MEANS Documentation

Release 1.0.0

Sisi Fan, Quentin Geissmann, Eszter Lakatos, Saulius Lukauskas

December 12, 2015

Contents

MEANS is python package for Moment Expansion Approximation, iNference and Simulation.

We present a free, user-friendly tool implementing an efficient moment expansion approximation with parametric closures that integrates well with the IPython interactive environment. Our package enables the analysis of complex stochastic systems without any constraints on the number of species and moments studied and the type of rate laws in the system. In addition to the approximation method our package provides numerous tools to help non-expert users in stochastic analysis.

CHAPTER 1

Installation

Please follow detailed instructions in Github.

CHAPTER 2

Tutorial

An interactive tutorial on getting started with MEANS can be found on our Github page. We recommend trying it out directly using Jupyter interactive environment.

CHAPTER 3

API Reference

3.1 means.approximation package

3.1.1 Subpackages

means.approximation.lna package

Submodules

class `means.approximation.lna.lna.`**`LinearNoiseApproximation`**`(`*model*`)`

Bases: *`means.approximation.approximation_baseclass.ApproximationBaseClass`*

A class to performs Linear Noise Approximation of a model.

Initialise the approximation.

Parameters **`model`** (*`Model`*) – Model to approximate

`run``()`

Overrides the default _run() private method. Performs the complete analysis :return: A fully computed set of Ordinary Differential Equations that can be used for further simulation :rtype: *`ODEProblem`*

`means.approximation.lna.lna.`**`lna_approximation`**`(`*model*`)`

A wrapper around *`LinearNoiseApproximation`*. It performs linear noise approximation (MEA).

Returns an ODE problem which can be further used in inference and simulation.

Return type *`ODEProblem`*

Module contents

Linear Noise Approximation This part of the package implements Linear Noise Approximation as described in *[Komorowski2009]*.

Example:

```
>>> from means.approximation.lna.lna import lna_approximation
>>> from means.examples.sample_models import MODEL_P53
>>> ode_problem = lna_approximation(MODEL_P53)
>>> print ode_problem
```

The result is an *means.core.problems.ODEProblem*. Typically, it would be further used to perform simulations (see *simulation*) and inference (see *inference*).

---

means.approximation.mea package

Submodules

Gamma moment closure This part of the package provides the original the Gamma closure.

class `means.approximation.mea.closure_gamma.`**`GammaClosure`**(*max_order*, *multivariate=True*)

Bases: *means.approximation.mea.closure_scalar.ClosureBase*

EXPERIMENTAL

A class providing gamma closure to *MomentExpansionApproximation*. Expression for higher order (max_order + 1) central moments are computed from expressions of higher order raw moments. As a result, any higher order moments will be replaced by a symbolic expression depending on mean and variance only.

Parameters

- **`max_order`** (*int*) – the maximal order of moments to be modelled.
- **`type`** – 0 for univariate (ignore covariances), 1 and 2 for

the two types of multivariate gamma distributions. :type type: *int* :return:

Log-normal moment closure This part of the package provides the original the Log-normal closure.

class `means.approximation.mea.closure_log_normal.`**`LogNormalClosure`**(*max_order*, *multivariate=True*)

Bases: *means.approximation.mea.closure_scalar.ClosureBase*

A class providing log-normal closure to *MomentExpansionApproximation*. Expression for higher order (max_order + 1) central moments are computed from expressions of higher order raw moments. As a result, any higher order moments will be replaced by a symbolic expression depending on mean and variance only.

Parameters

- **`max_order`** (*int*) – the maximal order of moments to be modelled.
- **`multivariate`** – whether to consider covariances

Returns

Normal moment closure This part of the package provides the original the Normal (Gaussian) closure.

class `means.approximation.mea.closure_normal.`**`NormalClosure`**(*max_order*, *multivariate=True*)

Bases: *means.approximation.mea.closure_scalar.ClosureBase*

A class providing normal closure to *MomentExpansionApproximation*. Expression for higher order (max_order + 1) central moments are directly computed using Isserlis' Theorem. As a result, any higher order moments will be replaced by a symbolic expression depending on mean and variance only.

Parameters

- **`max_order`** (*int*) – the maximal order of moments to be modelled.

- **multivariate** – whether to consider covariances

Returns

Scalar moment closure This part of the package provides the original (and default) closure `ScalarClosure` as well as the base class for all closers.

class `means.approximation.mea.closure_scalar.`**`ClosureBase`**(*max_order*, *multivariate=True*)

Bases: `object`

A virtual class for closure methods. An implementation of *_compute_raw_moments()* must be provided in subclasses.

Parameters

- **max_order** (*int*) – the maximal order of moments to be modelled.
- **multivariate** – whether to consider covariances

Returns

`close`(*mfk*, *central_from_raw_exprs*, *n_counter*, *k_counter*)

In MFK, replaces symbol for high order (order == max_order+1) by parametric expressions. That is expressions depending on lower order moments such as means, variances, covariances and so on.

Parameters

- **mfk** – the right hand side equations containing symbols for high order central moments
- **central_from_raw_exprs** – expressions of central moments in terms of raw moments
- **n_counter** (list[`Moment`]) – a list of `Moment`s representing central moments
- **k_counter** (list[`Moment`]) – a list of `Moment`s representing raw moments

Returns the modified MFK

Return type *sympy.Matrix*

`is_multivariate`

`max_order`

class `means.approximation.mea.closure_scalar.`**`ScalarClosure`**(*max_order*, *value=0*)

Bases: `means.approximation.mea.closure_scalar.ClosureBase`

A class providing scalar closure to `MomentExpansionApproximation`. Expression for higher order (max_order + 1) central moments are set to a scalar. Typically, higher order central moments are replaced by zero.

Parameters

- **max_order** (*int*) – the maximal order of moments to be modelled.
- **value** – a scalar value for higher order moments

`value`

`means.approximation.mea.dmu_over_dt.`**`generate_dmu_over_dt`**(*species*, *propensity*, *n_counter*, *stoichiometry_matrix*)

Calculate $\frac{d\mu_i}{dt}$ in eq. 6 (see Ale et al. 2013).

$$\frac{d\mu_i}{dt} = S \left[\sum_l \sum_{n_1=0}^{\infty} ... \sum_{n_d=0}^{\infty} \frac{1}{\mathbf{n}!} \frac{\partial^n \mathbf{n} a_l(\mathbf{x})}{\partial \mathbf{x^n}} |_{x=\mu} \mathbf{M_{x^n}} \right]$$

Parameters

- **species** (list[*sympy.Symbol*]) – the name of the species/variables (typically *['y_0', 'y_1', ..., 'y_n']*)
- **propensity** – the reactions describes by the model
- **n_counter** (list[*Moment*]) – a list of *Moment*s representing central moments
- **stoichiometry_matrix** (*sympy.Matrix*) – the stoichiometry matrix

Returns a matrix in which each row corresponds to a reaction, and each column to an element of counter.

`means.approximation.mea.eq_central_moments.`**`eq_central_moments`**(*n_counter*, *k_counter*, *dmu_over_dt*, *species*, *propensities*, *stoichiometry_matrix*, *max_order*)

Function used to calculate the terms required for use in equations giving the time dependence of central moments.

The function returns the list Containing the sum of the following terms in in equation 9, for each of the $[n_1, ..., n_d]$ combinations in eq. 9 where ... is ... # FIXME

$$\binom{\mathbf{n}}{\mathbf{k}} (-1)^{\mathbf{n-k}} [\alpha \frac{d\beta}{dt} + \beta \frac{d\alpha}{dt}]$$

Parameters

- **n_counter** (list[*Moment*]) – a list of *Moment*s representing central moments
- **k_counter** (list[*Moment*]) – a list of *Moment*s representing raw moments
- **dmu_over_dt** – du/dt in paper
- **species** – species matrix: y_0, y_1,..., y_d
- **propensities** – propensities matrix
- **stoichiometry_matrix** – stoichiometry matrix

Returns central_moments matrix with *(len(n_counter)-1)* rows and one column per each $[n_1, ...n_d]$ combination

class `means.approximation.mea.eq_mixed_moments.`**`DBetaOverDtCalculator`**(*propensities*, *n_counter*, *stoichoimetry_matrix*, *species*)

Bases: `object`

A class providing a efficient way to recursively calculate $\frac{d\beta}{dt}$ (eq. 11 in *[Ale2013]*). A class was used here merely for optimisation reasons.

Parameters

- **propensities** – the rates/propensities of the reactions
- **n_counter** (list[*Moment*]) – a list of *Moment*s representing central moments
- **stoichoimetry_matrix** – The stoichiometry matrix. Explicitly provided by the model
- **species** – the names of the variables/species

get (*k_vec*, *e_counter*)
Provides the terms needed for equation 11 (see Ale et al. 2013). This gives the expressions for $\frac{d\beta}{dt}$ in equation 9, these are the time dependencies of the mixed moments

Parameters

- **k_vec** – k in eq. 11
- **e_counter** – e in eq. 11

Returns $\frac{d\beta}{dt}$

MEA helper functions. This part of the package provides a few small utility functions for the rest *mea*.

means.approximation.mea.mea_helpers.**derive_expr_from_counter_entry** (*expression*, *species*, *counter_entry*)
Derives an given expression with respect to arbitrary species and orders. This is used to compute $\frac{\partial^n \mathbf{n} a_l(\mathbf{x})}{\partial \mathbf{x^n}}$ in eq. 6

Parameters

- **expression** (Expr) – the expression to be derived
- **species** (list[Symbol]) – the name of the variables (typically {y_0, y_1, ..., y_n})
- **counter_entry** – an entry of counter. That is a tuple of integers of length equal to the number of variables.

For example, (0,2,1) means we derive with respect to the third variable (first order) and to the second variable (second order)

Returns the derived expression

means.approximation.mea.mea_helpers.**get_one_over_n_factorial** (**args*)
Calculates the $\frac{1}{\mathbf{n}!}$ of eq. 6 (see Ale et al. 2013). That is the invert of a product of factorials. :param counter_entry: an entry of counter. That is an array of integers of length equal to the number of variables. For instance, *counter_entry* could be *[1,0,1]* for three variables. :return: a scalar as a sympy expression

means.approximation.mea.mea_helpers.**make_k_chose_e** (*e_vec*, *k_vec*)
Computes the product $\binom{\mathbf{n}}{\mathbf{k}}$

Parameters

- **e_vec** (numpy.array) – the vector e
- **k_vec** (numpy.array) – the vector k

Returns a scalar

class `means.approximation.mea.moment_expansion_approximation.`**`MomentExpansionApproximation`**(*mod max clo- sure *clo sure **c sure*

Bases: *`means.approximation.approximation_baseclass.ApproximationBaseClass`*

A class to perform moment expansion approximation as described in *[Ale2013]* up to a given order of moment. In addition, it allows to close the Taylor expansion by using parametric values for last order central moments.

Parameters

- **`model`** (*`Model`*) – The model to be approximated
- **`max_order`** – the highest order of central moments in the resulting ODEs
- **`closure`** (*string*) – a string describing the type of closure to use. Currently, the supported closures are:

  'scalar' higher order central moments are set to zero. See *`ScalarClosure`*.

  'normal' uses normal distribution to compute last order central moments. See *`NormalClosure`*.

  'log-normal' uses log-normal distribution. See *`LogNormalClosure`*.

  'gamma' *EXPERIMENTAL*, uses gamma distribution. See *`GammaClosure`*.

- **`closure_args`** – arguments to be passed to the closure
- **`closure_kwargs`** – keyword arguments to be passed to the closure

`closure`

`run`()

Overrides the default run() method. Performs the complete analysis on the model specified during initialisation.

Returns an ODE problem which can be further used in inference and simulation.

Return type *`ODEProblem`*

`means.approximation.mea.moment_expansion_approximation.`**`mea_approximation`**(*model, max_order, closure='scalar', *closure_args, **closure_kwargs*)

A wrapper around *`MomentExpansionApproximation`*. It performs moment expansion approximation (MEA) up to a given order of moment. See *`MomentExpansionApproximation`* for details about the options.

Returns an ODE problem which can be further used in inference and simulation.

Return type *`ODEProblem`*

`means.approximation.mea.raw_to_central.`**`raw_to_central`**(*n_counter, species, k_counter*)

Expresses central moments in terms of raw moments (and other central moments). Based on equation 8 in the paper:

$$\mathbf{M}_{\mathbf{x^n}} = \sum_{k_1=0}^{n_1} \dots \sum_{k_d=0}^{n_d} \binom{\mathbf{n}}{\mathbf{k}} (-1)^{\mathbf{n}-\mathbf{k}} \mu^{\mathbf{n}-\mathbf{k}} \langle \mathbf{x^k} \rangle$$

The term $\mu^{\mathbf{n}-\mathbf{k}}$, so called alpha term is expressed with respect to *species* values that are equivalent to μ_i in the paper.

The last term, the beta term, $\langle \mathbf{x^n} \rangle$ is simply obtained from k_counter as it contains the symbols for raw moments.

Parameters

- **n_counter** (list[*Moment*]) – a list of *Moment*s representing central moments
- **species** – the symbols for species means
- **k_counter** (list[*Moment*]) – a list of *Moment*s representing raw moments

Returns a vector of central moments expressed in terms of raw moment

Module contents

Moment Expansion Approximation This part of the package implements Moment Expansion Approximation as described in *[Ale2013]*. In addition to the standard implementation, it allows to use different distribution (such as normal, log-normal and gamma) to close the moment expansion. The function `mea_approximation()` should provide all the necessary options.

Example:

```
>>> from means import mea_approximation
>>> from means.examples.sample_models import MODEL_P53
>>> ode_problem = mea_approximation(MODEL_P53,max_order=2)
>>> # equivalent to
>>> # ode_problem = mea_approximation(MODEL_P53, max_order=2, closure="scalar", value=0)
>>> print ode_problem
```

The result is an *means.core.problems.ODEProblem*. Typically, it would be further used to perform simulations (see *simulation*) and inference (see *inference*).

3.1.2 Submodules

class `means.approximation.approximation_baseclass.`**`ApproximationBaseClass`**(*model*)

Bases: `object`

A class of explicit generators for ordinary differential equations required to simulate the model provided.

Initialise the approximation.

Parameters **model** (*Model*) – Model to approximate

model

The model that is used in approximation

run()

Perform the approximation. Return a constructed `ODEProblem` object.

Returns a constructed set of equations, encoded in `ODEProblem` object.

Return type *ODEProblem*

3.1.3 Module contents

3.2 means.core package

3.2.1 Submodules

Descriptors

Descriptors are small classes that describe the terms in ODE equations or the meaning of the trajectory generations. They are the key objects in parameter inference as they provide ways for the system to know which trajectory generated to compare to which of the trajectories observed.

Descriptor objects can be initialised directly, and the appropriate class documentations should be viewed for instructions on how to do so.

If one would like to create a new descriptor object, such object must inherit from the `Descriptor` object defined below.

class `means.core.descriptors.`**`Descriptor`**
Bases: *`means.io.serialise.SerialisableObject`*

`mathtext``()`
Return the mathtext representation of this object. Used in the legend labels while plotting the trajectories. Should include the dollar signs (`$`) separating the mathtext from plaintext, e.g.

```
def mathtext(self):
    return "Some plain text string before the math representation of lambda $\lambda$"
```

Defaults to the same representation as provided by `__str__`.

`yaml_tag` **= u'!descriptor'**

class `means.core.descriptors.`**`Moment`**`(`*n_vector*, *symbol*`)`
Bases: *`means.core.descriptors.ODETermBase`*

An annotator for ODE expressions that describes that a particular expression in a set of ODEs corresponds to a Moment of the probability distribution. The particular moment is described by *`Moment.n_vector`*.

Creates an ODETerm that describes that a particular ODE term is a moment defined by the `n_vector` parameter. The said parameter should be a vector of ints describing the order of the particular species moment. For instance, for a two species system, the variance for the first species would have `n_vector=[2, 0]` as it is the expectation of the squared central moment for the first species.

Similarly, the covariance of the first and second species would have a `n_vector=[1, 1]` as it is the expectation of the product of the central moments for each of the species.

The sum of the terms of `n_vector` is referred as the *order* of the moment, and is accessible via the *`Moment.order`* attribute. The concept of `n_vector` is genearalisable to higher order moments as well.

It is worth noting that the concentrations of particular species are the first-order moments themselves, equivalent to vectors `[1, 0]` and `[0, 1]` for a two-species system. These moments are commonly referred to as the means of the systems.

When performing data inference, the `Trajectory` objects with appropriate descriptors have to be defined. In most cases, these descriptors will be the aforementioned first-order moments as most measurable data would be the mean concentrations of species, for instance:

```
>>> x_concentration_descriptor = Moment([1,0], symbol='x')
```

Note that both the moment's `n_vector` and the `symbol` parameters must match the parameters generated by the trajectory. In the example above the first species in the model need to be named `'x'`.

Parameters

- **`n_vector`** – a vector specifying the multidimensional moment e.g. `[1,0]`
- **`symbol`** – the symbol for the particular descriptor, e.g. `'x'`

`descriptor`

`is_mixed`
Returns whether the moment is a mixed moment, i.e. has a non-zero power to more than one species, or a raw moment (non-zero power to only one species).

`n_vector`
The n_vector this moment represents

`order`
The order of the moment

classmethod `to_yaml`(*dumper*, *data*)

`yaml_tag` **= u'!moment'**

class `means.core.descriptors.`**`ODETermBase`**(*symbol*)
Bases: *`means.core.descriptors.Descriptor`*

Base class for explaining terms in the ODE expressions. Instances of this class allow providing a description for each of the equations in the generated ODE system.

`descriptor`
Returns an uniquely identifying descriptor for this particular ODE term.

`mathtext`()

`symbol`

class `means.core.descriptors.`**`VarianceTerm`**(*position*, *symbol*)
Bases: *`means.core.descriptors.ODETermBase`*

Signifies that a particular equation generated from the model is part of a Variance Term

Creates a Descriptor for a particular ODE in the system that signifies that that particular equation computes the position-th term of a covariance matrix, where position is some tuple (row,column).

It is used in LNA approximation as there we need to deal with moment and variance terms differently

Parameters

- **`position`** – position in the covariance matrix
- **`symbol`** – symbol assigned to the term

`position`

classmethod `to_yaml`(*dumper*, *data*)

`yaml_tag` **= '!variance-term'**

Model

Model objects describe a system in terms of **stochastic** reaction propensities/rates, species/variables, constants and a stoichiometry matrix. Generally, describing a model is a pre-requisite for any subsequent analysis.

An example showing the p53 model could be encoded:

```
>>> from means import Model
>>> my_model = Model(parameters=['c_0',   # P53 production rate
>>>                              'c_1',   # MDM2-independent p53 degradation rate
>>>                              'c_2',   # saturating p53 degradation rate
>>>                              'c_3',   # P53-dependent MDM2 production rate
>>>                              'c_4',   # MDM2 maturation rate
>>>                              'c_5',   # MDM2 degradation rate
>>>                              'c_6'],  # P53 threshold of degradation by MDM2
>>>                  species=['y_0',   # Concentration of p53
>>>                           'y_1',   # Concentration of MDM2 precursor
>>>                           'y_2'],  # Concentration of MDM2
>>>                  stoichiometry_matrix=[[1, -1, -1, 0, 0, 0],
>>>                                        [0, 0, 0, 1, -1, 0],
>>>                                        [0, 0, 0, 0, 1, -1]],
>>>                  propensities=['c_0',
>>>                                'c_1*y_0',
>>>                                'c_2*y_2*y_0/(y_0+c_6)',
>>>                                'c_3*y_0',
>>>                                'c_4*y_1',
>>>                                'c_5*y_2'])
```

Printing the model to ensure everything is all right:

```
>>> print my_model
```

Typically, a model would be used for approximation (e.g. `moment_expansion_approximation`, or `lna`) and stochastic simulations (e.g. `ssa`).

---

class `means.core.model.`**`Model`**(*species*, *parameters*, *propensities*, *stoichiometry_matrix*)

Bases: `means.io.serialise.SerialisableObject`, `means.io.latex.LatexPrintableObject`

Stores the model of reactions we want to analyse

Creates a *Model* object that stores the model of reactions we want to analyse :param species: variables of the model, as *sympy.Symbol's, i.e. species :param parameters: parameters of the model, as 'sympy* symbols :param propensities: a matrix of propensities for each of the reaction in the model. :param stoichiometry_matrix: stoichiometry matrix for the model

`number_of_parameters`

`number_of_reactions`

`number_of_species`

`parameters`

`propensities`

`species`

`stoichiometry_matrix`

classmethod `to_yaml`(*dumper*, *data*)

`validate`()

Validates whether the particular model is created properly

`yaml_tag` = **u'!model'**

Problems

This part of the package implement classes describing "problems". Problems are required inputs for simulation and inference. Currently, there are two types of problems:

- **A *ODEProblem* is a system of differential equations describing** the temporal behaviour of the system. They are typically obtained through approximation (e.g. *moment_expansion_approximation*, or *lna*)
- A *StochasticProblem* can be used for stochastic simulationsand can be simply built from a *Model*:

```
>>> from means import StochasticProblem
>>> from means.examples.sample_models import MODEL_P53
>>> my_stoch_prob = StochasticProblem(MODEL_P53)
```

class means.core.problems.**ODEProblem**(*method*, *left_hand_side_descriptors*, *right_hand_side*, *parameters*)

Bases: *means.io.serialise.SerialisableObject*, *means.io.latex.LatexPrintableObject*, *means.util.memoisation.MemoisableObject*

Creates a *ODEProblem* object that stores a system of ODEs describing the kinetic of a system. Typically, *ODEProblem's will be further used in simulations (see :mod:'~means.simulation*) and inference (see *inference*).

Parameters **method** – a string describing the method used to generate the problem.

Currently, 'MEA' and 'LNA' are supported" :param left_hand_side_descriptors: the left hand side of equations as a list of

Descriptor objects (such as *Moment*)

Parameters

- **right_hand_side** – the right hand side of equations
- **parameters** – the parameters of the model

descriptor_for_symbol(*symbol*)

Given the symbol associated with the problem. Returns the *Descriptor* associated with that symbol

Parameters **symbol** (basestring|:class:*sympy.Symbol*) – Symbol

Returns

latex

left_hand_side

left_hand_side_descriptors

method

number_of_equations

number_of_parameters

number_of_species

parameters

right_hand_side

`right_hand_side_as_function`

Generates and returns the right hand side of the model as a callable function that takes two parameters: values for variables and values for constants, e.g. 'f(values_for_variables=[1,2,3], values_for_constants=[3,4,5])

This function is directly used in *means.simulation.Simulation* :return: :rtype: function

classmethod `to_yaml`(*dumper*, *data*)

`validate`()

Validates whether the ODE equations provided make sense i.e. the number of right-hand side equations match the number of left-hand side equations.

`variables`

`yaml_tag` = **'!problem'**

class `means.core.problems.`**`StochasticProblem`**(*model*)

Bases: *`means.core.model.Model`*, *`means.util.memoisation.MemoisableObject`*

The formulation of a model for stochastic simulations such as GSSA (see *`means.simulation.ssa`*).

`change`

`propensities_as_function`

3.2.2 Module contents

This package defines the common classes that are used within all of the other means subpackages.

The module exposes the descriptor classes, such as `Descriptor`, `VarianceTerm`, `Moment` and `ODETermBase` that are used to describe the types of trajectories generated, as well as certain terms in the ODE equations.

Similarly, both the `StochasticProblem` and `ODEProblem` classes that are used in stochastic and deterministic simulations respectively are exposed by this module.

Finally, the `Model` class, which provides a standard interface to describe a biological model, and can be thought to be the center of the whole package, is also implemented here.

3.3 means.examples package

3.3.1 Submodules

3.3.2 Module contents

3.4 means.inference package

3.4.1 Submodules

Distances

This part of the package implements functions to compute distance between two set of trajectories. Distance functions are typically required for parameter inference (see *`means.inference.inference`*).

`means.inference.distances.`**`gamma`**(*simulated_trajectories*, *observed_trajectories_lookup*)
Returns the negative log-likelihood of the observed trajectories assuming a gamma distribution on the simulated trajectories values

Parameters

- **`simulated_trajectories`** (list[`means.simulation.Trajectory`]) – Simulated trajectories
- **`observed_trajectories_lookup`** (*dict*) – A dictionary of (trajectory.description: trajectory) of observed trajectories

Returns

`means.inference.distances.`**`get_distance_function`**(*distance*)
Returns the distance function from the string name provided

Parameters **`distance`** – The string name of the distributions

Returns

`means.inference.distances.`**`lognormal`**(*simulated_trajectories*, *observed_trajectories_lookup*)

Returns the negative log-likelihood of the observed trajectories assuming a log-normal distribution

on the simulated trajectories values

Parameters

- **`simulated_trajectories`** (list[`means.simulation.Trajectory`]) – Simulated trajectories
- **`observed_trajectories_lookup`** (*dict*) – A dictionary of (trajectory.description: trajectory) of observed trajectories

Returns

`means.inference.distances.`**`normal`**(*simulated_trajectories*, *observed_trajectories_lookup*)

Returns the negative log-likelihood of the observed trajectories assuming a normal distribution

on the simulated trajectories values

Parameters

- **`simulated_trajectories`** (list[`means.simulation.Trajectory`]) – Simulated trajectories
- **`observed_trajectories_lookup`** (*dict*) – A dictionary of (trajectory.description: trajectory) of observed trajectories

Returns

`means.inference.distances.`**`sum_of_squares`**(*simulated_trajectories*, *observed_trajectories_lookup*)
Returns the sum-of-squares distance between the simulated_trajectories and observed_trajectories

Parameters

- **`simulated_trajectories`** (list[`means.simulation.Trajectory`]) – Simulated trajectories
- **`observed_trajectories_lookup`** (*dict*) – A dictionary of (trajectory.description: trajectory) of observed trajectories

Returns the distance between simulated and observed trajectories

Return type float

`means.inference.hypercube.`**`hypercube`**(*number_of_samples*, *variables*)

This implements Latin Hypercube Sampling.

See https://mathieu.fenniak.net/latin-hypercube-sampling/ for intuitive explanation of what it is

Parameters

- **`number_of_samples`** – number of segments/samples
- **`variables`** – initial parameters and conditions (list of ranges, i.e. (70, 110), (0.1, 0.5) ..)

Returns

class `means.inference.inference.`**`Inference`**(*problem*, *starting_parameters*, *starting_conditions*, *variable_parameters*, *observed_trajectories*, *distance_function_type='sum_of_squares'*, ***simulation_kwargs*)

Bases: *`means.io.serialise.SerialisableObject`*, *`means.util.memoisation.MemoisableObject`*

Parameters

- **`problem`** (ODEProblem) – ODEProblem to infer data for
- **`starting_parameters`** (*iterable*) – A list of starting values for each of the model's parameters
- **`starting_conditions`** (*iterable*) – A list of starting values for each of the initial conditions. All unspecified initial conditions will be set to zero
- **`variable_parameters`** – A dictionary of variable parameters, in the format `{parameter_symbol: (min_value, max_value)}` where the range `(min_value, max_value)` is the range of the allowed parameter values. If the range is None, parameters are assumed to be unbounded.
- **`observed_trajectories`** – A list of *Trajectory* objects containing observed data values.
- **`distance_function_type`** – Method of calculating the data fit. Currently supported values are *'sum_of_squares'*

  minimisation of the sum of squares distance between trajectories

  'gamma' maximum likelihood optimisation assuming gamma distribution

  'normal' maximum likelihood optimisation assuming normal distribution

  'lognormal' maximum likelihood optimisation assuming lognormal distribution
- **`simulation_kwargs`** – Keyword arguments to pass to the `means.simulation.Simulation` instance

`constraints`

`distance_function_type`

`infer`(*return_intermediate_solutions=False*, *return_distance_landscape=False*, *solver_exceptions_limit=100*)

Parameters

- **`return_intermediate_solutions`** – Return the intermediate parameter solutions that optimisation
- **`return_distance_landscape`** – Return the distance landscape that was explored

`observed_timepoints`

`observed_trajectories`

`observed_trajectories_lookup`
Similar to observed_trajectories, but returns a dictionary of {description:trajectory}

`problem`

Return type *ODEProblem*

`simulation`

`simulation_kwargs`

`starting_conditions`

`starting_conditions_with_variability`

`starting_parameters`

`starting_parameters_with_variability`

classmethod `to_yaml`(*dumper*, *data*)

`variable_parameters`

`yaml_tag` = **'!inference'**

class `means.inference.inference.`**`InferenceWithRestarts`**(*problem*, *number_of_samples*, *starting_parameter_ranges*, *starting_conditions_ranges*, *variable_parameters*, *observed_trajectories*, *distance_function_type='sum_of_squares'*)

Bases: `means.util.memoisation.MemoisableObject`

Parameter Inference Method that utilises multiple seed points for the optimisation.

Parameters

- **`problem`** (`ODEProblem`) – Problem to infer parameters for
- **`number_of_samples`** – Number of the starting points to randomly pick
- **`starting_parameter_ranges`** – Valid initialisation ranges for the parameters
- **`starting_conditions_ranges`** – Valid initialisation ranges for the initial conditions. If some initial conditions are not set, they will default to 0.
- **`variable_parameters`** – A dictionary of variable parameters, in the format `{parameter_symbol: (min_value, max_value)}` where the range `(min_value, max_value)` is the range of the allowed parameter values. If the range is None, parameters are assumed to be unbounded.
- **`observed_trajectories`** – A list of *Trajectory* objects containing observed data values.
- **`distance_function_type`** – Method of calculating the data fit. Currently supported values are - 'sum_of_squares' - min sum of squares optimisation - 'gamma' - maximum likelihood optimisation assuming gamma distribution - 'normal'- maximum likelihood optimisation assuming normal distribution - 'lognormal' - maximum likelihood optimisation assuming lognormal distribution - any callable function, that takes two arguments: simulated trajectories (list)

  and observed trajectories lookup (dictionary of description: trajectory pairs) see `means.inference.distances.sum_of_squares()` for examples of such functions

distance_function_type

infer(*number_of_processes=1*, **args*, ***kwargs*)

Parameters

- **number_of_processes** – If set to more than 1, the inference routines will be paralellised using `multiprocessing` module
- **args** – arguments to pass to `Inference.infer()`
- **kwargs** – keyword arguments to pass to `Inference.infer()`

Returns

number_of_samples

observed_trajectories

problem

Return type *ODEProblem*

starting_conditions_ranges

starting_parameter_ranges

variable_parameters

Parameter Inference Parallelisation

This part of the package provides helper functions to make parameter inference run in parallel.

`means.inference.parallelisation.`**multiprocessing_apply_infer**(*object_id*)
Used in the InferenceWithRestarts class. Needs to be in global scope for multiprocessing module to pick it up

`means.inference.parallelisation.`**multiprocessing_pool_initialiser**(*objects*, *infer_args*, *infer_kwargs*)

`means.inference.parallelisation.`**raw_results_in_parallel**(*inference_objects*, *number_of_processes*, **args*, ***kwargs*)

`means.inference.plotting.`**plot_2d_trajectory**(*x*, *y*, *x_label='‘*, *y_label='‘*, *legend=False*, *ax=None*, *start_and_end_locations_only=False*, *start_marker='bo'*, *end_marker='rx'*, *start_label='Start'*, *end_label='End'*, **args*, ***kwargs*)

`means.inference.plotting.`**plot_contour**(*x*, *y*, *z*, *x_label*, *y_label*, *ax=None*, *fmt='%.3f'*, **args*, ***kwargs*)

Inference Results

This part of the package provides classes to store and manage the results of inference.

class `means.inference.results.`**ConvergenceStatusBase**(*convergence_achieved*)
Bases: `means.io.serialise.SerialisableObject`

convergence_achieved

class `means.inference.results.`**`InferenceResult`**(*inference*, *optimal_parameters*, *optimal_initial_conditions*, *distance_at_minimum*, *convergence_status*, *solutions*, *distance_landscape*)

Bases: *`means.io.serialise.SerialisableObject`*, *`means.util.memoisation.MemoisableObject`*

Parameters

- **`inference`** –
- **`optimal_parameters`** –
- **`optimal_initial_conditions`** –
- **`distance_at_minimum`** –
- **`convergence_status`** (*`ConvergenceStatusBase`*) –
- **`solutions`** –
- **`distance_landscape`** – distance landscape - all the distances

`convergence_status`

`distance_at_minimum`

`distance_landscape`

The distance to the observed values at each point of the parameter space that was checked. This is different from the solutions list as it returns all the values checked, not only the ones that were chosen as intermediate steps by the solver :return: a list of (parameters, conditions, distance) tuples or None if the inference did not track it :rtype: list[tuple]|None

`distance_landscape_as_3d_data`(*x_axis*, *y_axis*)

Returns the distance landscape as three-dimensional data for the specified projection.

Parameters

- **`x_axis`** – variable to be plotted on the x axis of projection
- **`y_axis`** – variable to be plotted on the y axis of projection

Returns a 3-tuple (x, y, z) where x and y are the lists of coordinates and z the list of distances at respective coordinates

`inference`

`intermediate_trajectories`

`observed_trajectories`

`optimal_initial_conditions`

`optimal_parameters`

`optimal_trajectories`

`parameter_index`(*parameter_name*)

`plot`(*plot_intermediate_solutions=True*, *plot_observed_data=True*, *plot_starting_trajectory=True*, *plot_optimal_trajectory=True*, *filter_plots_function=None*, *legend=True*, *kwargs_observed_data=None*, *kwargs_starting_trajectories=None*, *kwargs_optimal_trajectories=None*, *kwargs_intermediate_trajectories=None*)

Plot the inference result.

Parameters

- **`plot_intermediate_solutions`** – plot the trajectories resulting from the intermediate solutions as well
- **`filter_plots_function`** – A function that takes a trajectory object and returns True if it should be plotted and false if not. None plots all available trajectories
- **`legend`** – Whether to draw the legend or not
- **`kwargs_observed_data`** – Kwargs to be passed to the `trajectory.plot` function for the observed data
- **`kwargs_starting_trajectories`** – kwargs to be passed to the `trajectory.plot` function for the starting trajectories
- **`kwargs_optimal_trajectories`** – kwargs to be passed to the `trajectory.plot` function for the optimal trajectories
- **`kwargs_intermediate_trajectories`** – kwargs to be passed to the `trajectory.plot` function for the intermediate trajectories

`plot_distance_landscape_projection`(*x_axis*, *y_axis*, *ax=None*, **args*, ***kwargs*)
Plots the projection of distance landscape (if it was returned), onto the parameters specified

Parameters

- **`x_axis`** – symbol to plot on x axis
- **`y_axis`** – symbol to plot on y axis
- **`ax`** – axis object to plot onto
- **`args`** – arguments to pass to `matplotlib.pyplot.contourf()`
- **`kwargs`** – keyword arguments to pass to `matplotlib.pyplot.contourf()`

Returns

`plot_trajectory_projection`(*x_axis*, *y_axis*, *legend=False*, *ax=None*, *start_and_end_locations_only=False*, *start_marker='bo'*, *end_marker='rx'*, **args*, ***kwargs*)
Plots the projection of the trajectory through the parameter space the minimisation algorithm took.

Since parameter space is often high-dimensional and paper can realistically represent only two, one needs to specify the `x_axis`, and `y_axis` arguments with the variable names to project the high-dimensional grid, on, i.e. `x_axis='c_1'`, `y_axis='c_4'`

Parameters

- **`x_axis`** (str|:class:~*sympy.Symbol*) – variable name (parameter or left hand side of equation) to project x axis onto
- **`y_axis`** (str|:class:~*sympy.Symbol*) – variable name to project y axis onto
- **`legend`** (*bool*) – Whether to display legend or not
- **`ax`** – Axis to plot onto (defaults to `matplotlib.pyplot.gca()` if not set)
- **`start_and_end_locations_only`** – If set to true, will not plot the trajectory, but only start and end parameter
- **`start_marker`** – The marker to use for start of trajectory, defaults to blue circle
- **`end_marker`** – The marker to use for end of trajectory, defaults to red x
- **`args`** – Arguments to pass to `matplotlib.pyplot.plot()` function
- **`kwargs`** – Keyword arguments to pass to `matplotlib.pyplot.plot()` function

problem

Return type *ODEProblem*

solutions

Solutions at each each iteration of optimisation. :return: a list of (parameters, conditions) pairs :rtype: list[tuple]|None

solutions_as_2d_trajectories (*x_axis*, *y_axis*)

Returns the *InferenceResult.solutions* as a plottable 2d trajectory.

Parameters

- **x_axis** – the variable to be on the x axis of projection
- **y_axis** – the variable to be on the y axis of preojection

Returns a tuple x, y specifying lists of x and y coordinates of projection

starting_initial_conditions

starting_parameters

starting_trajectories

classmethod to_yaml (*dumper*, *data*)

yaml_tag = '!inference-result'

class `means.inference.results.`**InferenceResultsCollection** (*inference_results*)

Bases: *means.io.serialise.SerialisableObject*

best

classmethod from_yaml (*loader*, *node*)

number_of_results

plot ()

plot_distance_landscape_projection (*x_axis*, *y_axis*, *ax=None*, **args*, ***kwargs*)

Plots the distance landscape jointly-generated from all the results

Parameters

- **x_axis** – symbol to plot on x axis
- **y_axis** – symbol to plot on y axis
- **ax** – axis object to plot onto
- **args** – arguments to pass to `matplotlib.pyplot.contourf()`
- **kwargs** – keyword arguments to pass to `matplotlib.pyplot.contourf()`

Returns

plot_trajectory_projection (*x_axis*, *y_axis*, **args*, ***kwargs*)

Plots trajectory projection on the specified x and y axes See *InferenceResult.plot_trajectory_projection()* for information on the arguments and keyword arguments

Parameters

- **x_axis** – variable to be plotted on the x axis of the projection
- **y_axis** – variable to be plotted on the y axis of the projection
- **args** – arguments to be passed to *InferenceResult.plot_trajectory_projection()*

- **kwargs** – keyword arguments to be passed to *InferenceResult.plot_trajectory_projection()*

results

Returns The results of performed inferences

Return type list[*InferenceResult*]

classmethod to_yaml(*dumper*, *data*)

yaml_tag = '!serialisable-results-collection'

class means.inference.results.**NormalConvergenceStatus**(*warn_flag*, *iterations_taken*, *function_calls_made*)

Bases: *means.inference.results.ConvergenceStatusBase*

function_calls_made

iterations_taken

classmethod to_yaml(*dumper*, *data*)

warn_flag

yaml_tag = '!convergence-status'

class means.inference.results.**SolverErrorConvergenceStatus**

Bases: *means.inference.results.ConvergenceStatusBase*

classmethod to_yaml(*dumper*, *data*)

yaml_tag = '!multiple-solver-errors'

3.4.2 Module contents

Parameter Inference

This part of the package provides utilities for parameter inference. Parameter inference will try to find the set of parameters which produces trajectories with minimal distance to the observed trajectories. Different distance functions are implemented (such as functions minimising the sum of squares error or functions based on parametric likelihood), but it is also possible to use custom distance functions.

The package provides support for both inference from a single starting point (`Inference`) or inference from random starting points (`InferenceWithRestarts`).

Some basic inference result plotting functionality is also provided by the package, see the documentation for `InferenceResult` for more information on this.

3.5 means.io package

3.5.1 Submodules

class means.io.latex.**LatexPrintableObject**

Bases: `object`

latex

Returns the latex text that could be printed into .tex document :return:

output_latex(*filename_or_file_handle*)
Output the file to a latex document :param filename_or_file_handle: filename or already opened file handle to output to :return:

`means.io.sbml.`**read_sbml**(*filename*)
Read the model from a SBML file.

Parameters **filename** – SBML filename to read the model from

Returns A tuple, consisting of `Model` instance, set of parameter values, and set of initial conditions variables.

class `means.io.serialise.`**MeansDumper**(*stream*, *default_style=None*, *default_flow_style=None*, *canonical=None*, *indent=None*, *width=None*, *allow_unicode=None*, *line_break=None*, *encoding=None*, *explicit_start=None*, *explicit_end=None*, *version=None*, *tags=None*)
Bases: `yaml.dumper.Dumper`

class `means.io.serialise.`**MeansLoader**(*stream*)
Bases: `yaml.loader.Loader`

class `means.io.serialise.`**SerialisableObject**
Bases: `yaml.YAMLObject`

classmethod from_file(*filename_or_file_object*)
Create new instance of the object from the file. :param filename_or_file_object: the filename of the file to read from or already opened file buffer :type filename_or_file_object: basestring|file

classmethod from_yaml(*loader*, *node*)

to_file(*filename_or_file_object*)
Save the object to the file, specified by filename *file_* or the buffer provided in *file_* :param filename_or_file_object: filename of the file to save the object to, or already open file buffer

`means.io.serialise.`**dump**(*object_*)

`means.io.serialise.`**from_file**(*filename_or_file_object*)
Read data from the specified file. :param filename_or_file_object: filename of the file or an opened `file` buffer to that file :type filename_or_file_object: basestring|file :return:

`means.io.serialise.`**load**(*data*)

`means.io.serialise.`**to_file**(*data*, *filename_or_file_object*)
Write `data` to a file specified by either filename of the file or an opened `file` buffer.

Parameters

- **data** – Object to write to file
- **filename_or_file_object** (*basestring|file*) – filename/or opened file buffer to write to

3.5.2 Module contents

Module for Input/Output operations.

Human Readable Serialisation

The module implements common methods to serialise and deserialise MEANS objects. Namely the module provides functions `means.io.dump()` and `means.io.load()` that would serialise and deserialise the said objects into `yaml` format. These serialised representations can be written to or read from files with the help of `means.io.to_file()` and `means.io.from_file()` functions.

For the user's convenience, the said methods are also attached to all serialisable objects, e.g. `means.core.Model.from_file()` method would allow the user to read `means.core.Model` object from file directly.

Binary Serialisation

We do not provide any convenience functions for binary serialisation of the object, because `pickle` package, which is in the default distribution of Python, has no problems of performing these tasks on MEANS objects.

We recommend using `pickle`, rather than *`means.io`* whenever fast serialisation is preferred to human readability.

SBML

This module also provides support for the input from SBML files. If the `libsbml` is installed in the user's system and has the appropriate python bindings, the function `means.io.read_sbml()` can be used to parse the files in SBML format to `means.core.Model` objects.

3.6 means.simulation package

3.6.1 Submodules

Simulation Descriptors

Descriptors that are local to the simulation package

class `means.simulation.descriptors.`**`PerturbedTerm`**(*ode_term*, *parameter*, *delta=0.01*)

Bases: *`means.core.descriptors.Descriptor`*

A `Descriptor` term that describes a particular object represents the sensitivity of some ODE term with respect to some parameter. In other words, sensitivity term describes $s_{ij}(t) = \frac{\partial y_i(t)}{\partial p_j}$ where y_i is the ODE term described above and p_j is the parameter.

This class is used to describe sensitivity trajectories returned by *`means.simulation.simulate.Simulation`*

Parameters

- **`ode_term`** (`ODETermBase`) – the ode term whose sensitivity is being computed
- **`parameter`** (`sympy.Symbol`) – parameter w.r.t. which the sensitivity is computed

`delta`

`ode_term`

`parameter`

class `means.simulation.descriptors.`**`SensitivityTerm`**(*ode_term*, *parameter*)

Bases: *`means.core.descriptors.Descriptor`*

A `Descriptor` term that describes a particular object represents the sensitivity of some ODE term with respect to some parameter. In other words, sensitivity term describes $s_{ij}(t) = \frac{\partial y_i(t)}{\partial p_j}$ where y_i is the ODE term described above and p_j is the parameter.

This class is used to describe sensitivity trajectories returned by *`means.simulation.simulate.Simulation`*

Parameters

- **ode_term** (ODETermBase) – the ode term whose sensitivity is being computed
- **parameter** (sympy.Symbol) – parameter w.r.t. which the sensitivity is computed

mathtext()

ode_term

parameter

classmethod to_yaml(*dumper*, *data*)

yaml_tag = **'!sensitivity-term'**

Simulate

This part of the package provides utilities for simulate the dynamic of an `ODEProblem`.

A wide range of numerical solver are available:

```
>>> from means import Simulation
>>> print Simulation.supported_solvers()
```

In order to simulate a system, it is necessary to provide values for the initial conditions and parameters (constants):

```
>>> from means import mea_approximation
>>> from means.examples.sample_models import MODEL_P53
>>> from means import Simulation
>>> import numpy as np
>>>
>>> ode_problem = mea_approximation(MODEL_P53,max_order=2)
>>> # We provide initial conditions, constants and time range
>>> RATES = [90, 0.002, 1.7, 1.1, 0.93, 0.96, 0.01]
>>> INITIAL_CONDITIONS = [70, 30, 60]
>>> TMAX = 40
>>> TIME_RANGE = np.arange(0, TMAX, .1)
>>> #This is where we simulate the system to obtain trajectories
>>> simulator = Simulation(ode_problem)
>>> trajectories = simulator.simulate_system(RATES, INITIAL_CONDITIONS, TIME_RANGE)
```

A `TrajectoryCollection` object (see `trajectory`) is created. See the documentation of `Simulation` for additional information.

---

class means.simulation.simulate.**Simulation**(*problem*, *solver='ode15s'*, ***solver_options*)
Bases: `means.io.serialise.SerialisableObject`

Class that allows to perform simulations of the trajectories for a particular problem. Implements all ODE solvers supported by Assimulo package.

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **compute_sensitivities** – Whether the model should test parameter sensitivity or not
- **solver** (*basestring*) – the solver to use. Currently, the solvers that available are:

  'cvode' sundials CVode solver, as implemented in `assimulo.solvers.sundials.CVode`

'dopri5' Dopri5 solver, see `assimulo.solvers.runge_kutta.Dopri5`

'euler' Euler solver, see `assimulo.solvers.euler.ExplicitEuler`

***'ode15s*:** sundials CVODE solver, with default parameters set to mimick the MATLAB's See *`ODE15sMixin`* for the list of these parameters.

'lsodar' LSODAR solver, see `assimulo.solvers.odepack.LSODAR`

'radau5' Radau5 solver, see `assimulo.solvers.radau5.Radau5ODE`

'rodas' Rosenbrock method of order (3)4 with step-size control, see `assimulo.solvers.rosenbrock.RodasODE`

'rungekutta34' Adaptive Runda-Kutta of order four, see `assimulo.solvers.runge_kutta.RungeKutta34`

'rungekutta4' Runge-Kutta method of order 4, see `assimulo.solvers.runge_kutta.RungeKutta4`

The list of these solvers is always accessible at runtime from *`Simulation.supported_solvers()`* method.

- **`solver_options`** – options to set in the solver. Consult Assimulo documentation for available options for information on specific options available.

`problem`

`simulate_system`(*parameters*, *initial_conditions*, *timepoints*)

Simulates the system for each of the timepoints, starting at initial_constants and initial_values values

Parameters

- **`parameters`** – list of the initial values for the constants in the model. Must be in the same order as in the model
- **`initial_conditions`** – List of the initial values for the equations in the problem. Must be in the same order as these equations occur. If not all values specified, the remaining ones will be assumed to be 0.
- **`timepoints`** – A list of time points to simulate the system for

Returns a list of `Trajectory` objects, one for each of the equations in the problem

Return type list[`Trajectory`]

`solver`

`solver_options`

classmethod `supported_solvers`()

List the supported solvers for the simulations.

```
>>> Simulation.supported_solvers()
['cvode', 'dopri5', 'euler', 'lsodar', 'radau5', 'rodas', 'rungekutta34', 'rungekutta4', 'od
```

Returns the names of the solvers supported for simulations

classmethod `to_yaml`(*dumper*, *data*)

`yaml_tag` = **'!simulation'**

class `means.simulation.simulate.`**`SimulationWithSensitivities`**`(`*problem*, *solver='ode15s'*, ***solver_options*`)`

Bases: *`means.simulation.simulate.Simulation`*

A similar class to it's baseclass *`Simulation`*. Performs simulations of the trajectories for each of the ODEs in given problem and performs sensitivity simulations for the problem's parameters.

Parameters

- **`problem`** (ODEProblem) – Problem to simulate
- **`compute_sensitivities`** – Whether the model should test parameter sensitivity or not
- **`solver`** (*basestring*) – the solver to use. Currently, the solvers that available are:

  'cvode' sundials CVode solver, as implemented in `assimulo.solvers.sundials.CVode`

  ***'ode15s'*:** sundials CVODE solver, with default parameters set to mimick the MATLAB's See *`ODE15sMixin`* for the list of these parameters.

  The list of these solvers is always accessible at runtime from *`SimulationWithSensitivities.supported_solvers()`* method.

- **`solver_options`** – options to set in the solver. Consult Assimulo documentation for available options for information on specific options available.

`simulate_system``(`*parameters*, *initial_conditions*, *timepoints*`)`

Simulates the system for each of the timepoints, starting at initial_constants and initial_values values

Parameters

- **`parameters`** – list of the initial values for the constants in the model. Must be in the same order as in the model
- **`initial_conditions`** – List of the initial values for the equations in the problem. Must be in the same order as these equations occur. If not all values specified, the remaining ones will be assumed to be 0.
- **`timepoints`** – A list of time points to simulate the system for

Returns a list of `TrajectoryWithSensitivityData` objects, one for each of the equations in the problem

Return type list[`TrajectoryWithSensitivityData`]

classmethod **`supported_solvers`**`()`

List the supported solvers for the simulations.

```
>>> SimulationWithSensitivities.supported_solvers()
['cvode', 'ode15s']
```

Returns the names of the solvers supported for simulations

Solvers

This part of the package provides wrappers around Assimulo solvers.

class `means.simulation.solvers.`**`CVodeMixin`**

Bases: *`means.simulation.solvers.UniqueNameInitialisationMixin`*, `object`

classmethod `unique_name()`

class `means.simulation.solvers.`**`CVodeSolver`**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: `means.simulation.solvers.SolverBase`, `means.simulation.solvers.CVodeMixin`

Parameters

- **`problem`** (`ODEProblem`) – Problem to simulate
- **`parameters`** (`iterable`) – Parameters of the solver. One entry for each constant in *problem*
- **`initial_conditions`** (`iterable`) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **`starting_time`** (*float*) – Starting time for the solver, defaults to 0.0
- **`options`** – Options to be passed to the specific instance of the solver.

class `means.simulation.solvers.`**`CVodeSolverWithSensitivities`**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: `means.simulation.solvers.SensitivitySolverBase`, `means.simulation.solvers.CVodeMixin`

Parameters

- **`problem`** (`ODEProblem`) – Problem to simulate
- **`parameters`** (`iterable`) – Parameters of the solver. One entry for each constant in *problem*
- **`initial_conditions`** (`iterable`) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **`starting_time`** (*float*) – Starting time for the solver, defaults to 0.0
- **`options`** – Options to be passed to the specific instance of the solver.

class `means.simulation.solvers.`**`Dopri5Solver`**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: `means.simulation.solvers.SolverBase`, `means.simulation.solvers.UniqueNameInitialisat`

Parameters

- **`problem`** (`ODEProblem`) – Problem to simulate
- **`parameters`** (`iterable`) – Parameters of the solver. One entry for each constant in *problem*
- **`initial_conditions`** (`iterable`) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **`starting_time`** (*float*) – Starting time for the solver, defaults to 0.0
- **`options`** – Options to be passed to the specific instance of the solver.

classmethod `unique_name()`

class `means.simulation.solvers.`**`ExplicitEulerSolver`**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: `means.simulation.solvers.SolverBase`, `means.simulation.solvers.UniqueNameInitialisat`

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

simulate (*timepoints*)

classmethod unique_name ()

class means.simulation.solvers.**LSODARSolver** (*problem*, *parameters*, *initial_conditions*, *starting_time=0.0*, ***options*)

Bases: *means.simulation.solvers.SolverBase*, *means.simulation.solvers.UniqueNameInitialisat*

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

classmethod unique_name ()

class means.simulation.solvers.**ODE15sLikeSolver** (*problem*, *parameters*, *initial_conditions*, *starting_time=0.0*, ***options*)

Bases: *means.simulation.solvers.SolverBase*, *means.simulation.solvers.ODE15sMixin*

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

class means.simulation.solvers.**ODE15sMixin**

Bases: *means.simulation.solvers.CVodeMixin*

A CVODE solver that mimicks the parameters used in ode15s solver in MATLAB.

The different parameters that are set differently by default are:

discr Set to 'BDF' by default

atol Set to 1e-6

rtol Set to 1e-3

ATOL = 1e-06

MINH = 5.684342e-14

RTOL = 0.001

classmethod unique_name()

class means.simulation.solvers.**ODE15sSolverWithSensitivities**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: *means.simulation.solvers.SensitivitySolverBase*, *means.simulation.solvers.ODE15sMixin*

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

class means.simulation.solvers.**Radau5Solver**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: *means.simulation.solvers.SolverBase*, *means.simulation.solvers.UniqueNameInitialisat*

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

classmethod unique_name()

class means.simulation.solvers.**RodasSolver**(*problem, parameters, initial_conditions, starting_time=0.0, **options*)

Bases: *means.simulation.solvers.SolverBase*, *means.simulation.solvers.UniqueNameInitialisat*

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

classmethod unique_name()

class `means.simulation.solvers.`**`RungeKutta34Solver`**(*problem*, *parameters*, *initial_conditions*, *starting_time=0.0*, ***options*)

Bases: *`means.simulation.solvers.SolverBase`*, *`means.simulation.solvers.UniqueNameInitialisat`*

Parameters

- **`problem`** (`ODEProblem`) – Problem to simulate
- **`parameters`** (`iterable`) – Parameters of the solver. One entry for each constant in *problem*
- **`initial_conditions`** (`iterable`) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **`starting_time`** (*float*) – Starting time for the solver, defaults to 0.0
- **`options`** – Options to be passed to the specific instance of the solver.

classmethod **`unique_name`**()

class `means.simulation.solvers.`**`RungeKutta4Solver`**(*problem*, *parameters*, *initial_conditions*, *starting_time=0.0*, ***options*)

Bases: *`means.simulation.solvers.SolverBase`*, *`means.simulation.solvers.UniqueNameInitialisat`*

Parameters

- **`problem`** (`ODEProblem`) – Problem to simulate
- **`parameters`** (`iterable`) – Parameters of the solver. One entry for each constant in *problem*
- **`initial_conditions`** (`iterable`) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **`starting_time`** (*float*) – Starting time for the solver, defaults to 0.0
- **`options`** – Options to be passed to the specific instance of the solver.

`simulate`(*timepoints*)

classmethod **`unique_name`**()

class `means.simulation.solvers.`**`SensitivitySolverBase`**(*problem*, *parameters*, *initial_conditions*, *starting_time=0.0*, ***options*)

Bases: *`means.simulation.solvers.SolverBase`*

Parameters

- **`problem`** (`ODEProblem`) – Problem to simulate
- **`parameters`** (`iterable`) – Parameters of the solver. One entry for each constant in *problem*
- **`initial_conditions`** (`iterable`) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **`starting_time`** (*float*) – Starting time for the solver, defaults to 0.0
- **`options`** – Options to be passed to the specific instance of the solver.

class `means.simulation.solvers.`**`SolverBase`**(*problem*, *parameters*, *initial_conditions*, *starting_time=0.0*, ***options*)

Bases: *`means.util.memoisation.MemoisableObject`*

This acts as a base class for ODE solvers used in *means*. It wraps around the solvers available in **:module:‘assimulo‘** package, and provides some basic functionality that allows solvers be used with *means* objects.

Parameters

- **problem** (ODEProblem) – Problem to simulate
- **parameters** (iterable) – Parameters of the solver. One entry for each constant in *problem*
- **initial_conditions** (iterable) – Initial conditions of the system. One for each of the equations. Assumed to be zero, if not specified
- **starting_time** (*float*) – Starting time for the solver, defaults to 0.0
- **options** – Options to be passed to the specific instance of the solver.

simulate (*timepoints*)
Simulate initialised solver for the specified timepoints

Parameters **timepoints** – timepoints that will be returned from simulation

Returns a list of trajectories for each of the equations in the problem.

exception means.simulation.solvers.**SolverException** (*message*, *base_exception=None*)
Bases: exceptions.Exception

base_exception

class means.simulation.solvers.**UniqueNameInitialisationMixin**
Bases: object

classmethod unique_name ()

means.simulation.solvers.**available_solvers** (*with_sensitivity_support=False*)

means.simulation.solvers.**parse_flag** (*exception_message*)
Parse the flag from the solver exception. e.g.

```
>>> parse_flag("Exception: Dopri5 failed with flag -3")
-3
```

Parameters **exception_message** (*str*) – message from the exception

Returns flag id

Return type int

Gillespie Stochastic Simulation Algorithm

This part of the package provides a simple implementation of GSSA. This is designed for experimental purposes much more than for performance. If you would like to use SSA for parameter inference, or for high number of species, there are many superior implementations available.

class means.simulation.ssa.**SSASimulation** (*stochastic_problem*, *n_simulations*, *random_seed=None*)
Bases: *means.io.serialise.SerialisableObject*

A class providing an implementation of the exact Gillespie Stochastic Simulation Algorithm [Gillespie77].

```
>>> from means.examples import MODEL_P53
>>> from means import StochasticProblem, SSASimulation
>>> import numpy as np
>>> PROBLEM = StochasticProblem(MODEL_P53)
>>> RATES = [90, 0.002, 1.7, 1.1, 0.93, 0.96, 0.01]
>>> INITIAL_CONDITIONS = [70, 30, 60]
>>> TIME_RANGE = np.arange(0, 40, .1)
>>> N_SSA = 10
>>> ssas = SSASimulation(PROBLEM, N_SSA)
>>> mean_trajectories = ssas.simulate_system(RATES, INITIAL_CONDITIONS, TIME_RANGE)
```

Parameters

- **`stochastic_problem`** –
- **`n_simulations`** –
- **`random_seed`** –

`simulate_system`(*parameters*, *initial_conditions*, *timepoints*, *max_moment_order=1*, *number_of_processes=1*)

Perform Gillespie SSA simulations and returns trajectories for of each species. Each trajectory is interpolated at the given time points. By default, the average amounts of species for all simulations is returned.

Parameters

- **`parameters`** – list of the initial values for the constants in the model. Must be in the same order as in the model
- **`initial_conditions`** – List of the initial values for the equations in the problem. Must be in the same order as these equations occur.
- **`timepoints`** – A list of time points to simulate the system for
- **`number_of_processes`** – the number of parallel process to be run
- **`max_moment_order`** – the highest moment order to calculate the trajectories to. if set to zero, the individual trajectories will be returned, instead of the averaged moments.

E.g. a value of one will return means, a values of two, means, variances and covariance and so on.

Returns a list of `Trajectory` one per species in the problem, or a list of lists of trajectories (one per simulation) if *return_average == False*.

Return type list[`Trajectory`]

`means.simulation.ssa.`**`multiprocessing_apply_ssa`**(*x*)

Used in the SSASimulation class. Needs to be in global scope for multiprocessing module to pick it up

`means.simulation.ssa.`**`multiprocessing_pool_initialiser`**(*population_rates_as_function*, *change*, *species*, *initial_conditions*, *t_max*, *seed*)

Trajectories

This part of the package provide convenient utilities to manage trajectories. A *`Trajectory`* object is generally a time series containing the values of a given moment (e.g. mean, variance, ...) over a time range. Trajectories are typically returned by simulations (see *`simulate`* and *`ssa`*), or from observation/measurement.

The *TrajectoryCollection* class is a container of trajectories. It can be used like other containers such as lists.

Both *~means.simulation.trajectory.TrajectoryCollection* and *~means.simulation.trajectory.Trajectory* have there own *.plot()* method to help representation.

class `means.simulation.trajectory.`**`Trajectory`**(*timepoints*, *values*, *description*)

Bases: `means.io.serialise.SerialisableObject`

A single simulated or observed trajectory for an ODE term.

Parameters

- **`timepoints`** (`iterable`) – timepoints the trajectory was simulated for
- **`values`** (`iterable`) – values of the curve at each of the timepoints
- **`description`** (`Descriptor`) – description of the trajectory

`description`

Description of this trajectory. The same description as the description for particular ODE term.

Return type `Descriptor`

`plot`(**args*, ***kwargs*)

Plots the trajectory using `matplotlib.pyplot`.

Parameters

- **`args`** – arguments to pass to `plot()`
- **`kwargs`** – keyword arguments to pass to `plot()`

Returns the result of the `matplotlib.pyplot.plot()` function.

`png`

`resample`(*new_timepoints*, *extrapolate=False*)

Use linear interpolation to resample trajectory values. The new values are interpolated for the provided time points. This is generally before comparing or averaging trajectories.

Parameters

- **`new_timepoints`** – the new time points
- **`extrapolate`** – whether extrapolation should be performed when some new time points are out of the current time range. if extrapolate=False, it would raise an exception.

Returns a new trajectory.

Return type `Trajectory`

`set_description`(*description*)

`svg`

`timepoints`

The timepoints trajectory was simulated for.

Return type `numpy.ndarray`

`to_csv`(*file*)

Write this trajectory to a csv file with the headers 'time' and 'value'.

Parameters **`file`** (`file`) – a file object to write to

Returns

classmethod **`to_yaml`**(*dumper*, *data*)

values
The values for each of the timepoints in *timepoints*.

Return type numpy.ndarray

yaml_tag = u'!trajectory'

class means.simulation.trajectory.**TrajectoryCollection**(*trajectories*)
Bases: *means.io.serialise.SerialisableObject*

A container of trajectories with representation functions for matplotlib and IPythonNoteBook. In most cases, it simply behaves as list.

plot(*legend=True*)

png

svg

to_csv(*file*)
Write all the trajectories of a collection to a csv file with the headers 'description', 'time' and 'value'.

Parameters **file** (file) – a file object to write to

Returns

classmethod to_yaml(*dumper*, *data*)

trajectories
Return a list of all trajectories in the collection :rtype: list[*Trajectory*]

yaml_tag = '!trajectory-collection'

class means.simulation.trajectory.**TrajectoryWithSensitivityData**(*timepoints*, *values*, *description*, *sensitivity_data*)
Bases: *means.simulation.trajectory.Trajectory*

An extension to Trajectory that provides data about the sensitivity of said trajectory as well.

Parameters

- **timepoints** (numpy.ndarray) – timepoints the trajectory was simulated for
- **values** (numpy.ndarray) – values of the curve at each of the timepoints
- **description** (Descriptor) – description of the trajectory
- **sensitivity_data** (list[Trajectory]) – a list of Trajectory objects signifying the sensitivity change over time for each of the parameters.

classmethod from_trajectory(*trajectory*, *sensitivity_data*)

plot_perturbations(*parameter*, *delta=0.0001*, **args*, ***kwargs*)

sensitivity_data
THe sensitivity data for the trajectory

Return type list[Trajectory]

classmethod to_yaml(*dumper*, *data*)

yaml_tag = '!trajectory-with-sensitivity'

means.simulation.trajectory.**perturbed_trajectory**(*trajectory*, *sensitivity_trajectory*, *delta=0.0001*)
Slightly perturb trajectory wrt the parameter specified in sensitivity_trajectory.

Parameters

- **trajectory** (*Trajectory*) – the actual trajectory for an ODE term
- **sensitivity_trajectory** (*Trajectory*) – sensitivity trajectory (dy/dpi for all timepoints t)
- **delta** (*float*) – the perturbation size

Returns *Trajectory*

3.6.2 Module contents

Routines for stochastic and deterministic simulation.

3.7 means.util package

3.7.1 Submodules

`means.util.decorators.`**`cache`**(*func*)

`means.util.logs.`**`get_logger`**(*name*)

class `means.util.memoisation.`**`MemoisableObject`**

Bases: `object`

A wrapper around objects that support `memoised_property` decorator. It overrides the __getstate__ method to prevent pickling cached values.

`means.util.memoisation.`**`memoised_property`**(*function*)

MEANS Helpers

This part of the package provides a function to generate all mixed and "pure" raw and :class:'~means.core.descriptor.Moment's up to a maximal_order.

`means.util.moment_counters.`**`generate_n_and_k_counters`**(*max_order*, *species*, *central_symbols_prefix='M_'*, *raw_symbols_prefix='x_'*)

Makes a counter for central moments (n_counter) and a counter for raw moment (k_counter). Each is a list of `Moments` is represented by both a vector of integer and a symbol.

Parameters

- **max_order** – the maximal order of moment to be computer (will generate a list of moments up to *max_order + 1*)
- **species** – the name of the species

Returns a pair of lists of :class:'~means.core.descriptors.Moment's corresponding to central,

and raw moments, respectively. :rtype: (list[`Moment`],list[`Moment`])

Sympy Helpers

This part of the package provides functions to extend the functionality of sympy, or to make MEANS compatible with different versions of sympy.

`means.util.sympyhelpers.`**`assert_sympy_expressions_equal`**(*expr1*, *expr2*)

Raises *AssertionError* if *expr1* is not equal to *expr2*.

Parameters

- **`expr1`** – first expression
- **`expr2`** – second expression

Returns None

`means.util.sympyhelpers.`**`product`**(*list*)

`means.util.sympyhelpers.`**`quick_solve`**(*expr*, *var*)

A function that tries to solve a very simple equation in the quickest way. For instance, an expression like :math: *2*a + 3*b + c == 0* needed to be solved for :math: *c* may be simply solved by stating :math: *c = -(2*a + 3*b)*. The function checks if the right hand side does not contain :math: *c*. If it does, then, the classic *sympy.solve()* method is used. *sympy.solve()* uses simplify in the background (i.e. is slow), it is therefore preferable to avoid using it.

Parameters

- **`expr`** – an expression like implicitly equal to 0
- **`var`** – a variable to solve for

Returns the solution for *var*

`means.util.sympyhelpers.`**`substitute_all`**(*sp_object*, *pairs*)

Performs multiple substitutions in an expression :param expr: a sympy matrix or expression :param pairs: a list of pairs (a,b) where each a_i is to be substituted with b_i :return: the substituted expression

`means.util.sympyhelpers.`**`sum_of_cols`**(*mat*)

`means.util.sympyhelpers.`**`sum_of_rows`**(*mat*)

`means.util.sympyhelpers.`**`sympy_empirical_equal`**(*expr1*, *expr2*)

Compare long , complex, expressions by replacing all symbols by a set of arbitrary expressions

Parameters

- **`expr1`** – first expression
- **`expr2`** – second expression

Returns True if expressions are empirically equal, false otherwise

`means.util.sympyhelpers.`**`sympy_expressions_equal`**(*expr1*, *expr2*)

Compare two sympy expressions that are not necessarily expanded. :param expr1: a first expression :param expr2: a second expression :return: True if the expressions are similar, False otherwise

`means.util.sympyhelpers.`**`sympy_sum_list`**(*list*)

`means.util.sympyhelpers.`**`to_list_of_symbols`**(*values*)

`means.util.sympyhelpers.`**`to_one_dim_array`**(*iterable_*, *dtype=<Mock id='140572037976272'>*)

`means.util.sympyhelpers.`**`to_sympy_column_matrix`**(*matrix*)

Converts a sympy matrix to a column matrix (i.e. transposes it if it was row matrix) Raises ValueError if matrix provided is not a vector :param matrix: a vector to be converted to column :return:

`means.util.sympyhelpers.`**`to_sympy_matrix`**`(`*value*`)`

Converts value to a *sympy.Matrix* object, if possible. Leaves the value as *sympy.Matrix* if it already was :param value: value to convert :return: :rtype: *sympy.Matrix*

3.7.2 Module contents

Utilities

This package contains helper functions that did not fit into any other packages. These functions include helper functions for common operations when dealing with `sympy`, as well as functions that help with memoisation of CPU intensive function results.

These functions are designed to be package-specific. The users of the software are generally discouraged to use any of these functions.

[Komorowski2009] 13. Komorowski, B. Finkenstadt, C. V. Harper, and D. A. Rand,"Bayesian inference of biochemical kinetic parameters using the linear noise approximation,"BMC Bioinformatics, vol. 10, no. 1, p. 343, Oct. 2009.

[Ale2013] 1. Ale, P. Kirk, and M. P. H. Stumpf, "A general moment expansion method for stochastic kinetic models," The Journal of Chemical Physics, vol. 138, no. 17, p. 174101, 2013.

[Ale2013] 1. Ale, P. Kirk, and M. P. H. Stumpf, "A general moment expansion method for stochastic kinetic models," The Journal of Chemical Physics, vol. 138, no. 17, p. 174101, 2013.

Python Module Index