pyParticleEst Documentation

Release 1.1

Jerker Nordh

Aug 27, 2017

Contents

The introduction chapter provides a quick introduction to the ideas behind the framework and explanation of how it's expected to be used. The rest is an API documentation for those classes that the users of the frameowork are expected to come into contact with.

CHAPTER 1

Contents

Introduction

This is a library to assist with calculations for estimation problems using particle based methods, it contains a number of algorithms such as the Particle Filter, Auxiliary Particle Filter and support several variants of Particle Smoothing through the use of Backward Simulation (FFBSi) techniques but also methods such as the Metropolis-Hastings Backward Proposer (MHBP) and the Metropolis-Hastings Improved Particle Smoother (MHIPS).

It also provides a framework for doing parameter estimation in nonlinear models using Expectation Maximization combined with the particle smoothing algorithms presented above. (PS-EM).

The use of Rao-Blackwellized models is considered an importan special case and extensive support for it is provided.

The structure is based on presenting a number of interfaces that a problem specific class must implement in order to use the algorithms. To assisst the end user base classes for common model structures, such as Mixed Linear/Nonlinear Gaussian (MLNLG) models are provided to keep the implementation effort to a minimum.

The idea is to provide an easy prototyping enviroment for testing different algorithms and model formulations when solving a problem and to act as a stepping stone for a later more performance oriented problem specific implementation by the end user. (outside the scope of this framework)

There are three main areas of interest for the end user, where extra focus has been spent writing clear docstring explaining the expected usage and behavior

1. The Simulator class in the pyparticleest.simulator module is the main entry point for the application, it is used for running the different algorithms and accessing the results.

2. The abstract base classes in pyparticleest.interfaces, they define which operations that needs to be implemented when using different classes of algorithms.

3. pyparticleest.models, this modue contains support code for common model classes, currently: Nonlinar Gaussian (NLG), Mixed Linear/Nonlinear Gaussian (MLNLG), Hierarchical, Linear Time-Varying (LTV)

You are encourage to study the examples in the documentation, more can be under test/manual in the source code. They demonstrate how the framework is instended to be used.

A good starting point is test/manual/filtering/basic_model.py

followed by test/manual/filtering/nonlin_model.py

For an example of a simple MLNLG model see test/manual/smoothing/mlnlg_model.py

and for the commonly used "standard nonlinear model" see test/manual/smoothing/standard_nonlin_model.py

API

Simulator

Simulator

This is the main class used to run the different algorithms using a user specified model class

class `pyparticleest.simulator.`**`Simulator`**(*model*, *u*, *y*)

Class interfacing filters/smoothers to assisst in solving estimation problem

Args:

- model: object of class describing problem type
- u (array-like): inputs, first dimension is the time index, the rest is specific to the particlar model class being used
- y (array-like): measurements, first dimension is the time index, the rest is specific to the particlar model class being used

`get_filtered_estimates`()

Returns type (est, w) (must first have called 'simulate')

- est: (T, N, D) array containing all particles
- w: (T,D) array containing all particle weights

T is the length of the dataset, N is the number of particles and D is the dimension of each particle

`get_filtered_mean`()

Calculate mean of filtered estimates (must first have called 'simulate')

Returns:

- (T, D) array

T is the length of the dataset, N is the number of particles and D is the dimension of each particle

`get_smoothed_estimates`()

Return smoothed estimates (must first have called 'simulate')

Returns:

- (T, N, D) array

T is the length of the dataset, N is the number of particles D is the dimension of each particle

`get_smoothed_mean`()

Calculate mean of smoothed estimates (must first have called 'simulate')

Returns:

- (T, D) array

T is the length of the dataset, N is the number of particles and D is the dimension of each particle

set_params(*params*)

Set the parameters of the model (if any)

Args:

- params (array-like): Model specific paremeters

simulate(*num_part*, *num_traj*, *filter='PF'*, *filter_options=None*, *smoother='full'*, *smoother_options=None*, *res=0.67*, *meas_first=False*)

Solve the estimation problem

Args:

- num_part (int): Number of particles used in the forward filter.
- num_traj (int): Number of backward trajectories generated by the smoother.
- filter (string): The filter algorithm to use
- smooter (string): The smoothing algorithm to use
- smoother_options (dict): algorithm specific smoother options
- res (float): resampling threshold for the forward filter
- meas_first (bool): Is the first measurement of the initial state (true) or after the first time update? (false)

Supported filters:

- 'pf': regular particle filter
- 'apf': auxilliary particle filter

Supported smoothers:

- 'ancestor': return forward trajectories from particle filtier (no extra smoothing step)
- 'full': Backward simulation evaluating all particle weights
- **'rs': Rejection sampling (with early stopping)**

  Options:

  - R: number of rejection sampling steps before falling back to 'full'
- **'rsas': Rejection sampling with early stopping.**

  Options:

  - x1 (float): (default is 1.0)
  - P1 (float): (default is 1.0)
  - sv (float): (default is 1.0)
  - sw (float): (default is 1.0)
  - ratio (float): (default is 1.0)
- **'mcmc': Metropolis-Hastings FFBSi**

  Options:

  - R: number of iterations to run the Markov chain
- **'mhips': Metropolis-Hastings Improved Particle Smoother**

  Options:

  - R: number of passes of the dataset to run the algortithms

- **'mhbp': Metropolis-Hastings Backward Proposer**

  Options:

  - R: the number of iterations to run the Markov chain for each time step

Parameter Estimator

This is an extension of the Simulator class which allows parameter estimation

class `pyparticleest.paramest.paramest.`**`ParamEstimation`**(*model*, *u*, *y*)

Extension of the Simulator class to iterative perform particle smoothing combined with a gradienst search algorithms for maximizing the likelihood of the parameter estimates

`maximize`(*param0*, *num_part*, *num_traj*, *max_iter=1000*, *tol=0.001*, *callback=None*, *callback_sim=None*, *meas_first=False*, *filter='pf'*, *smoother='full'*, *smoother_options=None*)

Find the maximum likelihood estimate of the paremeters using an EM-algorihms combined with a gradient search algorithms

Args:

- param0 (array-like): Initial parameter estimate
- num_part (int/array-like): Number of particle to use in the forward filter if array each iteration takes the next element from the array when setting up the filter
- num_traj (int/array-like): Number of smoothed trajectories to create if array each iteration takes the next element from the array when setting up the smoother
- max_iter (int): Max number of EM-iterations to perform
- tol (float): When the different in loglikelihood between two iterations is less that his value the algorithm is terminated
- callback (function): Callback after each EM-iteration with new estimate
- callback_sim (function): Callback after each simulation
- bounds (array-like): Hard bounds on parameter estimates
- meas_first (bool): If true, first measurement occurs before the first time update
- smoother (string): Which particle smoother to use
- smoother_options (dict): Extra options for the smoother
- analytic_gradient (bool): Use analytic gradient (requires that the model implements ParamEstInterface_GradientSearch)

Interfaces

Particle Filter

class `pyparticleest.interfaces.`**`SIR`**

`copy_ind`(*particles*, *new_ind=None*)

Copy select particles, can be overriden for models that require special handling of the particle representations when copying them

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- new_ind (array-like): Array of ints, specifying indices to copy

Returns: (array-like) with first dimension = len(new_ind)

create_initial_estimate(*N*)

Sample particles from initial distribution

Args:

- N (int): Number of particles to sample

Returns: (array-like) with first dimension = N, model specific representation of all particles

class pyparticleest.interfaces.**ParticleFiltering**

Base class for particles to be used with particle filtering. particles are a model specific array where the first dimension indexes the different particles.

measure(*particles*, *y*, *t*)

Return the log-pdf value of the measurement

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y|x^i)

update(*particles*, *u*, *t*, *noise*)

Propagate estimate forward in time

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time-stamp
- noise (array-like): noise realization used for the calucations , with first dimension = N (number of particles)

Returns: (array-like) with first dimension = N, particle estimate at time t+1

class pyparticleest.interfaces.**FFProposeFromMeasure**

propose_from_y(*N*, *y*, *t*)

Create N particles from p(x_t|y_t)

Particle Filter Non-Markov

class `pyparticleest.interfaces.`**`ParticleFilteringNonMarkov`**

`cond_predict_single_step`(*part*, *past_trajs*, *pind*, *future_parts*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Propagate states in 'part' conditioned on that the future state is 'future_parts'. This is used for e.g. Rao-Blackwellized MHIPS, where we need to propagate forward in time conditioned on the nonlinear state, but we want to recompute the additional data stored, e.g to exclude measurements present in the sufficient statistics for future_parts.

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

`cond_sampled_initial`(*part*, *t*)

Sample from initial distribution conditioned on the states being 'part' This is used for e.g. Rao-Blackwellized MHIPS, where we need to recompute the sufficient statistics without being affected by the intial measurement

Args: part: particles t: time-step

`copy_ind`(*particles*, *new_ind=None*)

Copy select particles, can be overriden for models that require special handling of the particle representations when copying them

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- new_ind (array-like): Array of ints, specifying indices to copy

Returns: (array-like) with first dimension = len(new_ind)

`create_initial_estimate`(*N*)

Sample particles from initial distribution

Args:

- N (int): Number of particles to sample

Returns: (array-like) with first dimension = N, model specific representation of all particles

`measure_full`(*particles*, *traj*, *uvec*, *yvec*, *tvec*, *ancestors*)

Return the log-pdf value of the measurement

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- traj: array of trajectory step objects from previous time-steps, last index is step just before the current
- ancestors (array-like): index of the ancestor of each particle in part
- uvec (array-like): input signals for {0:t}
- yvec (array-like): measurements for {0:t}
- tvec (array-like): time stamps for {0:t}

Returns: (array-like) with first dimension = N

`sample_process_noise_full` (*ptraj*, *ancestors*, *ut*, *tt*)

Sample process noise

Args:

- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- ancestors (array-like): index of the ancestor of each particle in part
- ut (array-like): input signals for {0:T}
- tt (array-like): time stamps for {0:T}

Returns: (array-like) with first dimension = N

`sample_smooth` (*part*, *ptraj*, *anc*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Create sampled estimates for the smoothed trajectory. Allows the update representation of the particles used in the forward step to include additional data in the backward step, can also for certain models be used to update the points estimates based on the future information.

Default implementation uses the same format as forward in time it ss part of the ParticleFiltering interface since it is used also when calculating "ancestor" trajectories

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N

`update_full` (*particles*, *traj*, *uvec*, *yvec*, *tvec*, *ancestors*, *noise*)

Propagate estimate forward in time

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- traj: array of trajectory step objects from previous time-steps, last index is step just before the current
- ancestors (array-like): index of the ancestor of each particle in part
- uvec (array-like): input signals for {0:t}
- yvec (array-like): measurements for {0:t}
- tvec (array-like): time stamps for {0:t}
- noise (array-like): samples noise for time t

Returns: (array-like) with first dimension = N

Auxiliary Particle Filter

class `pyparticleest.interfaces.`**`AuxiliaryParticleFiltering`**

Base class for particles to be used with auxiliary particle filtering

`eval_1st_stage_weights` (*particles*, *u*, *y*, *t*)

Evaluate "first stage weights" for the auxiliary particle filter. (log-probability of measurement using some propagated statistic, such as the mean, for the future state)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y_{t+1}|hat{x}_{t+1|t}^i)

Forward-Filter Backward Simulator

class `pyparticleest.interfaces.`**`FFBSi`**

Base class for particles to be used with particle smoothing (Backward Simulation)

`logp_xnext` (*particles*, *next_part*, *u*, *t*)

Return the log-pdf value for the possible future state 'next' given input u

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_part (array-like): particle estimate for t+1
- u (array-like): input signal
- t (float): time stamps

Returns: (array-like) with first dimension = N, logp(x_{t+1}|x_t^i)

logp_xnext_full (*part*, *past_trajs*, *pind*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Return the log-pdf value for the entire future trajectory. Useful for non-markovian modeles, that result from e.g marginalized state-space models.

Default implemention just calls logp_xnext which is enough for Markovian models

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- past_trajs: array of trajectory step objects from previous time-steps, last index is step just before the current
- pind (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N, logp(x_{t+1:T}|x_t^i)

logp_xnext_singlestep (*part*, *past_trajs*, *pind*, *future_parts*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Return the log-pdf value for the first step of the future trajectory. Needed in e.g MHIPS

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- past_trajs: Trajectory leading up to current time
- pind: Indices relating part to past_trajs
- future_parts (array-like): particle estimate for {t+1}, stored using 'filtered' particle representation, ie. sample_smooth has not been performed on them
- find: Indices relatin part and future_parts
- ut (array-like): input signals for {1:T}
- yt (array-like): measurements for {1:T}
- tt (array-like): time stamps for {1:T}
- cur_ind: index for current time

Returns: (array-like) with first dimension = N, logp(x_{t+1:T}|x_t^i)

FFBSi Non-Markov

class pyparticleest.interfaces.**FFBSiNonMarkov**

FFBSi with Rejection Sampling

class `pyparticleest.interfaces.`**`FFBSiRS`**

Base class for models to be used with rejection sampling methods

`logp_xnext_max`(*particles*, *u*, *t*)

Return the max log-pdf value for all possible future states' given input u

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_part (array-like): particle estimate for t+1
- u (array-like): input signal
- t (float): time stamps

Returns: (array-like) with first dimension = N, argmax_{x_{t+1}} logp(x_{t+1}|x_t)

class `pyparticleest.interfaces.`**`FFBSiRSNonMarkov`**

Base class for models to be used with rejection sampling methods

`logp_xnext_max_full`(*part*, *past_trajs*, *pind*, *uvec*, *yvec*, *tvec*, *cur_ind*)

Return the max log-pdf value for all possible future states' given input u

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- past_trajs: Trajectory leading up to current time
- pind: Indices relating part to past_trajs
- uvec (array-like): input signals for {1:T}
- yvec (array-like): measurements for {1:T}
- tvec (array-like): time stamps for {1:T}
- cur_ind: index for current time

Returns: (array-like) with first dimension = N, argmax_{x_{t+1}} logp(x_{t+1}|x_t)

Proposal methods, e.g. MHIPS and MHBP

class `pyparticleest.interfaces.`**`SampleProposer`**

Base class for models to be used with methods that require drawing of new samples. Here 'q' is the name we give to the proposal distribtion.

`logp_proposal`(*prop_part*, *ptraj*, *anc*, *future_trajs*, *find*, *yt*, *ut*, *tt*, *cur_ind*)

Eval the log-propability of the proposal distribution

Args:

- prop_part (array-like): Proposed particle estimate, first dimension has length = N
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part

- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each generated sample
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns (array-like) with first dimension = N, log q(x_t | x_{t-1}, x_{t+1:T}, y_t:T)

propose_smooth (*ptraj*, *anc*, *future_trajs*, *find*, *yt*, *ut*, *tt*, *cur_ind*)

Sample from a distribution q(x_t | x_{t-1}, x_{t+1:T}, y_t:T)

Args:

- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each generated sample
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) of dimension N, wher N is the dimension of partp and/or future_trajs (one of which may be 'None' at the start/end of the dataset)

Parameter estimation

Numerical gradient

class pyparticleest.paramest.interfaces.**ParamEstInterface**

Interface s for particles to be used with the parameter estimation algorithm presented in [1] [1] - 'System identification of nonlinear state-space models' by Schon, Wills and Ninness

eval_logp_xnext (*particles*, *particles_next*, *u*, *t*)

Calculate gradient of a term of the I2 integral approximation as specified in [1].

Eg. Evaluate log p(x_{t+1}|x_t) or sum log p(x_{t+1}|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- x_next (array-like): future states
- t (float): time stamp

Returns: (array-like) or (float)

eval_logp_y (*particles*, *y*, *t*)

Calculate gradient of a term of the I3 integral approximation as specified in [1].

Eg. Evaluate log p(y_t|x_t) or sum log p(y_t|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

Returns: (array-like) or (float)

Analytic gradient

class pyparticleest.paramest.interfaces.**ParamEstInterface_GradientSearch**

Interface s for particles to be used with the parameter estimation algorithm presented in [1] using analytic gradients

eval_logp_x0_val_grad (*particles*, *t*)

Calculate term of the I1 integral approximation as specified in [1]. Eg. Evaluate log p(x_0) or sum log p(x_0)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

The gradient is an array where each element is the derivative with respect to the corresponding parameter

Returns: ((array-like) or (float), array-like) (value, gradient)

eval_logp_xnext_val_grad (*particles*, *particles_next*, *u*, *t*)

Calculate gradient of a term of the I2 integral approximation as specified in [1].

Eg. Evaluate log p(x_{t+1}|x_t) or sum log p(x_{t+1}|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- particles_next (array-like): future states
- u (array-like): input signal
- t (float): time stamp

The gradient is an array where each element is the derivative with respect to the corresponding parameter

Returns: ((array-like) or (float), array-like) (value, gradient)

eval_logp_y_val_grad (*particles*, *y*, *t*)

Calculate gradient of a term of the I3 integral approximation as specified in [1].

Eg. Evaluate log p(y_t|x_t) or sum log p(y_t|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

The gradient is an array where each element is the derivative with respect to the corresponding parameter

Returns: ((array-like) or (float), array-like) (value, gradient)

Base classes

Mixed Linear/Nonlinear Gaussian

Model definition for base class for Mixed Linear/Nonlinear Gaussian systems

@author: Jerker Nordh

class `pyparticleest.models.mlnlg.`**`MixedNLGaussianMarginalized`**(*lxi*, *lz*, *Az=None*, *C=None*, *Qz=None*, *R=None*, *fz=None*, *Axi=None*, *Qxi=None*, *Qxiz=None*, *fxi=None*, *h=None*, *params=None*, ***kwargs*)

This class implements a fully marginalized smoother for mixed linear/nonlinear models, in contrast to the MixedNLGaussian class it never samples the linear states.

This is somewhat slower, and doesn't readily admit using rejection sampling, it is up to the end user which method is best for their particular problem

`calc_prop1`(*particles*, *next_part*, *u*, *t*)
internal helper function

`calc_prop3`(*particles*, *Omega*, *Lambda*, *u*, *t*)
internal helper function

`logp_xnext_full`(*part*, *past_trajs*, *pind*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)
Return the log-pdf value for the entire future trajectory. Useful for non-markovian modeles, that result from e.g marginalized state-space models.

Default implemention just calls logp_xnext which is enough for Markovian models

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- past_trajs: array of trajectory step objects from previous time-steps, last index is step just before the current
- pind (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}

- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N, logp(x_{t+1:T}|x_t^i)

sample_smooth(*part*, *ptraj*, *anc*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Calculate statistics needed when evaluating the logp_xnext_full for the marginalized trajectory

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N

class pyparticleest.models.mlnlg.**MixedNLGaussianSampled**(*lxi*, *lz*, *Az=None*, *C=None*, *Qz=None*, *R=None*, *fz=None*, *Axi=None*, *Qxi=None*, *Qxiz=None*, *fxi=None*, *h=None*, *params=None*, ***kwargs*)

Base class for particles of the type mixed linear/non-linear with additive gaussian noise.

Implement this type of system by extending this class and provide the methods for returning the system matrices at each time instant.

xi_{t+1} = f_xi + A_xi*z_t + v_xi, z_{t+1} = f_z + A_z*z_t + v_z, y_t = h + C*z_t + e, (v_xi, v_z)^T ~ N(0, (Q_xi, Qxiz Qxiz^T, Qz)) e ~ N (0, R)

Args:

- lxi (int): number of nonlinear states
- lz (int): number of linear states
- Az (arraylike): Az (if constant)
- C (arraylike): C (if constant)
- Qz (arraylike): Qz (if constant)
- R (arraylike): R (if constant)
- fz (arraylike): fz (if constant)
- Axi (arraylike): Axi (if constant)

- Qxi (arraylike): Qxi (if constant)
- Qxiz (arraylike): Qxiz (if constant)
- fxi (arraylike): fxi (if constant)
- h (arraylike): h (if constant)
- params (array-like): model parameters (if any)

calc_A_f_Q (*particles*, *u*, *t*)

Calculate the A, f and Q matrices for the particles. Where A, f and Q are the stacked matrices of (A_xi, A_z) and so on

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- y (array-like): measurement
- t (float): time-stamp

Returns: (A, f, Q, A_identical f_identical, Q_identical). The last elements indicate if the matrices are identical for all particles

calc_cond_dynamics (*particles*, *xi_next*, *u*, *t*)

Calculates the linear dynamics for each particle

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- xi_next (array-like): next non linear state
- u (array-like): input signal
- t (float): time stamp

Returns:

(Az, fz, Qz):

- Az (array-like): Az matrix for each particle
- fz (array-like): fz vector for each particle
- Qz (array-lie): Noise covariance for each particle

calc_l2 (*xin*, *zn*, *Pn*, *zl*, *Pl*, *A*, *f*, *M*)

Internal helper function

calc_l2_grad (*xin*, *zn*, *Pn*, *zl*, *Pl*, *A*, *f*, *M*, *f_grad*, *A_grad*)

Internal helper function

calc_l3 (*y*, *zl*, *Pl*, *Cl*, *hl*)

internal helper function

calc_l3_grad (*y*, *zl*, *Pl*, *Cl*, *hl*, *C_grad*, *h_grad*)

internal helper function

calc_xi_next (*particles*, *noise*, *u*, *t*)

Calculate the next nonlinear state given the input and noise realization

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamp
- noise (array-like): noise realization for each particle

Returns: (array-like): xi values for future particles

eval_1st_stage_weights (*particles*, *u*, *y*, *t*)

Evaluate "first stage weights" for the auxiliary particle filter. (log-probability of measurement using some propagated statistic, such as the mean, for the future state)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y_{t+1}|hat{x}_{t+1|t}^i)

eval_logp_x0 (*particles*, *t*)

Evaluate sum log p(x_0)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

eval_logp_x0_val_grad (*particles*, *t*)

Evaluate gradient of sum log p(x_0)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

eval_logp_xi0 (*xil*)

Evaluate logprob of the initial non-linear state eta, default implementation assumes all are equal, override this if another behavior is desired

Args:

- xil (list): Initial xi states

eval_logp_xi0_grad (*xil*)

Evaluate logprob of the initial non-linear state eta, default implementation assumes all are equal, override this if another behavior is desired

Args:

- xil (list): Initial xi states

eval_logp_xnext (*particles*, *x_next*, *u*, *t*)
Evaluate sum log p(x_{t+1}|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- x_next (array-like): future states
- t (float): time stamp

Returns: (float)

eval_logp_xnext_val_grad (*particles*, *x_next*, *u*, *t*)
Evaluate value and gradient sum log p(x_{t+1}|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- x_next (array-like): future states
- t (float): time stamp

Returns: ((float), (array-like))

eval_logp_y (*particles*, *y*, *t*)
Evaluate value of sum log p(y_t|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

Returns: (float)

eval_logp_y_val_grad (*particles*, *y*, *t*)
Evaluate value and gradient of sum log p(y_t|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

Returns: ((float), (array-like))

get_cross_covariance (*particles*, *u*, *t*)
Return cross-covariance between noise for nonlinear and linear states

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamp

Returns:

- (array-like): Qxiz(xi_t, u_t, t) for each particle

get_meas_dynamics_grad (*particles*, *y*, *t*)
Override this method if (C, h, R) depends on the parameters

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurment
- t (float): time stamps

Returns: (C_grad, h_grad, R_grad): Element-wise gradients with respect to all the parameters for the system matrices

get_pred_dynamics_grad (*particles*, *u*, *t*)
Override this method if (A, f, Q) depends on the parameters

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamps

Returns: (A_grad, f_grad, Q_grad): Element-wise gradients with respect to all the parameters for the system matrices

logp_proposal (*prop_part*, *ptraj*, *anc*, *future_trajs*, *find*, *yt*, *ut*, *tt*, *cur_ind*)
Eval the log-propability of the proposal distribution

Args:

- prop_part (array-like): Proposed particle estimate, first dimension has length = N
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each generated sample
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns (array-like) with first dimension = N, log q(x_t | x_{t-1}, x_{t+1:T}, y_t:T)

logp_xnext (*particles*, *next_part*, *u*, *t*)
Return the log-pdf value for the possible future state 'next' given input u

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)

- next_part (array-like): particle estimate for t+1
- u (array-like): input signal
- t (float): time stamps

Returns: (array-like) with first dimension = N, logp(x_{t+1}|x_t^i)

logp_xnext_max (*particles*, *u*, *t*)

Return the max log-pdf value for all possible future states' given input u

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_part (array-like): particle estimate for t+1
- u (array-like): input signal
- t (float): time stamps

Returns: (array-like) with first dimension = N, argmax_{x_{t+1}} logp(x_{t+1}|x_t)

logp_xnext_singlestep (*part*, *past_trajs*, *pind*, *future_parts*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Return the log-pdf value for the first step of the future trajectory. Needed in e.g MHIPS

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- past_trajs: Trajectory leading up to current time
- pind: Indices relating part to past_trajs
- future_parts (array-like): particle estimate for {t+1}, stored using 'filtered' particle representation, ie. sample_smooth has not been performed on them
- find: Indices relatin part and future_parts
- ut (array-like): input signals for {1:T}
- yt (array-like): measurements for {1:T}
- tt (array-like): time stamps for {1:T}
- cur_ind: index for current time

Returns: (array-like) with first dimension = N, logp(x_{t+1:T}|x_t^i)

meas_xi_next (*particles*, *xi_next*, *u*, *t*)

Update estimate using observation of next state

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- xi_next (array-like): future nonlinear states
- u (array-like): input signal
- t (float): time stamp

measure (*particles*, *y*, *t*)

Return the log-pdf value of the measurement and update the statistics for the linear states

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y|x^i)

pred_xi (*particles*, *u*, *t*)

Predict the next nonlinear state given the input

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamp

Returns: (array-like): xi values for future particles

propose_smooth (*ptraj*, *anc*, *future_trajs*, *find*, *yt*, *ut*, *tt*, *cur_ind*)

Sample from a distribution q(x_t | x_{0:t-1}, x_{t+1:T}, y_0:T)

Args:

- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each generated sample
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) of dimension N, wher N is the dimension of partp and/or future_trajs (one of which may be 'None' at the start/end of the dataset)

sample_process_noise (*particles*, *u*, *t*)

Return sampled process noise for the non-linear states

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time-stamp

Returns: (array-like) with first dimension = N

sample_smooth (*part*, *ptraj*, *anc*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Create sampled estimates for the smoothed trajectory. Allows the update representation of the particles used in the forward step to include additional data in the backward step, can also for certain models be used to update the points estimates based on the future information.

Default implementation uses the same format as forward in time it ss part of the ParticleFiltering interface since it is used also when calculating "ancestor" trajectories

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N

set_dynamics (*Az=None*, *fz=None*, *Qz=None*, *R=None*, *Axi=None*, *fxi=None*, *Qxi=None*, *Qxiz=None*, *C=None*, *h=None*)

Update dynamics, typically used when changing the system dynamics due to a parameter change

Args:

- lxi (int): number of nonlinear states
- lz (int): number of linear states
- Az (arraylike): Az (if constant)
- C (arraylike): C (if constant)
- Qz (arraylike): Qz (if constant)
- R (arraylike): R (if constant)
- fz (arraylike): fz (if constant)
- Axi (arraylike): Axi (if constant)
- Qxi (arraylike): Qxi (if constant)
- Qxiz (arraylike): Qxiz (if constant)
- fxi (arraylike): fxi (if constant)
- h (arraylike): h (if constant)

set_params (*params*)

This methods should be overriden if the system dynamics depends on any parameters, this method should however be called to store the new parameter values correctly

Args:

- params (array-like): new parameter values

`pyparticleest.models.mlnlg.factor_psd(A)`
internal helper function

Nonlinear Gaussian

Model definition for base class for Nonlinear Gaussian systems

@author: Jerker Nordh

class `pyparticleest.models.nlg.NonlinearGaussian(lxi, f=None, g=None, Q=None, R=None)`

Base class for particles of the type mixed linear/non-linear with additive gaussian noise.

Implement this type of system by extending this class and provide the methods for returning the system matrices at each time instant.

x_{t+1} = f(x_t, u_t) + v, v ~ N(0, Q(x_t, u_t)) y_t = g(x_t) + e, e ~ N(=, R(x_t))

This class currently doesn't support analytic gradients when performing parameter estimation, however using numerical gradients is typically fine

Args:

- lxi (int): number of states in model
- f (array-like): f (if constaint)
- g (array-like): g (if constaint)
- Q (array-like): Q (if constaint)
- R (array-like): R (if constaint)

calc_Q(*particles*, *u*, *t*)
Calucate Q

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamp

Returns: (array-like): Q for all particles

calc_R(*particles*, *t*)
Calucate R

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

Returns: (array-like): R for all particles

calc_f(*particles*, *u*, *t*)
Calucate f

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)

- u (array-like): input signal
- t (float): time stamp

Returns: (array-like): f for all particles

calc_g (*particles*, *t*)
Calucate g

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

Returns: (array-like): g for all particles

eval_1st_stage_weights (*particles*, *u*, *y*, *t*)
Evaluate "first stage weights" for the auxiliary particle filter. (log-probability of measurement using some propagated statistic, such as the mean, for the future state)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y_{t+1}|hat{x}_{t+1|t}^i)

logp_proposal (*prop_part*, *ptraj*, *anc*, *future_trajs*, *find*, *yt*, *ut*, *tt*, *cur_ind*)
Eval the log-propability of the proposal distribution

Args:

- prop_part (array-like): Proposed particle estimate, first dimension has length = N
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each generated sample
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns (array-like) with first dimension = N, log q(x_t | x_{t-1}, x_{t+1:T}, y_t:T)

logp_xnext (*particles*, *next_part*, *u*, *t*)
Return the log-pdf value for the possible future state 'next' given input u

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_part (array-like): particle estimate for t+1
- u (array-like): input signal
- t (float): time stamps

Returns: (array-like) with first dimension = N, logp(x_{t+1}|x_t^i)

logp_xnext_max (*particles*, *u*, *t*)

Return the max log-pdf value for all possible future states' given input u

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_part (array-like): particle estimate for t+1
- u (array-like): input signal
- t (float): time stamps

Returns: (array-like) with first dimension = N, argmax_{x_{t+1}} logp(x_{t+1}|x_t)

measure (*particles*, *y*, *t*)

Return the log-pdf value of the measurement

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y|x^i)

propose_smooth (*ptraj*, *anc*, *future_trajs*, *find*, *yt*, *ut*, *tt*, *cur_ind*)

Sample from a distribution q(x_t | x_{0:t-1}, x_{t+1:T}, y_t:T)

Args:

- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each generated sample
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) of dimension N, wher N is the dimension of partp and/or future_trajs (one of which may be 'None' at the start/end of the dataset)

sample_process_noise (*particles*, *u*, *t*)
Sample process noise

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time-stamp

Returns: (array-like) with first dimension = N

set_params (*params*)
This methods should be overriden if the system dynamics depends on any parameters, this method should however be called to store the new parameter values correctly

Args:

- params (array-like): new parameter values

update (*particles*, *u*, *t*, *noise*)
Propagate estimate forward in time

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time-stamp
- noise (array-like): noise realization used for the calucations , with first dimension = N (number of particles)

Returns: (array-like) with first dimension = N, particle estimate at time t+1

class pyparticleest.models.nlg.**NonlinearGaussianInitialGaussian** (*x0=None*, *Px0=None*, *lxi=None*, ***kwargs*)
Nonlinear gaussian system with initial Gaussian distribution.

Args:

- x0 (array-like): mean value of initial state, defaults to 0
- Px0 (array-like): covariance of initial state, defaults to 0
- lxi (int): number of states, only needed if neither x0 or Px0 specified

create_initial_estimate (*N*)
Sample particles from initial distribution

Args:

- N (int): Number of particles to sample

Returns: (array-like) with first dimension = N, model specific representation of all particles

eval_logp_x0 (*particles*, *t*)
Evaluate log p(x_0)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

Linear Time-Varying

Model definition for base class for Linear Time-varying systems @author: Jerker Nordh

class `pyparticleest.models.ltv.`**`LTV`**(*z0*, *P0*, *A=None*, *C=None*, *Q=None*, *R=None*, *f=None*, *h=None*, *params=None*, ***kwargs*)

Base class for particles of the type linear time varying with additive gaussian noise.

Implement this type of system by extending this class and provide the methods for returning the system matrices at each time instant

z_{t+1} = A*z_t + f + v, v ~ N(0, Q) y_t = C*z_t + h + e, e ~ N(0,R)

Args:

- z0: Initial mean value of the state estimate
- P0: Coviariance of initial z estimate
- A (array-like): A matrix (if constant)
- C (array-like): C matrix (if constant)
- Q (array-like): Q matrix (if constant)
- R (array-like): R matrix (if constant)
- f (array-like): f vector (if constant)
- h (array-like): h vector (if constant)
- params (array-like): model parameters (if any)

`calc_l1`(*z*, *P*, *z0*, *P0*)

internal helper function

`calc_l1_grad`(*z*, *P*, *z0*, *P0*, *z0_grad*)

internal helper function

`calc_l2`(*zn*, *Pn*, *z*, *P*, *A*, *f*, *M*)

internal helper function

`calc_l2_grad`(*zn*, *Pn*, *z*, *P*, *A*, *f*, *M*, *A_grad*, *f_grad*)

internal helper function

`calc_l3`(*y*, *z*, *P*)

internal helper function

`calc_l3_grad`(*y*, *z*, *P*, *C_grad*, *h_grad*)

internal helper function

`create_initial_estimate`(*N*)

Sample particles from initial distribution

Args:

- N (int): Number of particles to sample, since the estimate is deterministic there is no reason for N > 1

Returns: (array-like) with first dimension = N, model specific representation of all particles

eval_logp_x0 (*particles*, *t*)
Evaluate sum log p(x_0)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

eval_logp_x0_val_grad (*particles*, *t*)
Evaluate gradient of sum log p(x_0)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- t (float): time stamp

eval_logp_xnext (*particles*, *x_next*, *u*, *t*)
Evaluate log p(x_{t+1}|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- x_next (array-like): future states
- t (float): time stamp

Returns: (array-like)

eval_logp_xnext_val_grad (*particles*, *x_next*, *u*, *t*)
Evaluate value and gradient of log p(x_{t+1}|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- x_next (array-like): future states
- t (float): time stamp

Returns: ((array-like), (array-like))

eval_logp_y (*particles*, *y*, *t*)
Evaluate value of log p(y_t|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

Returns: (array-like)

eval_logp_y_val_grad (*particles*, *y*, *t*)
Evaluate value and gradient of log p(y_t|x_t)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

Returns: ((array-like), (array-like))

fwd_peak_density(*u*, *t*)

No need for rejections sampling for this type of model, always returns 0.0 since all particles are equivalent

Args:

- u: Unused
- t: Unused

Returns (float) 0.0

get_initial_grad()

Default implementation has no dependence on xi, override if needed

Calculate gradient estimate of initial state for linear state condition on the nonlinear estimate

Args:

- xi0 (array-like): Initial xi states

Returns: (z,P): z is a list of element-wise gradients for the inital mean values, P is a list of element-wise gradients for the covariance matrices

get_meas_dynamics(*y*, *t*)

Return matrices describing affine relation of measurement and current state estimates

y_t = C*z_t + h + e, e ~ N(0,R)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time stamp

Returns: (y, C, h, R): y is a preprocessed measurement, the rest are lists with the corresponding matrix for each particle. None indicates that the matrix is identical for all particles and the value stored in this class should be used instead

get_meas_dynamics_grad(*y*, *t*)

Override this method if (C, h, R) depends on the parameters

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurment
- t (float): time stamps

Returns: (C_grad, h_grad, R_grad): Element-wise gradients with respect to all the parameters for the system matrices

get_pred_dynamics(*u*, *t*)

Return matrices describing affine relation of next nonlinear state conditioned on the current time and input signal

z_{t+1} = A*z_t + f + v, v ~ N(0, Q)

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamp

Returns: (A, f, Q) where each element is a list with the corresponding matrix for each particle. None indicates that the matrix is identical for all particles and the value stored in this class should be used instead

get_pred_dynamics_grad(*u*, *t*)

Override this method if (A, f, Q) depends on the parameters

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time stamps

Returns: (A_grad, f_grad, Q_grad): Element-wise gradients with respect to all the parameters for the system matrices

get_states(*particles*)

Return the estimates contained in the particles array

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)

Returns

(zl, Pl):

- zl: list of mean values for z
- Pl: list of covariance matrices for z

logp_xnext(*particles*, *next_part*, *u*, *t*)

Return the log-pdf value for the possible future state 'next' given input u.

Always returns zeros since all particles are always equivalent for this type of model

Args:

•particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)

•next_part: Unused

•u: Unused

•t: Unused

Returns: (array-like) with first dimension = N, numpu.zeros((N,))

measure (*particles*, *y*, *t*)

Return the log-pdf value of the measurement and update the statistics for the states

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y|x^i)

sample_process_noise (*particles*, *u*, *t*)

There is no need to sample noise for this type of model

Args:

- particles: Unused
- next_part: Unused
- u: Unused
- t: Unused

Returns: None

sample_smooth (*part*, *ptraj*, *anc*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Update sufficient statistics based on the future states

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N

set_states (*particles*, *z_list*, *P_list*)

Set the estimate of the states

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- z_list (list): list of mean values for z for each particle

- P_list (list): list of covariance matrices for z for each particle

update (*particles*, *u*, *t*, *noise*)

Propagate estimate forward in time

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal
- t (float): time-stamp
- noise: Unused for this type of model

Returns: (array-like) with first dimension = N, particle estimate at time t+1

Hierarchial

Model definition for base class for hierarchical systems

@author: Jerker Nordh

class pyparticleest.models.hierarchial.**HierarchicalBase** (*len_xi*, *len_z*, ***kwargs*)

Base class for Rao-Blackwellization of hierarchical models

Args:

- len_xi (int): number of nonlinear states
- len_z (int): number of linear states

calc_cond_dynamics (*particles*, *xi_next*, *u*, *t*)

Calculates the linear dynamics for each particle

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- xi_next (array-like): next non linear state
- u (array-like): input signal
- t (float): time stamp

Returns:

(Az, fz, Qz):

- Az (array-like): Az matrix for each particle
- fz (array-like): fz vector for each particle
- Qz (array-lie): Noise covariance for each particle

calc_xi_next (*particles*, *u*, *t*, *noise*)

Calculate the next nonlinear state given the input and noise realization

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- u (array-like): input signal

- t (float): time stamp
- noise (array-like): noise realization for each particle

Returns: (array-like): xi values for future particles

logp_xnext (*particles*, *next_part*, *u*, *t*)

Return the log-pdf value for the possible future state 'next_part' given input u

If Nn = 1 all particle are evaluated against the same future state, otherwise N must equal Nn and each particle is only evaluated against the future state with the same index.

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_part (array-like): future states, with first dimension = Nn
- u (array-like): input signal
- t (float): time stamp

Returns:

(array-like): log-probability of the future state for each particle

logp_xnext_xi (*particles*, *next_xi*, *u*, *t*)

Evaluate the log-probability of the next nonlinear state

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- next_xi (array-like): future nonlinear state
- u (array-like): input signal
- t (float): time stamp

Returns:

(array-like): log-probability of the future nonlinear state for each particle

measure (*particles*, *y*, *t*)

Return the log-pdf value of the measurement and update the statistics for the linear states

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- y (array-like): measurement
- t (float): time-stamp

Returns: (array-like) with first dimension = N, logp(y|x^i)

measure_nonlin (*particles*, *y*, *t*)

Measurement probability for the nonlinear parts of the measurement equations

Args:

- particles (array-like): Model specific representation of all particles, with first dimension = N (number of particles)

- y (array-like): measurement
- t (float): time stamp

Returns:

(array-like): log-probability of the measurement for each particle

sample_smooth (*part*, *ptraj*, *anc*, *future_trajs*, *find*, *ut*, *yt*, *tt*, *cur_ind*)

Sampled linear state conditioned on future_trajs

Args:

- part (array-like): Model specific representation of all particles, with first dimension = N (number of particles)
- ptraj: array of trajectory step objects from previous time-steps, last index is step just before the current
- anc (array-like): index of the ancestor of each particle in part
- future_trajs (array-like): particle estimate for {t+1:T}
- find (array-like): index in future_trajs corresponding to each particle in part
- ut (array-like): input signals for {0:T}
- yt (array-like): measurements for {0:T}
- tt (array-like): time stamps for {0:T}
- cur_ind (int): index of current timestep (in ut, yt and tt)

Returns: (array-like) with first dimension = N

Examples

Basic Model

Demonstrates a simple integrator model. Start by imported the interface classes from pyparticleest and the main simulator class

```
%matplotlib inline
import numpy
import pyparticleest.utils.kalman as kalman
import pyparticleest.interfaces as interfaces
import matplotlib.pyplot as plt
import pyparticleest.simulator as simulator
```

First we need the generate a dataset, it contains a true trajectory x and an array of measurements y. The goal is to estimate x using the data in y.

```
def generate_dataset(steps, P0, Q, R):
    x = numpy.zeros((steps + 1,))
    y = numpy.zeros((steps,))
    x[0] = 2.0 + 0.0 * numpy.random.normal(0.0, P0)
    for k in range(1, steps + 1):
        x[k] = x[k - 1] + numpy.random.normal(0.0, Q)
        y[k - 1] = x[k] + numpy.random.normal(0.0, R)

    return (x, y)
```

We need to specify the model which estimation is based upon, for this example we implement it directly on top of the interface specifications. More commonly one would use one of the base classes for a specific class of model to reduce the amount of code needed.

```
class Model(interfaces.ParticleFiltering):
    """ x_{k+1} = x_k + v_k, v_k ~ N(0,Q)
        y_k = x_k + e_k, e_k ~ N(0,R),
        x(0) ~ N(0,P0) """

    def __init__(self, P0, Q, R):
        self.P0 = numpy.copy(P0)
        self.Q = numpy.copy(Q)
        self.R = numpy.copy(R)

    def create_initial_estimate(self, N):
        return numpy.random.normal(0.0, self.P0, (N,)).reshape((-1, 1))

    def sample_process_noise(self, particles, u, t):
        """ Return process noise for input u """
        N = len(particles)
        return numpy.random.normal(0.0, self.Q, (N,)).reshape((-1, 1))

    def update(self, particles, u, t, noise):
        """ Update estimate using 'data' as input """
        particles += noise

    def measure(self, particles, y, t):
        """ Return the log-pdf value of the measurement """
        logyprob = numpy.empty(len(particles), dtype=float)
        for k in range(len(particles)):
            logyprob[k] = kalman.lognormpdf(particles[k].reshape(-1, 1) - y, self.R)
        return logyprob

    def logp_xnext_full(self, part, past_trajs, pind,
                        future_trajs, find, ut, yt, tt, cur_ind):

        diff = future_trajs[0].pa.part[find] - part

        logpxnext = numpy.empty(len(diff), dtype=float)
        for k in range(len(logpxnext)):
            logpxnext[k] = kalman.lognormpdf(diff[k].reshape(-1, 1), numpy.
↪asarray(self.Q).reshape(1, 1))
        return logpxnext
```

Define length of dataset and some parameters for the model defined above

```
steps = 50
num = 50
P0 = 1.0
Q = 1.0
R = numpy.asarray(((1.0,),))
```

Generate the dataset, but first set the seed for the random number generator so we always get the same example

```
numpy.random.seed(1)
(x, y) = generate_dataset(steps, P0, Q, R)
```

Instantiate the model and create the simulator object using the model and measurement y. This example does not use an input signal therefore set u=None

```
model = Model(P0, Q, R)
sim = simulator.Simulator(model, u=None, y=y)
```

Perform the estimation, using 'num' as both the number of forward particle and backward trajectories. For the smoother simply use the ancestral paths of each particle at the end time.

```
sim.simulate(num, num, smoother='ancestor')
```

```
33
```

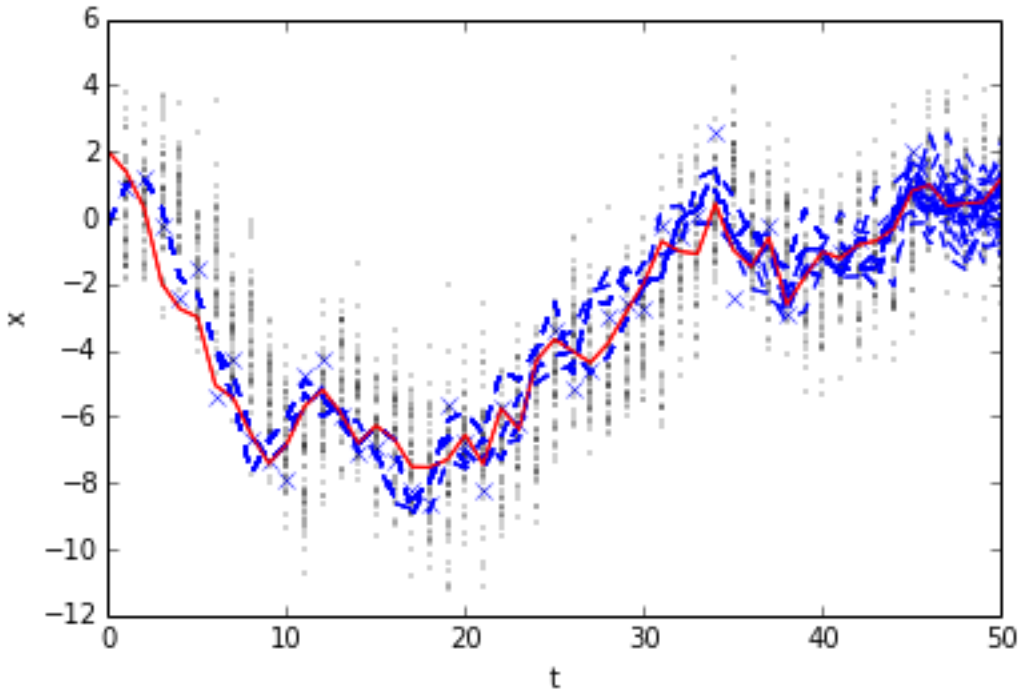
Extract filtered and smoothed estimates

```
(vals, _) = sim.get_filtered_estimates()
svals = sim.get_smoothed_estimates()
```

Plot true trajectory, measurements and the filtered and smoothed estimates

```
plt.plot(range(steps + 1), x, 'r-')
plt.plot(range(1, steps + 1), y, 'bx')
plt.plot(range(steps + 1), vals[:, :, 0], 'k.', markersize=0.8)
plt.plot(range(steps + 1), svals[:, :, 0], 'b--')
plt.plot(range(steps + 1), x, 'r-')
plt.xlabel('t')
plt.ylabel('x')
```

```
<matplotlib.text.Text at 0x7f654682bc10>
```



Standard Nonlinear Model

Example code for estimation the standard nonlinear model typically found in articles about particle filters

Import needed packages, from pyparticleest we use the base class for nonlinear models with additive Gaussian noise (NLG). We also use the simulator class as an easy interface to the actual algorithm inside pyparticleest.

```
%matplotlib inline
import numpy
import math
import pyparticleest.models.nlg as nlg
import pyparticleest.simulator as simulator
import matplotlib.pyplot as plt
```

Generate true trajectory x and measurements y. The goal is the create an estimate of x by only using the measurements y later in the code

```
def generate_dataset(steps, P0, Q, R):
    x = numpy.zeros((steps + 1,))
    y = numpy.zeros((steps + 1,))
    x[0] = numpy.random.multivariate_normal((0.0,), P0)
    y[0] = (0.05 * x[0] ** 2 +
            numpy.random.multivariate_normal((0.0,), R))
    for k in range(0, steps):
        x[k + 1] = (0.5 * x[k] +
                    25.0 * x[k] / (1 + x[k] ** 2) +
                    8 * math.cos(1.2 * k) +
                    numpy.random.multivariate_normal((0.0,), Q))
        y[k + 1] = (0.05 * x[k + 1] ** 2 +
                    numpy.random.multivariate_normal((0.0,), R))

    return (x, y)
```

Define the model for the standard nonlinaer example, it is of the type Nonlinear Gaussian with an initial Gaussian distribution of the states aswell. We therefore use the nlg.NonlinearGaussianInitialGaussian base class and only override the calc_g and calc_f methods.

```
class StdNonLin(nlg.NonlinearGaussianInitialGaussian):
    # x_{k+1} = 0.5*x_k + 25.0*x_k/(1+x_k**2) +
    #           8*math.cos(1.2*k) + v_k = f(x_k) + v:
    # y_k = 0.05*x_k**2 + e_k = g(x_k) + e_k,
    # x(0) ~ N(0,P0), v_k ~ N(0,Q), e_k ~ N(0,R)

    def __init__(self, P0, Q, R):
    # Set covariances in the constructor since they
    # are constant
        super(StdNonLin, self).__init__(Px0=P0, Q=Q, R=R)

    def calc_g(self, particles, t):
    # Calculate value of g(\xi_t,t)
        return 0.05 * particles ** 2

    def calc_f(self, particles, u, t):
    # Calculate value of f(xi_t,t)
        return (0.5 * particles +
                25.0 * particles / (1 + particles ** 2) +
                8 * math.cos(1.2 * t))
```

Define the length of the dataset and the noise parameters for our model.

```
T = 40
P0 = 5.0 * numpy.eye(1)
```

```
Q = 1.0 * numpy.eye(1)
R = 0.1 * numpy.eye(1)

# Forward particles
N = 100
# Backward trajectories
M = 10
```

Instantiate our model using the parameters defined above.

```
model = StdNonLin(P0, Q, R)
```

Set the seed of the random number generator so that the same dataset is generated each time the example is run.

```
numpy.random.seed(0)
(x, y) = generate_dataset(T, P0, Q, R)
```

Create a simulator object using our previously instatiated model combined with the measurements y. This example doesn't use any input signals u. Use N particle for the filter, M trajectories for the smoother. For the filtering algorithm use the standard bootstrap particle filter, for the smoothing use backward simulation with rejections sampling and adaptive stopping. Indicate that the first measurement is of the initial state. (If false the first measurement would have corresponding to after propagating the states forward in time once first).

```
sim = simulator.Simulator(model, u=None, y=y)
sim.simulate(N, M, filter='PF', smoother='rsas', meas_first=True)
```

```
25
```

Extract the filtered estimates, and computed the weighed mean of the filtered estimates.

```
(est_filt, w_filt) = sim.get_filtered_estimates()
mean_filt = sim.get_filtered_mean()
```

Extract smoothed estimates and mean and plot the mean.

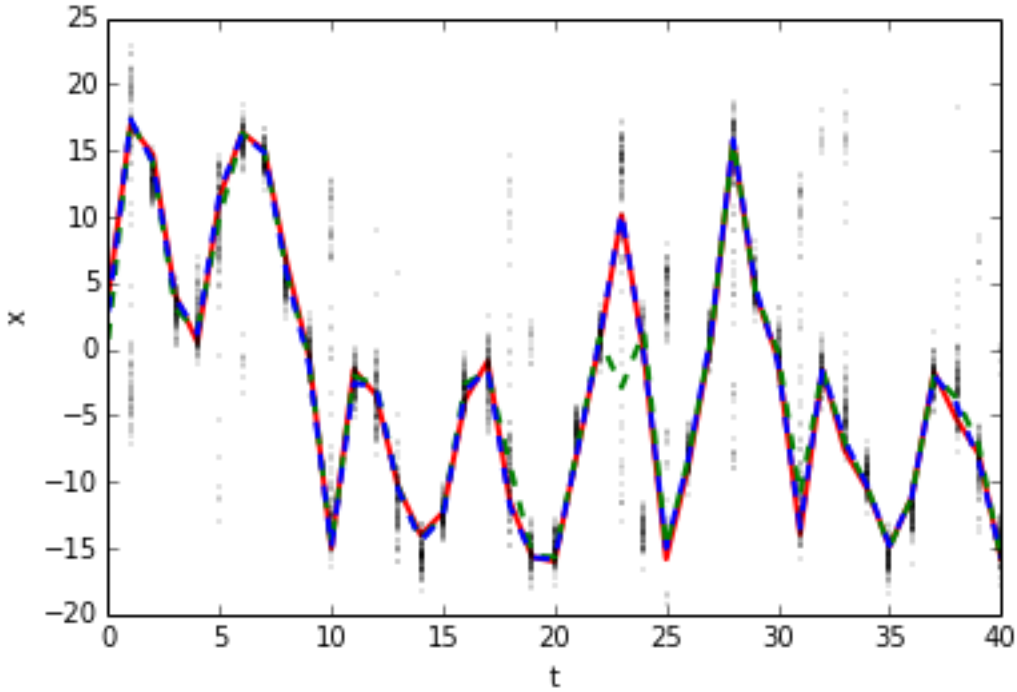
```
est_smooth = sim.get_smoothed_estimates()

mean_smooth = sim.get_smoothed_mean()
```

Plot the true state trajectory, particle estimates, weighted filtered mean and smoothed mean estimates

```
plt.plot(range(T + 1), x, 'r-', linewidth=2.0, label='True')
plt.plot((0,) * N, est_filt[0, :, 0].ravel(), 'k.',
        markersize=0.5, label='Particles')
for t in xrange(1, T + 1):
    plt.plot((t,) * N, est_filt[t, :, 0].ravel(),
             'k.', markersize=0.5)
plt.plot(range(T + 1), mean_filt[:, 0], 'g--',
         linewidth=2.0, label='Filter mean')
plt.plot(range(T + 1), mean_smooth[:, 0], 'b--',
         linewidth=2.0, label='Smoother mean')
plt.xlabel('t')
plt.ylabel('x')
```

```
<matplotlib.text.Text at 0x7f2a7848a3d0>
```


25
20
15
10
5
0
−5
−10
−15
−20
x
0
5
10
15
20
25
30
35
40
t

CHAPTER 2

Links

Project homepage: http://www.control.lth.se/Staff/JerkerNordh/pyparticleest.html

Source code: https://github.com/jerkern/pyParticleEst

Documentation: https://readthedocs.org/projects/pyparticleest

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

F

G

H

L

M

N

P

S

U