_{\rm basis}} A_k\, B_k(x) - \ln G_0 + \ln M \equiv Q(x; \boldsymbol{A}) - \ln G_0(\boldsymbol{A}) + \ln M, \qquad (17)$$

where A_k are the amplitudes – free parameters that are adjusted during the fit,
$B_k(x)$ are basis functions,

$Q(x;\boldsymbol{A}) \equiv \sum_k A_k\, B_k(x)$ is the weighted sum of basis function,
and $G_0(\boldsymbol{A}) \equiv \int \exp[Q(x;\boldsymbol{A})]\,\mathrm{d}x$ is the normalization constant determined from the condition that $\int P(x)\,\mathrm{d}x = M$. 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., $V_k = \sum_i w_i\, B_k(x_i)$), $\boldsymbol{W} \equiv \mathsf{B}^T\boldsymbol{w}$, $\mathsf{C} \equiv \mathsf{B}^T\,\mathsf{B}$. 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 a few dozen grid points in radius).

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

$$\begin{aligned}
\ln\mathcal{L} \;&\equiv\; \ln\mathcal{L}_\mathrm{data} - \lambda\mathcal{R}(\boldsymbol{A}) \;\equiv\; \sum_{i=1}^{N_\mathrm{data}} w_i \ln P(x_i;\,\boldsymbol{A}) - \lambda\int \left[\ln P''(x_i)\right]^2\,\mathrm{d}x \\
&= \sum_{i=1}^{N_\mathrm{data}} w_i \left(\sum_{k=1}^{N_\mathrm{ampl}} A_k B_k(x_i) - \ln G_0(\boldsymbol{A}) + \ln M\right) - \lambda \sum_{k=1}^{N_\mathrm{ampl}}\sum_{l=1}^{N_\mathrm{ampl}} A_k\, A_l\, R_{kl} \\
&= \left[\boldsymbol{V}^T\boldsymbol{A} - M\,\ln G_0(\boldsymbol{A}) + M\,\ln M\right] - \lambda\,\boldsymbol{A}^T\,\mathsf{R}\,\boldsymbol{A}, \qquad (18)
\end{aligned}$$

where $\lambda\mathcal{R}$ is the roughness penalty term, and the matrix $R_{kl} \equiv \int B_k''(x)\,B_l''(x)\,\mathrm{d}x$ is also pre-computed at the beginning of the procedure[16]. 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 $-\ln\mathcal{L}$ (18) are found by solving the system of equations $\partial\ln\mathcal{L}/\partial A_k = 0$ iteratively, using a multidimensional Newton

[16]It may be advantageous to use third derivatives here [62], in which case the solution in the limit of infinite smoothing ($\mathcal{R} \to 0$) corresponds to a Gaussian density profile; however, for our choice of *natural* cubic splines it is unattainable, since a quadratic function (logarithm of a Gaussian) has non-zero second derivatives at endpoints, and hence cannot be represented by a spline with natural boundary conditions.

method with explicit expressions for the gradient and hessian:

$$-\frac{\partial \ln \mathcal{L}}{\partial A_k} = -V_k + M\frac{\partial \ln G_0}{\partial A_k} + 2\lambda \sum_l R_{kl}\, A_l \ , \tag{19a}$$

$$-\frac{\mathrm{d}^2 \ln \mathcal{L}}{\partial A_k\, \partial A_l} = M\frac{\partial^2 \ln G_0}{\partial A_k\, \partial A_l} + 2\lambda R_{kl} \ \equiv H_{kl} \ , \tag{19b}$$

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

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

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

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.

$$\ln \mathcal{L}_{\mathrm{CV}}(\lambda) \equiv \sum_{i=1}^{N_{\mathrm{data}}} w_i\ \ln P(x_i;\ \boldsymbol{A}^{(i)}) = \sum_{i=1}^{N_{\mathrm{data}}} w_i \left(\sum_{k=1}^{N_{\mathrm{basis}}} A_k^{(i)} B_k(x_i) - \ln G_0(\boldsymbol{A}^{(i)}) \right) + M \ln M. \tag{20}$$

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 (19a) is zero for each i at the corresponding $\boldsymbol{A}^{(i)}$:

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

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

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 $\delta\mathsf{A}$ with N_{ampl} rows and N_{data} columns needs not be computed explicitly each time. Finally, the cross-validation score (20) is expressed as

$$\ln \mathcal{L}_{\mathrm{CV}}(\lambda) = \ln \mathcal{L}_{\mathrm{data}} - \mathrm{tr}(\mathsf{H}^{-1}\mathsf{C}) + \frac{\mathrm{d} \ln G_0(\boldsymbol{A})}{d\boldsymbol{A}}\, \mathsf{H}^{-1}\, \boldsymbol{W}. \tag{21}$$

Here $\ln \mathcal{L}_{\mathrm{data}}$ is the expression in brackets in (18). 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

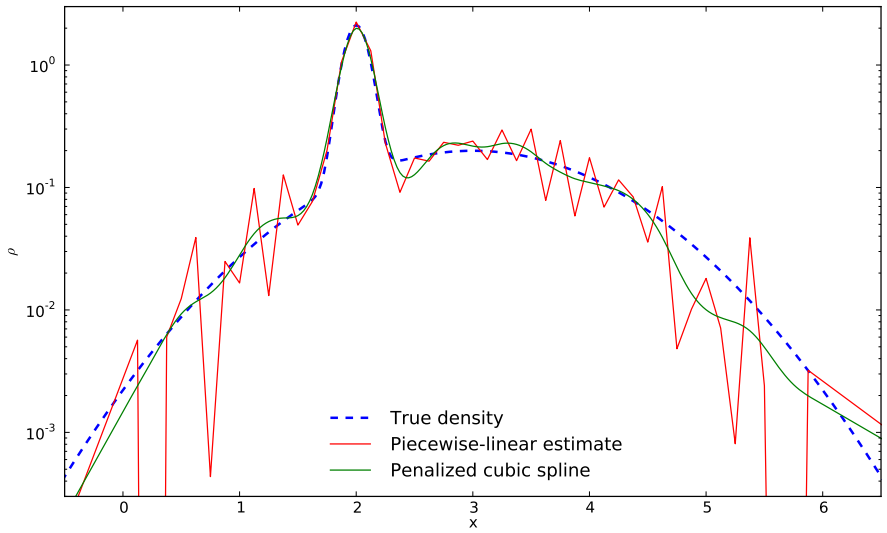

Figure 5: Penalized spline estimate of density from sample points. Here $N_{\rm data} = 1000$ were drawn from the original density profile (shown in dashed blue) described by a sum of two gaussians, with dispersions equal to 0.1 and 1. We reconstruct the logarithm of density using a linear ($N = 1$) 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.

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 5 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 the "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 $N_{\rm data}$, and allow the likelihood score to be worse than the best-fit score by an amount comparable to this expected scatter. In the case of uniform-weight samples and zero smoothing, the mean and dispersion in $\ln \mathcal{L}$ are

$$\left\langle \ln \mathcal{L} \right\rangle = \int P(x)\, \ln P(x)\, \mathrm{d}x = M\, \left[\frac{G_1}{G_0} + \ln M - \ln G_0 \right], \tag{22}$$

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

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

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.2.6 Gauss–Hermite series

Gauss–Hermite (GH) expansion is another type of basis set, useful for representing velocity distribution functions, which are typically not too dissimilar to a Gaussian profile. Unlike the B-spline basis set, which is specified by the number and location of grid nodes and the degree of polynomials, the GH basis is specified by the parameters of the zero-order function (a Gaussian with the given amplitude Ξ, mean value μ, and width σ) and the order of expansion M. The total number of basis functions is $M + 1$, and they are defined as

$$\mathcal{B}_m(x) \equiv \frac{\Xi}{\sqrt{2\pi}\, \sigma}\, \exp\left[-\frac{1}{2} \left(\frac{x - \mu}{\sigma} \right)^2 \right]\, \mathcal{H}_m\left(\frac{x - \mu}{\sigma} \right), \tag{23a}$$

where $\mathcal{H}_m(y)$ are the (astrophysical) Hermite polynomials [27, 70], defined by a recurrence relation

$$\mathcal{H}_0 = 1, \quad \mathcal{H}_1(y) = \sqrt{2}\, y, \quad \mathcal{H}_{m+1}(y) = \left[\sqrt{2}\, y\, \mathcal{H}_m(y) - \sqrt{m}\, \mathcal{H}_{m-1}(y) \right] \Big/ \sqrt{m+1}\,. \tag{23b}$$

They differ in normalization from the physicist's definition of Hermite polynomials $H(y)$: $\mathcal{H}_m(y) = H_m(y)/\sqrt{2^m\, m!}$. The GH basis functions are orthogonal on the real axis, with the

Gram matrix being

$$\mathcal{G}_{mn} \equiv \int_{-\infty}^{\infty} \mathcal{B}_m(x)\, \mathcal{B}_n(x)\, \mathrm{d}x = \frac{\Xi^2}{2\sqrt{\pi}\,\sigma}\, \delta_{mn}\,. \tag{24}$$

According to the general definition given in Section A.2.1, any function $f(x)$ can be approximated as a sum of GH basis functions multiplied by amplitudes $\boldsymbol{h}$ (also called GH coefficients):

$$\tilde{f}(x) = \sum_{m=0}^{M} h_m\, \mathcal{B}_m(x), \quad h_m = \frac{2\sqrt{\pi}\,\sigma}{\Xi^2} \int_{-\infty}^{\infty} f(x)\, \mathcal{B}_m(x)\, \mathrm{d}x\,. \tag{25}$$

For the fixed parameters Ξ, μ, σ, M, this expansion is defined in a unambiguous way. However, there is some redundancy (most obviously in the overall amplitude factor Ξ, which merely rescales the amplitudes $\boldsymbol{h}$). If one has freedom to adjust Ξ, μ and σ, it makes sense to define them in such a way that the first three coefficients are $h_0 = 1, h_1 = h_2 = 0$, which is always possible (more on this later). In this case, if one limits the order of expansion M to 2 (or, in fact, to 0), the approximated function $\tilde{f}(x)$ is actually the best-fit Gaussian for the original function $f(x)$. Addition of higher-order coefficients $h_m (m \geq 3)$ allows one to represent the deviations from this Gaussian, but does not change the first three coefficients, thanks to the orthogonality of the basis set.

On the other hand, if one needs to compare two functions represented by the GH series, this can only be done if the parameters of the basis set Ξ, μ, σ are the same. For instance, when the observational constraints on the velocity distribution come in the form of GH coefficients h_m (with the given Ξ, μ, σ and with the default values of $h_0 = 1, h_1 = h_2 = 0$), the same function in the model must be represented in the same basis, even though the coefficients $h_{0,1,2}$ do not have the default values.

In practice, it may be convenient to represent the velocity distribution of the model in terms of a B-spline expansion with amplitudes A_j, and then convert it into the GH coefficients h_m. This is, of course, a linear operation $\boldsymbol{h} = \mathcal{G}^{-1} \subset \boldsymbol{A}$ (for the given parameters of the two basis sets), with the matrix $C_{mj} = \langle \mathcal{B}_m, B_j \rangle$.

It is important to keep in mind that Ξ, μ, σ are *not* identical to the overall normalization, mean and dispersion of the function $\tilde{f}$. These are given by an amplitude-weighted sum over the corresponding values for all basis functions; for instance, the overall normalization is

$$\int_{-\infty}^{\infty} \tilde{f}(x)\, \mathrm{d}x = \sum_{m=0}^{M} h_m \int_{-\infty}^{\infty} \mathcal{B}_m(x)\, \mathrm{d}x = \Xi \sum_{m\ \mathrm{even}} h_m\, \frac{\sqrt{m!}}{m!!}\,, \tag{26a}$$

and the second moment times the above quantity is

$$\int_{-\infty}^{\infty} \tilde{f}(x)\, x^2\, \mathrm{d}x = \Xi \sum_{m\ \mathrm{even}} h_m\, \frac{(2m+1)\,\sqrt{m!}}{m!!}\,. \tag{26b}$$

As mentioned above, the GH expansion is defined by the amplitude Ξ, center μ and width σ of the zero-order function (a pure Gaussian). Given a sufficiently large number of terms M, it can approximate any reasonably smooth function $f(x)$ to a desired accuracy; however, the speed of convergence naturally depends on the choice of μ and σ. There are several possible approaches for setting these parameters.

1. van der Marel & Franx [70] choose the parameters μ, σ so that the lowest-order term approximates the function as closely as possible (this implies that only h_0 is nonzero, and hence the GH expansion is a pure Gaussian – note that the best-fit Gaussian is *not* a Gaussian with the width equal to the dispersion of the function!), then construct a full GH expansion with these parameters, computing the coefficients h_m according to (25). Thanks to the orthogonality of basis functions, the addition of subsequent terms does not change the previous expansion coefficients, hence in this (and *only in this*) case $h_1 = h_2 = 0$.

2. The same authors find it "more convenient in practice" to fit a truncated GH series $\mathscr{L}(x) \equiv \mathcal{B}_0(x) + \sum_{m=3}^{M} h_m \mathcal{B}_m(x)$ to the function $f(x)$ with $\Xi, \mu, \sigma, h_3 \dots h_M$ as free parameters adjusted during the fit to minimize the rms deviation between the function and its approximation, while still fixing $h_0 = 1, h_1 = h_2 = 0$. In this case, all parameters in the fit depend on the order of expansion M; in other words, this is the best approximation *at the given order*, not the approximation in which the lowest-order function is chosen to be the best-fit one. Naturally, this results in a better overall fit, but note that this is *not* a true GH expansion: if we compute the coefficients $h_{0,1,2}$ for the original function $f(x)$ with the parameters μ, σ having the best-fit values as described above, they will not have the values $1, 0, 0$ as implied during the fit, while the values of the remaining coefficients will be the same. Consequently, the actual GH expansion $\tilde{f}(x)$ constructed with these parameters will be somewhat less accurate than the fitted function $\mathscr{L}(x)$, but typically still more accurate than the GH expansion constructed around the best-fit Gaussian (as in the first approach). Since the fitted function $\mathscr{L}(x)$ has zero h_1, h_2, the best-fit values of μ, σ (and consequently all GH coefficients) will converge to the ones produced by the first method as $M \to \infty$, because only for this choice of σ the first two GH moments vanish.

3. Gerhard [27] independently introduced the GH parametrization of velocity profiles. In his approach, the scale parameter σ is not necessarily fixed to any particular value, hence h_2 (called s_2 in that paper) is not zero (his eq. 3.10). To avoid ambiguity, he suggests to use the true dispersion of the original function[17] $f(x)$ as the scale

[17] When fitting the noisy data, true dispersion is unknown a priori, so Gerhard suggests a two-step procedure: first perform a GH fit with the scale parameter σ set to some reasonable value (e.g. the width of the main peak of the function), then compute the true dispersion from this GH series, and re-fit another GH expansion with the scale set to the true dispersion. The total dispersion determined from the GH fit is less sensitive to the poorly measured wings of the profile. van der Marel & Franx caution that this procedure

parameter σ, but notices that sometimes a smaller value of σ could be preferred, e.g., when the function has a sharp and narrow peak. In common with the previous choice, all coefficients h_m are, in general, nonzero if σ is not equal to the width of the best-fit Gaussian.

4. However, if one dispenses with the constraint that $h_1 = h_2 = 0$, then a *still* better approximation (for a given order M) could be achieved by fitting an unrestricted GH series (25), simultaneously optimizing μ, σ and all $h_{m\geq 0}$.

Among these approaches, the first one (taking μ and σ to be the parameters of the best-fit Gaussian) is the only one that gives the same values for all h_m coefficients regardless of the order of expansion M (because its parameters are fixed and do not depend on M), and has $h_1 = h_2 = 0$ by construction. This makes it attractive from the conceptual point of view, because it can be "gracefully degraded" (truncated at a lower order while still producing a reasonable fit). In other approaches, μ, σ and all other coefficients depend on the choice of M, and because of two additional free parameters (h_1, h_2), the approximation is generally more accurate for a fixed M than in the first approach. Figure 6 illustrates the effect of different choices of σ on the approximation accuracy of the resulting GH series truncated at $M = 4$. The two examples shown in that figure are rather extreme, and in practice the difference is much smaller (and as mentioned before, vanishes for sufficiently high M, regardless of the choice of σ). We stress again that if the original function $f(x)$ is significantly non-Gaussian, one cannot interpret σ from the best-fit approximation as the "width" of the function, because the higher-order terms cannot be neglected. Moreover, there is no uniquely defined set of "true" GH parameters – different choices of σ will lead to different GH expansions, and it is not clear a priori which one will converge faster as the number of terms M increases (for the function shown on the right panel of the above figure, the optimal σ appears to be substantially smaller than either the width of the best-fit Gaussian or the true dispersion).

The above discussion assumed that we know perfectly the "true" original function. In practice, the GH expansion is often used to parametrize the velocity distribution functions extracted from or fitted to the spectra, which are often both noisy and undersampled. [15] find that the first approach in the above list (fitting the data by a Gaussian with free parameters μ and σ first and then using it to construct a GH expansion) becomes biased towards a pure Gaussian when the data is undersampled, even in the limit of zero noise, while the second approach (fitting a function $\mathscr{L}(x)$ with $\mu, \sigma, h_{m\geq 3}$) produces large and correlated uncertainties when the signal-to-noise ratio (SNR) is low. They design a hybrid method (the pPXF code) which uses $\mathscr{L}(x)$ at high SNR, but degenerates into fitting a pure Gaussian at low SNR. As explained above, the values of σ (and hence all GH coefficients) produced by the second approach will tend to those of the first approach as M increases (in the limit of

could underestimate of the dispersion if the truncated GH expansion has significant negative wings, and suggest to take $\max\big(\tilde{f}(x), 0\big)$ when computing the dispersion according to Equation 26b.

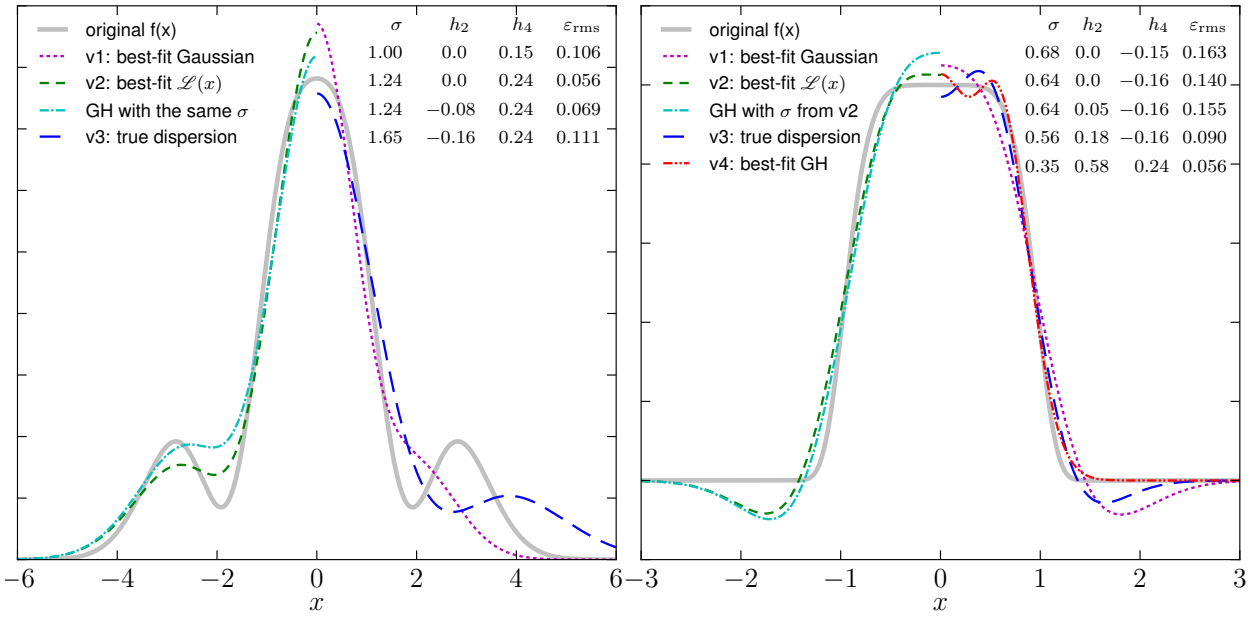

Figure 6: Gauss–Hermite expansions with $M = 4$ and different choices of σ.
Left panel: $f(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right) \left[1 + 0.15\,\mathcal{H}_4(x) + 0.2\,\mathcal{H}_6(x)\right]$ (same as in Appendix A of [36]) is actually a pure GH expansion with 6 terms, but we fit it with only 4 terms.
Right panel: $f(x) = \exp(-x^6)$ is a slightly smoothed top-hat function.
Solid gray line shows the original function, and other lines show various approximations (each one is plotted for either positive or negative x only, to reduce congestion):
Purple dotted line: GH expansion with the scale σ equal to that of the best-fit Gaussian (variant 1 in the list) has $h_2 = 0$ by construction, and recovers the true value of h_4 on the left panel, but is not the best fit by itself.
Dashed green line: best-fit function $\mathscr{L}(x)$ which looks like a GH expansion with $h_2 = 0$ and adjustable σ, h_4 (variant 2 in the list). It is not the *true* GH expansion of the function $f(x)$ with this scale σ, however: the latter is shown by dot-dashed cyan line, and has non-zero h_2.
Long-dashed blue line: GH expansion with the scale σ equal to the true dispersion of the original function (variant 3 in the list) also has non-zero h_2.
Dash-double-dotted red line: the absolute best-fit GH expansion with σ, h_2 and h_4 all being free parameters, which has the smallest deviation from the original function (not shown on the left panel because it is very similar to the variant 1).

well-sampled and noiseless data). Alternatively, one may use an entirely different method for extracting the velocity distribution from the spectrum, and only then fit a GH expansion to the resulting function. [36] use a nonparametric maximum penalized likelihood method to determine $f(x)$ and then follow the first approach (determine σ from the best-fit Gaussian) to construct a GH expansion $\tilde{f}(x)$.

A.2.7 Sampling

The sampling routine performs the following task: given an D-dimensional function $f(\boldsymbol{x}) \geq 0$ defined in a rectangular domain (without loss of generality may take it to be a unit hypercube $[0..1]^D$), generate an array of N points $\boldsymbol{x}_i$ such that their density in the vicinity of any point $\boldsymbol{x}$ is proportional to $f(\boldsymbol{x})$. In other words, f is interpreted as a probability distribution function and is sampled with equal-weight samples; in fact the integral $\int f(\boldsymbol{x})\,\mathrm{d}^D x = A$ needs not be unity, and is itself estimated by the routine.

We employ an adaptive rejection sampling method. The entire domain $\mathcal{V}$ is partitioned hierarchically into smaller hyperrectangles (cells) $\mathcal{V}_c$, an envelope function $\bar{f}_c$ (constant in each cell) is constructed, and a rejection sampling is used to draw output points from each cell. This requires several (up to 10–20) iterations, during each one this partitioning is further refined in regions with where the function values are largest, and more trial points are drawn. The cells are organized into a tree structure, where each non-leaf cell is split into two equal-volume child cells along some dimension. Each leaf cell keeps the list of trial points $\boldsymbol{x}_{c,k} \in \mathcal{V}_c$ that belong to this cell; when a cell is split in two halves, these points are distributed among the child cells. The procedure, illustrated in Figure 7, can be summarized as follows:

0. We start with only one cell $\mathcal{V}_1$ covering the entire domain, and sprinkle M_1 sampling points $\boldsymbol{x}_k$ uniformly in this cell. At this stage, the estimate of the integral is just

$$A = \frac{\mathrm{vol}(\mathcal{V}_1)}{M_1} \sum_{k=1}^{M_1} f(\boldsymbol{x}_k). \tag{27}$$

1. At each iteration, we first loop over all leaf cells $\mathcal{V}_c$ in the tree (those that are not split into child cells), and consider all M_c trial points $\boldsymbol{x}_{c,k}$ belonging to $\mathcal{V}_c$. The integral A is estimated as

$$A = \sum_{c=1}^{N_\mathrm{cell}} \frac{\mathrm{vol}(\mathcal{V}_c)}{M_c} \sum_{k=1}^{M_c} f(\boldsymbol{x}_{c,k}). \tag{28}$$

2. We then perform another scan over all leaf cells and check whether they contain enough trial points for the rejection sampling procedure. A trial point $\boldsymbol{x}_{c,k}$ can be selected as one of N output samples with probability $f(\boldsymbol{x}_{c,k})/\bar{f}_c$, where $\bar{f}_c \equiv \frac{A}{N}\frac{M_c}{\mathrm{vol}(\mathcal{V}_c)}$ is the

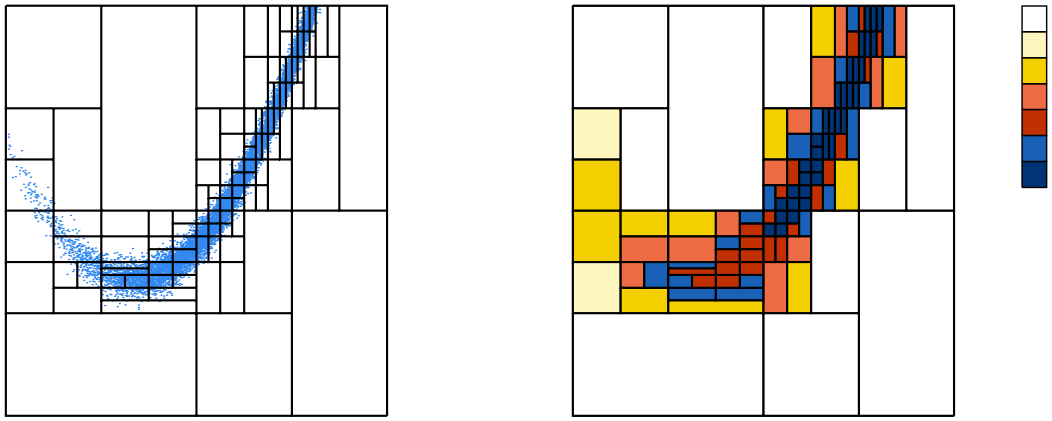

Figure 7: Illustration of the adaptive rejection sampling algorithm in the domain $[-1..2]^2$, recursively partitioned into ~ 100 rectangular cells. Left panel shows 10^4 points sampled from $f(x,y) = \exp[-R(x,y)]$, where $R(x,y) \equiv (1-x)^2 - 100(y-x^2)^2$ is the Rosenbrock function. In the right panel cells are colored according to the density of trial points (successive shades are $2\times$ denser).

"envelope" function, constant across the cell; clearly we need $\bar{f}_c$ to be larger than the maximum value of $f(\boldsymbol{x} \in \mathcal{V}_c)$. If this condition is not satisfied, the cell is scheduled for refinement.

3. For each such cell that needs to be refined, we have two choices: either to add more trial points into the entire cell (thus increasing M_c and hence pushing up the envelope function), or first split the cell in two halves and consider both of them in turn. There is a lower limit $M_{\rm min}$ on the number of trial points in a cell, and splitting is possible only if child cells would contain at least that many points. We examine all possible dimensions along which the cell could be split, and choose the one with the lowest entropy (i.e. the function varies more significantly along this dimension). The two new child cells inherit the parent cell's trial points, are added to the end of the list and will be scanned in turn during the same pass (at least one of them will need to be further refined). If the cell contains less than $2M_{\rm min}$ trial points and hence cannot be split, we double the number of trial points M_c in it (add the same number of new points uniformly distributed in the cell), so that it could be split (if necessary) on the next iteration.
4. If we have added new trial points into any cell, we repeat the procedure from step 1.
5. Otherwise the iterative refinement is complete, and we draw N output points from the trial points, with the probability described in step 2, and finish.

A.3 Coordinates

The coordinate transformation subsystem in AGAMA consists of several concepts. First, we define several commonly used coordinate systems (or *representations* in the terminology of ASTROPY): Cartesian, Cylindrical, Spherical, and Prolate Spheroidal. The same point in 3d space can be represented in any of these coordinate systems (having a common center and orientation). There are routines for transforming coordinates, velocities, gradients and hessians of scalar functions between these representations.

Second, we define rotations of coordinate basis. Consider a right-handed Cartesian reference frame defined by basis vectors (axes) $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$, and a rotated frame with the same origin defined by basis vectors $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$. The relative orientation of two frames can be specified by Euler angles (see Figure 8). The same point has coordinates x, y, z in the original frame and X, Y, Z in the rotated frame. The transformation between these coordinates (passive rotation) is given by the following orthogonal rotation matrix R, encapsulated in a `C++` class `coord::Orientation` or produced by the Python routine `makeRotationMatrix`:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \mathsf{R} \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \qquad \mathsf{R} \equiv \begin{pmatrix} c_\alpha c_\gamma - s_\alpha c_\beta s_\gamma & s_\alpha c_\gamma + c_\alpha c_\beta s_\gamma & s_\beta s_\gamma \\ -c_\alpha s_\gamma - s_\alpha c_\beta c_\gamma & -s_\alpha s_\gamma + c_\alpha c_\beta c_\gamma & s_\beta c_\gamma \\ s_\alpha s_\beta & -c_\alpha s_\beta & c_\beta \end{pmatrix}, \tag{29}$$

where we used $c_\circ, s_\circ$ as shortcuts for $\cos\circ, \sin\circ$. The inverse transformation is described by the same angles, but with negative signs and in reverse order: $-\gamma, -\beta, -\alpha$, and its rotation matrix is $\mathsf{R}^{-1} = \mathsf{R}^T$. The rotation matrix is invariant under the simultaneous substitution $\beta \to -\beta, \alpha \to \alpha+\pi, \gamma \to \gamma+\pi$; hence it is convenient to restrict the range of β to $[0, \pi]$ and the other two angles – to $(-\pi, \pi]$.

In the case of a triplanar symmetry (invariance under reflection about any of the three principal planes, or equivalently under change of sign of any coordinate), one may simultaneously change the sign of any two coordinates while preserving the right-handedness of the coordinate system, and this will not change any physical property of the system. These simultaneous sign changes and associated transformations of Euler angles are listed in the table on the right.

x	y	z	angles		
			α	β	γ
+	+	+	$\pi+\alpha$	$-\beta$	$\pi+\gamma$
			$\pi-\alpha$	$\pi-\beta$	$\pi+\gamma$
+	−	−	$-\alpha$	$\pi+\beta$	γ
			$-\alpha$	$\pi-\beta$	$\pi+\gamma$
−	+	−	$\pi-\alpha$	$\pi+\beta$	γ
			$\pi+\alpha$	β	γ
−	−	+	α	$-\beta$	$\pi+\gamma$

This kind of transformation is used to convert between the intrinsic coordinates of the stellar system (original frame $\boldsymbol{xyz}$) and the observed coordinates (rotated frame $\boldsymbol{XYZ}$), where the $\boldsymbol{Z}$ axis points along the line of sight away from the observer, the $\boldsymbol{Y}$ axis points upward / north in the image plane, and the $\boldsymbol{X}$ axis – leftward (!) / east in the image plane. Note that this creates a rather awkward situation that the $\boldsymbol{X}$ axis is directed opposite to the usual rightward orientation in conventional two-dimensional plots: this results from the unfortunate fact that the observer sits "inside" the celestial sphere. This transformation is often referred to as projection onto the sky or image plane XY. As another example

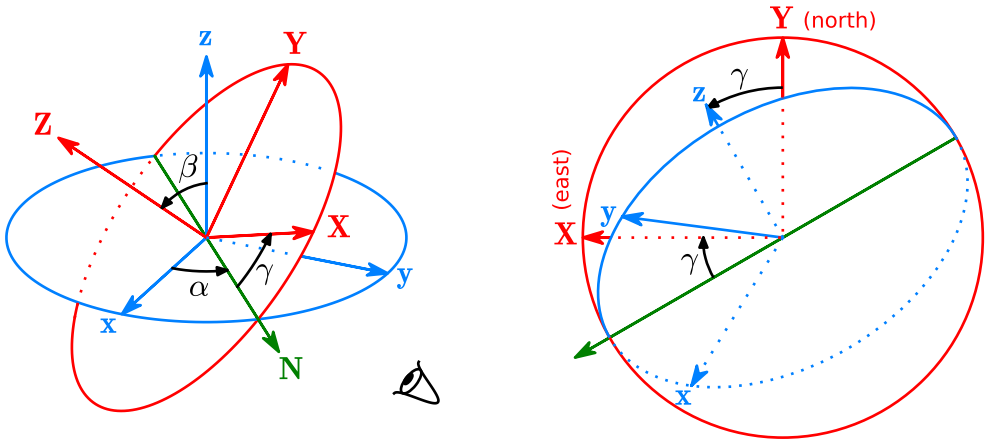

Figure 8: *Left panel:* specification of rotation of the cartesian coordinate system in terms of Euler angles α, β, γ.
Let the original reference frame be specified by the right-handed triplet of basis vectors (axes) $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$, and the rotated frame – by basis vectors $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$. The first rotation of the xy plane (blue) by angle α about the $\boldsymbol{z}$ axis creates an intermediate basis $\boldsymbol{x}', \boldsymbol{y}', \boldsymbol{z}'$, where the axis $\boldsymbol{x}'$ points along the line of nodes of the overall transformation (denoted by the green arrow N). The second rotation by angle β about the $\boldsymbol{x}'$ axis (line of nodes) tilts the $x'y'$ plane by angle β, creating a second intermediate basis $\boldsymbol{x}'', \boldsymbol{y}'', \boldsymbol{z}''$; the angle between basis vectors $\boldsymbol{z}''$ and $\boldsymbol{z}$ is also β. The third rotation of the $x''y''$ plane (red) by angle γ about the $\boldsymbol{z}''$ axis does not change the direction of that axis, hence the final axis $\boldsymbol{Z}$ is the same as $\boldsymbol{z}''$. The same scheme is used to specify the `orientation` of the `Tilted` potential modifier (Section 2.2.5). In this case, when evaluating the modified potential $\widetilde{\Phi}$ or its derivatives at a point in the blue (lowercase) frame, the input point is transformed into the rotated (red, uppercase) frame, fed into the underlying potential (whose native reference frame is red), and the result is transformed back into the lowercase frame.
Right panel: the same two reference frames as in the left panel, viewed in the image plane XY. The $\boldsymbol{Z}$ axis points perpendicular to the plane away from the observer; the $\boldsymbol{Y}$ axis points up / north, and the $\boldsymbol{X}$ axis – left / east. The equatorial plane xy of the intrinsic coordinate system is shown by blue ellipse (dashed when it is behind the image plane). Its intersection with the image plane is the line of nodes, marked by the green arrow. The angle of clockwise rotation of the $\boldsymbol{X}$ axis w.r.t. the line of nodes is γ (the last of the three rotations). The position angle (PA) in the XY plane is measured from the $\boldsymbol{Y}$ axis counter-clockwise (towards the $\boldsymbol{X}$ axis), hence the PA of the projection of $\boldsymbol{z}$ axis onto the image plane is also γ, and the PA of the line of nodes is $\gamma + \pi/2$.

of wrecked but venerable astronomical convention, position angles (PA) in the XY plane are measured from the $\boldsymbol{Y}$ (north) direction towards $\boldsymbol{X}$ (east), i.e., counter-clockwise in this configuration; hence the PA of $\boldsymbol{X}$ is $\pi/2$. In some studies, the orientation of the rotated frame is given by the spherical polar angles θ, ϕ of the line of sight $\boldsymbol{Z}$ in the intrinsic frame; in this convention, $\alpha = \phi + \pi/2$, $\beta = \theta$.

If the shape of an object in the intrinsic reference frame is a triaxial ellipsoid with the major axis $a\boldsymbol{x}$, intermediate axis $b\boldsymbol{y}$ and minor axis $c\boldsymbol{z}$, then the projection along the major axis corresponds to $\{\alpha, \beta\} = \{\pm\pi/2, \pi/2\}$, along the intermediate axis – to $\{0, \pi/2\}$ or $\pi, \pi/2$, and along the minor axis – to $\beta = 0$ and any α. In a general case, β is the inclination angle (0 is "face-on" orientation and $\pi/2$ is "edge-on"), and γ is the PA of the projection of the intrinsic minor axis $\boldsymbol{z}$ onto the sky plane. In the case of an axisymmetric system, the first rotation (α) does not change anything in its properties, hence we may set $\alpha = 0$. Then the intrinsic major axis $\boldsymbol{x}$ coincides with the line of nodes, and its PA is $\gamma + \pi/2$. In a general case, the projection of a triaxial ellipsoid is also an ellipse in the image plane, but its major and minor axes A, B and orientation (PA η) are related to the intrinsic shape a, b, c and viewing angles α, β, γ in a rather complicated way. In particular, the major axis of the ellipse may not be aligned with any of the three projected principal axes. There are routines `getProjectedEllipse`, `getIntrinsicShape` and `getViewingAngles` for computing one of these triplets from the other two, but note that the deprojection (determination of either the intrinsic shape or the viewing angles) is not always possible. For an axisymmetric system, there are two possible choices of viewing angles, depending on which side of the system is nearer, and for a triaxial system, there are four possible combinations of viewing angles.

In case that the distance to the stellar system is not overwhelmingly larger than its characteristic size (or equivalently, when it occupies a non-negligible area on the sky), one needs to employ more sophisticated transformations between the cartesian intrinsic coordinates within the stellar system and the spherical coordinates on the sky (no longer "sky plane"). An extreme case is our Galaxy, where the observer is located well within the stellar system, and it occupies the entire sky. Also in this case, we may need to consider shifts, not only rotations of the coordinate systems. The projection transformations will be extended to these cases in the future.

A.4 Potentials

A.4.1 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:

$$\Phi(r,\theta,\phi) = \sum_{l=0}^{l_{\max}} \sum_{m=-m_0(l)}^{m_0(l)} \Phi_{l,m}(r)\,\sqrt{4\pi}\tilde{P}_l^m(\cos\theta)\;\mathrm{trig}\,m\phi, \tag{30}$$

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

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, $l_{\max}$ is the order of expansion in meridional angle θ, and $m_0(l) = \min(l, m_{\max})$, where $m_{\max} \le l_{\max}$ 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 $m_{\max}=2$ may be sufficient, while $l_{\max}$ may be set to 8 or 10). 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 suitably scaled functions in log-scaled radius. The logarithmic transformation of radius is intended to attain high dynamic range with a moderate number (a few dozen) of grid points, while the transformation of amplitude of each term increases the accuracy of interpolation. These amplitude transformations are used only when the $l=0$ harmonic of the potential is everywhere negative (which is usually the case). The main $l=0$ term is log-scaled (i.e., the actual function to be interpolated is $\ln\left[1/\Phi(0) - 1/\Phi_{0,0}(\ln r)\right]$, where $\Phi(0)$ is the value of potential at origin, which may be finite or $-\infty$). The other terms are normalized to the value of the $l=0$ term (i.e., the spline is constructed for $[\Phi_{l,m}/\Phi_{0,0}](\ln r)$). Each term is interpolated as a quintic spline, 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 we locate the radius at which the logarithmic curvature of the spherically-symmetric part of the density profile ($d^2\ln(\rho_{0,0})/d(\ln r)^2$), weighted by the mass at the given radius ($\propto r^2\rho$), reaches the maximum. For most finite-mass models, this would be the near the half-mass radius, but even for models with infinite mass (such as NFW) this criterion still estimates the "radius of interest" quite well. This fiducial radius $r_\star$ is taken as the center of the logarithmic grid, a suitable grid spacing factor f is assigned, and the grid is extended both ways from this radius: $r_{\max/\min} = r_\star\, f^{\pm N_R/2}$. As N_R gets larger, both the dynamical range $D \equiv r_{\max}/r_{\min}$ 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 $f \sim 2$. If

the input density drops to zero beyond some radius, the upper extent of the grid is moved inward to this radius, Likewise, if the density is declining towards small r, the potential has a very flat core, so that the inner grid point is shifted up to a "safe" radius $r_{\rm min}$ at which the potential is sufficiently different from $\Phi(0)$ (at least in the last few digits), to prevent the loss of precision of floating-point numbers. This automatic algorithm gives reasonable results in vast majority of cases, but if necessary, the min/max radii may be provided by the user.

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; Table 5 lists the most common cases. All possible combinations of symmetries are encoded in the `coord::SymmetryType` class, and each one corresponds to a certain combination of non-trivial spherical-harmonic terms (`math::SphHarmIndices`), as described in `math_sphharm.h`. 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 by 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 $l_{\rm max}>2$. In this variant, the main log-scaled term is the 2d spline for the $m=0$ component, while the other azimuthal harmonics are normalized to the value of the main term (again, these scaling transformations are turned off if the $m=0$ term is non-negative or changes sign).

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):

$$\Phi_{l,m}(r) = -\frac{4\pi\,G}{2l+1}\left[r^{-l-1}\int_0^r \rho_{l,m}(r')\,{r'}^{l+2}\,\mathrm{d}r' + r^l\int_r^\infty \rho_{l,m}(r')\,{r'}^{1-l}\,\mathrm{d}r'\right], \tag{31}$$

$$\rho_{l,m}(r) \equiv \frac{1}{\sqrt{4\pi}}\int_{-1}^{1}\mathrm{d}\cos\theta\,\tilde{P}_l^m(\cos\theta)\int_0^{2\pi}\mathrm{d}\phi\ \mathrm{trig}\,m\phi\,\rho(r,\theta,\phi). \tag{32}$$

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 2.6.3) 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.2.5) 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.2.4) 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 (31,32) can be solved directly by summing over particles (the approach used in [71]), this results in a noisier and less accurate potential than the intermediate smoothed density can provide.

A.4.2 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 amplitudes of each interpolated term are also transformed in the same way as for the `Multipole` potential (for the same purpose – improving the accuracy of interpolation, and only when the $m=0$ term is everywhere negative), namely, the main $m=0$ term uses log-scaling of its amplitude, and the remaining ones are normalized to the value of the main term. 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 (currently $l_{\max}=8$) 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 9, 10).

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 $\gtrsim 10^3\, r_{\rm half-mass}$. The automatic procedure is somewhat less optimal than in case of Multipole, so it may be advisable to set up the grid manually, with two considerations in mind. First, the inner grid point should be comparable with the smallest scale of variation of the density profile (e.g., in the case of a thin disk, $z_{\min} \simeq$ `scaleHeight`), but not much smaller, because the relative difference between the potential at adjacent grid points should exceed the accuracy of its computation ($\sim 10^{-6}$), or else the high-order interpolation scheme greatly amplifies the errors in its derivatives. Second, the outer edge of the grid should be far enough from the region where the density is concentrated, so that the extrapolation outside the grid using only a few spherical-harmonic terms is accurate enough. The potential and its derivatives are discontinuous at the grid boundary, and it's important to keep this discontinuity at a negligible level.

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 `DensityAzimuthalHarmonic` 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 [18]:

$$\Phi_m(R,z) = -G \int_{-\infty}^{+\infty} \mathrm{d}z' \int_0^{\infty} \mathrm{d}R'\, 2\pi R'\, \rho_m(R',z')\, \Xi_m(R,z,R',z') \;, \tag{33}$$

$$\Xi_m \equiv \int_0^{\infty} \mathrm{d}k\, J_m(kR)\, J_m(kR')\, \exp(-k|z-z'|) \;, \quad \text{which evaluates to} \tag{34}$$

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

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

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

$$\Phi_m(R,z) = -G \sum_k m_k\, \Xi_m(R,z,R_k,z_k)\, \mathrm{trig}\, m\phi_k. \tag{35}$$

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 (33) or a sum over all particles in (35). 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). 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 $m \ge 0$ are used, etc.); and for models with z-reflection symmetry, coefficients are computed and stored only for the $z \ge 0$ half-space.

Figure 9 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 10 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.

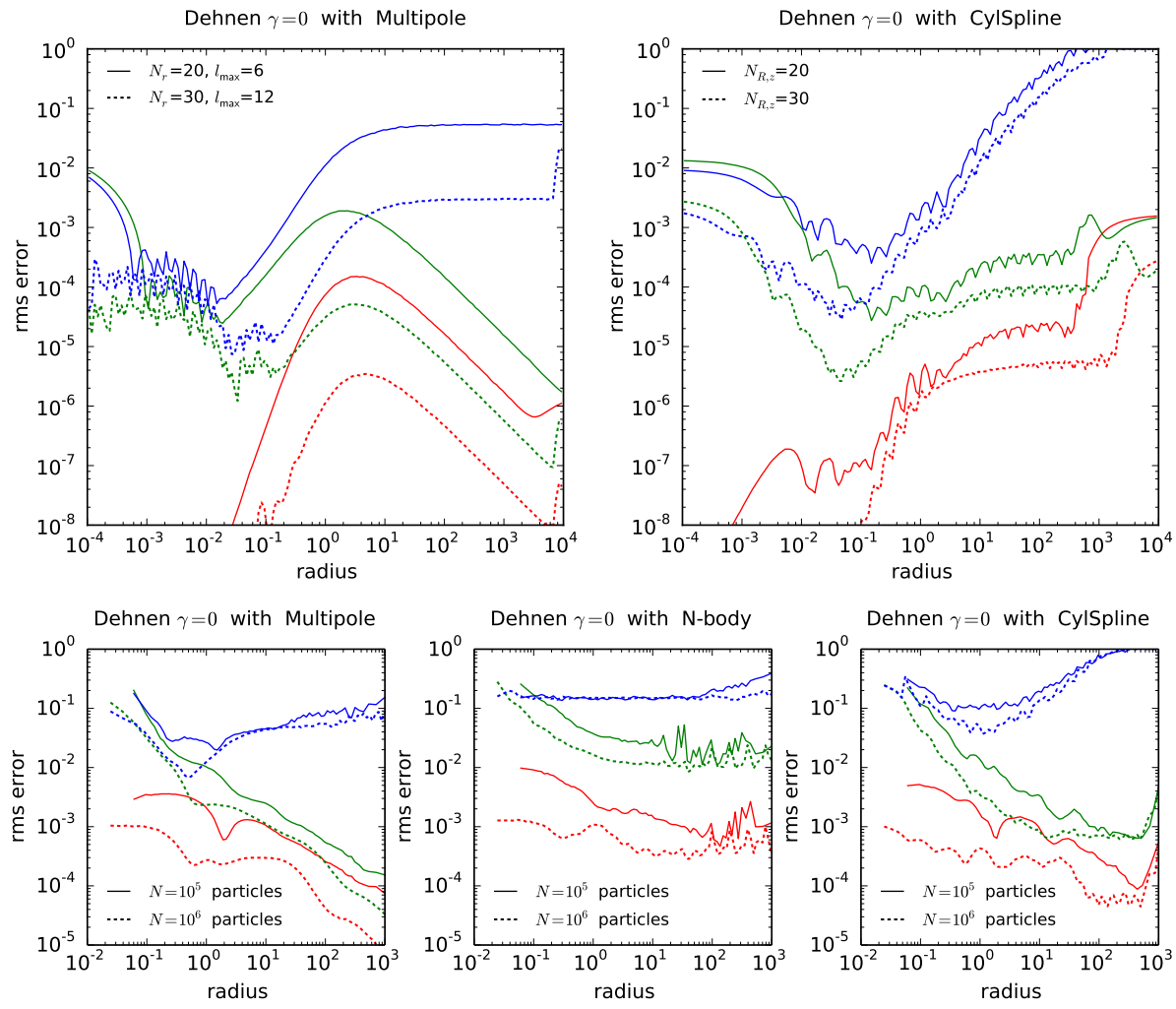

Figure 9: Accuracy of potential approximations in the case of initialization from a smooth density profile (top panels) and from an array of N particles (bottom panels). In both cases we compare the potential (red), force (green) and density (blue) computed using the potential expansions (left: Multipole, 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.

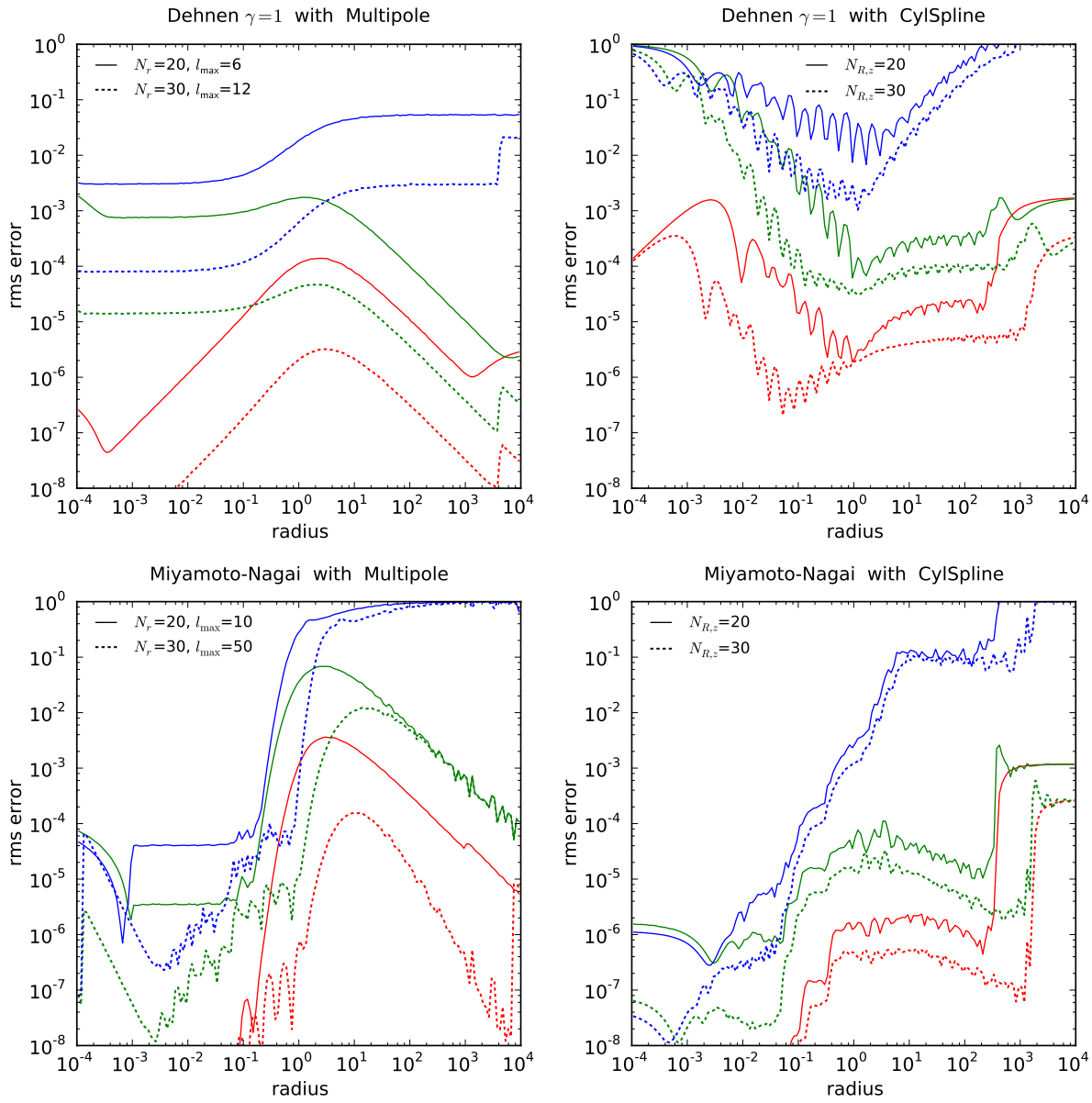

Figure 10: Accuracy of potential approximations for different types of density profiles. As in the previous figure, we plot the relative errors in potential (red), force (green) and density (blue) for Multipole (left) and CylSpline (right) potential expansions. Top panels are for a triaxial $\gamma = 1$ 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.

A separate class `DensityAzimuthalHarmonic` 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 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.5 Orbit integration and the variational equation

The orbit integration can be performed in all three standard coordinate systems (Cartesian, Cylindrical and Spherical), although only the Cartesian system is used in practice (in particular, by the `orbit` routine from the `Python` interface). In the reference frame that rotates with an angular frequency $\boldsymbol{\Omega}$, the Hamiltonian equations of motion for $\boldsymbol{w} \equiv \{\boldsymbol{x}, \boldsymbol{v}\}$ are

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{v} - \boldsymbol{\Omega} \times \boldsymbol{x}, \tag{36a}$$

$$\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = -\frac{\partial \Phi(\boldsymbol{x})}{\partial \boldsymbol{x}} - \boldsymbol{\Omega} \times \boldsymbol{v}. \tag{36b}$$

This ODE system is integrated using one of the available methods derived from the base class `math::BaseOdeSolver`, namely the 8th order Runge–Kutta with adaptive timestep (Dormand–Prince, DOP853) [31] and the 4th order Hermite method [38]. Other possibilities previously implemented in [71] include 15th order Gauss–Radau scheme [55], and several methods from ODEINT package [1], including Bulirsch–Stoer and various Runge–Kutta schemes. However, in practice all of them have quite similar performance in the appropriate range of tolerance parameters, thus we have only kept DOP853 and Hermite at the moment.

The DOP853 method has been slightly modified as follows. In the original method, each timestep has 12 Runge–Kutta stages (thus evaluates the forces that many times), plus another 3 force evaluations are needed to construct a 8th-order interpolator for the solution within the current timestep (so-called dense output feature). In our modification, these three evaluations are eliminated, at the expense of degrading the interpolation to 7th order, which is more than sufficient in practice. This reduces the number of force evaluations by 20% per accepted step. However, in the adaptive-timestep integrator, a significant fraction of timesteps ($\sim 1/3$) are rejected and repeated with a reduced step; rejected steps still cost 11 force evaluations, but would not incur the 3 additional evaluations for the dense output in the original method, so the total savings are closer to 15%.

The Hermite method needs only two evaluations per timestep, but uses both the gradient and the hessian of the potential, and the timestep is usually shorter than in the DOP853 method. Nevertheless, it may be more efficient if the required accuracy is not very tight ($\gtrsim 10^{-4}$, note that the default value is 10^{-8}).

The variational equation describes the evolution of infinitesimally small perturbations of initial conditions $\delta\boldsymbol{w} \equiv \{\delta\boldsymbol{x}, \delta\boldsymbol{v}\}$ along the orbit:

$$\frac{\mathrm{d}\,\delta\boldsymbol{x}}{\mathrm{d}t} = \delta\boldsymbol{v} - \boldsymbol{\Omega} \times \delta\boldsymbol{x}, \tag{37a}$$

$$\frac{\mathrm{d}\,\delta\boldsymbol{v}}{\mathrm{d}t} = -\frac{\partial^2\Phi(\boldsymbol{x})}{\partial\boldsymbol{x}^2}\,\delta\boldsymbol{x} - \boldsymbol{\Omega} \times \delta\boldsymbol{v}. \tag{37b}$$

We note that these equations describe a Hamiltonian system with a simple quadratic but time-dependent Hamiltonian (indirectly through the original trajectory):

$$\hat{H}(\delta\boldsymbol{x}, \delta\boldsymbol{v}, t) = \frac{1}{2}\delta\boldsymbol{x}^T \left.\frac{\partial^2\Phi(\boldsymbol{x})}{\partial\boldsymbol{x}^2}\right|_{\boldsymbol{x}=\boldsymbol{x}(t)} \delta\boldsymbol{x} + \frac{1}{2}\delta\boldsymbol{v}^2. \tag{37c}$$

Alternatively, they can be expressed as a second-order linear ODE with variable coefficients for $\delta\boldsymbol{x}$:

$$\frac{\mathrm{d}^2\delta\boldsymbol{x}(t)}{\mathrm{d}t^2} = \mathsf{A}(\mathsf{t})\,\delta\boldsymbol{x}(t) + \mathsf{B}(\mathsf{t})\,\frac{\mathrm{d}\,\delta\boldsymbol{x}(t)}{\mathrm{d}t}, \tag{37d}$$

where the matrices A and B, in the standard case of rotation about z axis $\boldsymbol{\Omega} \equiv \{0, 0, \Omega\}$, are

$$A_{ij}(t) = -\left.\frac{\partial^2\Phi(\boldsymbol{x})}{\partial x_i\,\partial x_j}\right|_{\boldsymbol{x}=\boldsymbol{x}(t)} - \begin{pmatrix} \Omega^2 & 0 & 0 \\ 0 & \Omega^2 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \qquad B_{ij} = \begin{pmatrix} 0 & 2\Omega & 0 \\ -2\Omega & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{37e}$$

Although one may follow the evolution of the deviation vector $\delta\boldsymbol{w}$ simultaneously with the trajectory $\boldsymbol{w}$ itself, using the same ODE integrator extended to 12 dimensions, this has two disadvantages: first, it makes no use of the linear character of the variational equation, and second, it would change the timestepping of the ODE solver compared to the case of integrating just the trajectory. If the orbit is chaotic, it will eventually look very different when integrated with or without the deviation vector, which is undesirable. Thus for the variational equation, we use an independent ODE solver derived from the base class `math::BaseOde2Solver`, which is specialized to the linear second-order ODE case. The main ODE solver integrates only the orbit and provides a high-accuracy interpolated solution $\boldsymbol{x}(t)$ within the current timestep, and the attached variational equation solver uses it to evolve $\delta\boldsymbol{x}$ on the same timestep.

There are two implementations based on the same principle, but with a different order: 6th-order method with 3 Hessian evaluations per timestep, and 8th-order method with 4 evaluations. In the 6th-order method, the second time derivative of $\delta\boldsymbol{x}$ is approximated by a quadratic polynomial on the current timestep $t_s..t_{s+1}$, using a normalized time variable $h \equiv (t - t_s)/(t_{s+1} - t_s)$:

$$\ddot{\delta\boldsymbol{x}}(h) \equiv \frac{\mathrm{d}^2\delta\boldsymbol{x}}{\mathrm{d}t^2} = \boldsymbol{c}_0 + (h - h_0)\big(\boldsymbol{c}_1 + (h - h_1)\,\boldsymbol{c}_2\big), \tag{38a}$$

and therefore $\delta\dot{\boldsymbol{x}}(h)$ and $\delta\boldsymbol{x}(h)$ are respectively third- and fourth-order polynomials in h. Here h_k are some fixed time moments within the current timestep, and the coefficients $\boldsymbol{c}_k$ are to be determined. First, we record the $D \times D$ matrices $\mathsf{A}(t)$ and $\mathsf{B}(t)$ at times $t_s + (t_{s+1} - t_s)\, h_k$, $k = 0, 1, 2$ – this is the only time the original trajectory is referenced and the potential derivatives are evaluated. We then recall that for each h_k,

$$\delta\ddot{x}_i(h_k) = \sum\nolimits_{j=1}^{D} \left[A_{ij}(h_k)\, \delta x_j(h_k) + B_{ij}(h_k)\, \delta\dot{x}_j(h_k) \right], \tag{38b}$$

where both left- and right-hand sides are linear combinations of the unknown coefficients $\boldsymbol{c}_k$. This linear algebraic system is most easily solved by iterations. At each iteration, we compute the values of $\delta\boldsymbol{x}(h_0)$ and $\delta\dot{\boldsymbol{x}}(h_0)$ on the right-hand side of (38b), using the current values of $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2$. From (38a) we see that this provides the updated value for $\boldsymbol{c}_0$ only, since the other terms in brackets are zero for $h = h_0$. Then we use the newly computed value of $\boldsymbol{c}_0$ and the values of $\boldsymbol{c}_1, \boldsymbol{c}_2$ from the previous iteration to construct $\delta\boldsymbol{x}(h_1)$ and $\delta\dot{\boldsymbol{x}}(h_1)$ on the right-hand side of (38b), and then update $\boldsymbol{c}_1$ from (38a) as $\boldsymbol{c}_1 = (\delta\ddot{\boldsymbol{x}}(h_1) - \boldsymbol{c}_0)/(h_1 - h_0)$. Finally, the same procedure is repeated for the point h_2 to update $\boldsymbol{c}_2$. The whole sequence is then iterated several times; we stress again that the matrices $\mathsf{A}(h_k)$ and $\mathsf{B}(h_k)$, which contain the expensive-to-compute Hessian of the potential, are fixed throughout this process. Finally, after the coefficients $\boldsymbol{c}_k$ have been determined with sufficient precision, the values of $\delta\boldsymbol{x}$ and $\delta\dot{\boldsymbol{x}}$ at $h = 1$ (end of the timestep) are computed and stored.

By choosing the three intermediate points h_k to be the nodes of the Gauss–Legendre quadrature, the whole scheme provides 6th order accuracy for $\delta\boldsymbol{x}$ at the timestep boundaries, while the interpolated solution within each timestep has 5th order accuracy for $\delta\boldsymbol{x}$ and 4th order for $\delta\dot{\boldsymbol{x}}$. The same approach is used to construct a scheme with four intermediate points h_k per timestep, in which $\delta\ddot{\boldsymbol{x}}$, $\delta\dot{\boldsymbol{x}}$ and $\delta\boldsymbol{x}$ are respectively cubic, quartic and quintinc polynomials, $\delta\boldsymbol{x}$ at the end of the timestep has 8th order accuracy, and the interpolated solution within the timestep has 6th order for $\delta\boldsymbol{x}$ and 5th order for $\delta\dot{\boldsymbol{x}}$. This latter scheme is used in practice to integrate the variational equation for one or more deviation vectors $\boldsymbol{w}$ (note that the matrices A, B containing the potential derivatives are only evaluated four times per timestep, irrespective of the number of independent deviation vectors). Note, however, that the high order of the scheme is only guaranteed for sufficiently smooth Hessians; this condition is not satisfied by the two commonly used potential expansions represented by quintic splines (`Multipole` and `CylSpline`), whose Hessians have only one continuous derivative.

The class `orbit::RuntimeVariational` uses the above described method to follow one or six deviation vectors $\delta\boldsymbol{w}_d$ during orbit integration. These vectors are initially chosen to be orthogonal and unit-normalized, so that the matrix $\mathsf{W}(t)$ assembled from these column-vectors is the Jacobian J of the mapping between the orbital initial conditions $\boldsymbol{w}^{(0)}$ and the current point on the trajectory $\boldsymbol{w}(t)$: $J_{kd} = \partial w_k(t)/\partial w_d^{(0)}$. For a regular orbit, deviation vectors grow linearly with time, and the basis remains sufficiently non-degenerate. However, if an orbit is chaotic, all vectors eventually start to grow exponentially and become more

and more aligned with each other (note that the determinant of the Jacobian remains unity, since the phase volume is conserved for any Hamiltonian system). The eigenvalues of the Jacobian

One may periodically orthogonalize the set of deviation vectors, using the QR factorization of $\mathsf{W}(t_1)$ at some time t_1 as $\mathsf{W}(t_1) = \mathsf{Q}^{(1)}\,\mathsf{R}^{(1)}$, where $\mathsf{Q}^{(1)}$ is an orthogonal matrix with values of order unity, and $\mathsf{R}^{(1)}$ is an upper triangular matrix with a large condition number (the ratio of maximal to minimal value on the main diagonal). The matrix $\mathsf{W}^{(1)}(t_1) \equiv \mathsf{Q}^{(1)}$ then replaces the original set of deviation vectors, and their evolution is followed until the matrix $\mathsf{W}^{(1)}(t)$ again becomes too large at some time t_2, at which point another orthogonalization is performed: $\mathsf{W}^{(1)}(t_2) = \mathsf{Q}^{(2)}\,\mathsf{R}^{(2)}$. The matrix $\mathsf{W}^{(2)}(t_2) \equiv \mathsf{Q}^{(2)}$ again replaces the previous deviation vectors, and the process continues as long as needed. At any point, the Jacobian can be reconstructed as $\mathsf{J}(t) = \mathsf{W}^{(N)}(t)\,\mathsf{R}^{(N)}\,\mathsf{R}^{(N-1)}\,\ldots\,\mathsf{R}^{(1)}$. Note that the product of upper triangular matrices $\mathsf{R} \equiv \prod_{n=1}^{N} \mathsf{R}^{(n)}$ is still an upper triangular matrix and can be updated after each orthogonalization step. Its condition number keeps growing with each step and may well exceed 10^{16} (but is still limited by the overall dynamical range of `double`, roughly 10^{300}). To prevent overflow, one may further renormalize R as $\exp(\Lambda)\,\hat{\mathsf{R}}$, where the largest diagonal element of $\hat{\mathsf{R}}$ is 1, and the smallest may eventually underflow to zero.

The full spectrum of Lyapunov exponents can be determined from the diagonal elements of the matrix R in the limit $t \to \infty$. However, in practice we are mostly interested in the largest of them, which indicates whether the orbit is chaotic or not. Since for a chaotic orbit all deviation vectors eventually become exponentially growing with the same rate, it is sufficient to follow the evolution of only one vector, renormalizing its magnitude to avoid overflow, but without the need for orthogonalization. On the other hand, if we are interested in the full Jacobian matrix J, we need to follow all six vectors. Even in this case there is still no particular advantage in carrying out the orthogonalization procedure, since the reconstructed Jacobian matrix will be fully degenerate (all deviation vectors will be equal to within floating-point precision) when the difference between the largest and the second-largest diagonal element of R exceeds 10^{16}, thus losing any extra dynamic range contained in R. Therefore, the orthogonalization procedure is not used by default.

Finally, the (largest) Lyapunov exponent is determined from the time evolution of the norm of the largest (or the only) deviation vector $|\boldsymbol{w}|$ as follows. Assuming that $|\boldsymbol{w}| \approx A\,t + B\,\exp\left(\lambda t / T_{\mathrm{orb}}\right)$, where the first term describes the phase mixing for regular orbits, and the optional second term becomes dominant for chaotic orbits at large times, we fit a linear regression $\ln\left(|\boldsymbol{w}(t)|\right) \approx \ln t + C + \lambda t / T_{\mathrm{orb}}$. If the linear term $\lambda t / T_{\mathrm{orb}}$ is significantly different from zero (that is, exceeds the other terms by at least 2 at the end of the time interval), λ is reported as the Lyapunov exponent normalized to the orbital period T_{orb}, otherwise the evidence for chaotic behaviour is considered insufficient and $\lambda = 0$ is reported.

A.6 Action/angle transformation

A.6.1 Stäckel approximation

A prolate spheroidal coordinate system is characterized by a single parameter Δ – the distance between the origin and any of the two focal points (located on the z axis). The coordinate lines are ellipses and hyperbolae defined by the focal points. How exactly the coordinate values along these lines are chosen is a matter of preference: various studies use different definitions, and we suggest yet another one for the reasons to be explained shortly.

- The triplet λ, ν, ϕ and their canonically conjugate momenta p_λ, p_ν, p_ϕ is used in [26, 57, 58]. The transformation between cylindrical and prolate spheroidal coordinates is given by

$$R^2 = (\lambda - \Delta^2)(\Delta^2 - \nu)/\Delta^2, \quad z^2 = \lambda\nu/\Delta^2, \tag{39a}$$

$$\lambda, \nu = \tfrac{1}{2}(R^2 + z^2 + \Delta^2) \pm \frac{1}{2}\sqrt{(R^2 + z^2 - \Delta^2)^2 + 4R^2\Delta^2} \quad (+ \text{ for } \lambda, - \text{ for } \nu), \tag{39b}$$

$$p_\lambda = \frac{R\,v_R}{2(\lambda - \Delta^2)} + \frac{z\,v_z}{2\lambda}, \quad p_\nu = \frac{R\,v_R}{2(\nu - \Delta^2)} + \frac{z\,v_z}{2\nu}, \quad p_\phi = Rv_\phi. \tag{39c}$$

  The allowed range of variables is $0 \le \nu \le \Delta^2 \le \lambda$ [18]. An arbitrary separable axisymmetric Stäckel potential is given by

$$\Phi(\lambda, \nu) = -\frac{f_\lambda(\lambda) - f_\nu(\nu)}{\lambda - \nu}. \tag{39d}$$

  The advantage of this choice is a near-symmetry between λ and ν and the fact that they occupy distinct ranges, which makes possible to use a single function f of one variable in place of both f_λ and f_ν. The disadvantages are that the coordinates only describe the half-space $z \ge 0$ (this may be amended by extending the range of ν to $-\Delta^2 \le \nu \le \Delta^2$ with the convention that $\nu < 0$ corresponds to $z < 0$), that the transformation is quadratic (not linear) at small R or z, and that it does not apply in the spherical limit ($\Delta = 0$).

- The triplet u, v, ϕ with the corresponding momenta is used in [6, 11]. The transformation is given by

$$R = \Delta \sinh u \sin v, \quad z = \Delta \cosh u \cos v, \qquad u \ge 0,\ 0 \le v \le \pi, \tag{40a}$$

[18] The above papers introduce different parameters of the coordinate system: $\alpha \equiv -a^2, \gamma \equiv -c^2$, such that $\Delta^2 = a^2 - b^2$, and the range of variables is $c^2 \le \nu \le a^2 \le \lambda$, but we may always set $c = 0$. A slightly different convention is used in [5]: λ in that paper corresponds to $\lambda - \Delta^2$ here, and similarly ν. Moreover, in some papers Δ stands for the squared focal distance.

thus $\lambda = \Delta^2 \cosh^2 u$, $\nu = \Delta^2 \cos^2 v$, and the expression for a generic axisymmetric Stäckel potential is

$$\Phi(u,v) = \frac{U(u) - V(v)}{\sinh^2 u + \sin^2 v}\,, \qquad U(u) \equiv -\frac{f_\lambda[u(\lambda)]}{\Delta^2},\ V(v) \equiv -\frac{f_\nu[v(\nu)]}{\Delta^2}. \tag{40b}$$

This form is advantageous because it covers the entire space, and v tends to the spherical polar angle θ at large radii, however u does not have an equally straightforward asymptotic meaning, and there is still no valid limit $\Delta \to 0$.

- Instead of u, one may use the quantity $\varpi \equiv \Delta \sinh u = \sqrt{\lambda - \Delta^2}$, which exactly equals the cylindrical radius R whenever $z = 0$ (equivalently $v = \pi/2$), regardless of Δ. At large distances ($\gg \Delta$), ϖ and v tend to the spherical radius r and the polar angle θ, respectively; thus in the limit $\Delta \to 0$ these are just the spherical coordinates. The transformation is thus

$$R = \varpi \sin v, \quad z = \sqrt{\varpi^2 + \Delta^2} \cos v, \qquad \varpi \geq 0,\ 0 \leq v \leq \pi, \tag{41a}$$

and the expression for the potential is

$$\Phi(\varpi,v) = -\frac{f_\varpi(\varpi) - f_v(v)}{\varpi^2 + \Delta^2 \sin^2 v}\,, \qquad f_\varpi(\varpi) \equiv f_\lambda[\varpi(\lambda)],\ f_v(v) \equiv f_\nu[v(\nu)]. \tag{41b}$$

In the subsequent discussion, we will use the coordinates λ and ν for consistency with the previous work, even though internally the calculations are performed in terms of ϖ and v.

Since the two functions f_λ, f_ν may be shifted by an arbitrary constant simultaneously, we assume that $f_\nu(0) = 0$. The continuity of potential at focal points requires that $f_\nu(\Delta^2) = f_\lambda(\Delta^2)$. Thus to obtain the values of these functions at an arbitrary point $\{\lambda,\nu\}$, we compute the potential at this point and at $\{\lambda, 0\}$ (in the equatorial plane), then take $f_\lambda(\lambda) = -\Phi(\lambda,0)\,\lambda$, $f_\nu(\nu) = \Phi(\lambda,\nu)\,(\lambda - \nu) - \Phi(\lambda,0)$.

The third integral is also introduced in different forms across various studies. Here we adopt the definition used by [57] (their eq. 3), given by

$$I_3 = f_\tau(\tau) + \left(E - \frac{L_z^2}{2(\tau - \Delta^2)} - 2(\tau - \Delta^2)\,p_\tau^2\right)\tau \qquad (\tau \text{ is either } \lambda \text{ or } \nu) \tag{42a}$$

$$= f_\tau(\tau) + \Phi(\lambda,\nu)\,\tau + \tfrac{1}{2}\big(L^2 - L_z^2 + v_z^2\Delta^2\big)$$

$$= [\Phi(\lambda,\nu) - \Phi(\lambda,0)]\,\lambda + \tfrac{1}{2}\big(z^2 v_\phi^2 + (Rv_z - zV_R)^2 + v_z^2\Delta^2\big). \tag{42b}$$

A slightly different expression is used in [5]: their eq. 1 introduces I_s which equals $-I_3/\Delta^2$ (they denote the focal distance as z_0). The quantity introduced in [11] (eq. 3.248), and also used in [6] (where it was also called I_3), is equivalent to $(I_3 + L_z^2/2)/\Delta^2 - E$.

The actions J_τ, where $\tau = \{\lambda, \nu\}$, are computed as

$$J_\tau = \frac{1}{\pi} \int_{\tau_{\min}}^{\tau_{\max}} p_\tau \, \mathrm{d}\tau \qquad (\tau \text{ is either } \lambda \text{ or } \nu), \tag{43}$$

where the canonical momentum $p_\tau(\lambda, \nu; E, L_z, I_3)$ is expressed from (42a), and the limits of integration $\tau_{\min,\max}$ are defined by the condition $p_\tau^2 = 0$. In the spherical limit, $J_\lambda = J_r$, $J_\nu = J_z = L - L_z$.

The essence of the Stäckel approximation is to pretend that the potential is of the Stäckel form and use the above expressions to compute the actions, substituting the actual potential where needed. The procedure is the following:

1. Choose the focal distance Δ, presumably in such a way as to maximize the resemblance of the potential to a separable form (39d). This defines the transformation between $\{R, z\}$ and $\{\lambda, \nu\}$.

2. Compute the potential at two points, $\{\lambda, \nu\}$ and $\{\lambda, 0\}$, and assign the three integrals of motion E, L_z and I_3 (the latter from 42b).

3. Find the integration limits $\{\lambda, \nu\}_{\min,\max}$ (assuming that the orbits looks like a rectangle in the $\{\lambda, \nu\}$ plane, which is of course only an approximation if the potential is not separable). In doing so, we solve for $p_\tau^2 = 0$ in (42a), where τ is either λ or ν, and the other coordinate is kept at its initial value. This step costs ~ 30 potential evaluations to find the three roots (the fourth one is always $\nu_{\min} = 0$).

4. Compute the actions from (43), again integrating along each of the two coordinates $\{\lambda, \nu\}$ while keeping the other one fixed at its initial value. We use a fixed-order Gauss–Legendre integration (with ten points) in a suitably scaled coordinate (to neutralize the singular behaviour of $p_\tau(\tau)$ near endpoints), hence this step costs 20 potential evaluations.

5. If the frequencies and angles are needed, follow the procedure described in the Appendix A of [57]. This involves computation of six additional integrals for frequencies, and further six – for the angles (note that they are carried along the same paths as the integrals for the actions, so one could store and re-use the values of potential, but this is not yet implemented).

The accuracy of this approximation may be judged by computing numerically an orbit in the given potential, determining the actions at each point along the orbit, and estimating their variation. If actions returned by the Stäckel approximations were true integrals, their variation would be zero. In practice, the variation depends crucially on the choice of the only free parameter – the focal distance Δ. Clearly, the best choice may depend on the two classical integrals of motion (E and L_z). There are two alternative approaches to assigning Δ:

- If $\Phi(\lambda,\nu)$ is a Stäckel potential, then from (39d) it follows that

$$\frac{\partial^2[(\lambda-\nu)\Phi(\lambda,\nu)]}{\partial\lambda\,\partial\nu}=0\ ,\quad \text{or, in the cylindrical coordinates,} \tag{44a}$$

$$3z\,\frac{\partial\Phi}{\partial R}-3R\,\frac{\partial\Phi}{\partial z}+R\,z\left(\frac{\partial^2\Phi}{\partial R^2}-\frac{\partial^2\Phi}{\partial z^2}\right)+(z^2-R^2-\Delta^2)\,\frac{\partial^2\Phi}{\partial R\,\partial z}=0. \tag{44b}$$

  Thus one may seek the value of Δ that minimizes the deviation of the above quantity from zero in the region occupied by the orbit.

- A shell orbit (the one with $J_r=0$) in a Stäckel potential has $\lambda=\text{const}$; thus one may find such an orbit for each E and L_z, and assign Δ from the condition that $p_\lambda^2(R,z=0)$ has a maximum value 0 reached at $R=R_{\rm shell}$ (in other words, the range of oscillation in λ shrinks to zero).

In either case, to make the action computation procedure efficient, we need to pre-compute a suitable table of $\Delta(E,L_z)$ and calculate the suitable value of Δ by 2d interpolation as the first step of the above procedure. We found that the second approach generally leads to a somewhat better action conservation accuracy.

A further significant speedup may be achieved by pre-computing a 3d interpolation table for the actions as functions of three integrals of motion – E, L_z and I_3. In this way, we only follow the first two steps of the procedure, avoiding the costly part of finding the integration limits and performing the integration itself. The table may be constructed by following the entire procedure for a family of orbits started at $R=R_{\rm shell}(E,L_z)$, $z=0$ and velocity directed at various angles in the meridional plane. From (42b) it follows that in this case, $I_3=(R^2+\Delta^2)\,v_z^2$, and since $v_z^2=2[E-\Phi(R,0)]-L_z^2/R^2-v_R^2$, the maximum value of I_3 at the given E and L_z is

$$I_3^{(\max)}(E,L_z)=\big(R_{\rm shell}^2+\Delta^2\big)\big(2[E-\Phi(R_{\rm shell},0)]-L_z^2/R_{\rm shell}^2\big), \tag{45}$$

where both $R_{\rm shell}$ and Δ are also functions of E and L_z. The interpolation table is constructed in terms of scaled variables: E, $L_z/L_{\rm circ}(E)$, $I_3/I_3^{(\max)}(E,L_z)$, and the values to be interpolated are $J_{r,z}/(L_{\rm circ}-L_z)$. The cost of construction of this 3d table is comparable to the cost of pre-computing the 2d table for $\Delta(E,L_z)$, and both take only a few CPU seconds (and are trivially parallelized).

However, since I_3 is only an approximate integral (which furthermore also depends on Δ), this introduces an additional error in the approximation, which cannot be reduces by making the interpolation grid finer. The error comes from the fact that the accuracy of conservation of I_3 along the orbit is typically worse than the accuracy of conservation of J_r, J_z computed at each point numerically. It turns out that the variation of I_3 from one point to another is largely balanced by performing the integration along the lines of constant λ and ν that pass through the given point, as depicted in the left panel of the same figure. By contrast,

in performing the interpolation from a pre-computed table, we essentially always follow the integration paths passing through $\{R_{\rm shell}(E,L_z),0\}$, but for a "wrong" I_3.

Nevertheless, the overall accuracy of the Stäckel approximation is reasonably good (for low-eccentricity orbits it is typically much better than shown in the above example). For disk orbits, the relative errors are typically better than 1%, while for halo orbits they may reach 10% – this happens mostly at resonances, when the orbit is not at all well described by a box in any prolate spheroidal coordinate system. The error in the interpolated approximation is a factor of 1.5–3 worse, but still tolerable in many contexts (e.g., construction of equilibrium models), and it leads to a $\sim 10\times$ speedup in action computation, for a very moderate overhead in construction of interpolation tables. In terms of accuracy and speed, the interpolated Stäckel approximation is thus similar to the use of un-adorned I_3 as the approximate third integral of motion, as advocated in [5]; however, actions have clear conceptual advantages.

A.7 Distribution functions

A.7.1 Spherical anisotropic DFs

This type of DF, represented by the class `df::QuasiSpherical` and its descendants, is constructed from a given combination of density and potential, under certain assumptions about the functional form of the DF. At the moment, only one specific subtype is implemented in the `df::QuasiSphericalCOM` class: the Cuddeford–Osipkov–Merritt model:

$$f(E,L) = \hat{f}(Q)\; L^{-2\beta_0}, \qquad Q \equiv E + L^2/(2r_a^2), \tag{46}$$

$$\hat{f}(Q) = \begin{cases} \dfrac{2^{\beta_0}}{(2\pi)^{3/2}\,\Gamma(1-\beta_0)\,\Gamma(3/2-\beta)} \displaystyle\int_Q^0 \frac{\mathrm{d}\hat{\rho}}{\mathrm{d}\Phi}\frac{\mathrm{d}\Phi}{(\Phi-Q)^{3/2-\beta_0}}, & 1/2<\beta_0<1, \\ \dfrac{1}{2\pi^2}\;\dfrac{\mathrm{d}\hat{\rho}}{\mathrm{d}\Phi}\Big|_{\Phi=Q}, & \beta_0=1/2, \\ \dfrac{2^{\beta_0}}{(2\pi)^{3/2}\,\Gamma(1-\beta_0)\,\Gamma(1/2-\beta)} \displaystyle\int_Q^0 \frac{\mathrm{d}^2\hat{\rho}}{\mathrm{d}\Phi^2}\frac{\mathrm{d}\Phi}{(\Phi-Q)^{1/2-\beta_0}}, & -1/2<\beta_0<1/2, \\ \dfrac{1}{2\pi^2}\;\dfrac{\mathrm{d}^2\hat{\rho}}{\mathrm{d}\Phi^2}\Big|_{\Phi=Q}, & \beta_0=-1/2, \end{cases}$$

$$\hat{\rho}(\Phi) \equiv \rho(r)\,r^{2\beta_0}\left[1+(r/r_a)^2\right]^{1-\beta_0}\Big|_{r=r(\Phi)}.$$

Here $\hat{\rho}$ is the augmented density, expressed as a function of potential and then differentiated once or twice. We use a finite-difference estimate for the radial derivatives of the original density $\rho(r)$, but this becomes inaccurate at small r when both ρ and Φ tend to finite limiting values. To cope with this issue, we fit a Taylor series expansion for $\rho(\Phi)$ as $\Phi\to\Phi(r=0)$,

and use it at small radii where it can be differentiated analytically (only if the series produce a reasonable approximation of the density). We limit the range of the anisotropy coefficient to $\beta_0 \geq -1/2$, since lower values would need a third or even higher derivative of density, becoming too challenging to compute accurately. The energy-dependent part of the DF $\hat{f}(Q)$ is represented by a log-scaled cubic spline in the scaled energy coordinate $\mathscr{E}$ defined below. Negative values of $f(Q)$ are replaced by zeros.

A.7.2 Spherical isotropic DFs and the phase-volume formalism

In the isotropic case ($\beta_0 = 0$, $r_a = \infty$), the DF is a function of E alone, but it can also be expressed in terms of an action-like variable h.

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

$$h(E) = \frac{16\pi^2}{3} \int_0^{r_{\max}(E)} r^2\, v^3(E,r)\, \mathrm{d}r = 8\pi^3 \int_0^{L^2_{\rm circ}(E)} J_r(E,L)\, \mathrm{d}L^2 = \int_{\Phi(0)}^{E} g(E')\, \mathrm{d}E', \quad (47a)$$

$$g(E) = 16\pi^2 \int_0^{r_{\max}(E)} r^2\, v(E,r)\, \mathrm{d}r = 4\pi^2 \int_0^{L^2_{\rm circ}(E)} T_{\rm rad}(E,L)\, \mathrm{d}L^2, \quad (47b)$$

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 $T_{\rm rad}(E,L) \equiv 2\int_{r_-}^{r_+} \mathrm{d}r/v_r \equiv 2\int_{r_-}^{r_+} \mathrm{d}r/\sqrt{v^2 - L^2/r^2} = 2\pi\, \partial J_r/\partial E$ is the radial period (its dependence on L at a fixed E is usually weak). In other words, phase volume is literally the volume of phase space enclosed by the energy hypersurface. Equations 2 and 3 in [19], or Equations 5.178 and 5.179 in [41], define two similarly related quantities $p \equiv g/(4\pi^2)$, $q \equiv h/(4\pi^2)$, with $p = \mathrm{d}q/\mathrm{d}E$.

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 finite-mass model in which the density drops faster than r^{-3} at large radii), $h(E) \propto (-E)^{-3/2}$ 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 $2 \leq \gamma \leq 3$ (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 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 `createSphericalIsotropicDF` and `fitSphericalIsotropicDF` defined in `df_spherical.h`.

The main application of these DFs is for simulating the effect of two-body relaxation, used in the Monte Carlo code Raga [72] and in the Fokker–Planck code PhaseFlow [73]. In general, the entire stellar system may consist of one or more components described by DFs $f_c(h)$, with m_c being the mass of individual stars in each component c. One needs to consider two kinds of the composite DF of the field stars: *plain* $f(h) \equiv \sum_c f_c(h)$ and *stellar mass-weighted* $\widehat{f}(h) \equiv \sum_c m_c\, f_c(h)$. Various derived quantities (integrals of the DF) will also be denoted by $\widehat{\cdot}$ if they use $\widehat{f}$. In [41] (Equation 5.173), f and $\widehat{f}$ are denoted by ν and μ, respectively.

There are two possible descriptions of relaxation phenomena: either locally, as a perturbation to the velocity $v \equiv \sqrt{2(E-\Phi(r))}$ of a test star with mass m 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

$$v\langle \Delta v_\parallel \rangle = -\Gamma\,(m\,J_{1/2} + \widehat{J}_{1/2})\,, \tag{48a}$$

$$\langle \Delta v_\parallel^2 \rangle = \tfrac{2}{3}\Gamma\,\left(\widehat{I}_0 + \widehat{J}_{3/2}\right), \tag{48b}$$

$$\langle \Delta v_\perp^2 \rangle = \tfrac{2}{3}\Gamma\,\left(2\widehat{I}_0 + 3\widehat{J}_{1/2} - \widehat{J}_{3/2}\right), \quad \text{where} \tag{48c}$$

$$I_0(E) \equiv \int_E^0 f(E')\,\mathrm{d}E' = \int_{h(E)}^\infty \frac{f(h')}{g(h')}\,\mathrm{d}h', \tag{48d}$$

$$J_{n/2}(E,\Phi) \equiv \int_{\Phi(r)}^E f(E')\left(\frac{E'-\Phi}{E-\Phi}\right)^{n/2}\mathrm{d}E' = \int_{h(E)}^\infty \frac{f(h')}{g(h')}\left(\frac{E'(h')-\Phi}{E-\Phi}\right)^{n/2}\mathrm{d}h', \tag{48e}$$

and similar definitions for $\widehat{I}_0, \widehat{J}_{n/2}$ using $\widehat{f}$ instead of f.

Orbit-averaged energy drift and diffusion coefficients are given by

$$\langle \Delta E \rangle_{\rm av} = \Gamma \left[\widehat{I}_0 - m\, K_g/g \right], \tag{49a}$$

$$\langle \Delta E^2 \rangle_{\rm av} = 2\Gamma \left[\widehat{I}_0\, h + \widehat{K}_h \right] / g, \tag{49b}$$

$$K_g(E) \equiv \int_{\Phi(0)}^{E} f(E')\, g(E')\, {\rm d}E' = \int_0^{h(E)} f(h')\, {\rm d}h', \tag{49c}$$

$$K_h(E) \equiv \int_{\Phi(0)}^{E} f(E')\, h(E')\, {\rm d}E' = \int_0^{h(E)} \frac{f(h')\, h'}{g(h')}\, {\rm d}h'. \tag{49d}$$

In these expressions, $\Gamma \equiv 16\pi^2\, G^2\, \ln\Lambda$, where $\ln\Lambda \sim \ln N = \mathcal{O}(10)$ 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 $^2/_3$ times their kinetic energy.

Of course, an efficient evaluation of diffusion coefficients again requires interpolation from pre-computed tables, which are provided by the class `SphericalIsotropicModelLocal`. 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: $\rho(r,t) = \int f(h,t) {\rm d}^3v$. In a multicomponent system, the DFs of individual components f_c evolve independently in a common potential, which is still determined by the total *plain* DF $f \equiv \sum_c f_c$. 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 (49), 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., [19]). 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:

$$\frac{\partial f_c(h,t)}{\partial t} = \frac{\partial}{\partial h}\left[A_c(h)\, f_c(h,t) + D(h)\frac{\partial f_c(h,t)}{\partial h}\right], \tag{50}$$

$$A_c(h) = \Gamma\, m_c\, K_g(h), \qquad D(h) = \Gamma\, g(h)\,\big[h\,\widehat{I}_0(h) + \widehat{K}_h(h)\big]. \tag{51}$$

Note that the advection coefficient A_c for c-th component is proportional to the mass of individual stars m_c of this component and involves the *plain* total DF f, whereas the diffusion coefficient D is the same for all components, and is given by integrals over the *stellar mass-weighted* total DF $\widehat{f}$. 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 [73], has several ingredients:

- The distribution functions $f_c(h,t)$ for all evolving populations of stars, represented by their values on a grid in $\ln h$. Values of $f_c(h)$ for an arbitrary argument are obtained from a cubic spline interpolator for $\ln f_c$ 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 $\rho(\Phi(r)) = \int_\Phi^0 f\big(h(E)\big)\, 4\pi\sqrt{2(E-\Phi)}\,\mathrm{d}E$ is obtained from the *plain* total DF *in the given potential*. We recompute the density after the DF has been evolved in the Fokker–Planck step, using the potential extrapolated from its previous evolution to the current time. This extrapolation makes in unnecessary to iteratively improve the solution by substituting the self-consistent potential back to the r.h.s. of this equation.

- The advection and diffusion coefficients A, D are computed using a dedicated class `SphericalIsotropicModel` which combines the two kinds of the total DF – $f(h)$ and $\widehat{f}(h)$ – with a potential $\Phi(r)$ and a mapping $h \leftrightarrow E$ (`PhaseVolume`) constructed for the given Φ.

- The FP equation (50) in the discretized form is solved with the Chang–Cooper scheme (e.g., [46]) or with a finite-element method, described in the appendix of [73].

A.8 Schwarzschild modelling

The framework for constructing Schwarzschild orbit-superposition models is centered around the concept of `Target` – an abstract interface for representing some features of the model in a discretized form. We denote the required values U_n of these features as $N_{\rm cons}$ model constraints, and the contributions of i-th orbit as $u_{i,n}$. The goal of the modelling procedure is to reproduce the constraints by a weighted superposition of orbit contributions. There are two categories of targets: density and kinematic.

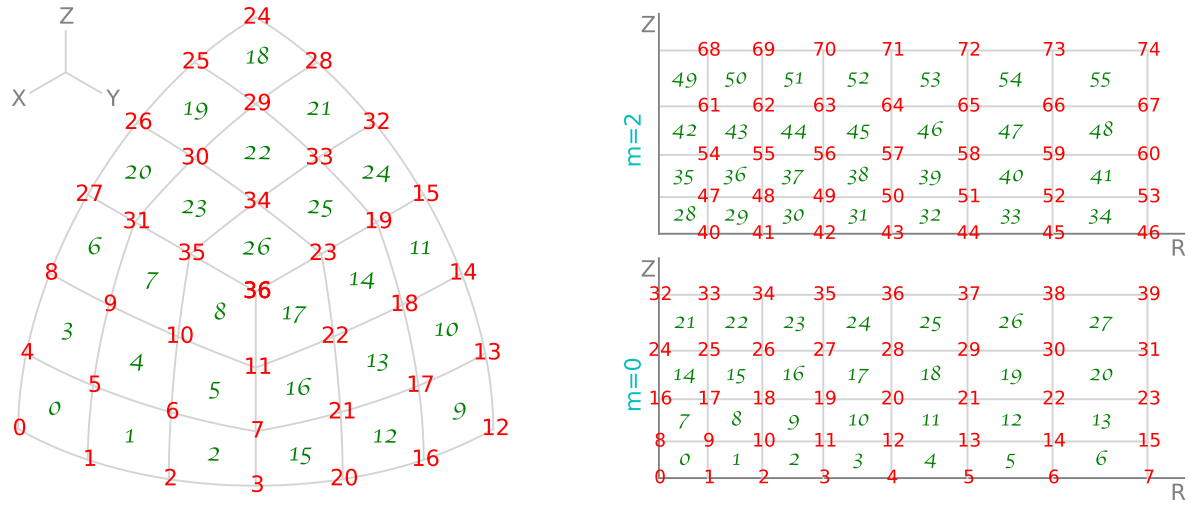

Figure 11: Discretization schemes for the 3d density profile in Schwarzschild models.
Left panel shows one radial shell in the "classic" grid scheme: one octant of a sphere is divided into three equal panes, and each pane – into $K \times K$ cells, where K is given by the parameter `stripsPerPane`. The volume discretization elements are 0th degree B-splines ($\sqcap$-shaped blocks) in the case of `type="DensityClassicTopHat"` (green indices at cell centers), or 1st degree B-splines ($\wedge$-shaped elements) in the case of `type="DensityClassicLinear"` (red indices at grid vertices). Indices further increase along the third dimension (radius), with radial shells that could be placed at arbitrary intervals (parameter `gridR`); for `DensityClassicLinear`, a single vertex at origin is added as the 0th element. In both cases, the grid may be further stretched along Y and Z axes by an amount controlled by `axisRatioY`, `axisRatioZ`; the radial grid then refers to the elliptical radius.
Right panels show the meridional section (R, Z) of the grid in the "cylindrical" scheme; the grid is rectangular but arbitrarily spaced in both directions (parameters `gridR`, `gridz`), and each even azimuthal harmonic term $m \le$ `mmax` has a separate set of indices. Again, the volume discretization elements are 0th degree B-splines in the case of `type="DensityCylindricalTopHat"` (green indices at cell centers) and 1st degree B-splines in the case of `type="DensityCylindricalLinear"` (red indices at grid vertices, excluding the Z axis for $m > 0$, where the coefficients of the Fourier expansion of density are always zero).

Density targets are used to produce a gravitationally self-consistent solution, in which the total density of the weighted superposition of orbits agrees with the Laplacian of the gravitational potential in which the orbits are integrated. There are three discretization schemes, which differ in the geometry of the spatial grid: classic, cylindrical, and spherical-harmonic. The first two have two variants each, differing in the degree of B-spline basis set (0 – top-hat, 1 – linear), and the latter is always 1st degree. In the classic scheme, the volume is divided into spherical or concentric ellipsoidal shells, the surface of each shell – into three panes, and each pane – into $K \times K$ cells, as shown in Figure 11, left panel. In `DensityClassicTopHat`, the volume of the model is divided into these 3d cells, and the mass of each cell (density integrated over the volume of the cell) serves as a single constraint. Equivalently, the volume discretization elements are non-overlapping quasi-rectangular blocks, and the basis functions have amplitude 1 in each cell and 0 elsewhere. In `DensityClassicLinear`, the basis functions are $\wedge$-shaped in all three directions (radial and two angular), with amplitude 1 reached at a single vertex of the grid and linearly tapering to zero at adjacent vertices. At any point, there are several (up to 8) overlapping basis functions, with their amplitudes summing up to unity. The density integrated with these weight functions still has the meaning of mass, but is associated with a grid vertex rather than grid segment, as shown in the above figure.

In the cylindrical scheme, the density is first expanded into Fourier series in the azimuthal angle ϕ, and then each m-th term ($m = 0, 2, \dots, m_{\max}$) is represented with a 2d B-spline basis set on the orthogonal grid in R, z, as shown in Figure 11, right panel. Similarly to the previous scheme, `DensityCylindricalTopHat` has basis functions that are non-overlapping and have amplitude 1 inside each cell of the 2d grid, while `DensityCylindricalLinear` has $\wedge$-shaped basis functions associated with grid vertices rather than segments. There is an additional factor $2\pi R$ in the integration of density times the basis function, so that the constraint values for the $m = 0$ term have the meaning of the mass in each grid cell or vertex (hence $\sum_{n=1}^{N_{\rm cons}} U_n$ is the total mass within the entire grid, similarly to the classic scheme). The constraint values for higher Fourier terms ($m > 0$) can have positive or negative sign, and should not be summed up to get the total mass. Since these higher-m terms must be zero on the z axis from regularity conditions, 1st-degree basis functions at the leftmost vertex in R are excluded from the basis set.

Finally, the `DensitySphHarm` scheme represents one radial coordinate with a B-spline basis set, and two angular coordinates with the spherical-harmonic basis set with order $l_{\max}, m_{\max}$. Both this and the previous schemes are conceptually similar to the `DensitySphericalHarmonic` (Section A.4.1) and `DensityAzimuthalHarmonic` (Section A.4.2) classes, which represent a 3d density profile as a corresponding interpolated function. The difference is that in those classes, the free parameters are the values of Fourier or multipole coefficients of density expansion at grid points, while in the target classes discussed in this section, the free parameters have the dimension of mass (density integrated over some volume).

The choice of a particular discretization scheme should be tailored to the density profile that is being represented: in the case of a spheroidal profile, classic or spherical-harmonic schemes work best, while for a disky profile (possibly with a non-axisymmetric bar), cylindri-

cal grid is preferred. In all cases, the radial (and vertical, in the cylindrical scheme) grids are defined by the user, typically with uniform or exponential spacing, and enclosing $\gtrsim 90-99\%$ of the total mass.

Kinematic targets come in two flavors. One is `KinemShell`, which represents the density-weighted radial and tangential velocity dispersions $\rho\sigma_{r,t}^2$ as functions of radius, projected onto the basis set of B-splines of `degree=0..3` defined by `gridr` in spherical radius. It is useful to constrain the velocity anisotropy profile $\beta \equiv 1 - \frac{1}{2}\sigma_t^2/\sigma_r^2$: if U_n^r and U_n^t are two equal-length arrays of these projections, then the constraints to be satisfied are written as $0 = 2(1-\beta_n)\,U_n^r - U_n^t$, where β_n is the value of β associated with n-th radial basis element. This can be used in the context of "theoretical" Schwarzschild models, when the goal is to construct a dynamically self-consistent model with the given density profile and have some control on its kinematic structure by assuming some functional form of $\beta(r)$.

More important in practice is the `LOSVD` target, which is used to constrain the kinematic structure of the model by observed line-of-sight velocity distributions in the image plane. There are several related ways of representing a LOSVD, and the relation between them is explained below.

The orientation of the image plane in the intrinsic coordinate system associated with the galaxy model is specified by three Euler angles α, β, γ, see Section A.3 for the definition, and Figure 8 for an illustration. The intrinsic (model) coordinate system is denoted as xyz, and the observational coordinate system – as XYZ, with Y axis pointing up in the image plane, X axis pointing left (note the opposite of the usual convention!), and Z axis pointing perpendicular to the image plane (along the line of sight) away from the observer. This unusual sign convention for the X axis is a consequence of the right-handedness of the coordinate frame. β is the usual inclination angle; γ is the angle between the line of nodes (intersection of xy and XY planes) and the X axis, and α is the angle between the line of nodes and the x (major) axis of the galaxy (it is relevant only for non-axisymmetric systems). If there are several observational datasets, each one needs a separate instance of a `LOSVD` target, with the parameters `gamma` possibly different between instances, and the other two angles `alpha`, `beta` being identical.

The LOSVDs are recorded in several spatial regions (apertures), which are arbitrary polygonal regions Ω_a in the image plane. The `apertures` parameter should contain a list of $N_a \times 2$ two-dimensional arrays specifying the N_a coordinates X, Y of a-th boundary polygon. Often these apertures come from binning up pixels in a regular two-dimensional grid in the image plane, e.g., using the Voronoi binning approach [14] or some other scheme. There is a `Python` routine `getBinnedApertures` that reconstructs the boundary polygons from an array describing the correspondence between bin indices and pixel coordinates. But any alternative way of defining apertures (e.g. circular fibers, sectors in polar coordinates, or spaxels of a long slit) is equally well possible to describe with arbitrary boundary polygons.

The effect of finite spatial resolution in the image plane is encoded in the point-spread function (PSF) of the instrument. The parameter `psf` can be specified either as a single

number (the width of a circular Gaussian – *not* the full width at half maximum (FWHM), which is 2.35× larger), or as a 2d array of several Gaussian components (the first column is the width and the second is the relative fraction of this component, which should sum up to unity).

The kinematic datacube is three-dimensional: two image-plane coordinates X, Y and the line-of-sight velocity V_Z. Accordingly, it is first recorded on a rectangular 3d grid in these variables, and represented internally as a 3d tensor-product B-spline:

$$\mathfrak{f}^{(\mathrm{int})}(X,Y,V_Z) = \sum_{i,j,k} A_{ijk}\; B_i^{(X)}(X)\; B_j^{(Y)}(Y)\; B_k^{(V)}(V_Z), \tag{52}$$

where A_{ijk} are the amplitudes and $B^{(X)}, B^{(Y)}, B^{(V)}$ are basis functions in each of the three directions. Then the two spatial directions are convolved with the PSF and rebinned onto the array of apertures. The output functions to be represented are the integrals of the LOSVD, convolved with the spatial PSF, over the area of each aperture Ω_a:

$$\mathfrak{f}_a(V_Z) = \sum_k A_{a,k}\; B_k^{(V)}(V_Z), \tag{53a}$$

$$A_{a,k} = \sum_{i,j} \iint_{X,Y\in\Omega_a} \mathrm{d}X\,\mathrm{d}Y \iint \mathrm{d}X'\,\mathrm{d}Y' \tag{53b}$$
$$\times\; A_{ijk}\; B_i^{(X)}(X')\; B_j^{(Y)}(Y')\;\; PSF\big(\sqrt{(X-X')^2+(Y-Y')^2}\big).$$

The amplitudes $A_{a,k}$ of B-spline expansion in the velocity dimension (indexed by $k = 1..N_V$) for each aperture (indexed by a) are stored in the flattened one-dimensional output array: each consecutive N_V numbers refer to one aperture. The conversion between the internal and the output representations is performed transparently to the user, using a single matrix multiplication, for which the matrix is pre-computed in advance and combines the spatial convolution and rebinning steps. The parameters provided by the user are: the grids in the image plane `gridx`, `gridy` and velocity `gridv`, and the degree of B-spline basis set ranging from 0 to 3 (although `degree=2` or `3` is strongly recommended for a much greater accuracy at the same grid size, as illustrated in the appendix of [76]). The image-plane grid should cover all apertures, and preferrably have an extra margin of $2-3$ times the PSF width for a more accurate convolution, but needs not be aligned with any of the apertures: the integration over Ω_a is performed exactly no matter what is the overlap between grid segments and aperture polygons. The spatial size of the grid should be comparable with the PSF width or the aperture size, whichever is larger. For instance, in the typical case that an adaptive binning scheme is employed, the apertures may consist of a single spaxel of the detector in the central area, typically smaller than the PSF width, and contain many such spaxels in the outer parts of the image. Then one may define a non-uniform `gridx` with smaller segments $\simeq$PSF width in the central part, which gradually become comparable to sizes of outermost apertures towards the endpoints in X. The parameter `gridy` may be omitted if it is identical to `gridx`.

One may also perform smoothing along the velocity axis by providing a nonzero `velpsf` parameter. Since the integrated-light LOSVDs are usually produced by a spectral fitting code in a velocity-deconvolved form, this is not needed (although won't hurt if the smoothing width is set to a significantly smaller value than the velocity dispersion). However, if the LOSVD is computed from individual stellar velocities, this parameter may represent the typical observational error (unfortunately, it is not easy to account for variable error bars between individual measurements).

Finally, another important parameter is `symmetry`, which determines how the model LOSVDs are symmetrized before producing the output array. Possible values are: `symmetry='t'` for the triaxial geometry, in which a fourfold discrete symmetry $\{x,y\} \leftrightarrow \{-x,-y\}, z \leftrightarrow -z$ holds even in the case of figure rotation; `symmetry='a'` for axisymmetric systems, which is approximately enforced by randomizing the azimuthal angle ϕ for each recorded point; or `symmetry='s'` for spherical systems, when both angles are randomized. This is performed in the intrinsic coordinate system xyz before projection: for instance, two out of four possible identical points in the triaxial case correspond to a reflection symmetry $\mathfrak{f}(X,Y,V_Z) = \mathfrak{f}(-X,-Y,-V_Z)$, but two other points project to coordinates unrelated to X,Y. This parameter should be in agreement with the symmetry of the potential, but needs to be provided separately, as the `Target` object has no knowledge of the potential.

Surface density is the integral of $\mathfrak{f}(X,Y,V_Z)$ along the velocity axis. When the `Target` is applied to a `Density` object, it produces an array of aperture masses – integrals of PSF-convolved LOSVDs over the area of each aperture and over velocity:

$$\mathfrak{M}_a \equiv \int \mathfrak{f}(V_Z)\,\mathrm{d}V_Z. \tag{54}$$

In terms of the B-spline representation of the LOSVD, it can be expressed as the dot product of the vector of amplitudes $A_{a,k}$ (53b) by the vector of integrals of basis functions

$$\mathcal{I}_k \equiv \int B_k^{(V)}(V_Z)\,\mathrm{d}V_Z. \tag{55}$$

Observational constraints on the LOSVD do not come in the form of B-splines, therefore one needs to convert the coefficients $A_{a,k}$ (output by the `Target` object as a 1d flattened array U_n for the entire model, or a 2d array $u_{i,n}$ of coefficients for each i-th orbit) into a form suitable for comparison with observations.

The observed LOSVDs are usually not normalized, i.e., they provide the distribution of stars in velocities, but not the total luminosity in the given aperture. This quantity needs to be computed from the model, by applying the LOSVD `Target` to the `Density` object.

Typically, the mass-to-light ratio of stars Υ is a free parameter in the models. When changing the mass normalization of all galactic components (including dark matter, central black hole, etc.) by the same factor Υ, one can reuse the same orbit library, but rescale the

model velocities by a factor $\sqrt{\Upsilon}$ before comparing the model LOSVDs to the observed ones. The rescaled LOSVD is represented by a B-spline: $\mathfrak{f}'_a(V_Z) = \sum_k A'_{a,k}\, B_k\big(\sqrt{\Upsilon}\, V_Z\big)$, where the new set of basis functions is defined by the velocity grid multiplied by $\sqrt{\Upsilon}$, and the new amplitudes are $A'_{a,k} = A_{a,k}/\sqrt{\Upsilon}$.

The observed LOSVD in a-th aperture is usually represented by some sort of basis-set expansion $\mathfrak{f}_a^{(\mathrm{obs})}(V_Z) = \sum_l C_{a,l}\, F_{a,l}(V_Z)$, with a vector of coefficients $\boldsymbol{C}_a \equiv C_{a,l}$ and the set of basis functions $F_{a,l}(V_Z)$. This could be a B-spline basis, e.g., a velocity histogram (0th-degree B-spline), in which case the basis functions are the same for all apertures, or a Gauss–Hermite (GH) expansion (Section A.2.6), in which case the basis functions depend on three additional parameters in each aperture – amplitude Ξ_a, center μ_a and width σ_a of the Gaussian function, which is the zeroth term in the expansion. In either case, the model LOSVDs can be *reinterpolated* onto the observational basis set(s), as explained in the last paragraph of Section A.2.1. The transformation between the vector of (rescaled) B-spline amplitudes of the model LOSVD $\boldsymbol{A}'_a$ and the vector of expansion coefficients in the observational basis set $\boldsymbol{C}_a$ is described by a matrix multiplication: $\boldsymbol{C}_a = \mathcal{G}^{-1}\,\mathsf{H}\,\boldsymbol{A}'_a$, where $H_{lk} \equiv \langle F_{a,l}, B_k\rangle$ and $\mathcal{G}_{lm} \equiv \langle F_{a,l}, F_{a,m}\rangle$ are the matrices of inner products of corresponding basis functions.

Example of all these steps is provided below (a more elaborate `Python` script is given in `example_forstand.py`). The apertures are Voronoi-binned as described in the file `voronoi_bins.txt` containing N_{aper} rows and 3 columns: X and Y coordinates of bin centers, and bin index. The kinematic measurements are provided in two alternative forms: (1) LOSVD histograms in the file `losvd_histograms.txt`, containing N_{aper} rows and $2\,N_{\mathrm{bins}}$ columns, each pair of consecutive columns giving the amplitude of LOSVD in a given bin of velocity grid and its error estimate; (2) Gauss–Hermite moments in the file `losvd_ghmoments.txt`, containing N_{aper} rows and 12 columns – values and error estimates of $\mu, \sigma, h_{3..6}$.

```
vorbins   = numpy.loadtxt("voronoi_bins.txt")
apertures = agama.schwarzlib.getBinnedApertures(
    xcoords=vorbins[:,0], ycoords=vorbins[:,1], bintags=vorbins[:,2])
histfile  = numpy.loadtxt("losvd_histograms.txt")
obs_gridv = numpy.linspace(-v_max, v_max, 16)  # observational vel. grid, N_bins = 15
obs_degree= 0  # histograms are 0th-degree B-splines
hist_val  = histfile[:,0::2];  hist_err = histfile[:,1::2]  # odd/even columns
ghmfile   = numpy.loadtxt("losvd_ghmoments.txt")
ghm_val   = ghmfile[:,0::2];  ghm_err = ghmfile[:,1::2]
num_aper  = len(apertures)  # number of apertures = len(histfile) = len(ghmfile)
```

Define the density and LOSVD `Target` objects (only the necessary parameters are provided):

```
mod_gridx = numpy.linspace(-r_max, r_max, 50)   # grid should cover all apertures
mod_gridv = numpy.linspace(-v_max, v_max, 25)   # and all velocities in the model
mod_degree= 2  # degree of B-splines for representing model LOSVDs (use 2 or 3)
```

```
tar_los   = agama.Target(type="LOSVD", gridx=mod_gridx, gridv=mod_gridv,
    degree=mod_degree, apertures=apertures, symmetry="s", psf=psf)
mod_gridr = agama.nonuniformGrid(30, 0.01*r_max, 5.*r_max)
tar_den   = agama.Target(type="DensitySphHarm", lmax=0, gridr=mod_gridr)
```

Assume we have a model potential `pot`, which also doubles as the density profile of stars, and have constructed initial conditions for the orbit library in `ic` (2d array of shape `num_orbits`$\times 6$). Integrate the orbits while collecting the matrices $u_{i,n}^{(t)}$ containing the contribution of i-th orbit to n-th discretization element of t-th target:

```
mat_den, mat_los = agama.orbit(potential=pot, ic=ic, time=100.*pot.Tcirc(ic),
    targets=[tar_den, tar_los])
```

Next we compute the required values of density and kinematic constraints. As explained above, the observational kinematic profiles are not normalized, so we compute the overall scaling factors $\mathfrak{M}_a$ (54) from the model density profile. For GH moments, a couple of extra steps are needed. First, the normalization is translated into the amplitude of the base Gaussian, summing the contributions from all even GH moments (26a). Second, the uncertainties on the mean and width of the Gaussian are converted into (approximate) uncertainties on h_1, h_2, which can be incorporated into the linear equation system. Finally, we may need to constrain the PSF-convolved surface density profile (aperture masses) separately from the 3d density profile, at least when working with GH moments (alternatively, one may add a set of constraints that h_0 be close to unity). This is expressed by a separate matrix constructed by dot-multiplying the LOSVD matrix of B-spline amplitudes of each orbit by the vector of B-spline integrals $\mathcal{I}$ (55). When fitting to LOSVD histograms directly, they will be already normalized, so separate aperture mass constraints are not necessary.

```
cons_den  = tar_den(pot)  # required values of 3d density constraints
cons_sur  = tar_los(pot)  # aperture masses (surface density integrated over apertures)
# row-normalize the provided histograms and multiply by aperture masses
obs_bsint = agama.bsplineIntegrals(degree=obs_degree, grid=obs_gridv)
num_obs_bs= len(obs_bsint)  # number of bins in observed velocity histograms
hist_norm = hist_val.dot(obs_bsint)  # ∫ f_a^(obs)(V_Z) dV_Z from the provided histograms
hist_val *= (cons_sur / hist_norm).reshape(num_aper, 1)
hist_err *= (cons_sur / hist_norm).reshape(num_aper, 1)
# normalization of the GH series in each aperture with contributions from h4, h6
ghm_norm  = 1 + ghm_val[:,3] * (24**0.5 / 8) + ghm_val[:,5] * (720**0.5 / 48)
# parameters of GH series (amplitude, center, width)
gh_params = numpy.array([ cons_sur/ghm_norm, ghm_val[:,0], ghm_val[:,1] ]).T
ghm_err[:,0:2] /= 2**0.5 * ghm_val[:,1:2]  # δh1 ≈ δv/√2σ, δh2 ≈ δσ/√2σ
ghm_val[:,0:2] *= 0  # set h1 = h2 = 0
num_obs_gh= ghm_val.shape[1]  # order of GH expansion (6 in our case)
# matrix of orbital contributions to aperture masses
```

```
mod_bsint = agama.bsplineIntegrals(degree=mod_degree, grid=mod_gridv)
num_mod_bs= len(mod_bsint)  # number of velocity basis functions in each aperture N_V
mat_sur   = mat_los.reshape(num_orbits, num_aper, num_mod_bs).dot(mod_bsint)
```

Now construct a Schwarzschild model for a particular value of mass-to-light ratio Υ, scaling the velocity grid of orbit LOSVDs by $\sqrt{\Upsilon}$ and their amplitudes by $1/\sqrt{\Upsilon}$ before converting them into the form compatible with the observational constraints.

```
mod_gridv_scaled = mod_gridv * Upsilon**0.5
```

If using LOSVD histograms, transform the matrix of B-spline amplitudes of orbit LOSVDs into the matrix of histogram values (amplitudes of 0th-degree B-spline) defined by the observational velocity grid. This is achieved by the following conversion matrix:

```
conv      = numpy.linalg.solve(
    agama.bsplineMatrix(obs_degree, obs_gridv),  # same as obs_bsint
    agama.bsplineMatrix(obs_degree, obs_gridv, mod_degree, mod_gridv_scaled))
mat_kin   = mat_los.reshape(num_orbits * num_aper, num_mod_bs).  \
    dot(conv.T). \
    reshape(num_orbits, num_aper * num_obs_bs) * Upsilon**-0.5
cons_kin  = hist_val.reshape(num_aper * num_obs_bs)
err_kin   = hist_err.reshape(num_aper * num_obs_bs)
```

If using GH moments, the routine `ghMoments` transforms the matrix of B-spline amplitudes of orbit LOSVDs into the matrix of GH moments computed in the observed GH basis defined by parameter `ghbasis` (different in each aperture). This matrix has moments $h_0..h_6$ (in this example `num_obs_gh=6`), but we don't use h_0 because it is not available observationally (instead, we constrain the aperture mass), so we reshape the matrix and eliminate the 0th column:

```
mat_kin   = agama.ghMoments(degree=mod_degree, gridv=mod_gridv_scaled,
    matrix=mat_los, ghorder=num_obs_gh, ghbasis=gh_params).  \
    reshape(num_orbits, num_aper,  num_obs_gh+1)[:,:,1:].    \
    reshape(num_orbits, num_aper * num_obs_gh) * Upsilon**-0.5
cons_kin  = ghm_val.reshape(num_aper * num_obs_gh)
err_kin   = ghm_err.reshape(num_aper * num_obs_gh)
```

Finally, solve the quadratic optimization problem to determine orbit weights. In this example, we require that the 3d density and 2d aperture mass constraints be satisfied exactly (set an infinite penalty for them), while the observational kinematic constraints should be satisfied as closely as possible, with the penalty proportional to the inverse squared observational error:

```
weights = agama.solveOpt(
    matrix=[mat_den.T, mat_sur.T, mat_kin.T],  # list of matrices
    rhs=[cons_den, cons_sur, cons_kin],        # list of RHS vectors (constraints)
```

```
    rpenq=[cons_den*numpy.inf, cons_sur*numpy.inf, 2*err_kin**-2]) # penalties
```

Many of the above steps are generally applicable to all observationally-constrained Schwarzschild models. The relevant routines and classes reside in the submodule `agama.schwarzlib`, and the user- and model-specific tasks may be kept in a separate script adapted from the template `example_forstand.py`.

References

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

[2] An J.H., Evans N.W., 2006, ApJ, 642, 752

[3] The Astropy collaboration, 2013, A&A, 558, A33

[4] Aumer M., Binney J., 2009, MNRAS, 397, 1286

[5] Bienaymé O., Robin A., Famaey B., 2015, A&A, 581, 123

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

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

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

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

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

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

[12] Binney J., Vasiliev E., 2023, MNRAS, 520, 1832

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

[14] Cappellari M., Copin Y., 2003, MNRAS, 342, 345

[15] Cappellari M., Emsellem E., 2004, PASP, 116, 138

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

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

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

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

[20] Cole D., Binney J., 2017, MNRAS, 465, 798

[21] Cuddeford P., 1991, MNRAS, 253, 414

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

[23] Dehnen W., 1999, AJ, 118, 1201

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

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

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

[27] Gerhard O., 1993, MNRAS, 265, 213

[28] Gieles M., Zocchi A., 2015, MNRAS, 454, 576

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

[30] Hahn T., 2005, Comput. Phys. Commun., 168, 78

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

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

[33] Hunter G., Sormani M., Beckmann J., et al., 2024, submitted

[34] Hyman J., 1983, J. Sci. Stat. Comput., 4, 645

[35] Jeffreson S., Sanders J., Evans N., et al., 2017, MNRAS, 469, 4740

[36] Joseph C., Merritt D., Olling R., et al., 2001, ApJ, 550, 668

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

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

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

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

[41] Merritt D., 2013, *Dynamics and evolution of galactic nuclei*, Princeton Univ. press

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

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

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

[45] O'Sullivan F., 1988, J.Sci.Stat.Comput., 9, 363

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

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

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

[49] Pontzen A., et al., ascl:1305.002

[50] Portail M., Gerhard O., Wegg C., Ness M., 2017, MNRAS, 465, 1621

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

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

[53] Price-Whelan A., 2017, J. Open Source Software, 2, 388; ascl:1707.006

[54] Read J., Mamon G., Vasiliev E., et al., 2021, MNRAS, 501, 978

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

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

[57] Sanders J., 2012, MNRAS, 426, 128

[58] Sanders J., Binney J., 2016, MNRAS, 457, 2107

[59] Sanders J., Lilley E., Vasiliev E., Evans N.W., Erkal D., 2020, MNRAS, 499, 4973

[60] Schwarzschild M., 1979, ApJ, 232, 236

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

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

[63] Sormani M., Sanders J., Fritz T., et al., 2022, MNRAS, 512, 1857

[64] Sormani M., Gerhard O., Portail M., Vasiliev E., Clarke J., 2022, MNRAS, 514, L5

[65] Springel V., 2010, MNRAS, 401, 791

[66] Springel V., Pakmor R., Zier O., Reinecke M., 2021, MNRAS, 506, 2871

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

[68] 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

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

[70] van der Marel R., Franx M., 1993, ApJ, 407, 525

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

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

[73] Vasiliev E., 2017, ApJ, 848, 10

[74] Vasiliev E., 2019, MNRAS, 482, 1525

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

[76] Vasiliev E., Valluri M., 2020, ApJ, 889, 39

[77] Vasiliev E., Belokurov V., Erkal D., 2021, MNRAS, 501, 2279

[78] Zemp M., Gnedin O., Gnedin N., Kravtsov A., 2011, ApJS, 197, 30

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

Agama logo is designed by Tatiana Morozova.