Examples source code description - Python

Alex Becker
May 1, 2023

Abstract

This document describes the logic and architecture of the Matlab code employed in the examples featured in the book "Kalman Filter from the Ground Up."
It is expected that the reader possesses a certain level of familiarity with the Python programming language. Therefore, this document does not provide an explanation of Python commands or data types.
The provided Python functions include input/output descriptions and comments. Web-based resources can be utilized to understand specific Python commands.

Legal Disclaimer

Document Notation

- The font style of ***folders*** (directories) names is bold, italic, and light cyan.
- The font style of *files* or *function* names is italic and blue.
- The font style of `variable` names is orange typewriter.

Versions

This document describes version 1.0.0 of the Python code. The document version 1.0.0.00.

The code has been developed with the following Python and IDE versions:

- IDE PyCharm, version 2022.1
- Python version 3.8.3
- Libraries versions:
  - NumPy version 1.23.2
  - Matplotlib 3.5.3
  - SciPy version 1.5.0

1 Code Structure

As many scientists and engineers employ Structured Programming methods rather than Object-Oriented programming, the code has been developed with a functional structure where each function is considered a separate module. However, should Object-Oriented Programming be preferred, it is easily feasible to construct classes based on the existing

functions. Each function is a separate file to keep the Python code consistent with MATLAB code.

1.1 Folders Structure

The root directory contains the *main.py* file and folders. The *main* script is used for running examples.

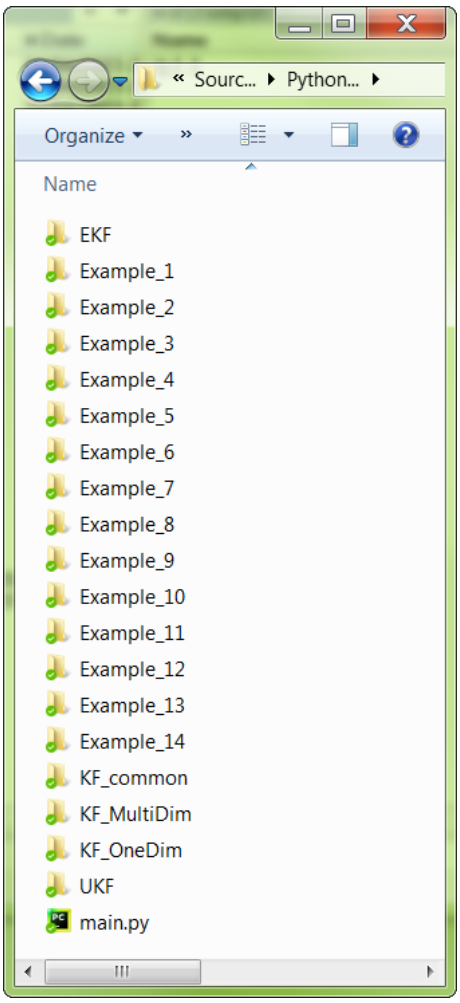

Figure 1: Folders structure.

- There are 14 folders named ***Examlpe_X***, which contain example-specific functions. For instance, the folder ***Example_6*** contains 'Example 6' functions.
- The ***KF_common*** folder contains auxiliary functions used by different examples.
- The ***KF_OneDim*** folder contains functions used in the univariate examples (Examples 1-8).
- The ***KF_MultiDim*** folder contains functions used in the multivariate Kalman Filter examples (Examples 9-14).
- The ***EKF*** folder contains the *EKF* function used in the Extended Kalman Filter examples (Examples 11-12).
- The ***UKF*** folder contains functions used in the Unscented Kalman Filter examples (Examples 13-14).

1.2 Example folder structure

Every ***Examlpe_X*** folder comprises functions specific to that particular example. For instance, the ***Example_9*** folder contains the following functions:

- *example_9* - the primary example function that executes relevant modules.
- *initParams_9* – parameters initiation for Example 9.
- *scenario_9* – this function generates the scenario (ground truth) for Example 9.
- *H_matrix_9* – this function creates the measurement matrix for Example 9.
- *plots_9* - this function creates results plots for Example 9 analysis.

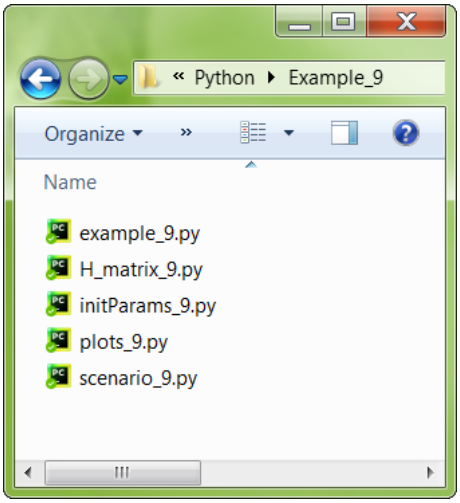

Figure 2: Example 9 folder.

2 Running example

There are two methods to run an example code.

1. Specify the example number in the *main* script (`exNum` variable) and run the script.

```
exNum = 1  # set example number

# run example
if exNum == 1:
    from Example_1.example_1 import example_1
    example_1()
elif exNum == 2:
    from Example_2.example_2 import example_2
    example_2()
elif exNum == 3:
    from Example_3.example_3 import example_3
    example_3()
elif exNum == 4:
    from Example_4.example_4 import example_4
    example_4()
elif exNum == 5:
    from Example_5.example_5 import example_5
    example_5()
elif exNum == 6:
    from Example_6.example_6 import example_6
```

```
    example_6()
elif exNum == 7:
    from Example_7.example_7 import example_7
    example_7()
elif exNum == 8:
    from Example_8.example_8 import example_8
    example_8()
elif exNum == 9:
    from Example_9.example_9 import example_9
    example_9()
elif exNum == 10:
    from Example_10.example_10 import example_10
    example_10()
elif exNum == 11:
    from Example_11.example_11 import example_11
    example_11()
elif exNum == 12:
    from Example_12.example_12 import example_12
    example_12()
elif exNum == 13:
    from Example_13.example_13 import example_13
    example_13()
elif exNum == 14:
    from Example_14.example_14 import example_14
    example_14()
```

2. Run *example_X* function (where 'X' is the example number).

3 Parameters initiation

Each example parameters are stored in a separate *initParams_X.py* file, where 'X' is the example number. The parameters are stored in the `params` dictionary.

Let us take a look at the *initParams_9* function.

```
def initParams_9(paramType):

    import numpy as np

    if paramType == 'KF':
        params = {
            "dim": 2,                          # 2D model (X,Y)
            "dt": 1,                           # Sample period
            "r": np.array([3]),                # Measurement Noise
            "sig_a": np.array([0.2]),          # Process Noise

            "strModel": 'quadratic',           # Constant acceleration
            "strNoiseModel": 'discrete',       # Discrete noise model

            # set filter initial conditions (state and variance)
            "x0": np.array([[0, 0, 0, 0, 0, 0]]).T,
            "P0": np.eye(6)*500
        }
    elif paramType == 'noiseGen':
        params = {
            "seed": np.array([123])     # Noise generator seed
        }
    elif paramType == 'scenario':
        params = {
            "isPlotScenario": False,
            # set scenario initial conditions
            "v": 25,                   # vehicle velocity (m/s)
            "L": 400,                  # trajectory straight part length
            "R": 300,                  # turn radius
            # process noise parameters
            "sig_a": np.array([0.2]),    # Process Noise
            "seed": np.array([789])      # Noise generator seed
        }

    return params
```

The function input is `paramType`. The function output is `params` dictionary.

The `paramType` variable is a string. `paramType` can be set to `'KF'`, `'noiseGen'`, or `'scenario'`.

When `paramType` is set to `'KF'`, the `params` dictionary is set with the Kalman Filter parameters.

When `paramType` is set to `'noiseGen'`, the `params` dictionary is set with parameters related to noise generation.

When `paramType` is set to `'scenario'`, the `params` dictionary is set with scenario parameters.

4 Example 1

In this example, we estimate the state of the static system (the gold weight measurement) using an averaging filter.

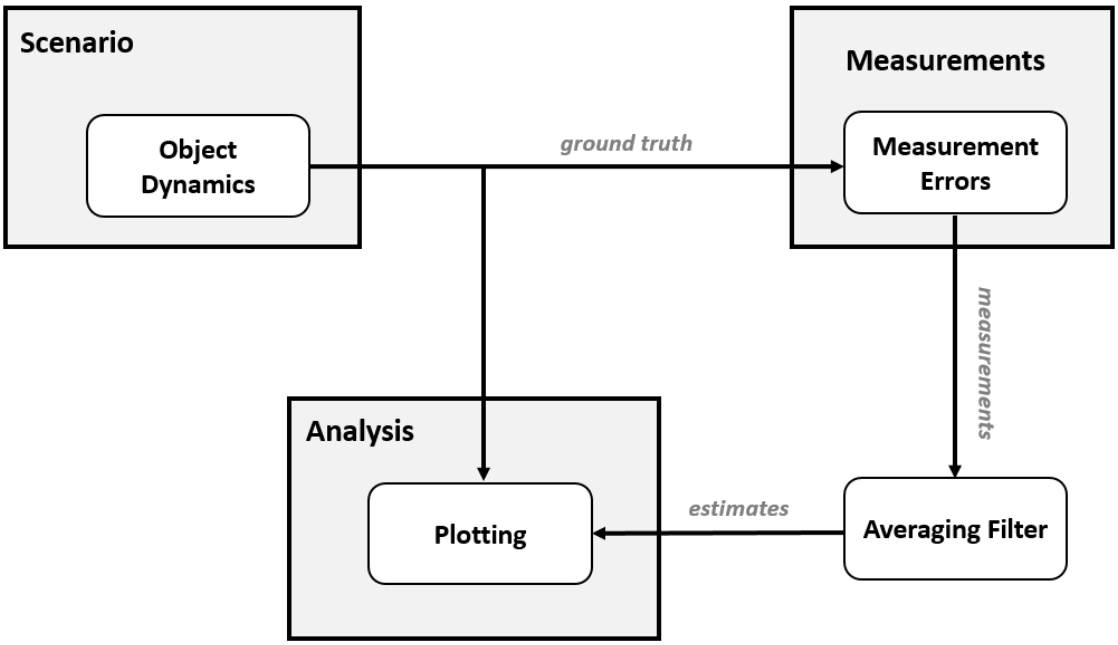

Figure 3: Example 1 flow chart.

1. First, we create the scenario (the ground truth) using *scenario_1* function. For this example, the scenario is a series of equal integers since the actual weight of the gold doesn't change.

   In order to change the number of iterations, modify `params["n"]` value in *initParams_1* function.

2. The next stage is measurements creation. The function *addNoise* (in folder ***KF_common***) adds normally distributed measurement noise to the ground truth. You can see the description of the *addNoise* function in subsection 16.1.

3. The gold weight is estimated by an averaging filter (function *avgFilter* in the ***KF_OneDim*** folder).

4. The function *plots_1* plots the estimates vs. the ground truth.

5 Example 2

In this example, we estimate the position and velocity of the aircraft moving at constant velocity using $\alpha - \beta$ filter.

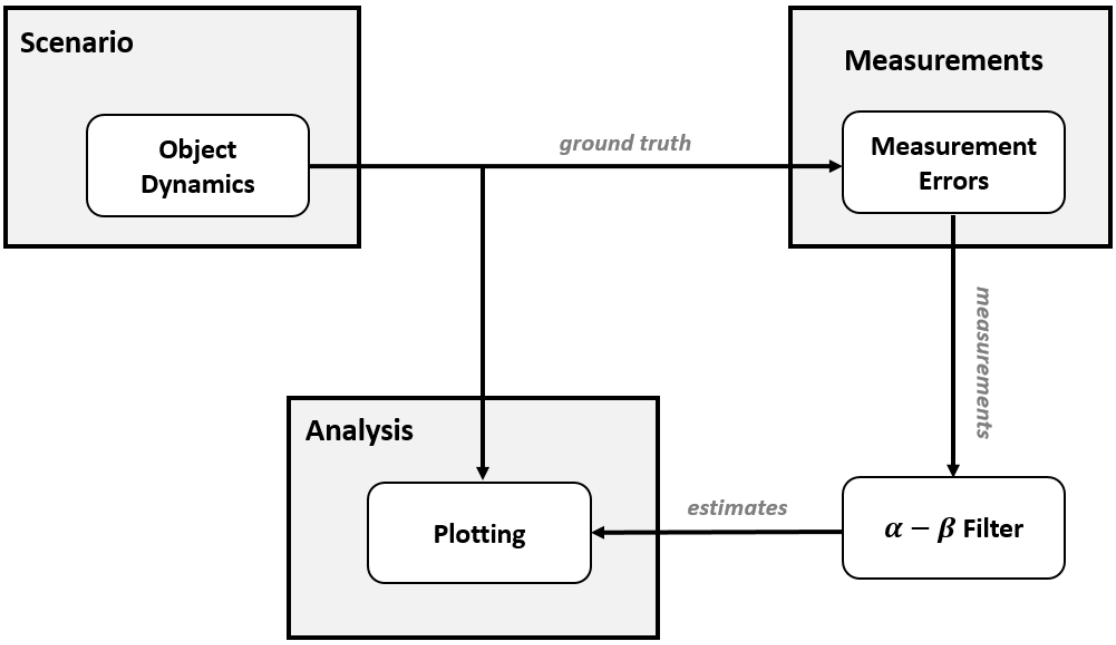

Figure 4: Example 2 flow chart.

1. First, we create the scenario (the ground truth) using *scenario_2* function. The target position is calculated using Newton's motion equation.
   - In order to change the number of iterations, modify the `params["n"]` value in the *initParams_2* function.
   - In order to plot the ground truth, set the `params["isPlotScenario"]` value in the *initParams_2* function to `'True'`.
2. The next stage is measurements creation. The function *addNoise* (in folder ***KF_common***) adds normally distributed measurement noise to the ground truth. You can see the description of the *addNoise* function in subsection 16.1.
3. The aircraft position and velocity are estimated by an averaging filter (function *alphaBetaGammaFilter* in the ***KF_OneDim*** folder).
4. The function *plots_2* plots the estimates vs. the ground truth.

6 Examples 3-4

In these examples, we estimate the position and velocity of accelerating aircraft using $\alpha - \beta$ and $\alpha - \beta - \gamma$ filters.

The flow is similar to Example 2.

7 Example 5

In this example, we estimate the height of a building height using Kalman Filter.

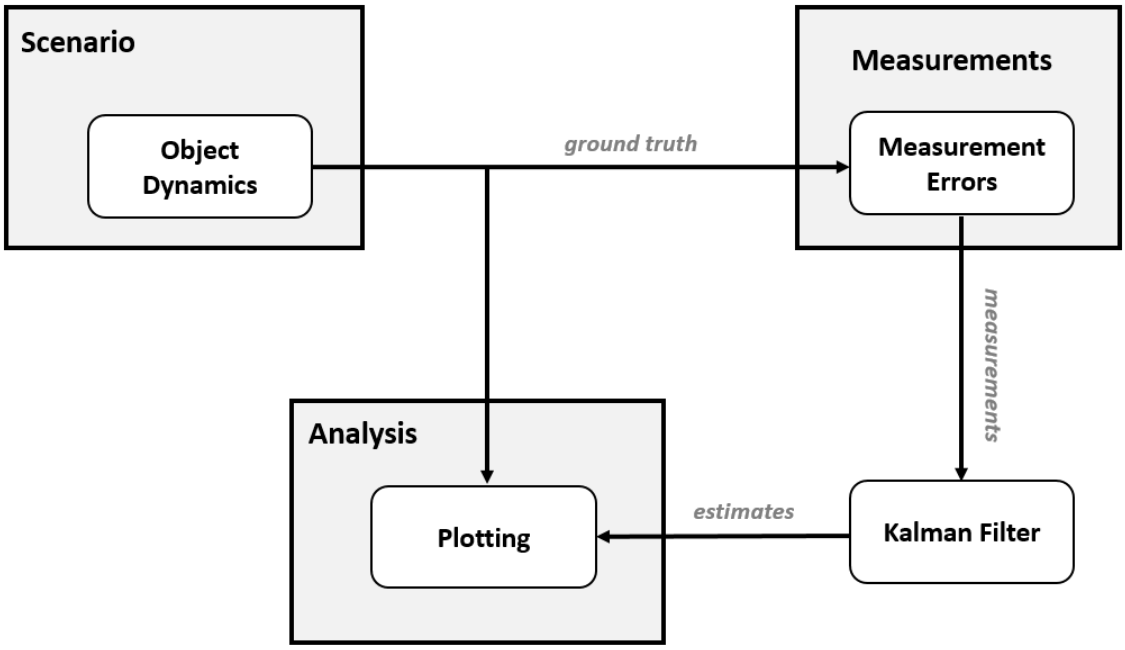

Figure 5: Example 5 flow chart.

1. First, we create the scenario (the ground truth) using *scenario_5* function. For this example, the scenario is a series of equal integers since the actual building height doesn't change.

   In order to change the number of iterations, modify the `params["n"]` value in the *initParams_5* function.

2. The next stage is measurements creation. The function *addNoise* (in folder ***K_common***) adds normally distributed measurement noise to the ground truth. You can see the description of the *addNoise* function in subsection 16.1.

3. The building height is estimated by the Kalman Filter filter (function *KF_1D* in the ***KF_OneDim*** folder).

4. The function *plots_5* plots the estimates vs. the ground truth. The plots also include confidence intervals. You can find an explanation of the confidence interval calculation in Appendix B of the book.

8 Example 6

In this example, we estimate the temperature of the liquid in a tank using Kalman Filter. Although the water temperature is constant, as opposed to the building height, it fluctuates slightly over time due to environmental influence. The temperature fluctuations are the process noise.

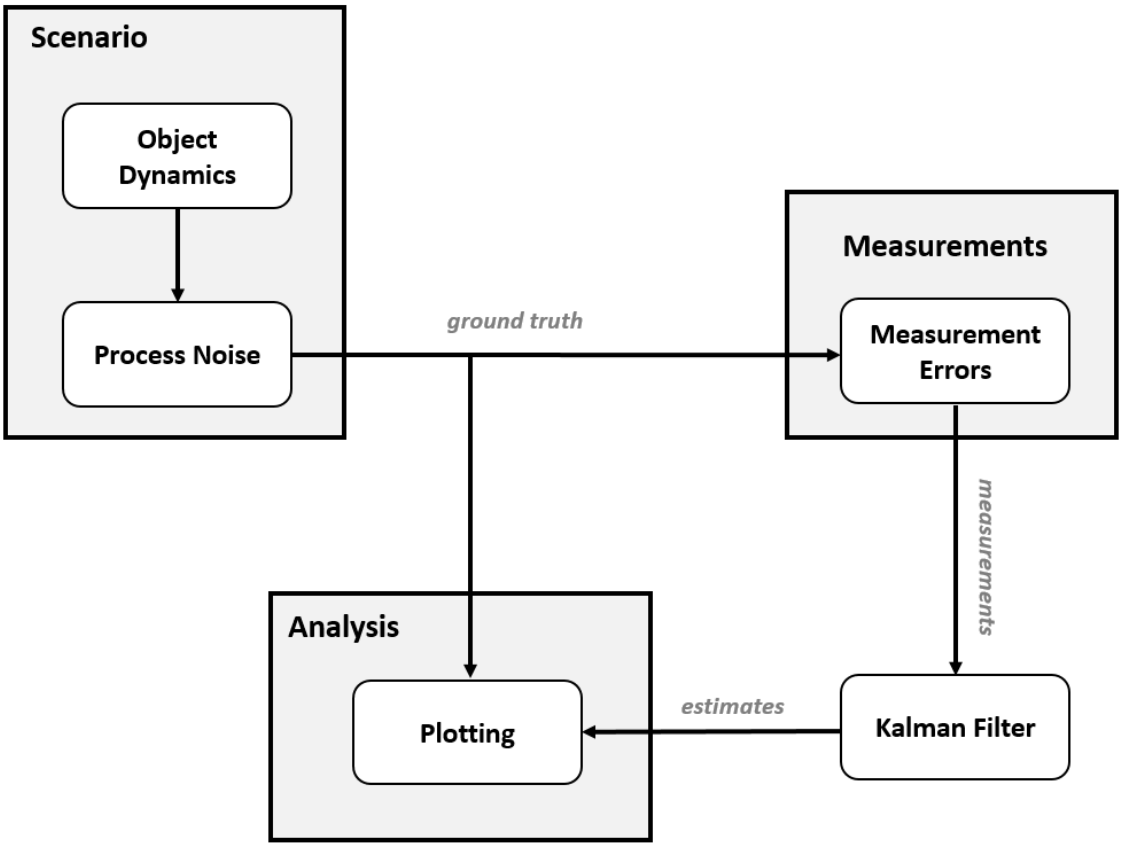

Figure 6: Example 6 flow chart.

1. First, we create the scenario (the ground truth) using *scenario_6* function. For this example, the scenario is a series of equal integers representing the liquid temperature. Then random process noise is added to the liquid temperature using the function *addNoise* (in folder ***KF_common***).

   In order to change the number of iterations, modify the `params["n"]` value in the *initParams_6 function*.

2. The next stage is measurements creation. The function *addNoise* (in folder ***KF_common***) adds normally distributed measurement noise to the ground truth. You can see the description of the *addNoise* function in subsection 16.1.

3. The liquid temperature is estimated by the Kalman Filter filter (function *KF_1D* in the ***KF_OneDim*** folder).

4. The function *plots_6* plots the estimates vs. the ground truth. The plots also include confidence intervals. You can find an explanation of the confidence interval calculation in Appendix B of the book.

9 Examples 7-8

In these examples, we estimate the temperature of a heating liquid in a tank using Kalman Filter.

The flow is similar to Example 6.

10 Example 9

In this example, we estimate the vehicle's location on the XY plane using a multivariate Kalman Filter.

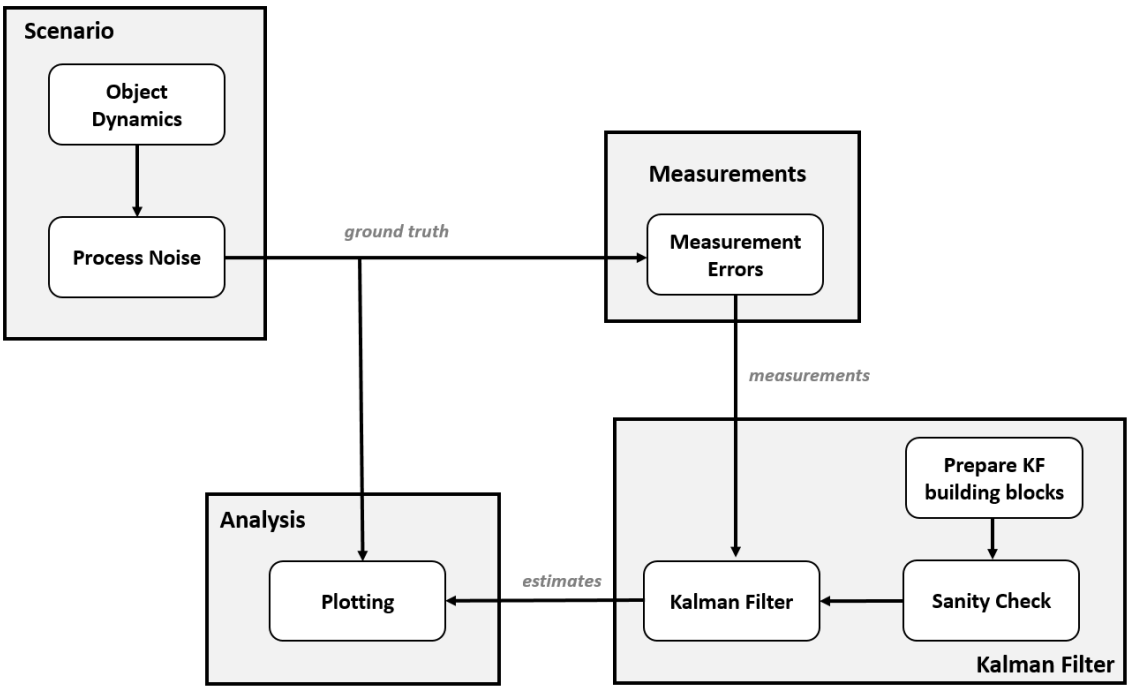

Figure 7: Example 9 flow chart.

10.1 Scenario

The scenario is divided into two parts:

1. The vehicle moves $400m$ in a straight line in the Y direction with a constant speed of $25m/s$.
2. The vehicle turns left, with a turning radius of $300m$, while moving at the same speed. The vehicle experiences acceleration due to the circular motion (angular acceleration). The scenario ends when the vehicle finishes the turning maneuver (quarter circle).

The vehicle trajectory simulation considerations:

- For the simplicity of circular movement generation, we want to have a circle center in the plane origin ($x = 0,\ y = 0$). Therefore, the turning maneuver should start at a point $x = 300m;\ y = 0$, and end at a point $x = 0;\ y = 300m$.
- Consequently, the straight-line part ends where the turning maneuver begins $x = 300m;\ y = 0m$.

The following figure describes the vehicle trajectory.

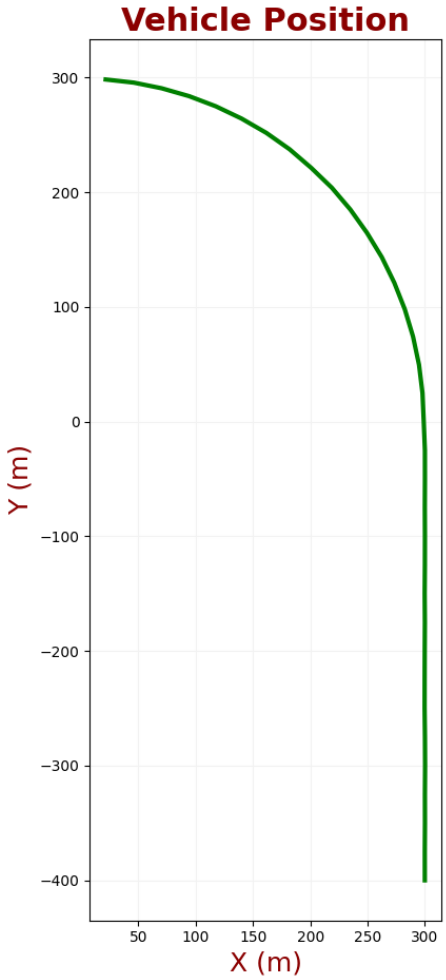

Figure 8: Example 9 vehicle trajectory.

The straight-line segment trajectory generation is straightforward.

The X-axis position is constant.

The Y-axis position equals: $y = v \cdot t$.

Where:

- v is the vehicle velocity
- t is time

Since the velocity is constant, the acceleration is zero. We generate process noise and add it to acceleration; therefore, the actual acceleration is the process noise.

For the turning maneuver trajectory generation, we use circular motion equations.

First, find the angular velocity:

$$\Omega = \frac{v}{R}$$

Where:

- Ω is the angular velocity

- v is the vehicle velocity
- R is the radius

The instant object angle is $\phi = \Omega t$

We should transform the polar coordinates to cartesian coordinates to find the object's position on the XY plane.

The X-axis position of the vehicle equals:

$$x = Rcos(\Omega t)$$

The Y-axis position of the vehicle equals:

$$y = Rsin(\Omega t)$$

The vehicle velocity is a derivative of the position:

$$v_x = \frac{dx}{dt} = \frac{d}{dt}\left(Rcos\left(\Omega t\right)\right) = -\Omega Rsin(\Omega t)$$

$$v_y = \frac{dy}{dt} = \frac{d}{dt}\left(Rsin\left(\Omega t\right)\right) = \Omega Rcos(\Omega t)$$

The vehicle acceleration is a derivative of the velocity:

$$a_x = \frac{dv_x}{dt} = \frac{d}{dt}\left(-\Omega Rsin\left(\Omega t\right)\right) = -\Omega^2 Rcos(\Omega t)$$

$$a_y = \frac{dv_y}{dt} = \frac{d}{dt}\left(\Omega Rcos\left(\Omega t\right)\right) = -\Omega^2 Rsin(\Omega t)$$

We generate process noise and add it to acceleration (a_x and a_y).

10.2 Measurements

We create measurements by adding the measurement noise to the actual vehicle position using the function *addNoise* (in folder ***KF_common***). You can see the description of the *addNoise* function in subsection 16.1.

10.3 Prepare KF building blocks

The following KF matrixes should be created:

- The state transition matrix $\boldsymbol{F}$ is created by the function *motion_F_matrix* (in folder ***KF_MultiDim***).
- The process noise matrix $\boldsymbol{Q}$ is created by the function *motion_Q_matrix* (in folder ***KF_MultiDim***).
- The measurement noise matrix $\boldsymbol{R}$ is created by the function *R_matrix* (in folder ***KF_MultiDim***).
- The observation matrix $\boldsymbol{H}$ is created by the function *H_matrix_9*.

In subsection 18.4, you can find the description of *motion_F_matrix*, *motion_Q_matrix*, and *R_matrix* functions.

10.4 Sanity Check

The function *sanityCheck* (in folder ***KF_MultiDim***) checks the correct dimensions of the $\boldsymbol{z}$ vector, $\boldsymbol{F}$, $\boldsymbol{Q}$, $\boldsymbol{R}$, and $\boldsymbol{H}$ matrices, and KF initiation $\boldsymbol{x}_0$ and $\boldsymbol{P}_0$. If there is any mismatch in the dimensions, the *sanityCheck* function throws an error. You can see the description of the *sanityCheck* function in subsection 18.1.

10.5 Kalman Filtering

The Kalman filtering is performed by the *KF* function (folder ***KF_common***).

10.6 Plotting

The *plots_9* function plots the vehicle position, velocity, and acceleration estimates vs. the ground truth. As well, the plot of the position includes the estimation confidence ellipses. You can find an explanation of the confidence ellipse calculation in Chapter 7 of the book.

11 Example 10

In this example, we estimate the altitude of a rocket. The rocket is equipped with an onboard altimeter that provides altitude measurements. In addition to the altimeter, the rocket is equipped with an accelerometer that measures the rocket's acceleration. The accelerometer measurements serve as a control input to the Kalman Filter.

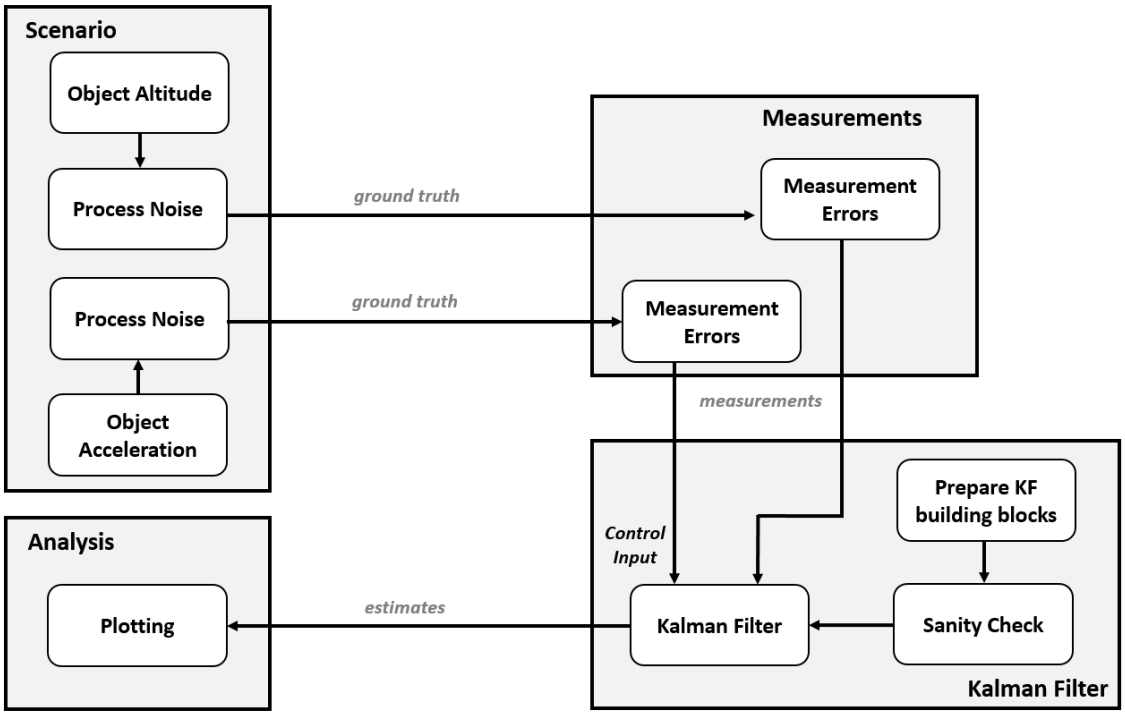

Figure 9: Example 10 flow chart.

- First, we create the scenario (the ground truth) using the *scenario_10* function. The scenario function output is the rocket altitude and acceleration. We add random process noise to altitude and acceleration measurements using the function addNoise (in folder ***KF_common***).

- The next stage is measurements creation. The function *addNoise* (in folder ***KF_common***) adds normally distributed measurement noise to the ground truth. You can see the description of the *addNoise* function in subsection 16.1.
- Create $\boldsymbol{F}$, $\boldsymbol{G}$, $\boldsymbol{Q}$, $\boldsymbol{R}$, and $\boldsymbol{H}$ matrices.
- The function *sanityCheck* (in folder ***KF_MultiDim***) checks the correct dimensions of the $\boldsymbol{z}$ vector, $\boldsymbol{F}$, $\boldsymbol{G}$, $\boldsymbol{Q}$, $\boldsymbol{R}$, and $\boldsymbol{H}$ matrices, and KF initiation $\boldsymbol{x}_0$, $\boldsymbol{u}_0$, and $\boldsymbol{P}_0$. If there is any mismatch in the dimensions, the *sanityCheck* function throws an error. You can see the description of the *sanityCheck* function in subsection 18.1.
- The rocket altitude is estimated by the Kalman Filter filter (function *KF* in the ***MultiDim*** folder).
- The function *plots_10* plots the estimates vs. the ground truth. The plots also include confidence intervals. You can find an explanation of the confidence interval calculation in Appendix B of the book.

12 Example 11

Example 11 scenario is similar to Example 9. scenario. The measurements are made by a radar that measures range and bearing angle. The Kalman filtering is performed by Extended Kalman Filter.

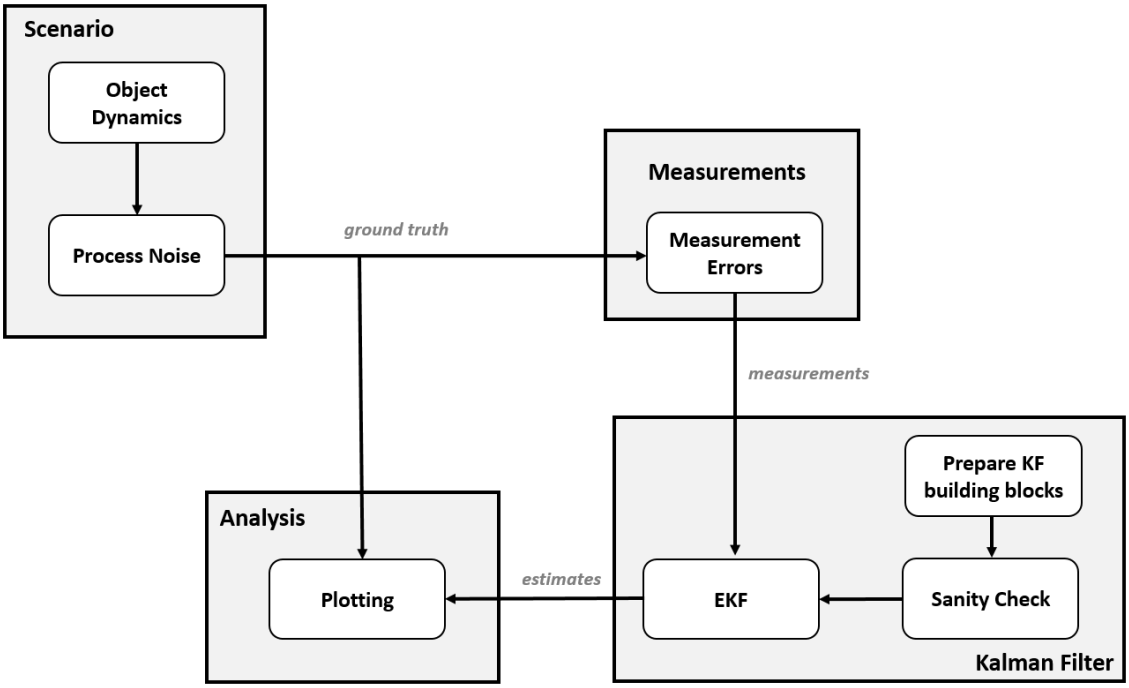

Figure 10: Example 11 flow chart.

12.1 Scenario

The *scenario_11* function creates the scenario. This function uses *scenario_9* for vehicle trajectory generation on the XY plane. Then the vehicle XY coordinates are transferred to polar coordinated (range and bearing angle):

$$R = \sqrt{x^2 + y^2}$$

$$\phi = atan\left(\frac{y}{x}\right)$$

Where:

- R is the vehicle range from the radar
- ϕ is the vehicle azimuthal angle relative to the radar

12.2 Measurements

We create measurements by adding the measurement noise to the actual vehicle range and bearing angle using the function *addNoise* (in folder ***KF_common***). You can see the description of the *addNoise* function in subsection 16.1.

12.3 Prepare KF building blocks

The following KF matrixes should be created:

- The process noise matrix $\boldsymbol{Q}$ is created by the function *motion_Q_matrix* (in folder ***KF_MultiDim***).
- The measurement noise matrix $\boldsymbol{R}$ is created by the function *R_matrix* (in folder ***KF_MultiDim***).

For EKF, the state transition matrix $\boldsymbol{F}$ and the observation matrix $\boldsymbol{H}$ depend on $\boldsymbol{x}$:

$$\boldsymbol{F} = \boldsymbol{f}(\boldsymbol{x})$$

$$\boldsymbol{H} = \boldsymbol{h}(\boldsymbol{x})$$

The $\boldsymbol{F}$ and $\boldsymbol{H}$ matrices and their Jacobians $\left(\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}}, \frac{\partial \boldsymbol{h}(\boldsymbol{x})}{\partial \boldsymbol{x}}\right)$ are created dynamically by *F_matrix_11* and *H_matrix_11* functions. The *EKF* function calls these functions on each filter iteration.

Although, at this stage, we don't create $\boldsymbol{F}$ and $\boldsymbol{H}$ matrices, we prepare input arguments (`Fargs`, `Hargs`) for $\boldsymbol{F}$ and $\boldsymbol{H}$ matrices.

12.4 Sanity Check

We get the Jacobians (`dFx`, `dHx`) of the $\boldsymbol{F}$ and $\boldsymbol{H}$ matrices at the filter initiation point for the sanity check purpose.

The function *sanityCheck* (in folder ***KF_MultiDim***) checks the correct dimensions of the $\boldsymbol{z}$, $\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}}$, $\boldsymbol{Q}$, $\boldsymbol{R}$, and $\frac{\partial \boldsymbol{h}(\boldsymbol{x})}{\partial \boldsymbol{x}}$ matrices and EKF initiation $\boldsymbol{x}_0$ and $\boldsymbol{P}_0$. If there is any mismatch in the dimensions, the *sanityCheck* function throws an error. You can see the description of the *sanityCheck* function in subsection 18.1.

12.5 Kalman Filtering

The Kalman filtering is performed by the *EKF* function (folder ***EKF***). The *EKF* function receives input arguments of *F_matrix_11* and *H_matrix_11* functions (`Fargs`, `Hargs`) and their handles (`F_matrix_11`, `H_matrix_11`).

12.6 Plotting

The *plots_11* function plots the vehicle position, velocity, and acceleration estimates vs. the ground truth. As well, the plot of the position includes the estimation confidence ellipses. You can find an explanation of the confidence ellipse calculation in Chapter 7 of the book.

13 Example 12

Example 12 is another example of an EKF application. In this example, we estimate the pendulum angle θ. The flow is similar to Example 11.

The *scenario_12* function creates the pendulum motion simulation according to the mathematical derivations in Appendix E of the book.

14 Example 13

Example 13 is similar to Example 11. The only difference is that the UKF algorithm performs the filtering.

15 Example 14

Example 14 is similar to Example 12. The only difference is that the modified UKF algorithm performs the filtering.

16 'KF_common' folder

The ***KF_common*** folder includes the following auxiliary functions:

- *addNoise*
- *uniform2normal*
- *mat2pyth*
- *confEllipse*
- *plotConfInt*

16.1 addNoise function

This function adds a normally distributed random noise to the input matrix (or vector). The *addNoise* function is used for process noise or measurement noise generation.

The function inputs are:

- `X` - the data without noise (actual system state). The rows of `X` represent different states (for example, x-position and y-position), and the columns of `X` represent states at different time samples.
- `sigma` - the uncertainty

  - If `sigma` is a scalar, the same measurement uncertainty is applied for each state and for each time sample.
  - If `sigma` is a column vector, each element represents the measurement uncertainties for different states, while the same measurement uncertainty is applied for each time sample.
  - If `sigma` is a row vector, each element represents the measurement uncertainties for different time samples, while the same measurement uncertainty is applied for the state.
  - If `sigma` is a matrix, each element represents a unique measurement uncertainty for each element in `X`.

- `params` - parameters structure

In order to create a reproducible random sequence, the random numbers generator uses a seed.

For example, to create a random sequence of 10 random values, we use the following command:

```
import numpy as np
np.random.rand(1,10)
```

This command creates different random values after each execution. We can reproduce the same random sequence by applying a seed before command execution.

```
import numpy as np
seed = np.array([1])
np.random.seed(seed[0])
np.random.rand(1,10)
```

For another seed value, we get other random values.

We can create uniformly distributed random values between 0 and 1 using the command:

```
np.random.rand(1,n)
```

We can create normally distributed random values between -1 and 1 with mean 0 and standard deviation 1 using the command:

```
np.random.randn(1,n)
```

For the book examples, I want to generate the same random values with the same seed using MATLAB and Python. MATLAB and Python generate the same random value sequences with the same seed only for uniform distribution. But for normal distribution, MATLAB and Python generate different random value sequences.

In order to keep the MATLAB and Python code consistent, we generate the random values with uniform distribution, then convert them to a normal distribution using the *uniform2normal* function.

16.2 uniform2normal function

This function converts uniformly distributed random values to normally distributed random values using the inverse transform sampling method.

You can find details on the inverse transform sampling method in Wikipedia.

Note

In real-life applications, there is no need for *addNoise* and *uniform2normal* functions. Just use the 'np.random.randn' command!

16.3 confEllipse function

This function creates 2D confidence ellipses for a set of estimations. You can find an explanation of the confidence ellipse calculation in Chapter 7 of the book.

16.4 plotConfEllipse function

This function plots confidence ellipses.

16.5 plotConfInt function

This function calculates and plots the confidence interval. You can find an explanation of the confidence interval calculation in Appendix B of the book.

17 'KF_OneDim' folder

The ***KF_OneDim*** folder includes functions used in the univariate examples (Examples 1-8). The following table describes the ***KF_OneDim*** folder functions functionality:

Function Name	Functionality	Used in
avgFilter	Cyclic run of Averaging Filter	Example 1
avgStateUpdate	Averaging Filter state update	Example 1
alphaBetaGammaFilter	Cyclic run of $\alpha - \beta - (\gamma)$ filter	Examples 2-4
alphaBetaGammaStateUpdate	State update for the $\alpha - \beta - (\gamma)$ filter	Examples 2-4
staicDynamicModel	Prediction of the next state for the static model	Examples 1, 5-8
motionDynamicModel	Prediction of the next state using the Newton mechanics motion equations	Examples 2-4
KF_1D	Cyclic run of univariate Kalman Filter	Examples 5-8

Table 1: KF_OneDim folder functions.

18 'KF_MultiDim' folder

The ***KF_MultiDim*** folder includes functions used in the multivariate Kalman Filter examples (Examples 9-14).

The ***KF_MultiDim*** folder includes the following functions:

- *sanityCheck*
- *R_matrix*
- *motion_Q_matrix*
- *motion_F_matrix*
- *motion_G_matrix*
- *KF*

18.1 sanityCheck function

The function *sanityCheck* checks the correct dimensions of the $\boldsymbol{z}$ vector, $\boldsymbol{F}$, $\boldsymbol{G}$, $\boldsymbol{U}$, $\boldsymbol{Q}$, $\boldsymbol{R}$, and $\boldsymbol{H}$ matrices, and KF initiation $\boldsymbol{x}_0$, $\boldsymbol{u}_0$, and $\boldsymbol{P}_0$. If there is any mismatch in the dimensions, the *sanityCheck* function throws an error.

The following table defines the correct dimensions.

Term	Name	Dimensions
$\boldsymbol{x}$	State Vector	$n_x \times 1$
$\boldsymbol{z}$	Measurements Vector	$n_z \times 1$
$\boldsymbol{F}$	State Transition Matrix	$n_x \times n_x$
$\boldsymbol{u}$	Input Variable	$n_u \times 1$
$\boldsymbol{G}$	Control Matrix	$n_x \times n_u$
$\boldsymbol{P}$	Estimate Covariance	$n_x \times n_x$
$\boldsymbol{Q}$	Process Noise Covariance	$n_x \times n_x$
$\boldsymbol{R}$	Measurement Covariance	$n_z \times n_z$
$\boldsymbol{H}$	Observation Matrix	$n_z \times n_x$

Table 2: Variables dimensions.

Dimensions notation:

- n_x is a number of states in a state vector
- n_z is a number of measured states
- n_u is a number of elements of the input variable

18.2 R_matrix function

This function creates a measurement noise matrix $\boldsymbol{R}$ (Measurement Covariance). We assume that measurements are uncorrelated, i.e., $\boldsymbol{R}$ is a diagonal matrix.

18.3 motion_Q_matrix function

This function creates a $\boldsymbol{Q}$ matrix (Process Noise Matrix) for the motion dynamic models.

The parameters that define the Process Noise Matrix structure are `strNoiseModel`, `strModel`, and `dim`.

`strModel` can be `'constant'`, `'linear'`, or `'quadratic'`.

`strNoiseModel` can be `'discrete'` or `'continuous'`.

A detailed description of the Process Noise Matrix construction is given in subsection 8.2.2 of the book.

For example, for `strModel` = `'quadratic'`, `strNoiseModel` = `'discrete'` and `dim` = `1`, the $\boldsymbol{Q}$ matrix is:

$$\boldsymbol{Q} = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \sigma_a^2$$

The `dim` variable defines the dimension that scales the $\boldsymbol{Q}$ matrix. For example, if `dim` = `2`:

$$\boldsymbol{Q} = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 \\ \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \sigma_a^2$$

18.4 motion_F_matrix function

This function creates an $\boldsymbol{F}$ matrix (State Transition Matrix) for the motion dynamic models. The parameters that define the State Transition Matrix structure are `strModel` and `dim`.

`strModel` can be `'constant'`, `'linear'`, or `'quadratic'`.

For example, if `strModel` = `'quadratic'` and `dim` = 1, the $\boldsymbol{F}$ matrix is, then:

$$\boldsymbol{F} = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

The `dim` variable defines the dimension that scales the $\boldsymbol{F}$ matrix. For example, if `dim` = `2`:

$$\boldsymbol{F} = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

18.5 motion_G_matrix function

This function creates a $\boldsymbol{G}$ matrix (Control Matrix) for the motion dynamic models. The parameter that defines the Control Matrix structure is `strModel`. The `strModel` can be `'constant'`, `'linear'`, or `'quadratic'`.

For example, for `strModel = 'linear'`, the $\boldsymbol{G}$ matrix is:

$$\boldsymbol{G} = \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix}$$

18.6 KF function

The *KF* function performs a cyclic run of the multivariate linear Kalman Filter.

19 'EKF' folder

The ***EKF*** folder includes the *EKF* function used in the Extended Kalman Filter examples (Examples 11-12). The *KF* function performs a cyclic run of the multivariate Extended Kalman Filter.

20 'UKF' folder

The ***UKF*** folder includes functions used in the Unscented Kalman Filter examples (Examples 13-14). The following table describes the ***UKF*** folder functions functionality:

Function Name	Functionality
UKF	Cyclic run of the multivariate Unscented Kalman Filter.
sigmaPoints	Sigma Points computation.
sigmaWeights	Compute the Sigma Points weights for 'original' or 'modified' algorithms.
sigmaTransform	Sigma Points Transform by computing transformed mean and covariance matrix.

Table 3: UKF folder functions.