FilterPy Documentation

Release 1.4.4

Roger R. Labbe

Aug 24, 2018

Contents

FilterPy is a Python library that implements a number of Bayesian filters, most notably Kalman filters. I am writing it in conjunction with my book *Kalman and Bayesian Filters in Python*[1], a free book written using Ipython Notebook, hosted on github, and readable via nbviewer. However, it implements a wide variety of functionality that is not described in the book.

As such this library has a strong pedalogical flavor. It is rare that I choose the most efficient way to calculate something unless it does not obscure exposition of the concepts of the filtering being done. I will always opt for clarity over speed. I do not mean to imply that this is a toy; I use it all of the time in my job.

I mainly develop in Python 3.x, but this should support both Python 2.x and 3.x flavors. At the moment I can not tell you the lowest required version; I tend to develop on the bleeding edge of the Python releases. I am happy to receive bug reports if it does not work with older versions, but testing backwards compatibility is not a high priority at the moment. As the package matures I will shift my focus in that direction.

FilterPy requires Numpy[2] and SciPy[3] to work. The tests and examples also use matplotlib[4]. For testing I use py.test[5].

[1] Labbe, Roger. "Kalman and Bayesian Filters in Python".
[2] NumPy http://www.numpy.org
[3] SciPy http://www.scipy.org
[4] matplotlib http://http://matplotlib.org/
[5] pytest http://pytest.org/latest/

CHAPTER 1

Installation

1.1 Installation with pip (recommended)

FilterPy is available on github (https://github.com/rlabbe/filterpy). However, it is also hosted on PyPi, and unless you want to be on the bleeding edge of development I recommend you get it from there. To install from the command line, merely type:

```
$ pip install filterpy
```

To test the installation, from a python REPL type:

```
>>> import filterpy
>>> filterpy.__version__
```

and it should display the version number that you installed.

1.2 Installation with GitHub

You can get the very latest code by getting it from GitHub and then performing the installation. I will say I am not following particularly stringent version control discipline. I mostly stay on **master** and commit things that are not entirely ready for prime-time, mostly because I'm the only one developing. I do not promise that any check in that is not tagged with a version number is usable.

```
$ git clone --depth=1 https://github.com/rlabbe/filterpy.git
$ cd filterpy
$ python setup.py install
```

–depth=1 just gets you the last few revisions that I made, which keeps the repo small. If you want the entire repo leave out the *depth* parameter, or fork the repo if you plan to modify it.

CHAPTER 2

Use

There are several submodules, each listed below. But in general you will need to import which classes and/or functions you need from the correct submodule, construct the objects, and then execute your code. Something lke

```
>>> from filterpy.kalman import KalmanFilter
>>> kf = KalmanFilter(dim_x=3, dim_z=1)
```

I try to provide examples in the help for each class, but this documentation needs a lot of work. For now I refer you to my book mentioned above if the documentation is not adequate. Better yet, write an issue on the GitHub issue tracker. I will respond with an answer as soon as I am online and available (minutes to a day, normally), and then revise the documentation. I shouldn't have to be prodded like this, but life is limited. So prod.

Raise issues here: https://github.com/rlabbe/filterpy/issues

CHAPTER 3

FilterPy's Naming Conventions

A word on variable names. I am an advocate for descriptive variable names. In the Kalman filter literature the measurement noise covariance matrix is called *R*. The name *R* is not descriptive. I could reasonably call it *measurement_noise_covariance*, and I've seen libraries do that. I've chosen not to.

In the end, Kalman filtering is math. To write a Kalman filter you are going to start by sitting down with a piece of paper and doing math. You will be writing and solving normal algebraic equations. Every Kalman filter text and source on the web uses the same equations. You cannot read about the Kalman filter without seeing this equation

$$\dot{\mathbf{x}} = \mathbf{Fx} + \mathbf{Gu} + w$$

One of my goals is to bring you to the point where you can read the original literature on Kalman filtering. For nontrivial problems the difficulty is not the implementation of the equations, but learning how to set up the equations so they solve your problem. In other words, every Kalman filter implements $\dot{\mathbf{x}} = \mathbf{Fx} + \mathbf{Gu} + w$; the difficult part is figuring out what to put in the matrices $\mathbf{F}$ and $\mathbf{G}$ to make your filter work for your problem. Vast amounts of work have been done to apply Kalman filters in various domains, and it would be tragic to be unable to avail yourself of this research.

So, like it or not you will need to learn that $\mathbf{F}$ is the *state transition matrix* and that $\mathbf{R}$ is the *measurement noise covariance*. Once you know that the code will become readable, and until then Kalman filter math, and all publications and web articles on Kalman filters will be inaccessible to you.

Finally, I think that mathematical programming is somewhat different than regular programming; what is readable in one domain is not readable in another. *q = x + m* is opaque in a normal context. On the other hand, *x = (.5*a)*t**2 + v_0*t + x_0* is to me the most readable way to program the Newtonian distance equation:

$$x = \frac{1}{2}at^2 + v_0t + x_0$$

We could write it as

```
distance = (.5 * constant_acceleration) * time_delta**2 +
           initial_velocity * time_delta + initial_distance
```

but I feel that obscures readability. This is debatable for this one equation; but most mathematical programs, and certainly Kalman filters, use systems of equations. I can most easily follow the code, and ensure that it does not have

bugs, when it reads as close to the math as possible. Consider this equation from the Kalman filter:

$$\mathbf{K} = \mathbf{P}\mathbf{H}^\mathsf{T}[\mathbf{H}\mathbf{P}\mathbf{H}^\mathsf{T} + \mathbf{R}]^{-1}$$

Python code for this would be

```
K = dot(P, H.T).dot(inv(dot(H, P).dot(H.T) + R))
```

It's already a bit hard to read because of the *dot* function calls (required because Python does not yet support an operator for matrix multiplication). But compare this to:

```
kalman_gain = (
    dot(apriori_state_covariance, measurement_function_transpose).dot(
    inv(dot(measurement_function, apriori_state_covariance).dot(
    measurement_function_transpose) + measurement_noise_covariance)))
```

which I adapted from a popular library. I grant you this version has more context, but I cannot glance at this and see what math it is implementing. In particular, the linear algebra $\mathbf{H}\mathbf{P}\mathbf{H}^\mathsf{T}$ is doing something very specific - multiplying $\mathbf{P}$ by $\mathbf{H}$ in a way that converts $\mathbf{P}$ from *world space* to *measurement space* (we'll learn what that means). It is nearly impossible to see that the Kalman gain (*K*) is just a ratio of one number divided by a second number which has been converted to a different basis. This statement may not convey a lot of information to you before reading the book, but I assure you that $\mathbf{K} = \mathbf{P}\mathbf{H}^\mathsf{T}[\mathbf{H}\mathbf{P}\mathbf{H}^\mathsf{T} + \mathbf{R}]^{-1}$ is saying something very succinctly. There are two key pieces of information here - we are finding a ratio, and we are doing it in measurement space. I can see that in my first Python line, I cannot see that in the second line. If you want a counter-argument, my version obscures the information that $\mathbf{P}$ is in this context is a *prior* .

These comments apply to library code. Calling code should use names like *sensor_noise*, or *gps_sensor_noise*, not *R*. Math code should read like math, and interface or glue code should read like normal code. Context is important.

I will not *win* this argument, and some people will not agree with my naming choices. I will finish by stating, very truthfully, that I made two mistakes the first time I typed the second version and it took me awhile to find it. In any case, I aim for using the mathematical symbol names whenever possible, coupled with readable class and function names. So, it is *KalmanFilter.P*, not *KF.P* and not *KalmanFilter.apriori_state_covariance*.

CHAPTER 4

Communication

Unless it is deeply private (you don't want someone else seeing propietary code, for example), please ask questions and such on the issue tracker, not by email. This is solely so that everyone gets to see the answer. "Issue" doesn't mean bug.

CHAPTER 5

Modules

5.1 filterpy.kalman

The classes in this submodule implement the various Kalman filters. There is also support for smoother functions. The smoothers are methods of the classes. For example, the KalmanFilter class contains rts_smoother to perform Rauch-Tung-Striebal smoothing.

5.1.1 Linear Kalman Filters

Implements various Kalman filters using the linear equations form of the filter.

KalmanFilter

Implements a linear Kalman filter. For now the best documentation is my free book Kalman and Bayesian Filters in Python[2]

The test files in this directory also give you a basic idea of use, albeit without much description.

In brief, you will first construct this object, specifying the size of the state vector with *dim_x* and the size of the measurement vector that you will be using with *dim_z*. These are mostly used to perform size checks when you assign values to the various matrices. For example, if you specified *dim_z=2* and then try to assign a 3x3 matrix to R (the measurement noise matrix you will get an assert exception because R should be 2x2. (If for whatever reason you need to alter the size of things midstream just use the underscore version of the matrices to assign directly: your_filter._R = a_3x3_matrix.)

After construction the filter will have default matrices created for you, but you must specify the values for each. It's usually easiest to just overwrite them rather than assign to each element yourself. This will be clearer in the example below. All are of type numpy.array.

[2] Labbe, Roger. "Kalman and Bayesian Filters in Python".

These are the matrices (instance variables) which you must specify. All are of type numpy.array (do NOT use numpy.matrix) If dimensional analysis allows you to get away with a 1x1 matrix you may also use a scalar. All elements must have a type of float.

Instance Variables

You will have to assign reasonable values to all of these before running the filter. All must have dtype of float.

x [ndarray (dim_x, 1), default = [0,0,0...0]] filter state estimate

P [ndarray (dim_x, dim_x), default eye(dim_x)] covariance matrix

Q [ndarray (dim_x, dim_x), default eye(dim_x)] Process uncertainty/noise

R [ndarray (dim_z, dim_z), default eye(dim_x)] measurement uncertainty/noise

H [ndarray (dim_z, dim_x)] measurement function

F [ndarray (dim_x, dim_x)] state transistion matrix

B [ndarray (dim_x, dim_u), default 0] control transition matrix

Optional Instance Variables

alpha : float

Assign a value > 1.0 to turn this into a fading memory filter.

Read-only Instance Variables

K [ndarray] Kalman gain that was used in the most recent update() call.

y [ndarray] Residual calculated in the most recent update() call. I.e., the different between the measurement and the current estimated state projected into measurement space (z - Hx)

S [ndarray] System uncertainty projected into measurement space. I.e., HPH' + R. Probably not very useful, but it is here if you want it.

likelihood [float] Likelihood of last measurment update.

log_likelihood [float] Log likelihood of last measurment update.

Example

Here is a filter that tracks position and velocity using a sensor that only reads position.

First construct the object with the required dimensionality.

```
from filterpy.kalman import KalmanFilter
f = KalmanFilter (dim_x=2, dim_z=1)
```

Assign the initial value for the state (position and velocity). You can do this with a two dimensional array like so:

```
f.x = np.array([[2.],    # position
                [0.]])   # velocity
```

or just use a one dimensional array, which I prefer doing.

```
f.x = np.array([2., 0.])
```

Define the state transition matrix:

```
f.F = np.array([[1.,1.],
                [0.,1.]])
```

Define the measurement function:

```
f.H = np.array([[1.,0.]])
```

Define the covariance matrix. Here I take advantage of the fact that P already contains np.eye(dim_x), and just multiply by the uncertainty:

```
f.P *= 1000.
```

I could have written:

```
f.P = np.array([[1000.,    0.],
                [   0., 1000.] ])
```

You decide which is more readable and understandable.

Now assign the measurement noise. Here the dimension is 1x1, so I can use a scalar

```
f.R = 5
```

I could have done this instead:

```
f.R = np.array([[5.]])
```

Note that this must be a 2 dimensional array, as must all the matrices.

Finally, I will assign the process noise. Here I will take advantage of another FilterPy library function:

```
from filterpy.common import Q_discrete_white_noise
f.Q = Q_discrete_white_noise(dim=2, dt=0.1, var=0.13)
```

Now just perform the standard predict/update loop:

while some_condition_is_true:

```
z = get_sensor_reading()
f.predict()
f.update(z)

do_something_with_estimate (f.x)
```

Procedural Form

This module also contains stand alone functions to perform Kalman filtering. Use these if you are not a fan of objects.

Example

```
while True:
    z, R = read_sensor()
    x, P = predict(x, P, F, Q)
    x, P = update(x, P, z, R, H)
```

References

github repo: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

read online: http://nbviewer.ipython.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

PDF version (often lags the two sources above) https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/Kalman_and_Bayesian_Filters_in_Python.pdf

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Kalman filter

class filterpy.kalman.**KalmanFilter**(*dim_x*, *dim_z*, *dim_u=0*)

Implements a Kalman filter. You are responsible for setting the various state variables to reasonable values; the defaults will not give you a functional filter.

For now the best documentation is my free book Kalman and Bayesian Filters in Python *[2]*. The test files in this directory also give you a basic idea of use, albeit without much description.

In brief, you will first construct this object, specifying the size of the state vector with dim_x and the size of the measurement vector that you will be using with dim_z. These are mostly used to perform size checks when you assign values to the various matrices. For example, if you specified dim_z=2 and then try to assign a 3x3 matrix to R (the measurement noise matrix you will get an assert exception because R should be 2x2. (If for whatever reason you need to alter the size of things midstream just use the underscore version of the matrices to assign directly: your_filter._R = a_3x3_matrix.)

After construction the filter will have default matrices created for you, but you must specify the values for each. It's usually easiest to just overwrite them rather than assign to each element yourself. This will be clearer in the example below. All are of type numpy.array.

Parameters

dim_x [int] Number of state variables for the Kalman filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4. This is used to set the default size of P, Q, and u

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

dim_u [int (optional)] size of the control input, if it is being used. Default value of 0 indicates it is not used.

compute_log_likelihood [bool (default = True)] Computes log likelihood by default, but this can be a slow computation, so if you never use it you can turn this computation off.

References

[1], *[2]*

Examples

Here is a filter that tracks position and velocity using a sensor that only reads position.

First construct the object with the required dimensionality.

```
from filterpy.kalman import KalmanFilter
f = KalmanFilter (dim_x=2, dim_z=1)
```

Assign the initial value for the state (position and velocity). You can do this with a two dimensional array like so:

```
f.x = np.array([[2.],    # position
                [0.]])   # velocity
```

or just use a one dimensional array, which I prefer doing.

```
f.x = np.array([2., 0.])
```

Define the state transition matrix:

```
f.F = np.array([[1.,1.],
                [0.,1.]])
```

Define the measurement function:

f.H = np.array([[1.,0.]])

Define the covariance matrix. Here I take advantage of the fact that P already contains np.eye(dim_x), and just multiply by the uncertainty:

```
f.P *= 1000.
```

I could have written:

```
f.P = np.array([[1000.,    0.],
                [   0., 1000.] ])
```

You decide which is more readable and understandable.

Now assign the measurement noise. Here the dimension is 1x1, so I can use a scalar

```
f.R = 5
```

I could have done this instead:

```
f.R = np.array([[5.]])
```

Note that this must be a 2 dimensional array, as must all the matrices.

Finally, I will assign the process noise. Here I will take advantage of another FilterPy library function:

```
from filterpy.common import Q_discrete_white_noise
f.Q = Q_discrete_white_noise(dim=2, dt=0.1, var=0.13)
```

Now just perform the standard predict/update loop:

while some_condition_is_true:

```
z = get_sensor_reading()
f.predict()
f.update(z)

do_something_with_estimate (f.x)
```

Procedural Form

This module also contains stand alone functions to perform Kalman filtering. Use these if you are not a fan of objects.

Example

```
while True:
    z, R = read_sensor()
    x, P = predict(x, P, F, Q)
    x, P = update(x, P, z, R, H)
```

See my book Kalman and Bayesian Filters in Python *[2]*.

You will have to set the following attributes after constructing this object for the filter to perform properly. Please note that there are various checks in place to ensure that you have made everything the 'correct' size. However, it is possible to provide incorrectly sized arrays such that the linear algebra can not perform an operation. It can also fail silently - you can end up with matrices of a size that allows the linear algebra to work, but are the wrong shape for the problem you are trying to solve.

Attributes

x [numpy.array(dim_x, 1)] Current state estimate. Any call to update() or predict() updates this variable.

P [numpy.array(dim_x, dim_x)] Current state covariance matrix. Any call to update() or predict() updates this variable.

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

z [numpy.array] Last measurement used in update(). Read only.

R [numpy.array(dim_z, dim_z)] Measurement noise matrix

Q [numpy.array(dim_x, dim_x)] Process noise matrix

F [numpy.array()] State Transition matrix

H [numpy.array(dim_z, dim_x)] Measurement function

y [numpy.array] Residual of the update step. Read only.

K [numpy.array(dim_x, dim_z)] Kalman gain of the update step. Read only.

S [numpy.array] System uncertainty (P projected to measurement space). Read only.

SI [numpy.array] Inverse system uncertainty. Read only.

log_likelihood [float] log-likelihood of the last measurement.

likelihood [float] Computed from the log-likelihood.

mahalanobis [float] "

inv [function, default numpy.linalg.inv] If you prefer another inverse function, such as the Moore-Penrose pseudo inverse, set it to that instead: kf.inv = np.linalg.pinv

This is only used to invert self.S. If you know it is diagonal, you might choose to set it to filterpy.common.inv_diagonal, which is several times faster than numpy.linalg.inv for diagonal matrices.

alpha [float] Fading memory setting.

__init__(*dim_x*, *dim_z*, *dim_u=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

predict(*u=None*, *B=None*, *F=None*, *Q=None*)
Predict next state (prior) using the Kalman filter state propagation equations.

Parameters

u [np.array] Optional control vector. If not *None*, it is multiplied by B to create the control input into the system.

B [np.array(dim_x, dim_z), or None] Optional control transition matrix; a value of None will cause the filter to use *self.B*.

F [np.array(dim_x, dim_x), or None] Optional state transition matrix; a value of None will cause the filter to use *self.F*.

Q [np.array(dim_x, dim_x), scalar, or None] Optional process noise matrix; a value of None will cause the filter to use *self.Q*.

update(*z*, *R=None*, *H=None*)
Add a new measurement (z) to the Kalman filter.

If z is None, nothing is computed. However, x_post and P_post are updated with the prior (x_prior, P_prior), and self.z is set to None.

Parameters

z [(dim_z, 1): array_like] measurement for this update. z can be a scalar if dim_z is 1, otherwise it must be convertible to a column vector.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

H [np.array, or None] Optionally provide H to override the measurement function for this one call, otherwise self.H will be used.

predict_steadystate(*u=0*, *B=None*)
Predict state (prior) using the Kalman filter state propagation equations. Only x is updated, P is left unchanged. See update_steadstate() for a longer explanation of when to use this method.

Parameters

u [np.array] Optional control vector. If non-zero, it is multiplied by B to create the control input into the system.

B [np.array(dim_x, dim_z), or None] Optional control transition matrix; a value of None will cause the filter to use *self.B*.

update_steadystate(*z*)
Add a new measurement (z) to the Kalman filter without recomputing the Kalman gain K, the state covariance P, or the system uncertainty S.

You can use this for LTI systems since the Kalman gain and covariance converge to a fixed value. Precompute these and assign them explicitly, or run the Kalman filter using the normal predict()/update(0 cycle until they converge.

The main advantage of this call is speed. We do significantly less computation, notably avoiding a costly matrix inversion.

Use in conjunction with predict_steadystate(), otherwise P will grow without bound.

Parameters

z [(dim_z, 1): array_like] measurement for this update. z can be a scalar if dim_z is 1, otherwise it must be convertible to a column vector.

Examples

```
>>> cv = kinematic_kf(dim=3, order=2) # 3D const velocity filter
>>> # let filter converge on representative data, then save k and P
>>> for i in range(100):
>>>     cv.predict()
>>>     cv.update([i, i, i])
>>> saved_k = np.copy(cv.K)
>>> saved_P = np.copy(cv.P)
```

later on:

```
>>> cv = kinematic_kf(dim=3, order=2) # 3D const velocity filter
>>> cv.K = np.copy(saved_K)
>>> cv.P = np.copy(saved_P)
>>> for i in range(100):
>>>     cv.predict_steadystate()
>>>     cv.update_steadystate([i, i, i])
```

update_correlated(*z*, *R=None*, *H=None*)

Add a new measurement (z) to the Kalman filter assuming that process noise and measurement noise are correlated as defined in the *self.M* matrix.

If z is None, nothing is changed.

Parameters

z [(dim_z, 1): array_like] measurement for this update. z can be a scalar if dim_z is 1, otherwise it must be convertible to a column vector.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

H [np.array, or None] Optionally provide H to override the measurement function for this one call, otherwise self.H will be used.

batch_filter(*zs*, *Fs=None*, *Qs=None*, *Hs=None*, *Rs=None*, *Bs=None*, *us=None*, *update_first=False*, *saver=None*)

Batch processes a sequences of measurements.

Parameters

zs [list-like]

list of measurements at each time step *self.dt*. Missing measurements must be represented by *None*.

Fs [None, list-like, default=None] optional value or list of values to use for the state transition matrix F.

If Fs is None then self.F is used for all epochs.

Otherwise it must contain a list-like list of F's, one for each epoch. This allows you to have varying F per epoch.

Qs [None, np.array or list-like, default=None] optional value or list of values to use for the process error covariance Q.

If Qs is None then self.Q is used for all epochs.

Otherwise it must contain a list-like list of Q's, one for each epoch. This allows you to have varying Q per epoch.

Hs [None, np.array or list-like, default=None] optional list of values to use for the measurement matrix H.

If Hs is None then self.H is used for all epochs.

If Hs contains a single matrix, then it is used as H for all epochs.

Otherwise it must contain a list-like list of H's, one for each epoch. This allows you to have varying H per epoch.

Rs [None, np.array or list-like, default=None] optional list of values to use for the measurement error covariance R.

If Rs is None then self.R is used for all epochs.

Otherwise it must contain a list-like list of R's, one for each epoch. This allows you to have varying R per epoch.

Bs [None, np.array or list-like, default=None] optional list of values to use for the control transition matrix B.

If Bs is None then self.B is used for all epochs.

Otherwise it must contain a list-like list of B's, one for each epoch. This allows you to have varying B per epoch.

us [None, np.array or list-like, default=None] optional list of values to use for the control input vector;

If us is None then None is used for all epochs (equivalent to 0, or no control input).

Otherwise it must contain a list-like list of u's, one for each epoch.

update_first [bool, optional, default=False]

controls whether the order of operations is update followed by predict, or predict followed by update. Default is predict->update.

saver [filterpy.common.Saver, optional] filterpy.common.Saver object. If provided, saver.save() will be called after every epoch

Returns

means [np.array((n,dim_x,1))]

array of the state for each time step after the update. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance [np.array((n,dim_x,dim_x))] array of the covariances for each time step after the update. In other words *covariance[k,:,:]* is the covariance at step *k*.

means_predictions [np.array((n,dim_x,1))] array of the state for each time step after the predictions. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance_predictions [np.array((n,dim_x,dim_x))] array of the covariances for each time step after the prediction. In other words *covariance[k,:,:]* is the covariance at step *k*.

Examples

```
# this example demonstrates tracking a measurement where the time
# between measurement varies, as stored in dts. This requires
# that F be recomputed for each epoch. The output is then smoothed
# with an RTS smoother.

zs = [t + random.randn()*4 for t in range (40)]
Fs = [np.array([[1., dt], [0, 1]] for dt in dts]

(mu, cov, _, _) = kf.batch_filter(zs, Fs=Fs)
(xs, Ps, Ks) = kf.rts_smoother(mu, cov, Fs=Fs)
```

rts_smoother(*Xs*, *Ps*, *Fs=None*, *Qs=None*, *inv=<Mock name='mock.linalg.inv' id='140694326135248'>*)

Runs the Rauch-Tung-Striebal Kalman smoother on a set of means and covariances computed by a Kalman filter. The usual input would come from the output of *KalmanFilter.batch_filter()*.

Parameters

Xs [numpy.array] array of the means (state variable x) of the output of a Kalman filter.

Ps [numpy.array] array of the covariances of the output of a kalman filter.

Fs [list-like collection of numpy.array, optional] State transition matrix of the Kalman filter at each time step. Optional, if not provided the filter's self.F will be used

Qs [list-like collection of numpy.array, optional] Process noise of the Kalman filter at each time step. Optional, if not provided the filter's self.Q will be used

inv [function, default numpy.linalg.inv] If you prefer another inverse function, such as the Moore-Penrose pseudo inverse, set it to that instead: kf.inv = np.linalg.pinv

Returns

x [numpy.ndarray] smoothed means

P [numpy.ndarray] smoothed state covariances

K [numpy.ndarray] smoother gain at each step

Pp [numpy.ndarray] Predicted state covariances

Examples

```
zs = [t + random.randn()*4 for t in range (40)]

(mu, cov, _, _) = kalman.batch_filter(zs)
(x, P, K, Pp) = rts_smoother(mu, cov, kf.F, kf.Q)
```

get_prediction(*u=0*)

Predicts the next state of the filter and returns it without altering the state of the filter.

Parameters

u [np.array] optional control input

Returns

(x, P) [tuple] State vector and covariance array of the prediction.

get_update(*z=None*)
Computes the new estimate based on measurement *z* and returns it without altering the state of the filter.

Parameters

z [(dim_z, 1): array_like] measurement for this update. z can be a scalar if dim_z is 1, otherwise it must be convertible to a column vector.

Returns

(x, P) [tuple] State vector and covariance array of the update.

residual_of(*z*)
Returns the residual for the given measurement (z). Does not alter the state of the filter.

measurement_of_state(*x*)
Helper function that converts a state into a measurement.

Parameters

x [np.array] kalman state vector

Returns

z [(dim_z, 1): array_like] measurement for this update. z can be a scalar if dim_z is 1, otherwise it must be convertible to a column vector.

log_likelihood
log-likelihood of the last measurement.

likelihood
Computed from the log-likelihood. The log-likelihood can be very small, meaning a large negative value such as -28000. Taking the exp() of that results in 0.0, which can break typical algorithms which multiply by this value, so by default we always return a number >= sys.float_info.min.

mahalanobis
" Mahalanobis distance of measurement. E.g. 3 means measurement was 3 standard deviations away from the predicted value.

Returns

mahalanobis [float]

log_likelihood_of(*z*)
log likelihood of the measurement *z*. This should only be called after a call to update(). Calling after predict() will yield an incorrect result.

alpha
Fading memory setting. 1.0 gives the normal Kalman filter, and values slightly larger than 1.0 (such as 1.02) give a fading memory effect - previous measurements have less influence on the filter's estimates. This formulation of the Fading memory filter (there are many) is due to Dan Simon [1]_.

test_matrix_dimensions(*z=None*, *H=None*, *R=None*, *F=None*, *Q=None*)
Performs a series of asserts to check that the size of everything is what it should be. This can help you debug problems in your design.

If you pass in H, R, F, Q those will be used instead of this object's value for those matrices.

Testing *z* (the measurement) is problamatic. x is a vector, and can be implemented as either a 1D array or as a nx1 column vector. Thus Hx can be of different shapes. Then, if Hx is a single value, it can be either a 1D array or 2D vector. If either is true, z can reasonably be a scalar (either '3' or np.array('3') are scalars under this definition), a 1D, 1 element array, or a 2D, 1 element array. You are allowed to pass in any combination that works.

filterpy.kalman.**update**(*x*, *P*, *z*, *R*, *H=None*, *return_all=False*)

Add a new measurement (z) to the Kalman filter. If z is None, nothing is changed.

This can handle either the multidimensional or unidimensional case. If all parameters are floats instead of arrays the filter will still work, and return floats for x, P as the result.

update(1, 2, 1, 1, 1) # univariate update(x, P, 1

Parameters

x [numpy.array(dim_x, 1), or float] State estimate vector

P [numpy.array(dim_x, dim_x), or float] Covariance matrix

z [(dim_z, 1): array_like] measurement for this update. z can be a scalar if dim_z is 1, otherwise it must be convertible to a column vector.

R [numpy.array(dim_z, dim_z), or float] Measurement noise matrix

H [numpy.array(dim_x, dim_x), or float, optional] Measurement function. If not provided, a value of 1 is assumed.

return_all [bool, default False] If true, y, K, S, and log_likelihood are returned, otherwise only x and P are returned.

Returns

x [numpy.array] Posterior state estimate vector

P [numpy.array] Posterior covariance matrix

y [numpy.array or scalar] Residua. Difference between measurement and state in measurement space

K [numpy.array] Kalman gain

S [numpy.array] System uncertainty in measurement space

log_likelihood [float] log likelihood of the measurement

filterpy.kalman.**predict**(*x*, *P*, *F=1*, *Q=0*, *u=0*, *B=1*, *alpha=1.0*)

Predict next state (prior) using the Kalman filter state propagation equations.

Parameters

x [numpy.array] State estimate vector

P [numpy.array] Covariance matrix

F [numpy.array()] State Transition matrix

Q [numpy.array, Optional] Process noise matrix

u [numpy.array, Optional, default 0.] Control vector. If non-zero, it is multiplied by B to create the control input into the system.

B [numpy.array, optional, default 0.] Control transition matrix.

alpha [float, Optional, default=1.0] Fading memory setting. 1.0 gives the normal Kalman filter, and values slightly larger than 1.0 (such as 1.02) give a fading memory effect - previous measurements have less influence on the filter's estimates. This formulation of the Fading memory filter (there are many) is due to Dan Simon

Returns

x [numpy.array] Prior state estimate vector

P [numpy.array] Prior covariance matrix

filterpy.kalman.**batch_filter**(*x*, *P*, *zs*, *Fs*, *Qs*, *Hs*, *Rs*, *Bs=None*, *us=None*, *update_first=False*, *saver=None*)

Batch processes a sequences of measurements.

Parameters

zs [list-like] list of measurements at each time step. Missing measurements must be represented by None.

Fs [list-like] list of values to use for the state transition matrix matrix.

Qs [list-like] list of values to use for the process error covariance.

Hs [list-like] list of values to use for the measurement matrix.

Rs [list-like] list of values to use for the measurement error covariance.

Bs [list-like, optional] list of values to use for the control transition matrix; a value of None in any position will cause the filter to use *self.B* for that time step.

us [list-like, optional] list of values to use for the control input vector; a value of None in any position will cause the filter to use 0 for that time step.

update_first [bool, optional] controls whether the order of operations is update followed by predict, or predict followed by update. Default is predict->update.

saver [filterpy.common.Saver, optional] filterpy.common.Saver object. If provided, saver.save() will be called after every epoch

Returns

means [np.array((n,dim_x,1))] array of the state for each time step after the update. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance [np.array((n,dim_x,dim_x))] array of the covariances for each time step after the update. In other words *covariance[k,:,:]* is the covariance at step *k*.

means_predictions [np.array((n,dim_x,1))] array of the state for each time step after the predictions. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance_predictions [np.array((n,dim_x,dim_x))] array of the covariances for each time step after the prediction. In other words *covariance[k,:,:]* is the covariance at step *k*.

Examples

```
zs = [t + random.randn()*4 for t in range (40)]
Fs = [kf.F for t in range (40)]
Hs = [kf.H for t in range (40)]

(mu, cov, _, _) = kf.batch_filter(zs, Rs=R_list, Fs=Fs, Hs=Hs, Qs=None,
                                  Bs=None, us=None, update_first=False)
(xs, Ps, Ks) = kf.rts_smoother(mu, cov, Fs=Fs, Qs=None)
```

Saver

This is a helper class designed to allow you to save the state of the Kalman filter for each epoch. Each instance variable is stored in a list when you call save().

This class is deprecated as of version 1.3.2 and will be deleted soon. Instead, see the class filterpy.common.Saver, which works for any class, not just a KalmanFilter object.

Example

```
saver = Saver(kf)
for i in range(N):
    kf.predict()
    kf.update(zs[i])
    saver.save()

saver.to_array() # convert all to np.array

# plot the 0th element of kf.x over all epoches
plot(saver.xs[:, 0])
```

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Kalman filter saver

class filterpy.kalman.**Saver**(*kf*, *save_current=True*)

Deprecated. Use filterpy.common.Saver instead.

Helper class to save the states of the KalmanFilter class. Each time you call save() the current states are appended to lists. Generally you would do this once per epoch - predict/update.

Once you are done filtering you can optionally call to_array() to convert all of the lists to numpy arrays. You cannot safely call save() after calling to_array().

Examples

```
kf = KalmanFilter(...whatever)
# initialize kf here

saver = Saver(kf) # save data for kf filter
for z in zs:
    kf.predict()
    kf.update(z)

    saver.save()

saver.to_array()
# plot the 0th element of the state
plt.plot(saver.xs[:, 0, 0])
```

__init__(*kf*, *save_current=True*)

Construct the save object, optionally saving the current state of the filter

save()

save the current state of the Kalman filter

to_array()

convert all of the lists into np.array

FixedLagSmoother

Introduction and Overview

This implements a fixed lag Kalman smoother.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.kalman.**FixedLagSmoother**(*dim_x*, *dim_z*, *N=None*)

Fixed Lag Kalman smoother.

Computes a smoothed sequence from a set of measurements based on the fixed lag Kalman smoother. At time k, for a lag N, the fixed-lag smoother computes the state estimate for time k-N based on all measurements made between times k-N and k. This yields a pretty good smoothed result with O(N) extra computations performed for each measurement. In other words, if N=4 this will consume about 5x the number of computations as a basic Kalman filter. However, the loops contain only 3 dot products, so it will be much faster than this sounds as the main Kalman filter loop involves transposes and inverses, as well as many more matrix multiplications.

Implementation based on Wikipedia article as it existed on November 18, 2014.

References

Wikipedia http://en.wikipedia.org/wiki/Kalman_filter#Fixed-lag_smoother

Simon, Dan. "Optimal State Estimation," John Wiley & Sons pp 274-8 (2006).

Examples

```
from filterpy.kalman import FixedLagSmoother
fls = FixedLagSmoother(dim_x=2, dim_z=1)

fls.x = np.array([[0.],
                  [.5]])

fls.F = np.array([[1.,1.],
                  [0.,1.]])

fls.H = np.array([[1.,0.]])

fls.P *= 200
fls.R *= 5.
fls.Q *= 0.001
```

(continues on next page)

(continued from previous page)

```
zs = [...some measurements...]
xhatsmooth, xhat = fls.smooth_batch(zs, N=4)
```

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

__init__(*dim_x*, *dim_z*, *N=None*)

Create a fixed lag Kalman filter smoother. You are responsible for setting the various state variables to reasonable values; the defaults below will not give you a functional filter.

Parameters

dim_x [int] Number of state variables for the Kalman filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

This is used to set the default size of P, Q, and u

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

N [int, optional] If provided, the size of the lag. Not needed if you are only using smooth_batch() function. Required if calling smooth()

smooth(*z*, *u=None*)

Smooths the measurement using a fixed lag smoother.

On return, self.xSmooth is populated with the N previous smoothed estimates, where self.xSmooth[k] is the kth time step. self.x merely contains the current Kalman filter output of the most recent measurement, and is not smoothed at all (beyond the normal Kalman filter processing).

self.xSmooth grows in length on each call. If you run this 1 million times, it will contain 1 million elements. Sure, we could minimize this, but then this would make the caller's code much more cumbersome.

This also means that you cannot use this filter to track more than one data set; as data will be hopelessly intermingled. If you want to filter something else, create a new FixedLagSmoother object.

Parameters

z [ndarray or scalar] measurement to be smoothed

u [ndarray, optional] If provided, control input to the filter

smooth_batch(*zs*, *N*, *us=None*)

batch smooths the set of measurements using a fixed lag smoother. I consider this function a somewhat pedalogical exercise; why would you not use a RTS smoother if you are able to batch process your data? Hint: RTS is a much better smoother, and faster besides. Use it.

This is a batch processor, so it does not alter any of the object's data. In particular, self.x is NOT modified. All date is returned by the function.

Parameters

zs [ndarray of measurements] iterable list (usually ndarray, but whatever works for you) of measurements that you want to smooth, one per time step.

N [int] size of fixed lag in time steps

us [ndarray, optional] If provided, control input to the filter for each time step

Returns

(xhat_smooth, xhat) [ndarray, ndarray] xhat_smooth is the output of the N step fix lag smoother xhat is the filter output of the standard Kalman filter

SquareRootKalmanFilter

Introduction and Overview

This implements a square root Kalman filter. No real attempt has been made to make this fast; it is a pedalogical exercise. The idea is that by computing and storing the square root of the covariance matrix we get about double the significant number of bits. Some authors consider this somewhat unnecessary with modern hardware. Of course, with microcontrollers being all the rage these days, that calculus has changed. But, will you really run a Kalman filter in Python on a tiny chip? No. So, this is for learning.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Square Root Kalman Filter

class filterpy.kalman.**SquareRootKalmanFilter**(*dim_x*, *dim_z*, *dim_u=0*)

Create a Kalman filter which uses a square root implementation. This uses the square root of the state covariance matrix, which doubles the numerical precision of the filter, Therebuy reducing the effect of round off errors.

It is likely that you do not need to use this algorithm; we understand divergence issues very well now. However, if you expect the covariance matrix P to vary by 20 or more orders of magnitude then perhaps this will be useful to you, as the square root will vary by 10 orders of magnitude. From my point of view this is merely a 'reference' algorithm; I have not used this code in real world software. Brown[1] has a useful discussion of when you might need to use the square root form of this algorithm.

You are responsible for setting the various state variables to reasonable values; the defaults below will not give you a functional filter.

Parameters

dim_x [int] Number of state variables for the Kalman filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

This is used to set the default size of P, Q, and u

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

dim_u [int (optional)] size of the control input, if it is being used. Default value of 0 indicates it is not used.

References

[1] Robert Grover Brown. Introduction to Random Signals and Applied Kalman Filtering. Wiley and sons, 2012.

Examples

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

x [numpy.array(dim_x, 1)] State estimate

P [numpy.array(dim_x, dim_x)] covariance matrix

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] covariance matrix of the prior

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] covariance matrix of the posterior

z [numpy.array] Last measurement used in update(). Read only.

R [numpy.array(dim_z, dim_z)] measurement uncertainty

Q [numpy.array(dim_x, dim_x)] Process uncertainty

F [numpy.array()] State Transition matrix

H [numpy.array(dim_z, dim_x)] Measurement function

y [numpy.array] Residual of the update step. Read only.

K [numpy.array(dim_x, dim_z)] Kalman gain of the update step. Read only.

S [numpy.array] Systen uncertaintly projected to measurement space. Read only.

__init__(*dim_x*, *dim_z*, *dim_u=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

update(*z*, *R2=None*)
Add a new measurement (z) to the kalman filter. If z is None, nothing is changed.

Parameters

z [np.array] measurement for this update.

R2 [np.array, scalar, or None] Sqrt of meaaurement noize. Optionally provide to override the measurement noise for this one call, otherwise self.R2 will be used.

predict(*u=0*)
Predict next state (prior) using the Kalman filter state propagation equations.

Parameters

u [np.array, optional] Optional control vector. If non-zero, it is multiplied by B to create the control input into the system.

residual_of(*z*)
returns the residual for the given measurement (z). Does not alter the state of the filter.

measurement_of_state(*x*)
Helper function that converts a state into a measurement.

Parameters

x [np.array] kalman state vector

Returns

z [np.array] measurement corresponding to the given state

Q1_2
Sqrt Process uncertainty

Q
Process uncertainty

P_prior
covariance matrix of the prior

P_post
covariance matrix of the posterior

P1_2
sqrt of covariance matrix

P
covariance matrix

R1_2
sqrt of measurement uncertainty

R
measurement uncertainty

InformationFilter

Introduction and Overview

This is a basic implementation of the information filter.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.kalman.**InformationFilter**(*dim_x*, *dim_z*, *dim_u=0*, *compute_log_likelihood=True*)

Create a linear Information filter. Information filters compute the inverse of the Kalman filter, allowing you to easily denote having no information at initialization.

You are responsible for setting the various state variables to reasonable values; the defaults below will not give you a functional filter.

Parameters

dim_x [int] Number of state variables for the filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

This is used to set the default size of P, Q, and u

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

dim_u [int (optional)] size of the control input, if it is being used. Default value of 0 indicates it is not used.

self.compute_log_likelihood = compute_log_likelihood

self.log_likelihood = math.log(sys.float_info.min)

Examples

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

x [numpy.array(dim_x, 1)] State estimate vector

P_inv [numpy.array(dim_x, dim_x)] inverse state covariance matrix

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_inv_prior [numpy.array(dim_x, dim_x)] Inverse prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_inv_post [numpy.array(dim_x, dim_x)] Inverse posterior (updated) state covariance matrix. Read Only.

z [ndarray] Last measurement used in update(). Read only.

R_inv [numpy.array(dim_z, dim_z)] inverse of measurement noise matrix

Q [numpy.array(dim_x, dim_x)] Process noise matrix

H [numpy.array(dim_z, dim_x)] Measurement function

y [numpy.array] Residual of the update step. Read only.

K [numpy.array(dim_x, dim_z)] Kalman gain of the update step. Read only.

S [numpy.array] Systen uncertaintly projected to measurement space. Read only.

log_likelihood [float] log-likelihood of the last measurement. Read only.

likelihood [float] likelihood of last measurment. Read only.

Computed from the log-likelihood. The log-likelihood can be very small, meaning a large negative value such as -28000. Taking the exp() of that results in 0.0, which can break typical algorithms which multiply by this value, so by default we always return a number >= sys.float_info.min.

inv [function, default numpy.linalg.inv] If you prefer another inverse function, such as the Moore-Penrose pseudo inverse, set it to that instead: kf.inv = np.linalg.pinv

__init__(*dim_x*, *dim_z*, *dim_u=0*, *compute_log_likelihood=True*)
x.__init__(...) initializes x; see help(type(x)) for signature

update(*z*, *R_inv=None*)
Add a new measurement (z) to the kalman filter. If z is None, nothing is changed.

Parameters

z [np.array] measurement for this update.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

predict(*u=0*)
Predict next position.

Parameters

u [ndarray] Optional control vector. If non-zero, it is multiplied by B to create the control input into the system.

batch_filter(*zs*, *Rs=None*, *update_first=False*, *saver=None*)
Batch processes a sequences of measurements.

Parameters

zs [list-like] list of measurements at each time step *self.dt* Missing measurements must be represented by 'None'.

Rs [list-like, optional] optional list of values to use for the measurement error covariance; a value of None in any position will cause the filter to use *self.R* for that time step.

update_first [bool, optional,] controls whether the order of operations is update followed by predict, or predict followed by update. Default is predict->update.

saver [filterpy.common.Saver, optional] filterpy.common.Saver object. If provided, saver.save() will be called after every epoch

Returns

means: np.array((n,dim_x,1)) array of the state for each time step. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance: np.array((n,dim_x,dim_x)) array of the covariances for each time step. In other words *covariance[k,:,:]* is the covariance at step *k*.

F
State Transition matrix

P
State covariance matrix

FadingKalmanFilter

Implements a fading memory Kalman filter.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Fading Memory Kalman filter

class filterpy.kalman.**FadingKalmanFilter**(*alpha*, *dim_x*, *dim_z*, *dim_u=0*)
Fading memory Kalman filter. This implements a linear Kalman filter with a fading memory effect controlled by *alpha*. This is obsolete. The class KalmanFilter now incorporates the *alpha* attribute, and should be used instead.

You are responsible for setting the various state variables to reasonable values; the defaults below will not give you a functional filter.

Parameters

alpha [float, >= 1] alpha controls how much you want the filter to forget past measurements. alpha==1 yields identical performance to the Kalman filter. A typical application might use 1.01

dim_x [int] Number of state variables for the Kalman filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

This is used to set the default size of P, Q, and u

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

dim_u [int (optional)] size of the control input, if it is being used. Default value of 0 indicates it is not used.

Examples

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

You will have to assign reasonable values to all of these before

running the filter. All must have dtype of float

x [ndarray (dim_x, 1), default = [0,0,0...0]] state of the filter

P [ndarray (dim_x, dim_x), default identity matrix] covariance matrix

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

z [ndarray] Last measurement used in update(). Read only.

Q [ndarray (dim_x, dim_x), default identity matrix] Process uncertainty matrix

R [ndarray (dim_z, dim_z), default identity matrix] measurement uncertainty

H [ndarray (dim_z, dim_x)] measurement function

F [ndarray (dim_x, dim_x)] state transistion matrix

B [ndarray (dim_x, dim_u), default 0] control transition matrix

y [numpy.array] Residual of the update step. Read only.

K [numpy.array(dim_x, dim_z)] Kalman gain of the update step. Read only.

S [numpy.array] System uncertainty (P projected to measurement space). Read only.

S [numpy.array] Inverse system uncertainty. Read only.

log_likelihood [float] log-likelihood of the last measurement.

likelihood [float] Computed from the log-likelihood.

mahalanobis [float] "

__init__(*alpha*, *dim_x*, *dim_z*, *dim_u=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

update(*z*, *R=None*)
Add a new measurement (z) to the kalman filter. If z is None, nothing is changed.

Parameters

z [np.array] measurement for this update.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

predict(*u=0*)
Predict next position.

Parameters

u [np.array] Optional control vector. If non-zero, it is multiplied by B to create the control input into the system.

batch_filter(*zs*, *Rs=None*, *update_first=False*)
Batch processes a sequences of measurements.

Parameters

zs [list-like] list of measurements at each time step *self.dt* Missing measurements must be represented by 'None'.

Rs [list-like, optional] optional list of values to use for the measurement error covariance; a value of None in any position will cause the filter to use *self.R* for that time step.

update_first [bool, optional,] controls whether the order of operations is update followed by predict, or predict followed by update. Default is predict->update.

Returns

means: np.array((n,dim_x,1)) array of the state for each time step after the update. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance: np.array((n,dim_x,dim_x)) array of the covariances for each time step after the update. In other words *covariance[k,:,:]* is the covariance at step *k*.

means_predictions: np.array((n,dim_x,1)) array of the state for each time step after the predictions. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance_predictions: np.array((n,dim_x,dim_x)) array of the covariances for each time step after the prediction. In other words *covariance[k,:,:]* is the covariance at step *k*.

get_prediction(*u=0*)
Predicts the next state of the filter and returns it. Does not alter the state of the filter.

Parameters

u [np.array] optional control input

Returns

(x, P) State vector and covariance array of the prediction.

residual_of(*z*)
returns the residual for the given measurement (z). Does not alter the state of the filter.

measurement_of_state(*x*)
Helper function that converts a state into a measurement.

Parameters

x [np.array] kalman state vector

Returns

z [np.array] measurement corresponding to the given state

alpha
scaling factor for fading memory

log_likelihood
log-likelihood of the last measurement.

likelihood
Computed from the log-likelihood. The log-likelihood can be very small, meaning a large negative value such as -28000. Taking the exp() of that results in 0.0, which can break typical algorithms which multiply by this value, so by default we always return a number >= sys.float_info.min.

mahalanobis
" Mahalanobis distance of innovation. E.g. 3 means measurement was 3 standard deviations away from the predicted value.

Returns

mahalanobis [float]

MMAE Filter Bank

needs documentation....

Example

```
from filterpy.kalman import MMAEFilterBank

pos, zs = generate_data(120, noise_factor=0.2)
z_xs = zs[:, 0]
t = np.arange(0, len(z_xs) * dt, dt)

dt = 0.1
filters = [make_cv_filter(dt), make_ca_filter(dt)]
H_cv = np.array([[1., 0, 0],
                 [0., 1, 0]])

H_ca = np.array([[1., 0., 0.],
                 [0., 1., 0.],
                 [0., 0., 1.]])


bank = MMAEFilterBank(filters, (0.5, 0.5), dim_x=3, H=(H_cv, H_ca))

xs, probs = [], []
for z in z_xs:
    bank.predict()
    bank.update(z)
    xs.append(bank.x[0])
    probs.append(bank.p[0])
```

(continues on next page)

(continued from previous page)

```
plt.subplot(121)
plt.plot(xs)
plt.subplot(122)
plt.plot(probs)
```

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.kalman.**MMAEFilterBank**(*filters*, *p*, *dim_x*, *H=None*)

Implements the fixed Multiple Model Adaptive Estimator (MMAE). This is a bank of independent Kalman filters. This estimator computes the likelihood that each filter is the correct one, and blends their state estimates weighted by their likelihood to produce the state estimate.

Parameters

filters [list of Kalman filters] List of Kalman filters.

p [list-like of floats] Initial probability that each filter is the correct one. In general you'd probably set each element to 1./len(p).

dim_x [float] number of random variables in the state X

H [Measurement matrix]

References

Zarchan and Musoff. "Fundamentals of Kalman filtering: A Practical Approach." AIAA, third edition.

Examples

..code: ca = make_ca_filter(dt, noise_factor=0.6) cv = make_ca_filter(dt, noise_factor=0.6) cv.F[:,2] = 0 # remove acceleration term cv.P[2,2] = 0 cv.Q[2,2] = 0

filters = [cv, ca] bank = MMAEFilterBank(filters, p=(0.5, 0.5), dim_x=3)

for z in zs: bank.predict() bank.update(z)

Also, see my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

x [numpy.array(dim_x, 1)] Current state estimate. Any call to update() or predict() updates this variable.

P [numpy.array(dim_x, dim_x)] Current state covariance matrix. Any call to update() or predict() updates this variable.

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

z [ndarray] Last measurement used in update(). Read only.

filters [list of Kalman filters] List of Kalman filters.

__init__ (*filters*, *p*, *dim_x*, *H=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

predict (*u=0*)
Predict next position using the Kalman filter state propagation equations for each filter in the bank.

Parameters

u [np.array] Optional control vector. If non-zero, it is multiplied by B to create the control input into the system.

update (*z*, *R=None*, *H=None*)
Add a new measurement (z) to the Kalman filter. If z is None, nothing is changed.

Parameters

z [np.array] measurement for this update.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

H [np.array, or None] Optionally provide H to override the measurement function for this one call, otherwise self.H will be used.

IMM Estimator

needs documentation....

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.kalman.**IMMEstimator** (*filters*, *mu*, *M*)
Implements an Interacting Multiple-Model (IMM) estimator.

Parameters

filters [(N,) array_like of KalmanFilter objects] List of N filters. filters[i] is the ith Kalman filter in the IMM estimator.

Each filter must have the same dimension for the state *x* and *P*, otherwise the states of each filter cannot be mixed with each other.

mu [(N,) array_like of float] mode probability: mu[i] is the probability that filter i is the correct one.

M [(N, N) ndarray of float] Markov chain transition matrix. M[i,j] is the probability of switching from filter j to filter i.

References

Bar-Shalom, Y., Li, X-R., and Kirubarajan, T. "Estimation with Application to Tracking and Navigation". Wiley-Interscience, 2001.

Crassidis, J and Junkins, J. "Optimal Estimation of Dynamic Systems". CRC Press, second edition. 2012.

Labbe, R. "Kalman and Bayesian Filters in Python". https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Examples

```
>>> import numpy as np
>>> from filterpy.common import kinematic_kf
>>> kf1 = kinematic_kf(2, 2)
>>> kf2 = kinematic_kf(2, 2)
>>> # do some settings of x, R, P etc. here, I'll just use the defaults
>>> kf2.Q *= 0   # no prediction error in second filter
>>>
>>> filters = [kf1, kf2]
>>> mu = [0.5, 0.5]  # each filter is equally likely at the start
>>> trans = np.array([[0.97, 0.03], [0.03, 0.97]])
>>> imm = IMMEstimator(filters, mu, trans)
>>>
>>> for i in range(100):
>>>     # make some noisy data
>>>     x = i + np.random.randn()*np.sqrt(kf1.R[0, 0])
>>>     y = i + np.random.randn()*np.sqrt(kf1.R[1, 1])
>>>     z = np.array([[x], [y]])
>>>
>>>     # perform predict/update cycle
>>>     imm.predict()
>>>     imm.update(z)
>>>     print(imm.x.T)
```

For a full explanation and more examples see my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

x [numpy.array(dim_x, 1)] Current state estimate. Any call to update() or predict() updates this variable.

P [numpy.array(dim_x, dim_x)] Current state covariance matrix. Any call to update() or predict() updates this variable.

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

N [int] number of filters in the filter bank

mu [(N,) ndarray of float] mode probability: mu[i] is the probability that filter i is the correct one.

M [(N, N) ndarray of float] Markov chain transition matrix. M[i,j] is the probability of switching from filter j to filter i.

cbar [(N,) ndarray of float] Total probability, after interaction, that the target is in state j. We use it as the # normalization constant.

likelihood: (N,) ndarray of float Likelihood of each individual filter's last measurement.

omega [(N, N) ndarray of float] Mixing probabilitity - omega[i, j] is the probabilility of mixing the state of filter i into filter j. Perhaps more understandably, it weights the states of each filter by:

x_j = sum(omega[i,j] * x_i)

with a similar weighting for P_j

__init__ (*filters*, *mu*, *M*)
x.__init__(...) initializes x; see help(type(x)) for signature

update(*z*)
Add a new measurement (z) to the Kalman filter. If z is None, nothing is changed.

Parameters

z [np.array] measurement for this update.

predict(*u=None*)
Predict next state (prior) using the IMM state propagation equations.

Parameters

u [np.array, optional] Control vector. If not *None*, it is multiplied by B to create the control input into the system.

5.1.2 Extended Kalman Filter

ExtendedKalmanFilter

Introduction and Overview

Implements a extended Kalman filter. For now the best documentation is my free book Kalman and Bayesian Filters in Python[1]

The test files in this directory also give you a basic idea of use, albeit without much description.

In brief, you will first construct this object, specifying the size of the state vector with *dim_x* and the size of the measurement vector that you will be using with *dim_z*. These are mostly used to perform size checks when you assign values to the various matrices. For example, if you specified *dim_z=2* and then try to assign a 3x3 matrix to R (the measurement noise matrix you will get an assert exception because R should be 2x2. (If for whatever reason you need to alter the size of things midstream just use the underscore version of the matrices to assign directly: your_filter._R = a_3x3_matrix.)

[1] Labbe, Roger. "Kalman and Bayesian Filters in Python".

After construction the filter will have default matrices created for you, but you must specify the values for each. It's usually easiest to just overwrite them rather than assign to each element yourself. This will be clearer in the example below. All are of type numpy.array.

References

github repo: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

read online: http://nbviewer.ipython.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

PDF version (often lags the two sources above) https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/Kalman_and_Bayesian_Filters_in_Python.pdf

---

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.kalman.**ExtendedKalmanFilter**(*dim_x*, *dim_z*, *dim_u=0*)

Implements an extended Kalman filter (EKF). You are responsible for setting the various state variables to reasonable values; the defaults will not give you a functional filter.

You will have to set the following attributes after constructing this object for the filter to perform properly. Please note that there are various checks in place to ensure that you have made everything the 'correct' size. However, it is possible to provide incorrectly sized arrays such that the linear algebra can not perform an operation. It can also fail silently - you can end up with matrices of a size that allows the linear algebra to work, but are the wrong shape for the problem you are trying to solve.

Parameters

dim_x [int] Number of state variables for the Kalman filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

This is used to set the default size of P, Q, and u

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

Examples

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

x [numpy.array(dim_x, 1)] State estimate vector

P [numpy.array(dim_x, dim_x)] Covariance matrix

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

R [numpy.array(dim_z, dim_z)] Measurement noise matrix

Q [numpy.array(dim_x, dim_x)] Process noise matrix

F [numpy.array()] State Transition matrix

H [numpy.array(dim_x, dim_x)] Measurement function

y [numpy.array] Residual of the update step. Read only.

K [numpy.array(dim_x, dim_z)] Kalman gain of the update step. Read only.

S [numpy.array] Systen uncertaintly projected to measurement space. Read only.

z [ndarray] Last measurement used in update(). Read only.

log_likelihood [float] log-likelihood of the last measurement.

likelihood [float] Computed from the log-likelihood.

mahalanobis [float] Mahalanobis distance of innovation.

__init__(*dim_x*, *dim_z*, *dim_u=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

predict_update(*z*, *HJacobian*, *Hx*, *args=()*, *hx_args=()*, *u=0*)
Performs the predict/update innovation of the extended Kalman filter.

Parameters

z [np.array] measurement for this step. If *None*, only predict step is perfomed.

HJacobian [function] function which computes the Jacobian of the H matrix (measurement function). Takes state variable (self.x) as input, along with the optional arguments in args, and returns H.

Hx [function] function which takes as input the state variable (self.x) along with the optional arguments in hx_args, and returns the measurement that would correspond to that state.

args [tuple, optional, default (,)] arguments to be passed into HJacobian after the required state variable.

hx_args [tuple, optional, default (,)] arguments to be passed into Hx after the required state variable.

u [np.array or scalar] optional control vector input to the filter.

update(*z*, *HJacobian*, *Hx*, *R=None*, *args=()*, *hx_args=()*, *residual=<Mock name='mock.subtract' id='140694326081744'>*)
Performs the update innovation of the extended Kalman filter.

Parameters

z [np.array] measurement for this step. If *None*, posterior is not computed

HJacobian [function] function which computes the Jacobian of the H matrix (measurement function). Takes state variable (self.x) as input, returns H.

Hx [function] function which takes as input the state variable (self.x) along with the optional arguments in hx_args, and returns the measurement that would correspond to that state.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

args [tuple, optional, default (,)] arguments to be passed into HJacobian after the required state variable. for robot localization you might need to pass in information about the map and time of day, so you might have *args=(map_data, time)*, where the signature of HCacobian will be *def HJacobian(x, map, t)*

hx_args [tuple, optional, default (,)] arguments to be passed into Hx function after the required state variable.

residual [function (z, z2), optional] Optional function that computes the residual (difference) between the two measurement vectors. If you do not provide this, then the built in minus operator will be used. You will normally want to use the built in unless your residual computation is nonlinear (for example, if they are angles)

predict_x(*u=0*)
Predicts the next state of X. If you need to compute the next state yourself, override this function. You would need to do this, for example, if the usual Taylor expansion to generate F is not providing accurate results for you.

predict(*u=0*)
Predict next state (prior) using the Kalman filter state propagation equations.

Parameters

u [np.array] Optional control vector. If non-zero, it is multiplied by B to create the control input into the system.

log_likelihood
log-likelihood of the last measurement.

likelihood
Computed from the log-likelihood. The log-likelihood can be very small, meaning a large negative value such as -28000. Taking the exp() of that results in 0.0, which can break typical algorithms which multiply by this value, so by default we always return a number >= sys.float_info.min.

mahalanobis
Mahalanobis distance of innovation. E.g. 3 means measurement was 3 standard deviations away from the predicted value.

Returns

mahalanobis [float]

5.1.3 Unscented Kalman Filter

These modules are used to implement the Unscented Kalman filter.

UnscentedKalmanFilter

Introduction and Overview

This implements the unscented Kalman filter.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

---

class filterpy.kalman.**UnscentedKalmanFilter**(*dim_x*, *dim_z*, *dt*, *hx*, *fx*, *points*, *sqrt_fn=None*, *x_mean_fn=None*, *z_mean_fn=None*, *residual_x=None*, *residual_z=None*)

Implements the Scaled Unscented Kalman filter (UKF) as defined by Simon Julier in [1], using the formulation provided by Wan and Merle in [2]. This filter scales the sigma points to avoid strong nonlinearities.

Parameters

dim_x [int] Number of state variables for the filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

This is for convience, so everything is sized correctly on creation. If you are using multiple sensors the size of *z* can change based on the sensor. Just provide the appropriate hx function

dt [float] Time between steps in seconds.

hx [function(x)] Measurement function. Converts state vector x into a measurement vector of shape (dim_z).

fx [function(x,dt)] function that returns the state x transformed by the state transistion function. dt is the time step in seconds.

points [class] Class which computes the sigma points and weights for a UKF algorithm. You can vary the UKF implementation by changing this class. For example, MerweScaledSigmaPoints implements the alpha, beta, kappa parameterization of Van der Merwe, and JulierSigmaPoints implements Julier's original kappa parameterization. See either of those for the required signature of this class if you want to implement your own.

sqrt_fn [callable(ndarray), default=None (implies scipy.linalg.cholesky)] Defines how we compute the square root of a matrix, which has no unique answer. Cholesky is the default choice due to its speed. Typically your alternative choice will be scipy.linalg.sqrtm. Different choices affect how the sigma points are arranged relative to the eigenvectors of the covariance matrix. Usually this will not matter to you; if so the default cholesky() yields maximal performance. As of van der Merwe's dissertation of 2004 [6] this was not a well reseached area so I have no advice to give you.

If your method returns a triangular matrix it must be upper triangular. Do not use numpy.linalg.cholesky - for historical reasons it returns a lower triangular matrix. The SciPy version does the right thing as far as this class is concerned.

x_mean_fn [callable (sigma_points, weights), optional] Function that computes the mean of the provided sigma points and weights. Use this if your state variable contains nonlinear values such as angles which cannot be summed.

```
def state_mean(sigmas, Wm):
    x = np.zeros(3)
    sum_sin, sum_cos = 0., 0.

    for i in range(len(sigmas)):
        s = sigmas[i]
        x[0] += s[0] * Wm[i]
        x[1] += s[1] * Wm[i]
        sum_sin += sin(s[2])*Wm[i]
```

(continues on next page)

(continued from previous page)

```
        sum_cos += cos(s[2])*Wm[i]
    x[2] = atan2(sum_sin, sum_cos)
    return x
```

z_mean_fn [callable (sigma_points, weights), optional] Same as x_mean_fn, except it is called for sigma points which form the measurements after being passed through hx().

residual_x [callable (x, y), optional]

residual_z [callable (x, y), optional] Function that computes the residual (difference) between x and y. You will have to supply this if your state variable cannot support subtraction, such as angles (359-1 degreees is 2, not 358). x and y are state vectors, not scalars. One is for the state variable, the other is for the measurement state.

```
def residual(a, b):
    y = a[0] - b[0]
    if y > np.pi:
        y -= 2*np.pi
    if y < -np.pi:
        y = 2*np.pi
    return y
```

References

[1], *[2]*, *[3]*, *[4]*, *[5]*, *[6]*

Examples

Simple example of a linear order 1 kinematic filter in 2D. There is no need to use a UKF for this example, but it is easy to read.

```
>>> def fx(x, dt):
>>>     # state transition function - predict next state based
>>>     # on constant velocity model x = vt + x_0
>>>     F = np.array([[1, dt, 0, 0],
>>>                   [0, 1, 0, 0],
>>>                   [0, 0, 1, dt],
>>>                   [0, 0, 0, 1]], dtype=float)
>>>     return np.dot(F, x)
>>>
>>> def hx(x):
>>>    # measurement function - convert state into a measurement
>>>    # where measurements are [x_pos, y_pos]
>>>    return np.array([x[0], x[2]])
>>>
>>> dt = 0.1
>>> # create sigma points to use in the filter. This is standard for Gaussian␣
↪processes
>>> points = MerweScaledSigmaPoints(4, alpha=.1, beta=2., kappa=-1)
>>>
>>> kf = UnscentedKalmanFilter(dim_x=4, dim_z=2, dt=dt, fx=fx, hx=hx,␣
↪points=points)
>>> kf.x = np.array([-1., 1., -1., 1]) # initial state
>>> kf.P *= 0.2 # initial uncertainty
```

(continues on next page)

(continued from previous page)

```
>>> z_std = 0.1
>>> kf.R = np.diag([z_std**2, z_std**2]) # 1 standard
>>> kf.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.01**2, block_size=2)
>>>
>>> zs = [[i+randn()*z_std, i+randn()*z_std] for i in range(50)] # measurements
>>> for z in zs:
>>>     kf.predict()
>>>     kf.update(z)
>>>     print(kf.x, 'log-likelihood', kf.log_likelihood)
```

For in depth explanations see my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Also see the filterpy/kalman/tests subdirectory for test code that may be illuminating.

Attributes

x [numpy.array(dim_x)] state estimate vector

P [numpy.array(dim_x, dim_x)] covariance estimate matrix

x_prior [numpy.array(dim_x)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

z [ndarray] Last measurement used in update(). Read only.

R [numpy.array(dim_z, dim_z)] measurement noise matrix

Q [numpy.array(dim_x, dim_x)] process noise matrix

K [numpy.array] Kalman gain

y [numpy.array] innovation residual

log_likelihood [scalar] log-likelihood of the last measurement.

likelihood [float] Computed from the log-likelihood.

mahalanobis [float] “

inv [function, default numpy.linalg.inv] If you prefer another inverse function, such as the Moore-Penrose pseudo inverse, set it to that instead:

```
kf.inv = np.linalg.pinv
```

__init__(*dim_x*, *dim_z*, *dt*, *hx*, *fx*, *points*, *sqrt_fn=None*, *x_mean_fn=None*, *z_mean_fn=None*, *residual_x=None*, *residual_z=None*)

Create a Kalman filter. You are responsible for setting the various state variables to reasonable values; the defaults below will not give you a functional filter.

predict(*dt=None*, *UT=None*, *fx=None*, ***fx_args*)

Performs the predict step of the UKF. On return, self.x and self.P contain the predicted state (x) and covariance (P). ‘

Important: this MUST be called before update() is called for the first time.

Parameters

dt [double, optional] If specified, the time step to be used for this prediction. self._dt is used if this is not provided.

fx [callable f(x, **fx_args), optional] State transition function. If not provided, the default function passed in during construction will be used.

UT [function(sigmas, Wm, Wc, noise_cov), optional] Optional function to compute the unscented transform for the sigma points passed through hx. Typically the default function will work - you can use x_mean_fn and z_mean_fn to alter the behavior of the unscented transform.

****fx_args** [keyword arguments] optional keyword arguments to be passed into f(x).

update(*z*, *R=None*, *UT=None*, *hx=None*, ***hx_args*)

Update the UKF with the given measurements. On return, self.x and self.P contain the new mean and covariance of the filter.

Parameters

z [numpy.array of shape (dim_z)] measurement vector

R [numpy.array((dim_z, dim_z)), optional] Measurement noise. If provided, overrides self.R for this function call.

UT [function(sigmas, Wm, Wc, noise_cov), optional] Optional function to compute the unscented transform for the sigma points passed through hx. Typically the default function will work - you can use x_mean_fn and z_mean_fn to alter the behavior of the unscented transform.

****hx_args** [keyword argument] arguments to be passed into h(x) after x -> h(x, **hx_args)

cross_variance(*x*, *z*, *sigmas_f*, *sigmas_h*)

Compute cross variance of the state *x* and measurement *z*.

compute_process_sigmas(*dt*, *fx=None*, ***fx_args*)

computes the values of sigmas_f. Normally a user would not call this, but it is useful if you need to call update more than once between calls to predict (to update for multiple simultaneous measurements), so the sigmas correctly reflect the updated state x, P.

batch_filter(*zs*, *Rs=None*, *dts=None*, *UT=None*, *saver=None*)

Performs the UKF filter over the list of measurement in *zs*.

Parameters

zs [list-like] list of measurements at each time step *self._dt* Missing measurements must be represented by 'None'.

Rs [None, np.array or list-like, default=None] optional list of values to use for the measurement error covariance R.

If Rs is None then self.R is used for all epochs.

If it is a list of matrices or a 3D array where len(Rs) == len(zs), then it is treated as a list of R values, one per epoch. This allows you to have varying R per epoch.

dts [None, scalar or list-like, default=None] optional value or list of delta time to be passed into predict.

If dtss is None then self.dt is used for all epochs.

If it is a list where len(dts) == len(zs), then it is treated as a list of dt values, one per epoch. This allows you to have varying epoch durations.

UT [function(sigmas, Wm, Wc, noise_cov), optional] Optional function to compute the unscented transform for the sigma points passed through hx. Typically the default function will work - you can use x_mean_fn and z_mean_fn to alter the behavior of the unscented transform.

saver [filterpy.common.Saver, optional] filterpy.common.Saver object. If provided, saver.save() will be called after every epoch

Returns

means: ndarray((n,dim_x,1)) array of the state for each time step after the update. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance: ndarray((n,dim_x,dim_x)) array of the covariances for each time step after the update. In other words *covariance[k,:,:]* is the covariance at step *k*.

Examples

```
# this example demonstrates tracking a measurement where the time
# between measurement varies, as stored in dts The output is then smoothed
# with an RTS smoother.

zs = [t + random.randn()*4 for t in range (40)]

(mu, cov, _, _) = ukf.batch_filter(zs, dts=dts)
(xs, Ps, Ks) = ukf.rts_smoother(mu, cov)
```

rts_smoother(*Xs*, *Ps*, *Qs=None*, *dts=None*, *UT=None*)

Runs the Rauch-Tung-Striebal Kalman smoother on a set of means and covariances computed by the UKF. The usual input would come from the output of *batch_filter()*.

Parameters

Xs [numpy.array] array of the means (state variable x) of the output of a Kalman filter.

Ps [numpy.array] array of the covariances of the output of a kalman filter.

Qs: list-like collection of numpy.array, optional Process noise of the Kalman filter at each time step. Optional, if not provided the filter's self.Q will be used

dt [optional, float or array-like of float] If provided, specifies the time step of each step of the filter. If float, then the same time step is used for all steps. If an array, then each element k contains the time at step k. Units are seconds.

UT [function(sigmas, Wm, Wc, noise_cov), optional] Optional function to compute the unscented transform for the sigma points passed through hx. Typically the default function will work - you can use x_mean_fn and z_mean_fn to alter the behavior of the unscented transform.

Returns

x [numpy.ndarray] smoothed means

P [numpy.ndarray] smoothed state covariances

K [numpy.ndarray] smoother gain at each step

Examples

```
zs = [t + random.randn()*4 for t in range (40)]

(mu, cov, _, _) = kalman.batch_filter(zs)
(x, P, K) = rts_smoother(mu, cov, fk.F, fk.Q)
```

log_likelihood
log-likelihood of the last measurement.

likelihood
Computed from the log-likelihood. The log-likelihood can be very small, meaning a large negative value such as -28000. Taking the exp() of that results in 0.0, which can break typical algorithms which multiply by this value, so by default we always return a number >= sys.float_info.min.

mahalanobis
" Mahalanobis distance of measurement. E.g. 3 means measurement was 3 standard deviations away from the predicted value.

Returns

mahalanobis [float]

---

class filterpy.kalman.**MerweScaledSigmaPoints**(*n*, *alpha*, *beta*, *kappa*, *sqrt_method=None*, *subtract=None*)

Generates sigma points and weights according to Van der Merwe's 2004 dissertation[1] for the UnscentedKalmanFilter class.. It parametizes the sigma points using alpha, beta, kappa terms, and is the version seen in most publications.

Unless you know better, this should be your default choice.

Parameters

n [int] Dimensionality of the state. 2n+1 weights will be generated.

alpha [float] Determins the spread of the sigma points around the mean. Usually a small positive value (1e-3) according to [3].

beta [float] Incorporates prior knowledge of the distribution of the mean. For Gaussian x beta=2 is optimal, according to [3].

kappa [float, default=0.0] Secondary scaling parameter usually set to 0 according to [4], or to 3-n according to [5].

sqrt_method [function(ndarray), default=scipy.linalg.cholesky] Defines how we compute the square root of a matrix, which has no unique answer. Cholesky is the default choice due to its speed. Typically your alternative choice will be scipy.linalg.sqrtm. Different choices affect how the sigma points are arranged relative to the eigenvectors of the covariance matrix. Usually this will not matter to you; if so the default cholesky() yields maximal performance. As of van der Merwe's dissertation of 2004 [6] this was not a well reseached area so I have no advice to give you.

If your method returns a triangular matrix it must be upper triangular. Do not use numpy.linalg.cholesky - for historical reasons it returns a lower triangular matrix. The SciPy version does the right thing.

subtract [callable (x, y), optional] Function that computes the difference between x and y. You will have to supply this if your state variable cannot support subtraction, such as angles (359-1 degreees is 2, not 358). x and y are state vectors, not scalars.

References

[1]

Examples

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

Wm [np.array] weight for each sigma point for the mean

Wc [np.array] weight for each sigma point for the covariance

__init__(*n*, *alpha*, *beta*, *kappa*, *sqrt_method=None*, *subtract=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

num_sigmas()
Number of sigma points for each variable in the state x

sigma_points(*x*, *P*)
Computes the sigma points for an unscented Kalman filter given the mean (x) and covariance(P) of the filter. Returns tuple of the sigma points and weights.

Works with both scalar and array inputs: sigma_points (5, 9, 2) # mean 5, covariance 9 sigma_points ([5, 2], 9*eye(2), 2) # means 5 and 2, covariance 9I

Parameters

x [An array-like object of the means of length n] Can be a scalar if 1D. examples: 1, [1,2], np.array([1,2])

P [scalar, or np.array] Covariance of the filter. If scalar, is treated as eye(n)*P.

Returns

sigmas [np.array, of size (n, 2n+1)] Two dimensional array of sigma points. Each column contains all of the sigmas for one dimension in the problem space.

Ordered by Xi_0, Xi_{1..n}, Xi_{n+1..2n}

---

class filterpy.kalman.**JulierSigmaPoints**(*n*, *kappa=0.0*, *sqrt_method=None*, *subtract=None*)
Generates sigma points and weights according to Simon J. Julier and Jeffery K. Uhlmann's original paper[1]. It parametizes the sigma points using kappa.

Parameters

n [int] Dimensionality of the state. 2n+1 weights will be generated.

kappa [float, default=0.] Scaling factor that can reduce high order errors. kappa=0 gives the standard unscented filter. According to [Julier], if you set kappa to 3-dim_x for a Gaussian x you will minimize the fourth order errors in x and P.

sqrt_method [function(ndarray), default=scipy.linalg.cholesky] Defines how we compute the square root of a matrix, which has no unique answer. Cholesky is the default choice due to its speed. Typically your alternative choice will be scipy.linalg.sqrtm. Different choices affect how the sigma points are arranged relative to the eigenvectors of the covariance matrix. Usually this will not matter to you; if so the default cholesky() yields maximal performance.

As of van der Merwe's dissertation of 2004 [6] this was not a well reseached area so I have no advice to give you.

If your method returns a triangular matrix it must be upper triangular. Do not use numpy.linalg.cholesky - for historical reasons it returns a lower triangular matrix. The SciPy version does the right thing.

subtract [callable (x, y), optional] Function that computes the difference between x and y. You will have to supply this if your state variable cannot support subtraction, such as angles (359-1 degreees is 2, not 358). x and y

References

[1]

Attributes

Wm [np.array] weight for each sigma point for the mean

Wc [np.array] weight for each sigma point for the covariance

__init__(*n*, *kappa=0.0*, *sqrt_method=None*, *subtract=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

num_sigmas()
Number of sigma points for each variable in the state x

sigma_points(*x*, *P*)
Computes the sigma points for an unscented Kalman filter given the mean (x) and covariance(P) of the filter. kappa is an arbitrary constant. Returns sigma points.

Works with both scalar and array inputs: sigma_points (5, 9, 2) # mean 5, covariance 9 sigma_points ([5, 2], 9*eye(2), 2) # means 5 and 2, covariance 9I

Parameters

x [array-like object of the means of length n] Can be a scalar if 1D. examples: 1, [1,2], np.array([1,2])

P [scalar, or np.array] Covariance of the filter. If scalar, is treated as eye(n)*P.

kappa [float] Scaling factor.

Returns

sigmas [np.array, of size (n, 2n+1)] 2D array of sigma points χ. Each column contains all of the sigmas for one dimension in the problem space. They are ordered as:

$$\chi[0] = x \tag{5.1}$$
$$\chi[1..n] = x + [\sqrt{(n+\kappa)P}]_k \tag{5.2}$$
$$\chi[n+1..2n] = x - [\sqrt{(n+\kappa)P}]_k \tag{5.3}$$

---

class filterpy.kalman.**SimplexSigmaPoints**(*n*, *alpha=1*, *sqrt_method=None*, *subtract=None*)
Generates sigma points and weights according to the simplex method presented in [1].

Parameters

n [int] Dimensionality of the state. n+1 weights will be generated.

sqrt_method [function(ndarray), default=scipy.linalg.cholesky] Defines how we compute the square root of a matrix, which has no unique answer. Cholesky is the default choice due to its speed. Typically your alternative choice will be scipy.linalg.sqrtm

If your method returns a triangular matrix it must be upper triangular. Do not use numpy.linalg.cholesky - for historical reasons it returns a lower triangular matrix. The SciPy version does the right thing.

subtract [callable (x, y), optional] Function that computes the difference between x and y. You will have to supply this if your state variable cannot support subtraction, such as angles (359-1 degreees is 2, not 358). x and y are state vectors, not scalars.

References

[1]

Attributes

Wm [np.array] weight for each sigma point for the mean

Wc [np.array] weight for each sigma point for the covariance

__init__(*n*, *alpha=1*, *sqrt_method=None*, *subtract=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

num_sigmas()
Number of sigma points for each variable in the state x

sigma_points(*x*, *P*)
Computes the implex sigma points for an unscented Kalman filter given the mean (x) and covariance(P) of the filter. Returns tuple of the sigma points and weights.

Works with both scalar and array inputs: sigma_points (5, 9, 2) # mean 5, covariance 9 sigma_points ([5, 2], 9*eye(2), 2) # means 5 and 2, covariance 9I

Parameters

x [An array-like object of the means of length n] Can be a scalar if 1D. examples: 1, [1,2], np.array([1,2])

P [scalar, or np.array] Covariance of the filter. If scalar, is treated as eye(n)*P.

Returns

sigmas [np.array, of size (n, n+1)] Two dimensional array of sigma points. Each column contains all of the sigmas for one dimension in the problem space.

Ordered by Xi_0, Xi_{1..n}

unscented_transform

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

filterpy.kalman.**unscented_transform**(*sigmas*, *Wm*, *Wc*, *noise_cov=None*, *mean_fn=None*, *residual_fn=None*)

Computes unscented transform of a set of sigma points and weights. returns the mean and covariance in a tuple.

This works in conjunction with the UnscentedKalmanFilter class.

Parameters

sigmas: ndarray, of size (n, 2n+1) 2D array of sigma points.

Wm [ndarray [# sigmas per dimension]] Weights for the mean.

Wc [ndarray [# sigmas per dimension]] Weights for the covariance.

noise_cov [ndarray, optional] noise matrix added to the final computed covariance matrix.

mean_fn [callable (sigma_points, weights), optional] Function that computes the mean of the provided sigma points and weights. Use this if your state variable contains nonlinear values such as angles which cannot be summed.

```
def state_mean(sigmas, Wm):
    x = np.zeros(3)
    sum_sin, sum_cos = 0., 0.

    for i in range(len(sigmas)):
        s = sigmas[i]
        x[0] += s[0] * Wm[i]
        x[1] += s[1] * Wm[i]
        sum_sin += sin(s[2])*Wm[i]
        sum_cos += cos(s[2])*Wm[i]
    x[2] = atan2(sum_sin, sum_cos)
    return x
```

residual_fn [callable (x, y), optional] Function that computes the residual (difference) between x and y. You will have to supply this if your state variable cannot support subtraction, such as angles (359-1 degreees is 2, not 358). x and y are state vectors, not scalars.

```
def residual(a, b):
    y = a[0] - b[0]
    y = y % (2 * np.pi)
    if y > np.pi:
        y -= 2*np.pi
    return y
```

Returns

x [ndarray [dimension]] Mean of the sigma points after passing through the transform.

P [ndarray] covariance of the sigma points after passing throgh the transform.

Examples

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

5.1.4 Ensemble Kalman Filter

EnsembleKalmanFilter

Introduction and Overview

This implements the Ensemble Kalman filter.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

---

class filterpy.kalman.**EnsembleKalmanFilter**(*x*, *P*, *dim_z*, *dt*, *N*, *hx*, *fx*)

This implements the ensemble Kalman filter (EnKF). The EnKF uses an ensemble of hundreds to thousands of state vectors that are randomly sampled around the estimate, and adds perturbations at each update and predict step. It is useful for extremely large systems such as found in hydrophysics. As such, this class is admittedly a toy as it is far too slow with large N.

There are many versions of this sort of this filter. This formulation is due to Crassidis and Junkins [1]. It works with both linear and nonlinear systems.

Parameters

x [np.array(dim_x)] state mean

P [np.array((dim_x, dim_x))] covariance of the state

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x,y), dim_z would be 2.

dt [float] time step in seconds

N [int] number of sigma points (ensembles). Must be greater than 1.

K [np.array] Kalman gain

hx [function hx(x)] Measurement function. May be linear or nonlinear - converts state x into a measurement. Return must be an np.array of the same dimensionality as the measurement vector.

fx [function fx(x, dt)] State transition function. May be linear or nonlinear. Projects state x into the next time period. Returns the projected state x.

References

- [1] John L Crassidis and John L. Junkins. "Optimal Estimation of Dynamic Systems. CRC Press, second edition. 2012. pp, 257-9.

Examples

```
def hx(x):
   return np.array([x[0]])

F = np.array([[1., 1.],
              [0., 1.]])
def fx(x, dt):
    return np.dot(F, x)

x = np.array([0., 1.])
P = np.eye(2) * 100.
dt = 0.1
f = EnKF(x=x, P=P, dim_z=1, dt=dt, N=8,
         hx=hx, fx=fx)

std_noise = 3.
f.R *= std_noise**2
f.Q = Q_discrete_white_noise(2, dt, .01)

while True:
    z = read_sensor()
    f.predict()
    f.update(np.asarray([z]))
```

See my book Kalman and Bayesian Filters in Python https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

Attributes

x [numpy.array(dim_x, 1)] State estimate

P [numpy.array(dim_x, dim_x)] State covariance matrix

x_prior [numpy.array(dim_x, 1)] Prior (predicted) state estimate. The *_prior and *_post attributes are for convienence; they store the prior and posterior of the current epoch. Read Only.

P_prior [numpy.array(dim_x, dim_x)] Prior (predicted) state covariance matrix. Read Only.

x_post [numpy.array(dim_x, 1)] Posterior (updated) state estimate. Read Only.

P_post [numpy.array(dim_x, dim_x)] Posterior (updated) state covariance matrix. Read Only.

z [numpy.array] Last measurement used in update(). Read only.

R [numpy.array(dim_z, dim_z)] Measurement noise matrix

Q [numpy.array(dim_x, dim_x)] Process noise matrix

fx [callable (x, dt)] State transition function

hx [callable (x)] Measurement function. Convert state x into a measurement

K [numpy.array(dim_x, dim_z)] Kalman gain of the update step. Read only.

inv [function, default numpy.linalg.inv] If you prefer another inverse function, such as the Moore-Penrose pseudo inverse, set it to that instead: kf.inv = np.linalg.pinv

__init__(*x*, *P*, *dim_z*, *dt*, *N*, *hx*, *fx*)
x.__init__(...) initializes x; see help(type(x)) for signature

initialize(*x*, *P*)
Initializes the filter with the specified mean and covariance. Only need to call this if you are using the filter to filter more than one set of data; this is called by __init__

Parameters

x [np.array(dim_z)] state mean

P [np.array((dim_x, dim_x))] covariance of the state

update(*z*, *R=None*)
Add a new measurement (z) to the kalman filter. If z is None, nothing is changed.

Parameters

z [np.array] measurement for this update.

R [np.array, scalar, or None] Optionally provide R to override the measurement noise for this one call, otherwise self.R will be used.

predict()
Predict next position.

5.2 filterpy.common

Contains various useful functions that support the filtering classes and functions. Most useful are functions to compute the process noise matrix Q. It also implements the Van Loan discretization of a linear differential equation.

5.2.1 common

A collection of functions used throughout FilterPy, and/or functions that you will find useful when you build your filters.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

---

filterpy.common.**Saver**(*kf*, *save_current=False*, *skip_private=False*, *skip_callable=False*, *ignore=()*)
Helper class to save the states of any filter object. Each time you call save() all of the attributes (state, covariances, etc) are appended to lists.

Generally you would do this once per epoch - predict/update.

Then, you can access any of the states by using the [] syntax or by using the . operator.

```
my_saver = Saver()
... do some filtering

x = my_saver['x']
x = my_save.x
```

Either returns a list of all of the state *x* values for the entire filtering process.

If you want to convert all saved lists into numpy arrays, call to_array().

Parameters

kf [object] any object with a __dict__ attribute, but intended to be one of the filtering classes

save_current [bool, default=True] save the current state of *kf* when the object is created;

skip_private: bool, default=False Control skipping any private attribute (anything starting with '_') Turning this on saves memory, but slows down execution a bit.

skip_callable: bool, default=False Control skipping any attribute which is a method. Turning this on saves memory, but slows down execution a bit.

ignore: (str,) tuple of strings list of keys to ignore.

Examples

```
kf = KalmanFilter(...whatever)
# initialize kf here

saver = Saver(kf) # save data for kf filter
for z in zs:
    kf.predict()
    kf.update(z)
    saver.save()

x = np.array(s.x) # get the kf.x state in an np.array
plt.plot(x[:, 0], x[:, 2])

# ... or ...
s.to_array()
plt.plot(s.x[:, 0], s.x[:, 2])
```

---

filterpy.common.**Q_discrete_white_noise**(*dim*, *dt=1.0*, *var=1.0*, *block_size=1*, *order_by_dim=True*)

Returns the Q matrix for the Discrete Constant White Noise Model. dim may be either 2, 3, or 4 dt is the time step, and sigma is the variance in the noise.

Q is computed as the G * G^T * variance, where G is the process noise per time step. In other words, G = [[.5dt^2][dt]]^T for the constant velocity model.

Parameters

dim [int (2, 3, or 4)] dimension for Q, where the final dimension is (dim x dim)

dt [float, default=1.0] time step in whatever units your filter is using for time. i.e. the amount of time between innovations

var [float, default=1.0] variance in the noise

block_size [int >= 1] If your state variable contains more than one dimension, such as a 3d constant velocity model [x x' y y' z z']^T, then Q must be a block diagonal matrix.

order_by_dim [bool, default=True] Defines ordering of variables in the state vector. *True* orders by keeping all derivatives of each dimensions)

[x x' x" y y' y"]

whereas *False* interleaves the dimensions

[x y z x' y' z' x" y" z"]

References

Bar-Shalom. "Estimation with Applications To Tracking and Navigation". John Wiley & Sons, 2001. Page 274.

Examples

```
>>> # constant velocity model in a 3D world with a 10 Hz update rate
>>> Q_discrete_white_noise(2, dt=0.1, var=1., block_size=3)
array([[0.000025, 0.0005  , 0.      , 0.      , 0.      , 0.      ],
       [0.0005  , 0.01    , 0.      , 0.      , 0.      , 0.      ],
       [0.      , 0.      , 0.000025, 0.0005  , 0.      , 0.      ],
       [0.      , 0.      , 0.0005  , 0.01    , 0.      , 0.      ],
       [0.      , 0.      , 0.      , 0.      , 0.000025, 0.0005  ],
       [0.      , 0.      , 0.      , 0.      , 0.0005  , 0.01    ]])
```

---

filterpy.common.**Q_continuous_white_noise**(*dim*, *dt=1.0*, *spectral_density=1.0*, *block_size=1*, *order_by_dim=True*)

Returns the Q matrix for the Discretized Continuous White Noise Model. dim may be either 2, 3, 4, dt is the time step, and sigma is the variance in the noise.

Parameters

dim [int (2 or 3 or 4)] dimension for Q, where the final dimension is (dim x dim) 2 is constant velocity, 3 is constant acceleration, 4 is constant jerk

dt [float, default=1.0] time step in whatever units your filter is using for time. i.e. the amount of time between innovations

spectral_density [float, default=1.0] spectral density for the continuous process

block_size [int >= 1] If your state variable contains more than one dimension, such as a 3d constant velocity model [x x' y y' z z']^T, then Q must be a block diagonal matrix.

order_by_dim [bool, default=True] Defines ordering of variables in the state vector. *True* orders by keeping all derivatives of each dimensions)

[x x' x" y y' y"']

whereas *False* interleaves the dimensions

[x y z x' y' z' x" y" z"']

Examples

```
>>> # constant velocity model in a 3D world with a 10 Hz update rate
>>> Q_continuous_white_noise(2, dt=0.1, block_size=3)
array([[0.00033333, 0.005     , 0.        , 0.        , 0.        , 0.        ],
       [0.005     , 0.1       , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.00033333, 0.005     , 0.        , 0.        ],
       [0.        , 0.        , 0.005     , 0.1       , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.00033333, 0.005     ],
       [0.        , 0.        , 0.        , 0.        , 0.005     , 0.1       ]])
```

---

filterpy.common.**van_loan_discretization**(*F*, *G*, *dt*)

Discretizes a linear differential equation which includes white noise according to the method of C. F. van Loan [1]. Given the continuous model

x' = Fx + Gu

where u is the unity white noise, we compute and return the sigma and Q_k that discretizes that equation.

References

[1] C. F. van Loan. "Computing Integrals Involving the Matrix Exponential." IEEE Trans. Automomatic Control, AC-23 (3): 395-404 (June 1978)

[2] Robert Grover Brown. "Introduction to Random Signals and Applied Kalman Filtering." Forth edition. John Wiley & Sons. p. 126-7. (2012)

Examples

Given y" + y = 2u(t), we create the continuous state model of

x' = [0 1] * x + [0]*u(t) [-1 0] [2]

and a time step of 0.1:

```
>>> F = np.array([[0,1],[-1,0]], dtype=float)
>>> G = np.array([[0.],[2.]])
>>> phi, Q = van_loan_discretization(F, G, 0.1)
```

```
>>> phi
array([[ 0.99500417,  0.09983342],
       [-0.09983342,  0.99500417]])
```

```
>>> Q
array([[ 0.00133067,  0.01993342],
       [ 0.01993342,  0.39866933]])
```

(example taken from Brown[2])

---

filterpy.common.**linear_ode_discretation**(*F*, *L=None*, *Q=None*, *dt=1.0*)

---

filterpy.common.**kinematic_kf**(*dim*, *order*, *dt=1.0*, *dim_z=1*, *order_by_dim=True*)

Returns a KalmanFilter using newtonian kinematics of arbitrary order for any number of dimensions. For example, a constant velocity filter in 3D space would have order 1 dimension 3.

Parameters

dim [int, >= 1] number of dimensions (2D space would be dim=2)

order [int, >= 0] order of the filter. 2 would be a const acceleration model with a stat

dim_z [int, default 1] size of z vector *per* dimension *dim*. Normally should be 1

dt [float, default 1.0] Time step. Used to create the state transition matrix

order_by_dim [bool, default=True] Defines ordering of variables in the state vector. *True* orders by keeping all derivatives of each dimensions)

[x x' x" y y' y'']

whereas *False* interleaves the dimensions

[x y z x' y' z' x" y" z'']

Examples

A constant velocity filter in 3D space with delta time = .2 seconds would be created with

```
>>> kf = kinematic_kf(dim=3, order=1, dt=.2)
>>> kf.F
>>> array([[1. , 0.2, 0. , 0. , 0. , 0. ],
           [0. , 1. , 0. , 0. , 0. , 0. ],
           [0. , 0. , 1. , 0.2, 0. , 0. ],
           [0. , 0. , 0. , 1. , 0. , 0. ],
           [0. , 0. , 0. , 0. , 1. , 0.2],
           [0. , 0. , 0. , 0. , 0. , 1. ]])
```

which will set the state *x* to be interpreted as

[x, x', y, y', z, z'].T

If you set *order_by_dim* to False, then *x* is ordered as

[x y z x' y' z'].T

As another example, a 2D constant jerk is created with

>> kinematic_kf(2, 3)

Assumes that the measurement z is position in each dimension. If this is not true you will have to alter the H matrix by hand.

P, Q, R are all set to the Identity matrix.

H is assigned assuming the measurement is position, one per dimension *dim*.

```
>>> kf = kinematic_kf(2, 1, dt=3.0)
>>> kf.F
array([[1., 3., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 3.],
       [0., 0., 0., 1.]])
```

---

filterpy.common.**kinematic_state_transition**(*order*, *dt*)

create a state transition matrix of a given order for a given time step *dt*.

---

filterpy.common.**runge_kutta4**(*y*, *x*, *dx*, *f*)

computes 4th order Runge-Kutta for dy/dx.

Parameters

y [scalar] Initial/current value for y

x [scalar] Initial/current value for x

dx [scalar] difference in x (e.g. the time step)

f [ufunc(y,x)] Callable function (y, x) that you supply to compute dy/dx for the specified values.

---

filterpy.common.**inv_diagonal**(*S*)

Computes the inverse of a diagonal NxN np.array S. In general this will be much faster than calling np.linalg.inv().

However, does NOT check if the off diagonal elements are non-zero. So long as S is truly diagonal, the output is identical to np.linalg.inv().

Parameters

S [np.array] diagonal NxN array to take inverse of

Returns

S_inv [np.array] inverse of S

Examples

This is meant to be used as a replacement inverse function for the KalmanFilter class when you know the system covariance S is diagonal. It just makes the filter run faster, there is

```
>>> kf = KalmanFilter(dim_x=3, dim_z=1)
>>> kf.inv = inv_diagonal  # S is 1x1, so safely diagonal
```

5.3 filterpy.stats

Contains statistical functions useful for Kalman filtering such as multivariate Gaussian multiplication, computing the log-likelihood, NESS, and mahalanobis distance, along with plotting routines to plot multivariate Gaussians CDFs, PDFs, and covariance ellipses.

5.3.1 stats

A collection of functions used to compute and plot statistics relevant to Bayesian filters.

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

---

filterpy.stats.**gaussian**(*x*, *mean*, *var*, *normed=True*)

returns normal distribution (pdf) for x given a Gaussian with the specified mean and variance. All must be scalars.

gaussian (1,2,3) is equivalent to scipy.stats.norm(2,math.sqrt(3)).pdf(1) It is quite a bit faster albeit much less flexible than the latter.

Parameters

x [scalar or array-like] The value for which we compute the probability

mean [scalar] Mean of the Gaussian

var [scalar] Variance of the Gaussian

norm [bool, default True] Normalize the output if the input is an array of values.

Returns

probability [float] probability of x for the Gaussian (mean, var). E.g. 0.101 denotes 10.1%.

Examples

```
>>> gaussian(8, 1, 2)
1.3498566943461957e-06
```

```
>>> gaussian([8, 7, 9], 1, 2)
array([1.34985669e-06, 3.48132630e-05, 3.17455867e-08])
```

`filterpy.stats.`**`mul`**(*mean1*, *var1*, *mean2*, *var2*)

Multiply Gaussian (mean1, var1) with (mean2, var2) and return the results as a tuple (mean, var).

Strictly speaking the product of two Gaussian PDFs is a Gaussian function, not Gaussian PDF. It is, however, proportional to a Gaussian PDF, so it is safe to treat the output as a PDF for any filter using Bayes equation, which normalizes the result anyway.

Parameters

mean1 [scalar] mean of first Gaussian

var1 [scalar] variance of first Gaussian

mean2 [scalar] mean of second Gaussian

var2 [scalar] variance of second Gaussian

Returns

mean [scalar] mean of product

var [scalar] variance of product

References

Bromily. "Products and Convolutions of Gaussian Probability Functions", Tina Memo No. 2003-003. http://www.tina-vision.net/docs/memos/2003-003.pdf

Examples

```
>>> mul(1, 2, 3, 4)
(1.6666666666666667, 1.3333333333333333)
```

---

filterpy.stats.**add**(*mean1*, *var1*, *mean2*, *var2*)

Add the Gaussians (mean1, var1) with (mean2, var2) and return the results as a tuple (mean,var).

var1 and var2 are variances - sigma squared in the usual parlance.

---

filterpy.stats.**log_likelihood**(*z*, *x*, *P*, *H*, *R*)

Returns log-likelihood of the measurement z given the Gaussian posterior (x, P) using measurement function H and measurement covariance error R

---

filterpy.stats.**likelihood**(*z*, *x*, *P*, *H*, *R*)

Returns likelihood of the measurement z given the Gaussian posterior (x, P) using measurement function H and measurement covariance error R

---

filterpy.stats.**logpdf**(*x*, *mean=None*, *cov=1*, *allow_singular=True*)

Computes the log of the probability density function of the normal N(mean, cov) for the data x. The normal may be univariate or multivariate.

Wrapper for older versions of scipy.multivariate_normal.logpdf which don't support support the allow_singular keyword prior to verion 0.15.0.

If it is not supported, and cov is singular or not PSD you may get an exception.

x and *mean* may be column vectors, row vectors, or lists.

---

filterpy.stats.**multivariate_gaussian**(*x*, *mu*, *cov*)

This is designed to replace scipy.stats.multivariate_normal which is not available before version 0.14. You may either pass in a multivariate set of data:

```
multivariate_gaussian (array([1,1]), array([3,4]), eye(2)*1.4)
multivariate_gaussian (array([1,1,1]), array([3,4,5]), 1.4)
```

or unidimensional data:

```
multivariate_gaussian(1, 3, 1.4)
```

In the multivariate case if cov is a scalar it is interpreted as eye(n)*cov

The function gaussian() implements the 1D (univariate)case, and is much faster than this function.

equivalent calls:

```
multivariate_gaussian(1, 2, 3)
scipy.stats.multivariate_normal(2,3).pdf(1)
```

Parameters

x [float, or np.array-like] Value to compute the probability for. May be a scalar if univariate, or any type that can be converted to an np.array (list, tuple, etc). np.array is best for speed.

mu [float, or np.array-like] mean for the Gaussian . May be a scalar if univariate, or any type that can be converted to an np.array (list, tuple, etc).np.array is best for speed.

cov [float, or np.array-like] Covariance for the Gaussian . May be a scalar if univariate, or any type that can be converted to an np.array (list, tuple, etc).np.array is best for speed.

Returns

probability [float] probability for x for the Gaussian (mu,cov)

---

filterpy.stats.**multivariate_multiply**(*m1*, *c1*, *m2*, *c2*)

Multiplies the two multivariate Gaussians together and returns the results as the tuple (mean, covariance).

Parameters

m1 [array-like] Mean of first Gaussian. Must be convertable to an 1D array via numpy.asarray(), For example 6, [6], [6, 5], np.array([3, 4, 5, 6]) are all valid.

c1 [matrix-like]

Covariance of first Gaussian. Must be convertable to an 2D array via numpy.asarray().

m2 [array-like] Mean of second Gaussian. Must be convertable to an 1D array via numpy.asarray(), For example 6, [6], [6, 5], np.array([3, 4, 5, 6]) are all valid.

c2 [matrix-like] Covariance of second Gaussian. Must be convertable to an 2D array via numpy.asarray().

Returns

m [ndarray] mean of the result

c [ndarray] covariance of the result

Examples

```
m, c = multivariate_multiply([7.0, 2], [[1.0, 2.0], [2.0, 1.0]],
                             [3.2, 0], [[8.0, 1.1], [1.1,8.0]])
```

---

filterpy.stats.**plot_gaussian_cdf**(*mean=0.0*, *variance=1.0*, *ax=None*, *xlim=None*, *ylim=(0.0, 1.0)*, *xlabel=None*, *ylabel=None*, *label=None*)

Plots a normal distribution CDF with the given mean and variance. x-axis contains the mean, the y-axis shows the cumulative probability.

Parameters

mean [scalar, default 0.] mean for the normal distribution.

variance [scalar, default 0.] variance for the normal distribution.

ax [matplotlib axes object, optional] If provided, the axes to draw on, otherwise plt.gca() is used.

xlim, ylim: (float,float), optional specify the limits for the x or y axis as tuple (low,high). If not specified, limits will be automatically chosen to be 'nice'

xlabel [str,optional] label for the x-axis

ylabel [str, optional] label for the y-axis

label [str, optional] label for the legend

Returns

axis of plot

filterpy.stats.**plot_gaussian_pdf**(*mean=0.0*, *variance=1.0*, *std=None*, *ax=None*, *mean_line=False*, *xlim=None*, *ylim=None*, *xlabel=None*, *ylabel=None*, *label=None*)

Plots a normal distribution PDF with the given mean and variance. x-axis contains the mean, the y-axis shows the probability density.

Parameters

mean [scalar, default 0.] mean for the normal distribution.

variance [scalar, default 1., optional] variance for the normal distribution.

std: scalar, default=None, optional standard deviation of the normal distribution. Use instead of *variance* if desired

ax [matplotlib axes object, optional] If provided, the axes to draw on, otherwise plt.gca() is used.

mean_line [boolean] draws a line at x=mean

xlim, ylim: (float,float), optional specify the limits for the x or y axis as tuple (low,high). If not specified, limits will be automatically chosen to be 'nice'

xlabel [str,optional] label for the x-axis

ylabel [str, optional] label for the y-axis

label [str, optional] label for the legend

Returns

axis of plot

---

filterpy.stats.**plot_discrete_cdf**(*xs*, *ys*, *ax=None*, *xlabel=None*, *ylabel=None*, *label=None*)

Plots a normal distribution CDF with the given mean and variance. x-axis contains the mean, the y-axis shows the cumulative probability.

Parameters

xs [list-like of scalars] x values corresponding to the values in *y's. Can be 'None*, in which case range(len(ys)) will be used.

ys [list-like of scalars] list of probabilities to be plotted which should sum to 1.

ax [matplotlib axes object, optional] If provided, the axes to draw on, otherwise plt.gca() is used.

xlim, ylim: (float,float), optional specify the limits for the x or y axis as tuple (low,high). If not specified, limits will be automatically chosen to be 'nice'

xlabel [str,optional] label for the x-axis

ylabel [str, optional] label for the y-axis

label [str, optional] label for the legend

Returns

axis of plot

---

filterpy.stats.**plot_gaussian**(*mean=0.0*, *variance=1.0*, *ax=None*, *mean_line=False*, *xlim=None*, *ylim=None*, *xlabel=None*, *ylabel=None*, *label=None*)

DEPRECATED. Use plot_gaussian_pdf() instead. This is poorly named, as there are multiple ways to plot a Gaussian.

---

filterpy.stats.**covariance_ellipse**(*P*, *deviations=1*)

Returns a tuple defining the ellipse representing the 2 dimensional covariance matrix P.

Parameters

P [nd.array shape (2,2)] covariance matrix

deviations [int (optional, default = 1)] # of standard deviations. Default is 1.

Returns (angle_radians, width_radius, height_radius)

---

filterpy.stats.**plot_covariance_ellipse**(*mean*, *cov=None*, *variance=1.0*, *std=None*, *ellipse=None*, *title=None*, *axis_equal=True*, *show_semiaxis=False*, *facecolor=None*, *edgecolor=None*, *fc=u'none'*, *ec=u'#004080'*, *alpha=1.0*, *xlim=None*, *ylim=None*, *ls=u'solid'*)

Deprecated function to plot a covariance ellipse. Use plot_covariance instead.

See also:

`plot_covariance`

---

filterpy.stats.**norm_cdf**(*x_range*, *mu*, *var=1*, *std=None*)

Computes the probability that a Gaussian distribution lies within a range of values.

Parameters

x_range [(float, float)] tuple of range to compute probability for

mu [float] mean of the Gaussian

var [float, optional] variance of the Gaussian. Ignored if *std* is provided

std [float, optional] standard deviation of the Gaussian. This overrides the *var* parameter

Returns

probability [float] probability that Gaussian is within x_range. E.g. .1 means 10%.

---

filterpy.stats.**rand_student_t**(*df*, *mu=0*, *std=1*)

return random number distributed by student's t distribution with *df* degrees of freedom with the specified mean and standard deviation.

---

filterpy.stats.**NESS**(*xs*, *est_xs*, *ps*)

Computes the normalized estimated error squared test on a sequence of estimates. The estimates are optimal if the mean error is zero and the covariance matches the Kalman filter's covariance. If this holds, then the mean of the NESS should be equal to or less than the dimension of x.

Parameters

xs [list-like] sequence of true values for the state x

est_xs [list-like] sequence of estimates from an estimator (such as Kalman filter)

ps [list-like] sequence of covariance matrices from the estimator

Returns

ness [list of floats] list of NESS computed for each estimate

Examples

`filterpy.stats.`**`mahalanobis`**(*x*, *mean*, *cov*)

Computes the Mahalanobis distance between the state vector x from the Gaussian *mean* with covariance *cov*. This can be thought as the number of standard deviations x is from the mean, i.e. a return value of 3 means x is 3 std from mean.

Parameters

x [(N,) array_like, or float] Input state vector

mean [(N,) array_like, or float] mean of multivariate Gaussian

cov [(N, N) array_like or float] covariance of the multivariate Gaussian

Returns

mahalanobis [double] The Mahalanobis distance between vectors *x* and *mean*

Examples

```
>>> mahalanobis(x=3., mean=3.5, cov=4.**2) # univariate case
0.125
```

```
>>> mahalanobis(x=3., mean=6, cov=1) # univariate, 3 std away
3.0
```

```
>>> mahalanobis([1., 2], [1.1, 3.5], [[1., .1],[.1, 13]])
0.42533327058913922
```

5.4 filterpy.monte_carlo

Routines for Markov Chain Monte Carlo (MCMC) computation, mainly for particle filtering.

5.4.1 resampling

Routines for resampling particles from particle filters based on their current weights. All these routines take a list of normalized weights and returns a list of indexes to the weights that should be chosen for resampling. The caller is responsible for performing the actual resample.

filterpy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

---

filterpy.monte_carlo.**residual_resample**(*weights*)

Performs the residual resampling algorithm used by particle filters.

Based on observation that we don't need to use random numbers to select most of the weights. Take int(N*w^i) samples of each particle i, and then resample any remaining using a standard resampling algorithm [1]

Parameters

weights [list-like of float] list of weights as floats

Returns

indexes [ndarray of ints] array of indexes into the weights defining the resample. i.e. the index of the zeroth resample is indexes[0], etc.

References

[1]

---

filterpy.monte_carlo.**stratified_resample**(*weights*)

Performs the stratified resampling algorithm used by particle filters.

This algorithms aims to make selections relatively uniformly across the particles. It divides the cumulative sum of the weights into N equal divisions, and then selects one particle randomly from each division. This guarantees that each sample is between 0 and 2/N apart.

Parameters

weights [list-like of float] list of weights as floats

Returns

indexes [ndarray of ints] array of indexes into the weights defining the resample. i.e. the index of the zeroth resample is indexes[0], etc.

---

filterpy.monte_carlo.**systematic_resample**(*weights*)

Performs the systemic resampling algorithm used by particle filters.

This algorithm separates the sample space into N divisions. A single random offset is used to to choose where to sample from for all divisions. This guarantees that every sample is exactly 1/N apart.

Parameters

weights [list-like of float] list of weights as floats

Returns

indexes [ndarray of ints] array of indexes into the weights defining the resample. i.e. the index of the zeroth resample is indexes[0], etc.

---

filterpy.monte_carlo.**multinomial_resample**(*weights*)

This is the naive form of roulette sampling where we compute the cumulative sum of the weights and then use binary search to select the resampled point based on a uniformly distributed random number. Run time is O(n log n). You do not want to use this algorithm in practice; for some reason it is popular in blogs and online courses so I included it for reference.

Parameters

weights [list-like of float] list of weights as floats

Returns

indexes [ndarray of ints] array of indexes into the weights defining the resample. i.e. the index of the zeroth resample is indexes[0], etc.

5.5 filterpy.discrete_bayes

Routines for performing discrete Bayes filtering.

5.5.1 Discrete Bayes

words go here

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

---

`filterpy.discrete_bayes.`**`normalize`**(*pdf*)

Normalize distribution *pdf* in-place so it sums to 1.0.

Returns pdf for convienence, so you can write things like:

```
>>> kernel = normalize(randn(7))
```

Parameters

pdf [ndarray] discrete distribution that needs to be converted to a pdf. Converted in-place, i.e., this is modified.

Returns

pdf [ndarray] The converted pdf.

`filterpy.discrete_bayes.`**`update`**(*likelihood*, *prior*)

Computes the posterior of a discrete random variable given a discrete likelihood and prior. In a typical application the likelihood will be the likelihood of a measurement matching your current environment, and the prior comes from discrete_bayes.predict().

Parameters

likelihood [ndarray, dtype=flaot] array of likelihood values

prior [ndarray, dtype=flaot] prior pdf.

Returns

posterior [ndarray, dtype=float] Returns array representing the posterior.

Examples

```
# self driving car. Sensor returns values that can be equated to positions
# on the road. A real likelihood compuation would be much more complicated
# than this example.

likelihood = np.ones(len(road))
likelihood[road==z] *= scale_factor

prior = predict(posterior, velocity, kernel)
posterior = update(likelihood, prior)
```

---

filterpy.discrete_bayes.**predict**(*pdf*, *offset*, *kernel*, *mode=u'wrap'*, *cval=0.0*)

Performs the discrete Bayes filter prediction step, generating the prior.

pdf is a discrete probability distribution expressing our initial belief.

offset is an integer specifying how much we want to move to the right (negative values means move to the left)

We assume there is some noise in that offset, which we express in *kernel*. For example, if offset=3 and kernel=[.1, .7., .2], that means we think there is a 70% chance of moving right by 3, a 10% chance of moving 2 spaces, and a 20% chance of moving by 4.

It returns the resulting distribution.

If *mode='wrap'*, then the probability distribution is wrapped around the array.

If *mode='constant'*, or any other value the pdf is shifted, with *cval* used to fill in missing elements.

Examples

```
belief = [.05, .05, .05, .05, .55, .05, .05, .05, .05, .05]
prior = predict(belief, offset=2, kernel=[.1, .8, .1])
```

5.6 filterpy.gh

These classes various g-h filters. The functions are helpers that provide settings for the *g* and *h* parameters for various common filters.

5.6.1 GHFilterOrder

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

class filterpy.gh.**GHFilterOrder**(*x0*, *dt*, *order*, *g*, *h=None*, *k=None*)

A g-h filter of aspecified order 0, 1, or 2.

Strictly speaking, the g-h filter is order 1, and the 2nd order filter is called the g-h-k filter. I'm not aware of any filter name that encompasses orders 0, 1, and 2 under one name, or I would use it.

Parameters

x0 [1D np.array or scalar] Initial value for the filter state. Each value can be a scalar or a np.array.

You can use a scalar for x0. If order > 0, then 0.0 is assumed for the higher order terms.

x[0] is the value being tracked x[1] is the first derivative (for order 1 and 2 filters) x[2] is the second derivative (for order 2 filters)

dt [scalar] timestep

order [int] order of the filter. Defines the order of the system 0 - assumes system of form x = a_0 + a_1*t 1 - assumes system of form x = a_0 +a_1*t + a_2*t^2 2 - assumes system of form x = a_0 +a_1*t + a_2*t^2 + a_3*t^3

g [float] filter g gain parameter.

h [float, optional] filter h gain parameter, order 1 and 2 only

k [float, optional] filter k gain parameter, order 2 only

Atrributes

———-

x [np.array] State of the filter.

x[0] is the value being tracked x[1] is the derivative of x[0] (order 1 and 2 only) x[2] is the 2nd derivative of x[0] (order 2 only)

This is always an np.array, even for order 0 where you can initialize x0 with a scalar.

y [np.array] Residual - difference between the measurement and the prediction

dt [scalar] timestep

order [int] order of the filter. Defines the order of the system 0 - assumes system of form x = a_0 + a_1*t 1 - assumes system of form x = a_0 +a_1*t + a_2*t^2 2 - assumes system of form x = a_0 +a_1*t + a_2*t^2 + a_3*t^3

g [float] filter g gain parameter.

h [float] filter h gain parameter, order 1 and 2 only

k [float] filter k gain parameter, order 2 only

z [1D np.array or scalar] measurement passed into update()

__init__(*x0*, *dt*, *order*, *g*, *h=None*, *k=None*)

Creates a g-h filter of order 0, 1, or 2.

update(*z*, *g=None*, *h=None*, *k=None*)

Update the filter with measurement z. z must be the same type or treatable as the same type as self.x[0].

5.6.2 GHFilter

Implements the g-h filter.

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

class filterpy.gh.**GHFilter**(*x*, *dx*, *dt*, *g*, *h*)

Implements the g-h filter. The topic is too large to cover in this comment. See my book "Kalman and Bayesian Filters in Python" [1] or Eli Brookner's "Tracking and Kalman Filters Made Easy" [2].

A few basic examples are below, and the tests in ./gh_tests.py may give you more ideas on use.

Parameters

x [1D np.array or scalar] Initial value for the filter state. Each value can be a scalar or a np.array.

You can use a scalar for x0. If order > 0, then 0.0 is assumed for the higher order terms.

x[0] is the value being tracked x[1] is the first derivative (for order 1 and 2 filters) x[2] is the second derivative (for order 2 filters)

dx [1D np.array or scalar] Initial value for the derivative of the filter state.

dt [scalar] time step

g [float] filter g gain parameter.

h [float] filter h gain parameter.

References

[1] Labbe, "Kalman and Bayesian Filters in Python" http://rlabbe.github.io/Kalman-and-Bayesian-Filters-in-Python

[2] Brookner, "Tracking and Kalman Filters Made Easy". John Wiley and Sons, 1998.

Examples

Create a basic filter for a scalar value with g=.8, h=.2. Initialize to 0, with a derivative(velocity) of 0.

```
>>> from filterpy.gh import GHFilter
>>> f = GHFilter (x=0., dx=0., dt=1., g=.8, h=.2)
```

Incorporate the measurement of 1

```
>>> f.update(z=1)
(0.8, 0.2)
```

Incorporate a measurement of 2 with g=1 and h=0.01

```
>>> f.update(z=2, g=1, h=0.01)
(2.0, 0.21000000000000002)
```

Create a filter with two independent variables.

```
>>> from numpy import array
>>> f = GHFilter (x=array([0,1]), dx=array([0,0]), dt=1, g=.8, h=.02)
```

and update with the measurements (2,4)

```
>>> f.update(array([2,4])
(array([ 1.6,  3.4]), array([ 0.04,  0.06]))
```

Attributes

x [1D np.array or scalar] filter state

dx [1D np.array or scalar] derivative of the filter state.

x_prediction [1D np.array or scalar] predicted filter state

dx_prediction [1D np.array or scalar] predicted derivative of the filter state.

dt [scalar] time step

g [float] filter g gain parameter.

h [float] filter h gain parameter.

y [np.array, or scalar] residual (difference between measurement and prior)

z [np.array, or scalar] measurement passed into update()

__init__ (*x*, *dx*, *dt*, *g*, *h*)
x.__init__(...) initializes x; see help(type(x)) for signature

update (*z*, *g=None*, *h=None*)
performs the g-h filter predict and update step on the measurement z. Modifies the member variables listed below, and returns the state of x and dx as a tuple as a convienence.

Modified Members

x filtered state variable

dx derivative (velocity) of x

residual difference between the measurement and the prediction for x

x_prediction predicted value of x before incorporating the measurement z.

dx_prediction predicted value of the derivative of x before incorporating the measurement z.

Parameters

z [any] the measurement

g [scalar (optional)] Override the fixed self.g value for this update

h [scalar (optional)] Override the fixed self.h value for this update

Returns

x filter output for x

dx filter output for dx (derivative of x

batch_filter(*data*, *save_predictions=False*, *saver=None*)

Given a sequenced list of data, performs g-h filter with a fixed g and h. See update() if you need to vary g and/or h.

Uses self.x and self.dx to initialize the filter, but DOES NOT alter self.x and self.dx during execution, allowing you to use this class multiple times without reseting self.x and self.dx. I'm not sure how often you would need to do that, but the capability is there. More exactly, none of the class member variables are modified by this function, in distinct contrast to update(), which changes most of them.

Parameters

data [list like] contains the data to be filtered.

save_predictions [boolean] the predictions will be saved and returned if this is true

saver [filterpy.common.Saver, optional] filterpy.common.Saver object. If provided, saver.save() will be called after every epoch

Returns

results [np.array shape (n+1, 2), where n=len(data)] contains the results of the filter, where results[i,0] is x , and results[i,1] is dx (derivative of x) First entry is the initial values of x and dx as set by __init__.

predictions [np.array shape(n), optional] the predictions for each step in the filter. Only retured if save_predictions == True

VRF_prediction()

Returns the Variance Reduction Factor of the prediction step of the filter. The VRF is the normalized variance for the filter, as given in the equation below.

$$VRF(\hat{x}_{n+1,n}) = \frac{VAR(\hat{x}_{n+1,n})}{\sigma_x^2}$$

References

Asquith, "Weight Selection in First Order Linear Filters" Report No RG-TR-69-12, U.S. Army Missle Command. Redstone Arsenal, Al. November 24, 1970.

VRF()

Returns the Variance Reduction Factor (VRF) of the state variable of the filter (x) and its derivatives (dx, ddx). The VRF is the normalized variance for the filter, as given in the equations below.

$$VRF(\hat{x}_{n,n}) = \frac{VAR(\hat{x}_{n,n})}{\sigma_x^2}$$

$$VRF(\hat{\dot{x}}_{n,n}) = \frac{VAR(\hat{\dot{x}}_{n,n})}{\sigma_x^2}$$

$$VRF(\hat{\ddot{x}}_{n,n}) = \frac{VAR(\hat{\ddot{x}}_{n,n})}{\sigma_x^2}$$

Returns

vrf_x VRF of x state variable

vrf_dx VRF of the dx state variable (derivative of x)

5.6.3 GHKFilter

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.gh.**GHKFilter**(*x*, *dx*, *ddx*, *dt*, *g*, *h*, *k*)

Implements the g-h-k filter.

Parameters

x [1D np.array or scalar] Initial value for the filter state. Each value can be a scalar or a np.array.

You can use a scalar for x0. If order > 0, then 0.0 is assumed for the higher order terms.

x[0] is the value being tracked x[1] is the first derivative (for order 1 and 2 filters) x[2] is the second derivative (for order 2 filters)

dx [1D np.array or scalar] Initial value for the derivative of the filter state.

ddx [1D np.array or scalar] Initial value for the second derivative of the filter state.

dt [scalar] time step

g [float] filter g gain parameter.

h [float] filter h gain parameter.

k [float] filter k gain parameter.

References

Brookner, "Tracking and Kalman Filters Made Easy". John Wiley and Sons, 1998.

Attributes

x [1D np.array or scalar] filter state

dx [1D np.array or scalar] derivative of the filter state.

ddx [1D np.array or scalar] second derivative of the filter state.

x_prediction [1D np.array or scalar] predicted filter state

dx_prediction [1D np.array or scalar] predicted derivative of the filter state.

ddx_prediction [1D np.array or scalar] second predicted derivative of the filter state.

dt [scalar] time step

g [float] filter g gain parameter.

h [float] filter h gain parameter.

k [float] filter k gain parameter.

y [np.array, or scalar] residual (difference between measurement and prior)

z [np.array, or scalar] measurement passed into update()

__init__ (*x*, *dx*, *ddx*, *dt*, *g*, *h*, *k*)
x.__init__(...) initializes x; see help(type(x)) for signature

update (*z*, *g=None*, *h=None*, *k=None*)
Performs the g-h filter predict and update step on the measurement z.

On return, self.x, self.dx, self.y, and self.x_prediction will have been updated with the results of the computation. For convienence, self.x and self.dx are returned in a tuple.

Parameters

z [scalar] the measurement

g [scalar (optional)] Override the fixed self.g value for this update

h [scalar (optional)] Override the fixed self.h value for this update

k [scalar (optional)] Override the fixed self.k value for this update

Returns

x filter output for x

dx filter output for dx (derivative of x

batch_filter (*data*, *save_predictions=False*)
Performs g-h filter with a fixed g and h.

Uses self.x and self.dx to initialize the filter, but DOES NOT alter self.x and self.dx during execution, allowing you to use this class multiple times without reseting self.x and self.dx. I'm not sure how often you would need to do that, but the capability is there. More exactly, none of the class member variables are modified by this function.

Parameters

data [list_like] contains the data to be filtered.

save_predictions [boolean] The predictions will be saved and returned if this is true

Returns

results [np.array shape (n+1, 2), where n=len(data)] contains the results of the filter, where results[i,0] is x , and results[i,1] is dx (derivative of x) First entry is the initial values of x and dx as set by __init__.

predictions [np.array shape(n), or None] the predictions for each step in the filter. Only returned if save_predictions == True

VRF_prediction ()
Returns the Variance Reduction Factor for x of the prediction step of the filter.

This implements the equation

$$VRF(\hat{x}_{n+1,n}) = \frac{VAR(\hat{x}_{n+1,n})}{\sigma_x^2}$$

References

Asquith and Woods, "Total Error Minimization in First and Second Order Prediction Filters" Report No RE-TR-70-17, U.S. Army Missle Command. Redstone Arsenal, Al. November 24, 1970.

bias_error (*dddx*)
Returns the bias error given the specified constant jerk(dddx)

Parameters

dddx [type(self.x)] 3rd derivative (jerk) of the state variable x.

References

Asquith and Woods, "Total Error Minimization in First and Second Order Prediction Filters" Report No RE-TR-70-17, U.S. Army Missle Command. Redstone Arsenal, Al. November 24, 1970.

VRF()

Returns the Variance Reduction Factor (VRF) of the state variable of the filter (x) and its derivatives (dx, ddx). The VRF is the normalized variance for the filter, as given in the equations below.

$$VRF(\hat{x}_{n,n}) = \frac{VAR(\hat{x}_{n,n})}{\sigma_x^2}$$

$$VRF(\hat{\dot{x}}_{n,n}) = \frac{VAR(\hat{\dot{x}}_{n,n})}{\sigma_x^2}$$

$$VRF(\hat{\ddot{x}}_{n,n}) = \frac{VAR(\hat{\ddot{x}}_{n,n})}{\sigma_x^2}$$

Returns

vrf_x [type(x)] VRF of x state variable

vrf_dx [type(x)] VRF of the dx state variable (derivative of x)

vrf_ddx [type(x)] VRF of the ddx state variable (second derivative of x)

5.6.4 optimal_noise_smoothing

filterpy.gh.**optimal_noise_smoothing**(*g*)

provides g,h,k parameters for optimal smoothing of noise for a given value of g. This is due to Polge and Bhagavan[1].

Parameters

g [float] value for g for which we will optimize for

Returns

(g,h,k) [(float, float, float)] values for g,h,k that provide optimal smoothing of noise

References

[1] Polge and Bhagavan. "A Study of the g-h-k Tracking Filter". Report No. RE-CR-76-1. University of Alabama in Huntsville. July, 1975

Examples

```
from filterpy.gh import GHKFilter, optimal_noise_smoothing

g,h,k = optimal_noise_smoothing(g)
f = GHKFilter(0,0,0,1,g,h,k)
f.update(1.)
```

5.6.5 least_squares_parameters

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

`filterpy.gh.`**`least_squares_parameters`**`(n)`

An order 1 least squared filter can be computed by a g-h filter by varying g and h over time according to the formulas below, where the first measurement is at n=0, the second is at n=1, and so on:

$$h_n = \frac{6}{(n+2)(n+1)}$$
$$g_n = \frac{2(2n+1)}{(n+2)(n+1)}$$

Parameters

n [int] the nth measurement, starting at 0 (i.e. first measurement has n==0)

Returns

(g,h) [(float, float)] g and h parameters for this time step for the least-squares filter

Examples

```
from filterpy.gh import GHFilter, least_squares_parameters

lsf = GHFilter (0, 0, 1, 0, 0)
z = 10
for i in range(10):
    g,h = least_squares_parameters(i)
    lsf.update(z, g, h)
```

5.6.6 critical_damping_parameters

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

filterpy.gh.**critical_damping_parameters**(*theta*, *order=2*)

Computes values for g and h (and k for g-h-k filter) for a critically damped filter.

The idea here is to create a filter that reduces the influence of old data as new data comes in. This allows the filter to track a moving target better. This goes by different names. It may be called the discounted least-squares g-h filter, a fading-memory polynomal filter of order 1, or a critically damped g-h filter.

In a normal least-squares filter we compute the error for each point as

$$\epsilon_t = (z - \hat{x})^2$$

For a crically damped filter we reduce the influence of each error by

$$\theta^{t-i}$$

where

$$0 <= \theta <= 1$$

In other words the last error is scaled by theta, the next to last by theta squared, the next by theta cubed, and so on.

Parameters

theta [float, 0 <= theta <= 1] scaling factor for previous terms

order [int, 2 (default) or 3] order of filter to create the parameters for. g and h will be calculated for the order 2, and g, h, and k for order 3.

Returns

g [scalar] optimal value for g in the g-h or g-h-k filter

h [scalar] optimal value for h in the g-h or g-h-k filter

k [scalar] optimal value for g in the g-h-k filter

References

Brookner, "Tracking and Kalman Filters Made Easy". John Wiley and Sons, 1998.

Polge and Bhagavan. "A Study of the g-h-k Tracking Filter". Report No. RE-CR-76-1. University of Alabama in Huntsville. July, 1975

Examples

```
from filterpy.gh import GHFilter, critical_damping_parameters

g,h = critical_damping_parameters(0.3)
critical_filter = GHFilter(0, 0, 1, g, h)
```

5.6.7 benedict_bornder_constants

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

`filterpy.gh.`**`benedict_bornder_constants`**(*g*, *critical=False*)

Computes the g,h constants for a Benedict-Bordner filter, which minimizes transient errors for a g-h filter.

Returns the values g,h for a specified g. Strictly speaking, only h is computed, g is returned unchanged.

The default formula for the Benedict-Bordner allows ringing. We can "nearly" critically damp it; ringing will be reduced, but not entirely eliminated at the cost of reduced performance.

Parameters

g [float] scaling factor g for the filter

critical [boolean, default False] Attempts to critically damp the filter.

Returns

g [float] scaling factor g (same as the g that was passed in)

h [float] scaling factor h that minimizes the transient errors

References

Brookner, "Tracking and Kalman Filters Made Easy". John Wiley and Sons, 1998.

Examples

```
from filterpy.gh import GHFilter, benedict_bornder_constants
g, h = benedict_bornder_constants(.855)
f = GHFilter(0, 0, 1, g, h)
```

5.7 filterpy.memory

Implements a polynomial fading memory filter. You can achieve the same results, and more, using the KalmanFilter class. However, some books use this form of the fading memory filter, so it is here for completeness. I suppose some would also find this simpler to use than the standard Kalman filter.

5.7.1 FadingMemoryFilter

Introduction and Overview

Implements a polynomial fading memory filter. You can achieve the same results, and more, using the KalmanFilter class. However, some books use this form of the fading memory filter, so it is here for completeness. I suppose some would also find this simpler to use than the standard Kalman filter.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

class filterpy.memory.**FadingMemoryFilter**(*x0*, *dt*, *order*, *beta*)

Creates a fading memory filter of order 0, 1, or 2.

The KalmanFilter class also implements a more general fading memory filter and should be preferred in most cases. This is probably faster for low order systems.

This algorithm is based on the fading filter algorithm developed in Zarcan's "Fundamentals of Kalman Filtering" [1].

Parameters

x0 [1D np.array or scalar] Initial value for the filter state. Each value can be a scalar or a np.array.

You can use a scalar for x0. If order > 0, then 0.0 is assumed for the higher order terms.

x[0] is the value being tracked x[1] is the first derivative (for order 1 and 2 filters) x[2] is the second derivative (for order 2 filters)

dt [scalar] timestep

order [int] order of the filter. Defines the order of the system 0 - assumes system of form x = a_0 + a_1*t 1 - assumes system of form x = a_0 +a_1*t + a_2*t^2 2 - assumes system of form x = a_0 +a_1*t + a_2*t^2 + a_3*t^3

beta [float] filter gain parameter.

References

Paul Zarchan and Howard Musoff. "Fundamentals of Kalman Filtering: A Practical Approach" American Institute of Aeronautics and Astronautics, Inc. Fourth Edition. p. 521-536. (2015)

Attributes

x [np.array] State of the filter. x[0] is the value being tracked x[1] is the derivative of x[0] (order 1 and 2 only) x[2] is the 2nd derivative of x[0] (order 2 only)

This is always an np.array, even for order 0 where you can initialize x0 with a scalar.

P [np.array] The diagonal of the covariance matrix. Assumes that variance is one; multiply by sigma^2 to get the actual variances.

This is a constant and will not vary as the filter runs.

e [np.array] The truncation error of the filter. Each term must be multiplied by the a_1, a_2, or a_3 of the polynomial for the system.

For example, if the filter is order 2, then multiply all terms of self.e by a_3 to get the actual error. Multipy by a_2 for order 1, and a_1 for order 0.

__init__(*x0*, *dt*, *order*, *beta*)

x.__init__(...) initializes x; see help(type(x)) for signature

update(*z*)
update the filter with measurement z. z must be the same type (or treatable as the same type) as self.x[0].

5.8 filterpy.hinfinity

5.8.1 HInfinityFilter

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

class filterpy.hinfinity.**HInfinityFilter**(*dim_x*, *dim_z*, *dim_u*, *gamma*)
H-Infinity filter. You are responsible for setting the various state variables to reasonable values; the defaults below will not give you a functional filter.

Parameters

dim_x [int] Number of state variables for the Kalman filter. For example, if you are tracking the position and velocity of an object in two dimensions, dim_x would be 4.

This is used to set the default size of *P*, *Q*, and *u*

dim_z [int] Number of of measurement inputs. For example, if the sensor provides you with position in (x, y), *dim_z* would be 2.

dim_u [int] Number of control inputs for the Gu part of the prediction step.

gamma [float]

.. warning:: I do not believe this code is correct. DO NOT USE THIS. In particular, note that predict does not update the covariance matrix.

__init__(*dim_x*, *dim_z*, *dim_u*, *gamma*)
x.__init__(...) initializes x; see help(type(x)) for signature

update(*z*)
Add a new measurement *z* to the H-Infinity filter. If *z* is None, nothing is changed.

Parameters

z [ndarray] measurement for this update.

predict(*u=0*)
Predict next position.

Parameters

u [ndarray] Optional control vector. If non-zero, it is multiplied by *B* to create the control input into the system.

batch_filter(*Zs*, *update_first=False*, *saver=False*)
Batch processes a sequences of measurements.

Parameters

Zs [list-like] list of measurements at each time step *self.dt* Missing measurements must be represented by 'None'.

update_first [bool, default=False, optional,] controls whether the order of operations is update followed by predict, or predict followed by update.

saver [filterpy.common.Saver, optional] filterpy.common.Saver object. If provided, saver.save() will be called after every epoch

Returns

means: ndarray ((n, dim_x, 1)) array of the state for each time step. Each entry is an np.array. In other words *means[k,:]* is the state at step *k*.

covariance: ndarray((n, dim_x, dim_x)) array of the covariances for each time step. In other words *covariance[k, :, :]* is the covariance at step *k*.

get_prediction(*u=0*)

Predicts the next state of the filter and returns it. Does not alter the state of the filter.

Parameters

u [ndarray] optional control input

Returns

x [ndarray] State vector of the prediction.

residual_of(*z*)

returns the residual for the given measurement (z). Does not alter the state of the filter.

measurement_of_state(*x*)

Helper function that converts a state into a measurement.

Parameters

x [ndarray] H-Infinity state vector

Returns

z [ndarray] measurement corresponding to the given state

V

measurement noise matrix

5.8.2 filterpy.leastsq

LeastSquaresFilter

Copyright 2015 Roger R Labbe Jr.

FilterPy library. http://github.com/rlabbe/filterpy

Documentation at: https://filterpy.readthedocs.org

Supporting book at: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

This is licensed under an MIT license. See the readme.MD file for more information.

class filterpy.leastsq.**LeastSquaresFilter**(*dt*, *order*, *noise_sigma=0.0*)

Implements a Least Squares recursive filter. Formulation is per Zarchan *[1]*.

Filter may be of order 0 to 2. Order 0 assumes the value being tracked is a constant, order 1 assumes that it moves in a line, and order 2 assumes that it is tracking a second order polynomial.

Parameters

dt [float] time step per update

order [int] order of filter 0..2

noise_sigma [float] sigma (std dev) in x. This allows us to calculate the error of the filter, it does not influence the filter output.

References

[1]

Examples

```
from filterpy.leastsq import LeastSquaresFilter

lsq = LeastSquaresFilter(dt=0.1, order=1, noise_sigma=2.3)

while True:
    z = sensor_reading()   # get a measurement
    x = lsq.update(z)      # get the filtered estimate.
    print('error: {}, velocity error: {}'.format(
          lsq.error, lsq.derror))
```

Attributes

n [int] step in the recursion. 0 prior to first call, 1 after the first call, etc.

K [np.array] Gains for the filter. K[0] for all orders, K[1] for orders 0 and 1, and K[2] for order 2

x: np.array (order + 1, 1) estimate(s) of the output. It is a vector containing the estimate x and the derivatives of x: [x x' x'']].T. It contains as many derivatives as the order allows. That is, a zero order filter has no derivatives, a first order has one derivative, and a second order has two.

y [float] residual (difference between measurement projection of previous estimate to current time).

__init__ (*dt*, *order*, *noise_sigma=0.0*)
x.__init__(...) initializes x; see help(type(x)) for signature

reset ()
reset filter back to state at time of construction

update (*z*)
Update filter with new measurement *z*

Returns

x [np.array] estimate for this time step (same as self.x)

errors ()
Computes and returns the error and standard deviation of the filter at this time step.

Returns

error [np.array size 1xorder+1]

std [np.array size 1xorder+1]

CHAPTER 6

References

github repo: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

read online: http://nbviewer.ipython.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

PDF version (often lags the two sources above) https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/Kalman_and_Bayesian_Filters_in_Python.pdf

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Dan Simon. “Optimal State Estimation.” John Wiley & Sons. p. 208-212. (2006)

[2] Roger Labbe. “Kalman and Bayesian Filters in Python” https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

[1] Julier, Simon J. “The scaled unscented transformation,” American Control Converence, 2002, pp 4555-4559, vol 6.

Online copy: https://www.cs.unc.edu/~welch/kalman/media/pdf/ACC02-IEEE1357.PDF

[2] E. A. Wan and R. Van der Merwe, “The unscented Kalman filter for nonlinear estimation,” in Proc. Symp. Adaptive Syst. Signal Process., Commun. Contr., Lake Louise, AB, Canada, Oct. 2000.

Online Copy: https://www.seas.harvard.edu/courses/cs281/papers/unscented.pdf

[3] S. Julier, J. Uhlmann, and H. Durrant-Whyte. “A new method for the nonlinear transformation of means and covariances in filters and estimators,” IEEE Transactions on Automatic Control, 45(3), pp. 477-482 (March 2000).

[4] E. A. Wan and R. Van der Merwe, “The Unscented Kalman filter for Nonlinear Estimation,” in Proc. Symp. Adaptive Syst. Signal Process., Commun. Contr., Lake Louise, AB, Canada, Oct. 2000.

https://www.seas.harvard.edu/courses/cs281/papers/unscented.pdf

[5] Wan, Merle “The Unscented Kalman Filter,” chapter in *Kalman Filtering and Neural Networks*, John Wiley & Sons, Inc., 2001.

[6] R. Van der Merwe “Sigma-Point Kalman Filters for Probabilitic Inference in Dynamic State-Space Models” (Doctoral dissertation)

[1] R. Van der Merwe “Sigma-Point Kalman Filters for Probabilitic Inference in Dynamic State-Space Models” (Doctoral dissertation)

[1] Julier, Simon J.; Uhlmann, Jeffrey “A New Extension of the Kalman Filter to Nonlinear Systems”. Proc. SPIE 3068, Signal Processing, Sensor Fusion, and Target Recognition VI, 182 (July 28, 1997)

[1] Phillippe Moireau and Dominique Chapelle “Reduced-Order Unscented Kalman Filtering with Application to Parameter Identification in Large-Dimensional Systems” DOI: 10.1051/cocv/2010006

[1] J. S. Liu and R. Chen. Sequential Monte Carlo methods for dynamic systems. Journal of the American Statistical Association, 93(443):1032–1044, 1998.

[1] Zarchan and Musoff. “Fundamentals of Kalman Filtering: A Practical Approach.” Third Edition. AIAA, 2009.

Python Module Index

f

Index

Symbols

A

B

C

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V