Cubature Kalman Filter Algorithm: Line-By-Line

**Lines 1-13: Set Up**

A screen shot of a computer code

AI-generated content may be incorrect.

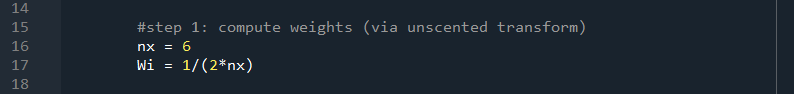
Define function, with state ‘XREF’ and time ‘tk’ reference trajectory input.

Import necessary functions and packages, noting that ‘NRHOmotion’ is explained in *CR3BP: Line-By-Line*.

Empty arrays set up to store results, ‘filter\_results’ to store state estimates, and ‘covariance\_results’ to store corresponding covariance determined by the EKF.

The rest of the function reads XREF state by state, storing the resulting estimates in filter\_results.

**Lines 14-18: Compute Weights**



The first step is setting the tuning parameters and therefore determining weights for the unscented transform and future point propagation.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | nx | Dimension of the state vector. |
|  | Wi | Weight for each sigma point. |

**Lines 19-34: Initialisation**

**A computer screen shot of text

AI-generated content may be incorrect.**

The first step in the algorithm is to set up the ‘a priori’ state. The filter takes this a priori ‘ith’ reference state provided in XREF array ‘XREFminus1’ (each row provides x, y, z coordinate) and corresponding time from tk ‘tkminus1’ and uses this to determine the next state at the next time read tk. How close the resulting state is to the corresponding truth state is the point of comparison for this experiment.

As the covariance is determined and carried on from the previous state, it is set to zero for the first read.

Below is a table outlining the naming conventions for the equations and the code.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | XREFminus1 | Reference state used to propagate from, most recent measurement of state |
|  | Pkminus1 | Last covariance (metric of how much we trust the estimate vs the measurement) determined |
|  | tkminus1 | Time at most recent measurement of state |
|  | tk | Time at which the future state estimate is being made to |

**Lines 50-62: Compute The Cubature Point Matrix**

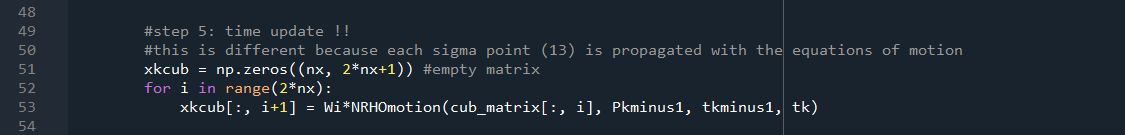
**A computer code on a black background

AI-generated content may be incorrect.**

The cubature point matrix, with dimensions nx by 2nx +1, is then computed via the following formula. Note that the values are defined in the accompanying table. Also note that the root of the covariance matrix was generated separately and named ‘Proot’ via Cholesky decomposition for optimal computational efficiency.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | cub\_matrix | Matrix containing the ‘cubature points’ or states computed from the reference state for later propagation based on the covariance. This allows for a number of possible states to be generated and compared. |

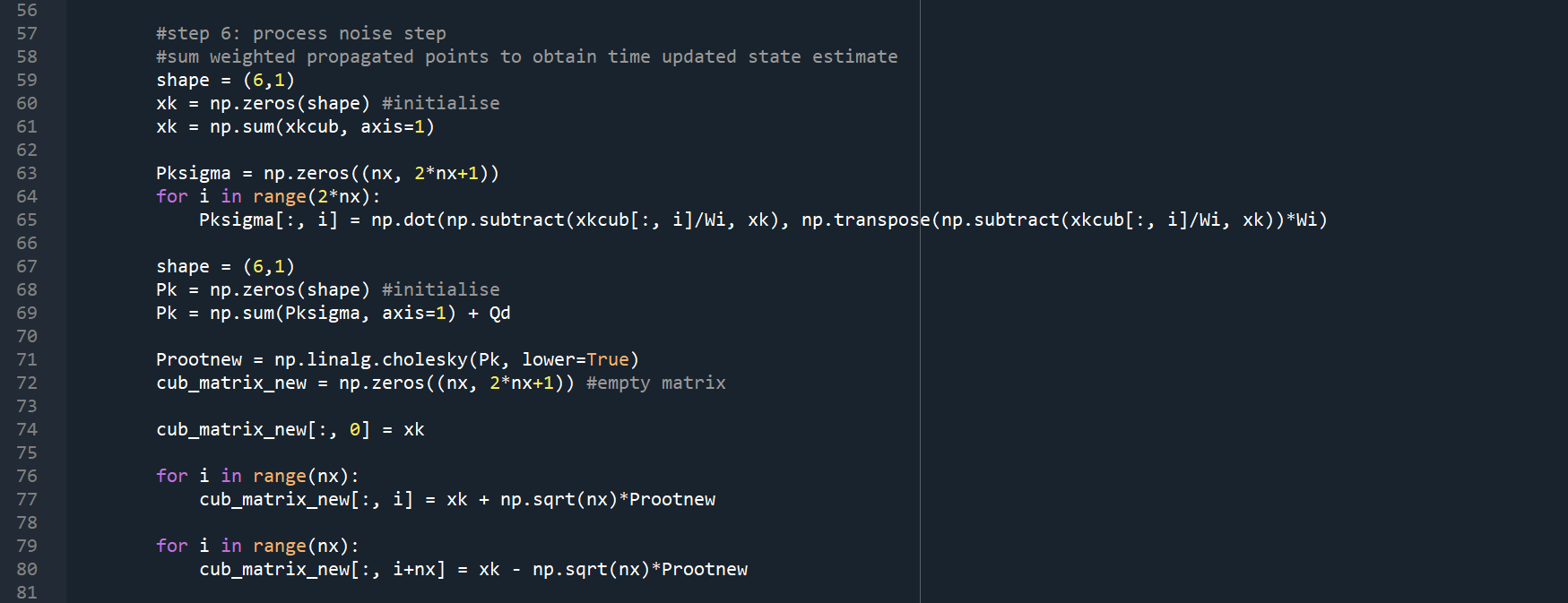
**Lines 48-54: Time Update**



Line 67 and 68 use the NRHOmotion function to propagate the sigma points and store the corresponding state estimates in xk, set up as an empty matrix prior in line 66.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | xcub | Matrix where propagated cubature points to the next time are stored. |

**Lines 70-107: Process Noise Step**



Next, a state and covariance is computed for the next time tk, including process noise via the following equations.

Then, a new set of sigma points are computed using this state at time k, again noting that the root of the covariance was calculated separately as ‘Prootnew’.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | xk | Resulting state estimate from summing the resulting propagated points from the time update, each multiplied by a scaling factor. |
|  | Pbark | Resulting covariance from propagated sigma points |
|  | Qd | Process noise matrix, an estimate of the errors accompanied with obtaining the results. |
|  | cub\_matrix\_new | Updated cubature point matrix using the new state estimate as the mean. |

**Lines 82-95: Predicted Measurement**

A computer screen shot of a computer code

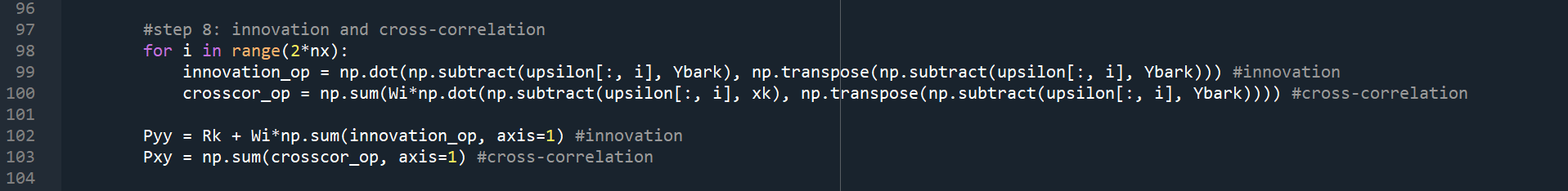
AI-generated content may be incorrect.

Propagating the new cubature point matrix, a new set of states are once again computed, under the matrix ‘upsilon’.

These points are then summed together in a similar fashion to as they were to obtain Xbark, determining a measurement updated state ‘Ybark’.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | upsilon | The propagated states of points stored in sigma\_matrix\_new. |
|  | Ybark | The measurement updated state as estimated by summing the propagated sigma points multiplied by their weighting factors. |

**Lines 127-143: Innovation & Cross-Correlation**



The above calculates the innovation and cross-correlation for the final corrective update. These are calculated via the below equations.

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | Pyy | Innovation |
|  | Pxy | Cross-correlation |
|  | Rk | Measurement noise matrix. |

**Lines 105-120: Corrector Update**

A computer screen shot of white text

AI-generated content may be incorrect.

Kalman gain determined:

State estimate:

Covariance:

|  |  |  |
| --- | --- | --- |
| Mathematical Notation | Python Variable Name | Definition |
|  | Kk | Kalman gain, weighting factor determining how much a measurement is trusted versus the model prediction. |
|  | Xk | Final state estimate for this iteration of the filter. |
|  | Yk | Actual state at time k. |
|  | Pk | Final covariance estimate for this iteration of the filter. |