UNIVERSITY OF MICHIGAN

AE567

Statistical Inference, Estimation, and Learning

Project 3: Filtering

Deniz Dost

UM ID: 82337583

Dec. 2, 2021
Deniz Dost, 82337583

1. Model

The time-discretized system dynamics for e pendulum with the angle measurements taken at every δ timesteps is governed by the equations below, where the two states are the angle (x_1) and the angular rate (x_2).

$$\Phi(X_{k+1}) = \begin{pmatrix} x_1^{k+1} \\ x_2^{k+1} \end{pmatrix} = \begin{pmatrix} x_1^k + x_2^k \Delta t \\ x_2^k - gsin(x_1^k)\Delta t \end{pmatrix} + q^k$$

Where $q^k \sim \mathcal{N}(0,\mathcal{Q})$

$$\mathcal{Q} = \begin{pmatrix} \frac{q^c \Delta t^3}{3} & \frac{q^c \Delta t^2}{2} \\ \frac{q^c \Delta t^2}{2} & q^c \Delta t \end{pmatrix}, \quad where \quad q_c = 0.1, \quad and \quad \Delta t = 0.01$$

$$h(X_k) = y^{\delta k} = sin(x_1^{\delta k}) + r^{\delta k}$$

Where $r^{\delta k} \sim \mathcal{N}(0,\mathcal{R})$

The dynamics and filtering are considered with the initial condition $(x_1^0, x_2^0) = (1.5,0)$ for 500 time steps. Gaussian filtering to be implemented follows the below equations for prediction

$$m_k^- = \mathbb{E}[X_k \,|\, \mathcal{Y}_{k-1}]$$

$$C_k^- = \mathbb{C}ov[X_k, X_k \,|\, \mathcal{Y}_{k-1}]$$

and update.

$$\mu = \mathbb{E}[Y_k]$$

$$U = \mathbb{C}ov[X_k, Y_k]$$

$$S = \mathbb{C}ov[Y_k, Y_k]$$

$$m_k = m_k^- + US^{-1}(y_k - \mu)$$

$$C_k = C_k^- - US^{-1}U^T$$

1.1. Linearized Extended Kalman Filtering Equations

The linearized system equations are shown below:

$$A_k = \nabla \Phi(X)|_{X_k = m_{k-1}}$$

$$A_k = \begin{bmatrix} 1 & \Delta t \\ -gcos(m_{k-1}^1)\Delta t & 1 \end{bmatrix}$$

$$H_k = \nabla h(X)|_{X_k = m_k^-}$$

$$H_k = \begin{bmatrix} cos(m_k^{1-}) & 0 \end{bmatrix}$$

For the filtering part, the prediction equations are as follows to compute the predicted mean and covariance from the linearized system dynamics:

$$m_k^- \approx \Phi(m_{k-1})$$

$$C_k^- \approx A_k C_{k-1} A_k^T + \Sigma$$

The update equations to compute the updated mean (estimates) and covariance from the predicted states and covariance are as follows:

$$\mu \approx h(m_k^-)$$

$$S \approx H_k C_k^- H_k^T + \Gamma$$

$$U \approx C_k^- H_k^T$$

$$m_k = m_k^- + US^{-1}(y_k - \mu)$$

$$C_k = C_k^- - US^{-1}U^T$$

1.2.Integrals Required for Gaussian Filtering

The integrals for the Gaussian filtering are introduced here with the given equations

Prediction equations computed with the previous state estimate to obtain mean and covariance predictions are

$$m_k^- = \int \Phi(X_{k-1}) \mathcal{N}(X_{k-1}; m_{k-1}, C_{k-1}) dX_{k-1}$$

$$C_k^- = \int (\Phi(X_{k-1}) - m_k^-)(\Phi(X_{k-1}) - m_k^-)^T \mathcal{N}(X_{k-1}; m_{k-1}, C_{k-1}) dX_{k-1} + \Sigma$$

Update equations computed with the predicted mean and covariance to obtain the updated mean (state estimates) and the covariance are as follows

$$\mu = \int h(X_k) \mathcal{N}(X_k; m_k^-, C_k^-) dX_k$$

$$U = \int (X_k - m_k^-)(h(X_k) - \mu)^T \mathcal{N}(X_k; m_k^-, C_k^-) dX_k$$

$$S = \int (h(X_k) - \mu)(h(X_k) - \mu)^T \mathcal{N}(X_k; m_k^-, C_k^-) dX_k + \Gamma$$

$$m_k = m_k^- + US^{-1}(y_k - \mu)$$

$$C_k = C_k^- - US^{-1}U^T$$

2. Gaussian Filtering

With the system introduced above, four types of filtering are done: Extended, Unscented, Gauss-Hermite (of order 3 and 5) filters. Gauss-Hermite is selected as the last type of filter here since it has been worked on in class material and similarities to the unscented, thus its familiarity. The actual initial state is used as the center of the prior, as the prior mean with standard deviation value as one by setting the prior covariance to the identity matrix. A total of 16 scenarios are run with the combinations of measurement timesteps defined as $\delta \in \{5,10,20,40\}$ and noise variance defined as $R \in \{1,0.1,0.01,0.001\}$ for each filtering algorithm. The system dynamics are simulated and data is created as shown in Figure 1.

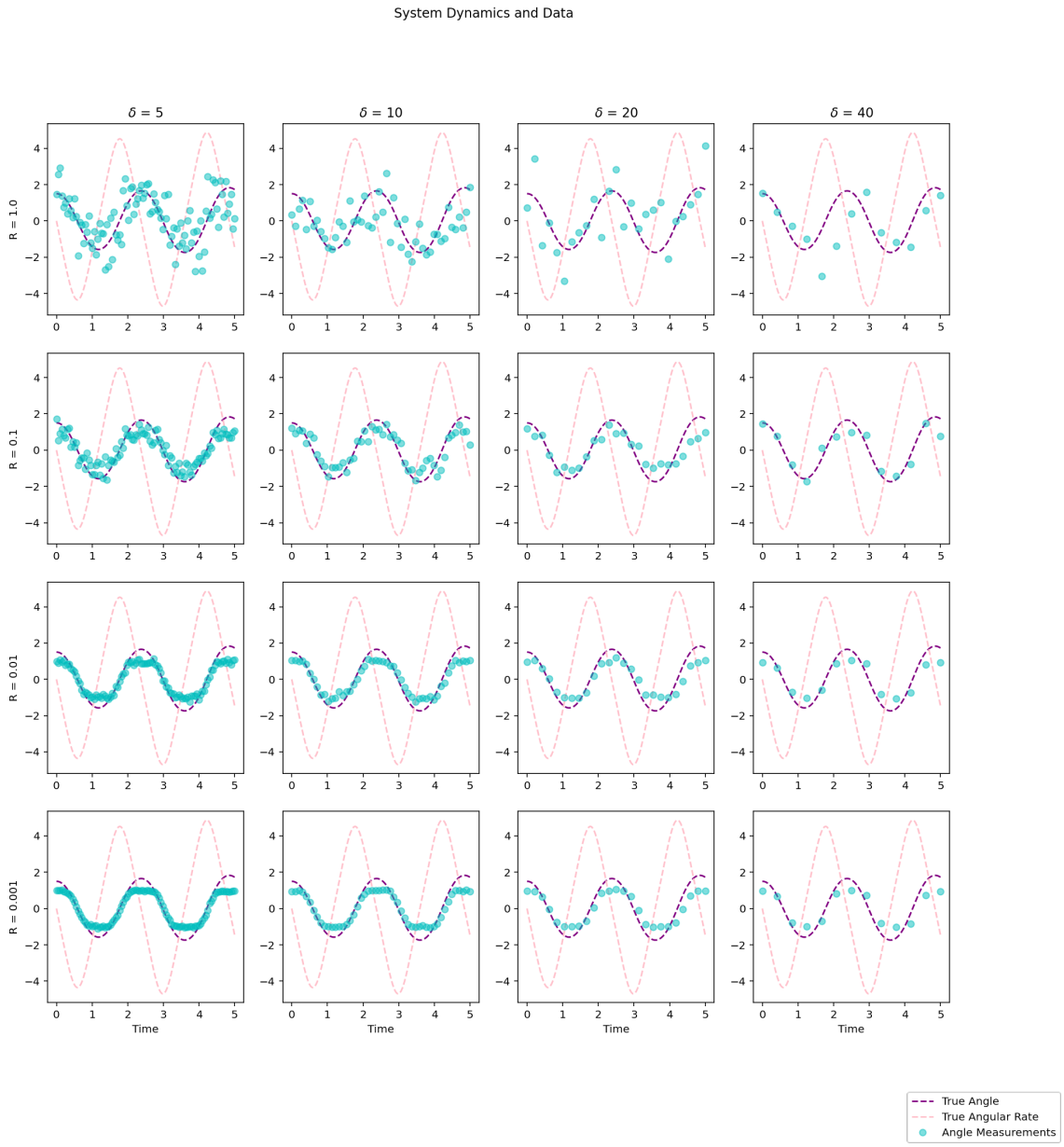
System Dynamics and Data
δ = 5
δ = 10
δ = 20
δ = 40
R = 1.0
R = 0.1
R = 0.01
R = 0.001
Time
True Angle
True Angular Rate
Angle Measurements

Figure 1

2.1.Extended Kalman Filter (EKF)

The EKF is based on integrating the linear integrands by linearizing the system around m_{k-1} for the dynamics and m_k^- for the measurement model with Taylor series approximation by using the Jacobians as shown above. The results with EKF is shown below in Figure 2.

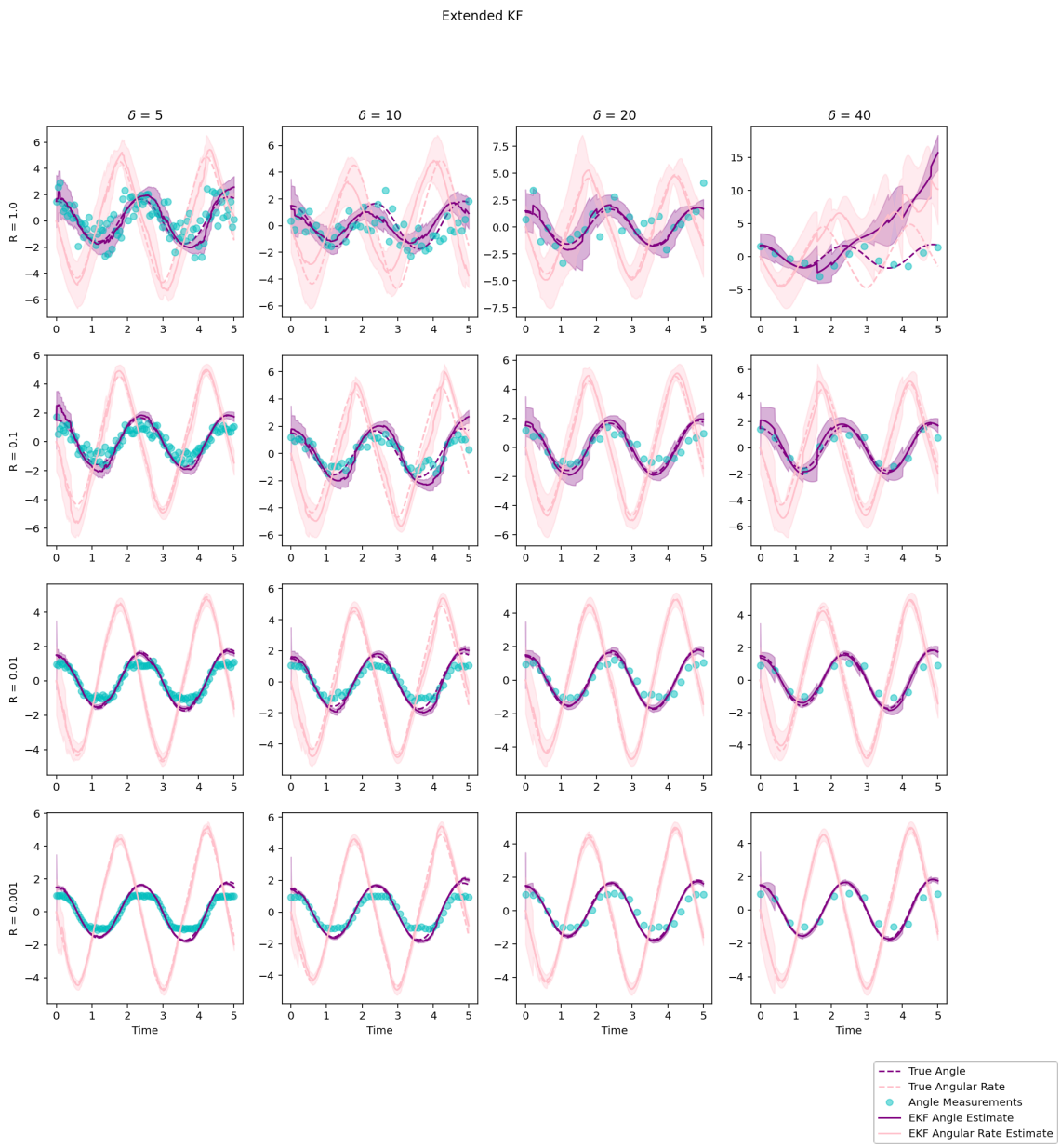

Figure 2

The mean squared error is calculated as shown below and the values for all of the scenarios with EKF are given for both of the states in Table 1.

$$MSE = \sum_{i=1}^{n} = \frac{(\hat{y}_i - y_i)^2}{n}$$

		Mean Squared Error	Measurement Timesteps			
		State	5	10	20	40
Noise Variance	1	1	0.1711	0.4165	0.0958	27.5325
		2	0.7877	2.8797	0.1300	22.8725
	0.1	1	0.0835	0.3000	0.0370	0.1046
		2	0.2272	1.4725	0.1305	0.3963
	0.01	1	0.0064	0.0523	0.0055	0.0123
		2	0.0158	0.2829	0.0205	0.0531
	0.001	1	0.0042	0.0135	0.0039	0.0033
		2	0.0372	0.0937	0.0173	0.0216

Table 1

2.2.Unscented Kalman Filter (UKF)

In order to evaluate the integrals approximately, quadrature rules are used.

$$\int h(X)dX \approx \sum_{i=0}^{2d} h\left(X^{(i)}\right) w^{(i)}$$

Where $w^{(i)}$ is the weight of each ith point, $x^{(i)}$. When integration is considered with respect to weights u in a standard Gaussian measure

$$\int h(X)p(X)dX = \int h(\mu + \Sigma u)\mathcal{N}(u\,|\,0,I)du = \int \tilde{h}(u)\mathcal{N}(u\,|\,0,I)du$$

Where u is transformed to X as

$$X = \mu + \sqrt{\Sigma}u$$

UKF uses $2d+1$ quadrature rule and it uses two types of weights: for mean and covariance. There is the additional point to the elements of the symmetric set $[M]$ at the origin, $u^{(0)}$.

$$[M] = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} -1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \ldots \right\}, \quad and \quad u^{(0)} \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

The weights for the mean are calculated as

$$w_m^{(i)} = \begin{cases} \frac{\lambda}{d+\lambda}, & i = 0) \\ \frac{1}{2(d+\lambda)}, & i = 1,2,...,2d \end{cases}$$

Whereas the weights for the covariance are calculated as

$$w_c^{(i)} = \begin{cases} \frac{\lambda}{d+\lambda} + (1-\alpha^2+\beta), & i = 0) \\ \frac{1}{2(d+\lambda)}, & i = 1,2,...,2d \end{cases}$$

Where α, β, and κ are free parameters to be chosen from literature and

$$\lambda = \alpha^2(d+\kappa) - d$$

Here are chosen as $\alpha = 0.001$, $\beta = 2$, $\kappa = 0$ from literature.

Then, the integration approximation becomes

$$\int h(X)\mathcal{N}(X\,|\,\mu,\Sigma)dX \approx \sum_{i=0}^{2d} h\left(\mu + \sqrt{\Sigma}\sqrt{d+\lambda}u^{(i)}\right) w^{(i)}$$

The results with the UKF are shown below in Figure 3.

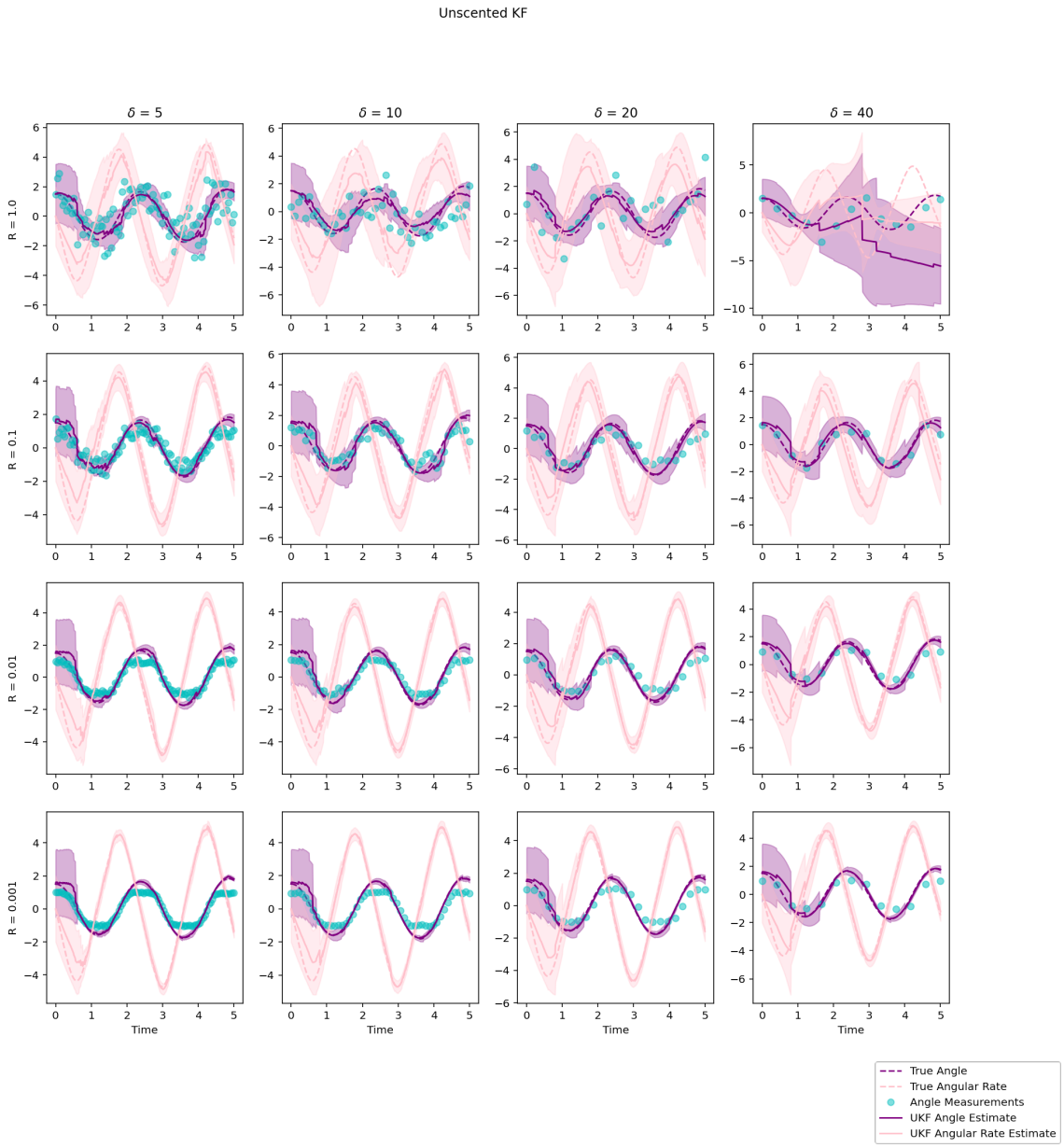
Unscented KF
δ = 5
δ = 10
δ = 20
δ = 40
R = 1.0
R = 0.1
R = 0.01
R = 0.001
Time
True Angle
True Angular Rate
Angle Measurements
UKF Angle Estimate
UKF Angular Rate Estimate

Figure 3

The mean squared error for all of the scenarios with UKF are given for both of the states in Table 2.

		Mean Squared Error	Measurement Timesteps			
		State	5	10	20	40
Noise Variance	1	1	0.1081	0.2052	0.1436	11.3669
		2	0.5219	1.3182	0.9611	8.4126
	0.1	1	0.0682	0.1631	0.0626	0.1158
		2	0.4386	0.4009	0.5053	0.5312
	0.01	1	0.0454	0.0560	0.0592	0.0723
		2	0.2145	0.2064	0.2320	0.2617
	0.001	1	0.0363	0.0406	0.0440	0.0767
		2	0.1925	0.2038	0.2394	0.2309

Table 2

2.3.Gauss-Hermite Kalman Filter of order 3 (GHKF3)

GHKF is discussed to be the most accurate and stable filters among the others introduced so far. However, it also happens to be the most expensive one because of the exponential increase in the evaluation points with respect to the dimensions. The Hermite polynomials are used recursively as:

$$H_{k+1}(x) = xH_k(x) - kH_{k-1}(x)$$

$$H_0(x) = 1$$

$$H_1(x) = x$$

$$H_2(x) = xH_1(x) - 1H_0(x) = x^2 - 1$$

$$H_3(x) = xH_2(x) - 2H_1(x) = x^3 - 3x$$

$$H_4(x) = xH_3(x) - 3H_2(x) = x^4 - 6x^2 + 3$$

$$H_5(x) = xH_4(x) - 4H_3(x) = x^5 - 10x^3 + 15x$$

And it is exact for polynomials up to order $2n-1$ with the nth order Hermite polynomial $H_n(x)$. When the integrands are approximated in an Hermite polynomial expansion and integrated, the multi-dimensional Gaussian distribution can be approximated as

$$\int h(X)\mathcal{N}(X\,|\,\mu,\Sigma)dX \approx \sum_{i=1}^{d} h\left(\mu+\sqrt{\Sigma}u^{(i)}\right)w^{(i)}$$

Here, two orders (of three and five) are implemented where

$$order = 2n-1$$

$$n = \frac{order+1}{2}$$

The results with GHKF with third order are shown below in Figure 4.

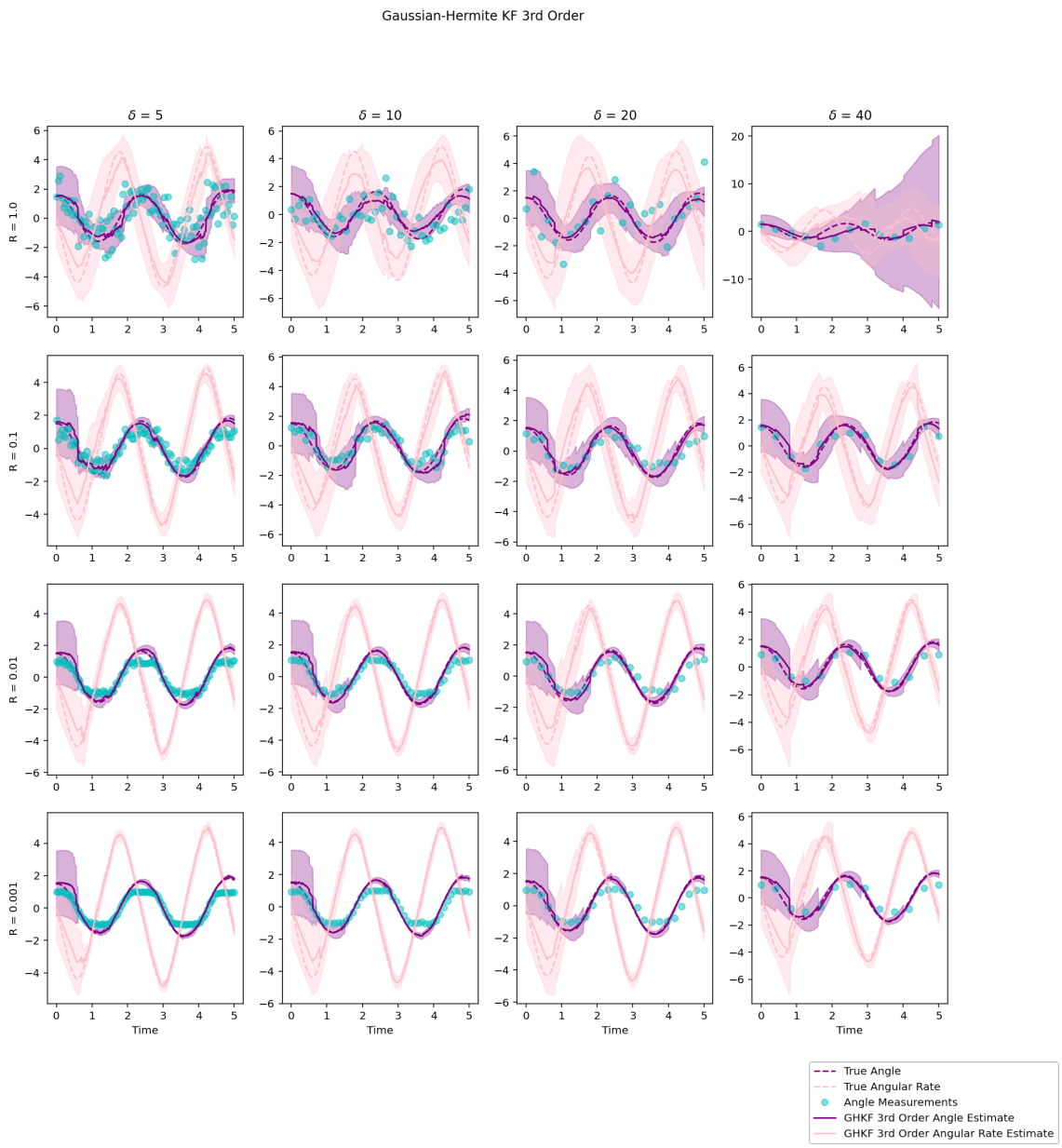

Figure 4

The mean squared error for all of the scenarios with GHKF third order are given for both of the states in Table 3.

		Mean Squared Error	Measurement Timesteps			
		State	5	10	20	40
Noise Variance	1	1	0.0970	0.1722	0.0783	0.7345
		2	0.4526	1.1291	0.6233	6.4787
	0.1	1	0.0621	0.1490	0.0474	0.0711
		2	0.4068	0.4460	0.4668	0.4613
	0.01	1	0.0349	0.0383	0.0431	0.0557
		2	0.1809	0.1706	0.2096	0.2646
	0.001	1	0.0263	0.0266	0.0285	0.0547
		2	0.1641	0.1690	0.2172	0.2347

Table 3

2.4.Gauss-Hermite Kalman Filter of order 5 (GHKF5)

The results with GHKF with fifth order are shown below in Figure 5.

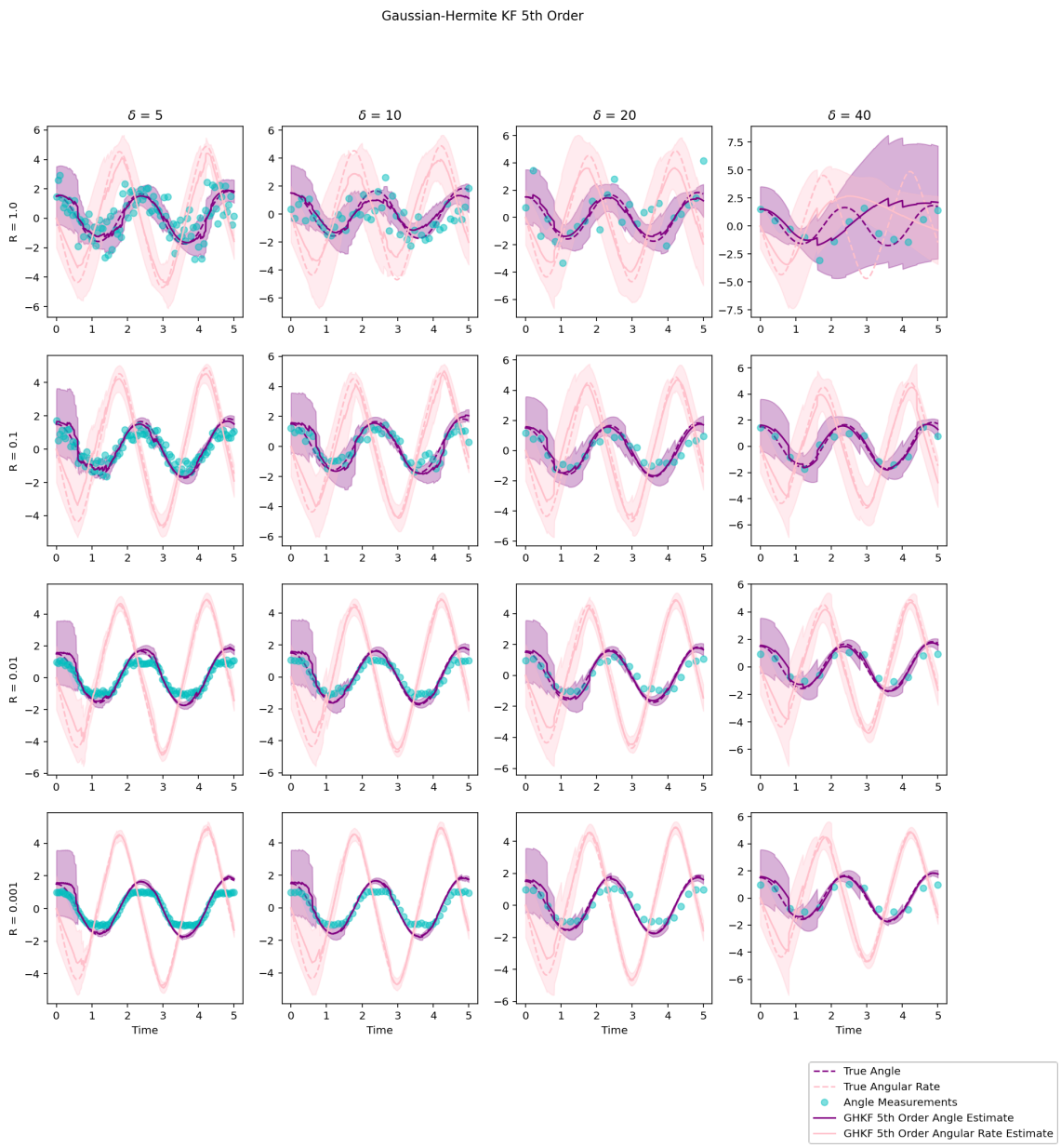

Figure 5

The mean squared error for all of the scenarios with GHKF fifth order are given for both of the states in Table 4.

		Mean Squared Error	Measurement Timesteps			
		State	5	10	20	40
Noise Variance	1	1	0.0982	0.1812	0.0906	3.4058
		2	0.4562	1.1877	0.6689	9.9906
	0.1	1	0.0649	0.1535	0.0527	0.0853
		2	0.3992	0.4197	0.4474	0.4820
	0.01	1	0.0393	0.0467	0.0495	0.0619
		2	0.1865	0.1770	0.2008	0.2564
	0.001	1	0.0306	0.0327	0.0354	0.0639
		2	0.1667	0.1719	0.2123	0.2041

Table 4

In order to compare the computational cost, the run times of each algorithm for all 16 scenarios are given below in Table 5.It can easily be seen that the computational cost drastically increases from EKF to GHKF of fifth order. This was expected considering the algorithm complexities of each filter and evaluation points.

Filter Type	Duration [sec]
EKF	0.12
UKF	0.64
GHKF3	0.73
GHKF5	1.44

Table 5

It can be seen from the mean squared errors and the graphs that as the time interval between the measures and the measurement noise gets bigger, the estimates get worse. The converse also holds, where the measurement noise and time between measurements are small, the estimates are better. This makes sense intuitively where good and more frequent measurements would lead to better estimates. With the lowest quality and least frequent measurements, the filters diverge (EKF almost always

whereas the others sometimes). The narrowing of the 2σ bands with better measurements can also be observed visually. GHKF is the most robust as expected theoretically and this can be observed when different runs are done where it does not diverge whereas the others (mostly EKF) sometimes diverge with cases other than the worst case, i.e. the least frequent and least quality measurements. Given the time is doubled from third to fifth order, and the filter performance on the estimates, third order GHKF can be the choice of filtering.

3. Particle Filtering

There is also another filtering alternative, particle filtering, which can be use when Gaussian approximations are not working well. Here, importing sampling is done to be used as the empirical distribution to have points and weights.Importance sampling was discussed on the previous projects, and it is once again implemented here. It uses importance sampling sequentially. When new data is available, the distribution is updated. The marginal likelihood is

$$\hat{P}(y_n, \mathscr{Y}_{n-1}) = \frac{1}{N}\sum_{i=1}^{N} v^{(i)}\bar{w}_{n-1}^{(i)}$$

$$where \quad w^{(i)} = v^{(i)}\bar{w}_{n-1}^{(i)} \quad and \quad \bar{w}_n^{(i)} = \sum_{i=1}^{N} w_n^{(i)}$$

$$v_n^{(i)} = \frac{P(y_n \,|\, x_n^{(i)})P(x_n^{(i)} \,|\, x_{n-1}^{(i)})}{\pi(x_n^{(i)} \,|\, \mathscr{X}_{n-1}^{(i)})}$$

Filter basically puts more importance with places with greater weights. The samples are got from these areas and use to propagate. Particle filtering for the pendulum is done with the system dynamics as the proposal. If no data is available, filter basically uses the proposal, the system dynamics. The three scenarios chosen for this purpose are:

$$\delta_1 = 40,\ R_1 = 1$$

$$\delta_2 = 10,\ R_2 = 0.1$$

$$\delta_3 = 5,\ R_3 = 0.001$$

which are the worst, medium and best cases respectively.

The particle filter run for the pendulum results in the estimates given below in Figure 6.

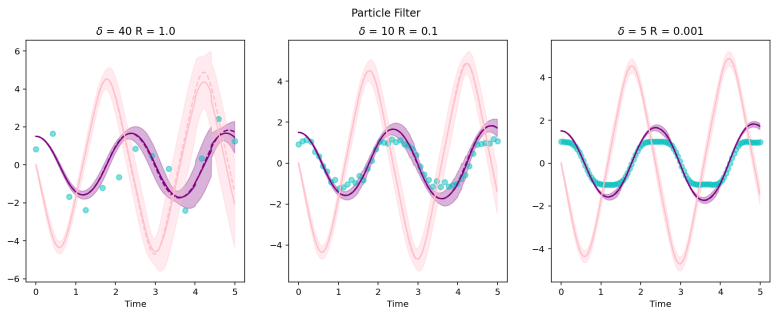

Figure 6

The joint posterior at five equally spaced time instances are shown below in Figure 7 for all of the scenarios as indicated.

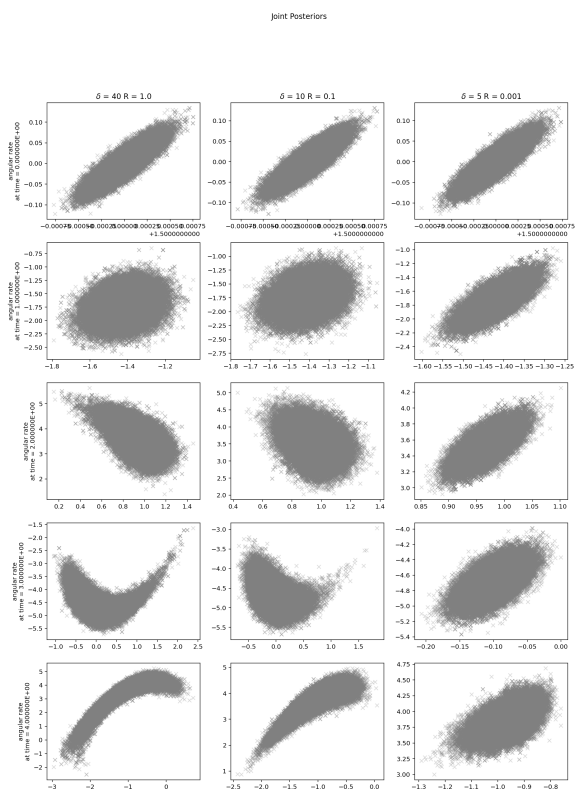

Figure7

The UKF run for the same conditions results in the estimates shown below in Figure 8 and the overlay of UKF approximations at the time instances shown in Figure 7 are given in Figure 9.

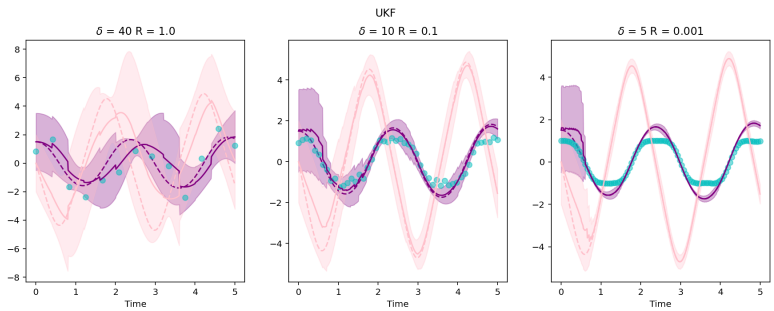

Figure 8

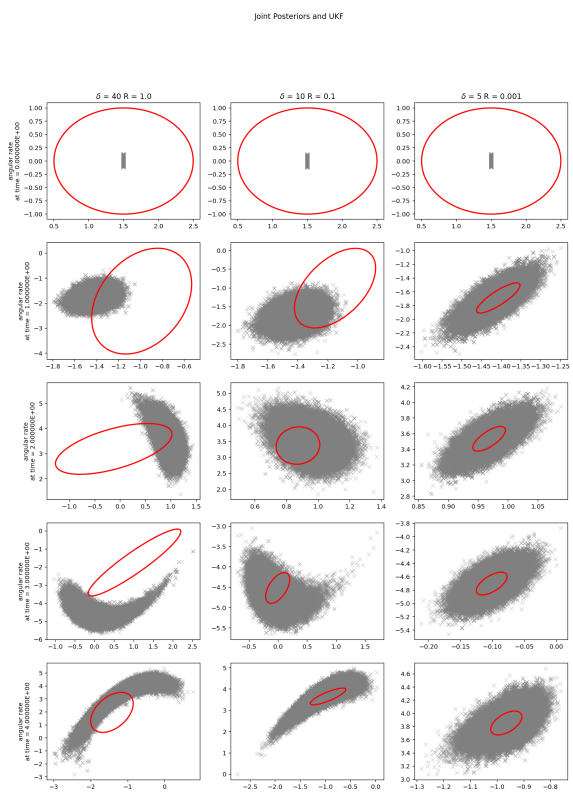

Figure 9

It can be seen that the first row encapsulates the points with the eye matrix initial covariance circle. Then, the last scenario with the most frequent and accurate measurements have the most Gaussian looking posteriors whereas the first one has the least. If all of the filters introduced are run for the fifth time instance, i.e. t=4 seconds, where the joint posterior is non-Gaussian, the moments are as shown in Figure 10. It can be seen that GHKF and UKF is better than EKF. Along different trials, GHKF is the most stable one in this graph. So, this also supports the claim that GHKF is the most robust one among the others.

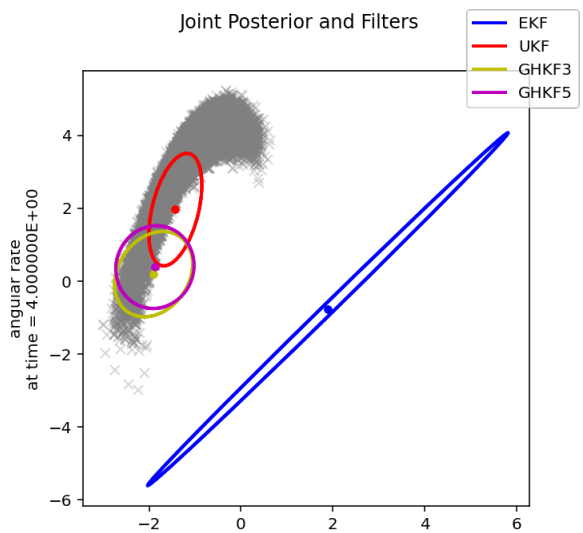

Figure 10

For the last scenario, a performance analysis is done with varying sample numbers. It is expected to have better convergence with increased number of samples. The convergence for the mean and covariance of the PF can be seen in Figure 11.

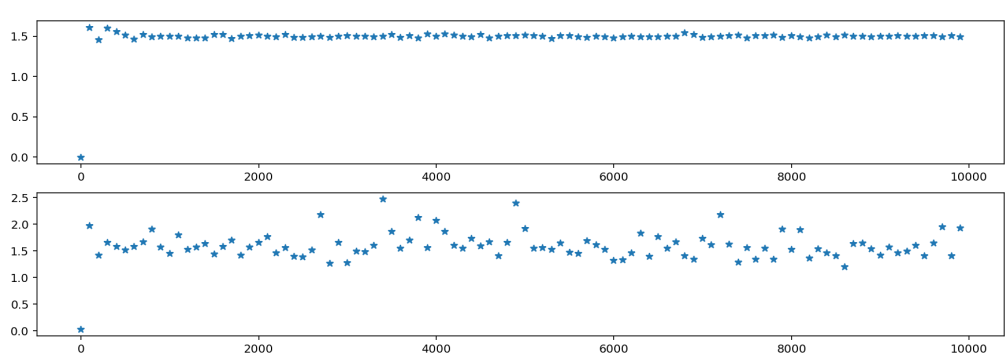

Figure 11

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mpl_toolkits.mplot3d.axes3d, time, numpy, pandas
import numpy.random as rnd
import numpy.linalg as la
import matplotlib.pyplot as plt
import pandas
import matplotlib.mpl_toolkits.mplot3d.axes3d
import sklearn.cross_validation
import sklearn.tree
import sklearn.ensemble
import sklearn.metrics
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.cross_validation import train_test_split
from sklearn.metrics import roc_curve, auc
```

Features

```
def gradientplot(ax, df, target_column, bins=10):
    ax = ax.ravel()[:-1]
    for i, c in enumerate(df.columns):
        ax[i].hist(df[c], bins=bins)
        ax[i].set_title(c)
        ax[i].set_xticks([])
        ax[i].set_yticks([])
    for ii in range(len(df.columns), len(ax)):
        ax[ii].hist(df[df[target_column].isnull()][c], bins=bins)
        ax[ii].set_title("%s|nan" % c)
        ax[ii].set_xticks([])
        ax[ii].set_yticks([])
    print_text = "Blue: %s, Green: %s" % (str(df[target_column].unique()[0])[:20],
                                          str(df[target_column].unique()[1])[:20])
    if len(df[target_column].unique()) > 2:
        print_text = "Too many values in %s" % target_column
    ax[-1].text(0.5, 0.5, print_text, ha='center')
    ax[-1].set_xticks([])
    ax[-1].set_yticks([])
    return ax[:len(df.columns)]
```

2d

```
def linear_predictor(df, mean, std, target, debug=):
    '''Generate a linear predictor based on the data and the mean and the stdev
    df: a dataframe
    mean: a series of means, indexed by column
    std: a series of stdevs, indexed by column
    target: the column to predict

    Returns:
    lin_df: the linear model
    lin_mean: the mean of the model
    lin_std: the stdev of the model, normalized against mean=0 stdev=1
    lin_predictor: a function that takes a dataframe and returns a predictor dataseries

    Warning:
    lin_mean, lin_std: a dataframe indexed by column
    lin_predictor: a function that takes a dataframe

    print_debug = lin_debug(debug, 'linear')
    lin_df = df[mean.index].dropna() * mean[df[mean.index].dropna().columns] +
                                       std[df[mean.index].dropna().columns]
    print_debug(lin_df.describe())

    lin_mean = lin_df.mean(), lin_df.mean(axis=1).mean()
    lin_std = lin_df.std(), lin_df.std(axis=1).mean()

    return lin_df, lin_mean, lin_std, lin_predictor
```

2d

```
def linear_predictor_step(df, mean, std, target, debug=):
    '''Generate a linear predictor based on the data and the mean and the stdev
    df: a dataframe
    mean: a series of means, indexed by column
    std: a series of stdevs, indexed by column
    target: the column to predict

    Returns:
    lin_df: the linear model
    lin_mean: the mean of the model
    lin_std: the stdev of the model, normalized against mean=0 stdev=1
    lin_predictor: a function that takes a dataframe and returns a predictor dataseries

    Warning:
    lin_mean, lin_std: a dataframe indexed by column
    lin_predictor: a function that takes a dataframe

    print_debug = lin_debug(debug, 'linear')
    lin_df = df[mean.index].dropna() * mean[df[mean.index].dropna().columns] +
                                       std[df[mean.index].dropna().columns]
    print_debug(lin_df.describe())

    lin_mean = lin_df.mean(), lin_df.mean(axis=1).mean()
    lin_std = lin_df.std(), lin_df.std(axis=1).mean()

    return lin_df, lin_mean, lin_std, lin_predictor
```

2d

```
def binary_predictor(df, mean, std, target, bins=10, print_bins, debug=):
    '''Generate a binary predictor based on the data and the mean and the stdev
    df: a dataframe
    mean: a series of means, indexed by column
    std: a series of stdevs, indexed by column
    target: the column to predict
    bins: the number of bins to use when generating the predictor
    print_bins: if, along with debug, to print the bins
    debug: if to print debug messages

    Returns:
    bin_df: the binary model
    bin_mean: the mean of the model
    bin_std: the stdev of the model, normalized against mean=0 stdev=1
    bin_predictor: a function that takes a dataframe and returns a predictor dataseries

    print_debug = lin_debug(debug, 'binary')
    bin_df = df[mean.index].dropna() * mean[df[mean.index].dropna().columns] +
                                       std[df[mean.index].dropna().columns]
    print_debug(bin_df.describe())

    bin_mean = bin_df.mean(), bin_df.mean(axis=1).mean()
    bin_std = bin_df.std(), bin_df.std(axis=1).mean()

    print_bins = lin_debug(print_bins, 'bins')
    for c in bin_df.columns:
        bin_df[c] = pandas.cut(bin_df[c], bins, retbins=True)[1][:-1]
        print_bins(c, bin_df[c].unique())

    bin_predictor = lambda df: df[bin_df.columns].isin(bin_df[bin_df.columns])

    return bin_df, bin_mean, bin_std, bin_predictor
```

2d

```
def binary(df):
    '''Binarize df
    df: a dataframe

    Returns:
    bin_df: a binarized version of df

    Warning:
    bin_df = df.copy()
    for c in df.columns:
        bin_df[c] = df[c].isnull()
        if len(df[c].unique()) == 2:
            bin_df[c] = df[c] == df[c].unique()[0]
        elif len(df[c].unique()) > 2:
            for u in df[c].unique():
                bin_df[c + '|' + str(u)] = df[c] == u
    return bin_df
```

2d

```
def binary_predictor_step(df, mean, std, target, debug=):
    '''Generate a binary predictor based on the data and the mean and the stdev
    df: a dataframe
    mean: a series of means, indexed by column
    std: a series of stdevs, indexed by column
    target: the column to predict
    debug: if to print debug messages

    Returns:
    bin_df: the binary model
    bin_mean: the mean of the model
    bin_std: the stdev of the model, normalized against mean=0 stdev=1
    bin_predictor: a function that takes a dataframe and returns a predictor dataseries

    print_debug = lin_debug(debug, 'binary')
    bin_df = df[mean.index].dropna() * mean[df[mean.index].dropna().columns] +
                                       std[df[mean.index].dropna().columns]
    print_debug(bin_df.describe())

    bin_mean = bin_df.mean(), bin_df.mean(axis=1).mean()
    bin_std = bin_df.std(), bin_df.std(axis=1).mean()

    bin_predictor = lambda df: df[bin_df.columns].isin(bin_df[bin_df.columns])

    return bin_df, bin_mean, bin_std, bin_predictor
```

2d

```
def binary_predictor_step(df, mean, std, target, bins=10, print_bins, debug=):
    '''Generate a binary predictor based on the data and the mean and the stdev
    df: a dataframe
    mean: a series of means, indexed by column
    std: a series of stdevs, indexed by column
    target: the column to predict
    bins: the number of bins to use when generating the predictor
    print_bins: if, along with debug, to print the bins
    debug: if to print debug messages

    Returns:
    bin_df: the binary model
    bin_mean: the mean of the model
    bin_std: the stdev of the model, normalized against mean=0 stdev=1
    bin_predictor: a function that takes a dataframe and returns a predictor dataseries

    print_debug = lin_debug(debug, 'binary')
    bin_df = df[mean.index].dropna() * mean[df[mean.index].dropna().columns] +
                                       std[df[mean.index].dropna().columns]
    print_debug(bin_df.describe())

    print_bins = lin_debug(print_bins, 'bins')
    for c in bin_df.columns:
        bin_df[c] = pandas.cut(bin_df[c], bins, retbins=True)[1][:-1]
        print_bins(c, bin_df[c].unique())

    bin_predictor = lambda df: df[bin_df.columns].isin(bin_df[bin_df.columns])

    return bin_df, bin_mean, bin_std, bin_predictor
```

2d

```
def binarize(df):
    '''Binarize df
    df: a dataframe

    Returns:
    bin_df: a binarized version of df

    Warning:
    bin_df = df.copy()
    for c in df.columns:
        bin_df[c] = df[c].isnull()
        if len(df[c].unique()) == 2:
            bin_df[c] = df[c] == df[c].unique()[0]
        elif len(df[c].unique()) > 2:
            for u in df[c].unique():
                bin_df[c + '|' + str(u)] = df[c] == u
    return bin_df
```

2d

```
def binary(df):
    '''Binarize df
    df: a dataframe

    Returns:
    bin_df: a binarized version of df

    Warning:
    bin_df = df.copy()
    for c in df.columns:
        bin_df[c] = df[c].isnull()
        if len(df[c].unique()) == 2:
            bin_df[c] = df[c] == df[c].unique()[0]
        elif len(df[c].unique()) > 2:
            for u in df[c].unique():
                bin_df[c + '|' + str(u)] = df[c] == u
    return bin_df
```

In [28]:

```
    Mean[ii, i] = np.dot(weights[:, ii], samples[:, i, ii])
    Var[ii, i] = np.exp(np.dot(weights[:, ii], (samples[:, i, ii] - np.tile(Mean[ii, i], (N, 1)))**2))
    return means, stds


def step(proxy, propgdf, current_samples, current_weights, likelihood, data, propagator, R):
    """
    Propagate a particle filter

    @param:[in] prop            - proposal function (current_state, data)
    @param:[in] propgdf         - proposal function logpdf
    @param:[in] current_samples - ensemble of samples
    @param:[in] current_weights - ensemble of weights
    @param:[in] likelihood      - function to evaluate the log likelihood (samples, data)
    @param:[in] data            - observation
    @param:[in] propagator      - dynamics logpdf

    Evercurse samples and weights after assimilating the data
    """
    new_samples = prop(current_samples, data)
    new_weights = likelihood(new_samples, data, R) + propagator(new_samples, current_samples) - \
                  propgdf(new_samples, current_samples, data)
    new_weights = np.exp(new_weights) * current_weights
    new_weights = new_weights / np.sum(new_weights)
    return new_samples, new_weights


def particle_filter(data, prior_mean, prior_cov,
                    prop, propgdf, likelihood, propagator, delta,R,
                    nsamples=100, resampling_threshold_frac=0.2):
    """Particle Filter

    Inputs
    ------
    data: (nsteps, m) array of data points, N is the time index, m is the dimensionality of the data
    prior_mean: (d), prior mean
    prior_cov: (d, d), prior mean
    Nsamples: integer, number of samples in the empirical distribution
    resampling_threshold_frac: float between 0 and 1 indicating to resample when effective sample size below frac of nsamples

    Returns
    -------
    samples: (nsamples, d, nsteps)
    weights: (nsamples, nsteps)
    eff: (nsamples), effective sample size

    Notes
    -----
    For documentation of prop, propgdf, likelihood, and propagator -- see the step function
    """
    d = prior_mean.shape[0]
    nsteps = data.shape[0]
    # Allocate memory
    samples = np.zeros((nsamples, d, nsteps+1))
    weights = np.zeros((nsamples, nsteps+1))
    eff = np.zeros((nsteps+1)) # keep track of effective sample size at each step
    # Generate initial samples from the prior
    L = np.linalg.cholesky(prior_cov)
    samples[:, :, 0] = np.tile(prior_mean, (nsamples, 1))+ np.dot(L, np.random.randn(d, nsamples),).T
    weights[:, 0] = 1.0 / nsamples # all weights are equal because of independent sampling from prior
    eff[0] = nsamples
    resamp_thresh = int(nsamples * resampling_threshold_frac)
    for ii in range(0, nsteps):
        if ii%delta==1:
            samples[:, :, ii], weights[:, ii] = step(prop,  propgdf, samples[:, :, ii-1], weights[:, ii-1],
                                                     likelihood, data[ii-1,:], propagator,R)

        else:
            samples[:,:,ii] = prop(samples[:, :, ii-1], data[ii-1,:])
            weights[:,ii]=weights[:,ii-1]

        # compute the effective sample size
        eff[ii] = 1.0 / np.sum(weights[:, ii]**2)

        # if ii % 50 == 0:
            #print('eff = ', ii, eff[ii])
        #print('eff/mean = ', eff[ii], mean(ii, :))

        # resample if effective sample size is below threshold
        if eff[ii] < resamp_threshold:
            samples[:, :, ii], weights[:, ii] = resample(nsamples, samples[:, :, ii], weights[:, ii])
    return samples, weights, eff
```

In [30]:

```
# Process noise
proc_var=0.1
proc_mat = np.zeros((1,2))
proc_mat[0, 0] = proc_var/3.0*dt**3
proc_mat[0, 1] = proc_var/2.0*dt**2
proc_mat[1, 0] = proc_var/2.0*dt**2
proc_mat[1, 1] = proc_var*dt
proc_mat_inv = np.linalg.pinv(proc_mat)
lprocc = np.linalg.cholesky(proc_mat)

def proposal(current_state, data=None, dt=dt):
    """ Bootstrap Particle Filter the proposal is the dynamics!"""

    if current_state.ndim == 1:
        return pendulum_dyn(current_state, dt=dt) + np.dot(lprocc, np.random.randn(2))
    else:
        nsamples = current_state.shape[0]
        return pendulum_dyn(current_state, dt=dt) + np.dot(lprocc, np.random.randn(2, nsamples).T)

def proposal_logpdf(current, previous, data=None, proc_var=0.1):
    """ Bootstrap Particle Filter: the proposal is the dynamics"""
    next1 = pendulum_dyn(previous, dt=dt)
    delta = next1 - current
    if current.ndim == 1:
        return -0.5 * np.dot(delta, np.dot(proc_mat_inv, delta))
    else:
        return -0.5 * np.sum(delta * np.dot(delta, proc_mat_inv.T), axis=1)

def likelihood(state, data, noise_var):
    """Gaussian Likelihood through nonlinear model"""
    dpropose = observe(state)
    delta = dpropose - data
    if state.ndim == 1:
        return -0.5 * np.dot(delta, delta) / noise_var
    else:
        return -0.5 * np.sum(delta * delta, axis=1) / noise_var
```

In [34]:

```
custom_lines = [Line2D([0], [0], color='b', lw=2),
                Line2D([0], [0], color='r', lw=2),
                Line2D([0], [0], color='y', lw=2),
                Line2D([0], [0], color='m', lw=2)]
```

In [36]:

```
# random particle
x0 = np.array([1.5, 0])
Nsteps = 500
N = Nsteps
qc = 0.1
deltat = 0.01
dt = deltat
delta_array = np.array([45,10,5])
Marray = np.array([1, 0.1, 0.001])
Q = np.array([[(qc*deltat**3)/3 , (qc*deltat**2)/2 ],
              [(qc*deltat**2)/2 ,  qc*deltat]])

proc_var = qc
xlim = [-6, 6]
ylim = [-6, 6]

fig0, ap0 = plt.subplots(1,3,figsize=(15,5))
fig1, ap1 = plt.subplots(1,3,figsize=(15,5))
fig2, ap2 = plt.subplots(1,3,figsize=(15,20))
fig3, ap3 = plt.subplots(1,3,figsize=(15,20))
fig4, ap4 = plt.subplots(1,3,figsize=(5,5))


for i in range(3):
    delta = deltaArray[i]
    R = Marray[i]
    obs = np.zeros((Nsteps, 2))
    true[0, :] = x0
    times = np.arange(0, Nsteps*dt, dt)
    data = np.zeros((Nsteps-1, 1))
    na = R
    noise_var = na*na

    proc_mat = Q
    proc_mat_inv = np.linalg.pinv(proc_mat)
    lprocc = np.linalg.cholesky(proc_mat)

    for ii in range(1, Nsteps):
        true[ii, :] = pendulum_dyn(true[ii-1, :],dt=dt)
        data[ii-1] = observe(true[ii, :]) + np.random.randn()*R

    T = N*deltat
    T = np.linspace(0, T, int(np.ceil(N/delta)))

    # PF
    prior_mean = true[0, :] # start at the truth -- this is obviously not possible in reality
    prior_cov = Q #np.eye(2) # identity covariance
    samples, weights, eff = particle_filter(data, prior_mean, prior_cov,
                                            proposal, proposal_logpdf, likelihood, proposal_logpdf,delta,R,
                                            nsamples=10000, resampling_threshold_frac=0.1)

    MEANPF, STDPF= compute_mean_std(samples, weights)
    MERANPF = MEANPF.T
    j=0
    ap0[i].plot(times, true[:, 0],'--', color='purple', label='True Angle')
    ap0[i].plot(times, true[:, 1],'--', color='pink', label='True Angular Rate')
    ty = np.linspace(0, T, int(np.ceil(N/delta)))
    ap1[i].plot(ty, data[:-deltaj, 0], 'o', alpha=0.5, label='Angle Measurements')
    ap1[i].plot(ty, MERANPF[:,0], color='purple', label='UKF Angular Rate Estimate')
    ap1[i].plot(times, MERANPF[:,1], color='pink', label='UKF Angular Rate Estimate')
    ap1[i].fill_between(times, MERANPF[:,0] - 2 * STDPF[:, 0], MERANPF[:,0] + 2*STDPF[:, 0],
                        color='purple', alpha=0.2)
    ap1[i].fill_between(times, MERANPF[:,1] - 2 * STDPF[:, 1],  MERANPF[:,1] + 2*STDPF[:, 1],
                        color='pink', alpha=.3)
    if j == 0: #ap.set(deltaArray[i])
        ap0[i].set_xlabel('Time', fontsize=10)
    if j == 0:
        ap0[i].set_title('$\deltat = ' +str(delta) + ' R = ' +str(R)

    # UKF
    prior_mean = x0.T # prior mean
    prior_cov = np.eye(2) # prior covariance
    proc_cov = Q # lw=1 * np.eye(2) # process noise covariance
    meas_cov = R # lw=1 * np.eye(R.shape[0]) #measurement noise covariance
    ukf = UKF(dt=dt,
    meanpost, covpost = UKF(R, yuse, proc_cov, meas_cov, prior_mean, prior_cov, delta)
    std_dera = get_std(covpost) #extract the standard deviations at all the states
    T = N*deltat
    T = np.linspace(0, T, N+1)
    ap1[i].plot(times, true[:, 0],'--', color='purple', label='True Angle')
    ap1[i].plot(times, true[:, 1],'--', color='pink', label='True Angular Rate')
    ap1[i].plot(ty, yuse, 'oo', alpha=0.5, label='Angle Measurements')
    ap1[i].plot(t, meanpost[0,:], color='purple', label='UKF Angle Estimate')
    ap1[i].plot(t, meanpost[1,:], color='pink', label='UKF Angular Rate Estimate')
    ap1[i].fill_between(t, meanpost[0,:] - 2 * std_dera[:, 0],  meanpost[0,:] + 2*std_dera[:, 0],
                        color='purple', alpha=0.3)
    ap1[i].fill_between(t, meanpost[1,:] - 2 * std_dera[:, 1],  meanpost[1,:] + 2*std_dera[:, 1],
                        color='pink', alpha=0.3)
    if j == 0: #ap.set(deltaArray[i])
        ap1[i].set_xlabel('Time', fontsize=10)
    if j == 0:
        ap1[i].set_title('$\deltat = ' +str(delta) + ' R = ' +str(R)

    # PART 2
    d, N = MEANPF.shape
    kk = 0
    for ii in range(N):
        if ii % 100 == 0:
            Nsamples = 100000
            weightsii = weights[:, ii]
            samplesii = samples[:, :, ii]
            s, w = resample(Nsamples, samplesii, weightsii) # resample to obtain equal weights
            xspace = np.linspace(xlim[0], xlim[1],100)
            yspace = np.linspace(ylim[0], ylim[1],100)
            XX, YY = np.meshgrid(xspace, yspace)
            positions = np.vstack([XX.ravel(), YY.ravel()])
            values = np.vstack([s[:, 0], s[:, 1]])
            kernel = kde(values, bw=[0.1, 0.1])
            f = np.reshape(kernel(positions).T, XX.shape)
            ap0.plot(s[:, 0], s[:, 1], 'x', color='grey', alpha=0.3,zorder=0)

            if i == 0:
                ap0[kk,i].set_ylabel('angular rate\nat time = {:5E}'.format(times[ii]))
            kk = kk+1
    kk = 0
    for i in range(N):
        if i % 100 == 0:
            Nsamples = 100000
            weightsii = weights[:, ii]
            samplesii = samples[:, :, ii]
            s, w = resample(Nsamples, samplesii, weightsii) # resample to obtain equal weights

            xspace = np.linspace(xlim[0], xlim[1],100)
            yspace = np.linspace(ylim[0], ylim[1],100)
            XX, YY = np.meshgrid(xspace, yspace)

            positions = np.vstack([XX.ravel(), YY.ravel()])
            values = np.vstack([s[:, 0], s[:, 1]])
            kernel = stats.gaussian_kde(values) # kernel density estimate to get contours
            f = np.reshape(kernel(positions).T, XX.shape)
            ap1[kk,i].plot(s[:, 0], s[:, 1], 'x', color='grey', alpha=0.3,zorder=0)

            if i == 0:
                ap1[kk,i].set_ylabel('angular rate\nat time = {:5E}'.format(times[ii]))
            kk = kk+1
    if j == 0:
        ap0[j,i].set_title('$\deltat = ' +str(delta) + ' R = ' +str(R)

    if j == 0:
        ap0[j,i].set_title('$\deltat = ' +str(delta) + ' R = ' +str(R)

    # PART 3
    if i == 0:
        #EKF
        meanpostEKF, covpostEKF = kalman_filter(R, yuse, proc_cov, meas_cov, prior_mean, prior_cov, deltat, delta)
        std_deveEKF = get_std(covpostEKF) #extract the standard deviations at all the states

        #UKF
        meanpostUKF, covpostUKF = UKF(R, yuse, proc_cov, meas_cov, prior_mean, prior_cov, delta)
        std_deveUKF = get_std(covpostUKF) #extract the standard deviations at all the states

        #GHKF 1
        order = 3
        orderd = int(order+1)/2
        meanpostGHKF1, covpostGHKF1 = GHKF(R, yuse, proc_cov, meas_cov, prior_mean, prior_cov, delta)
        std_devaGHKF1 = get_std(covpostGHKF1) #extract the standard deviations at all the states

        #GHKF 5
        order = 5
        orderd = int(order+1)/2
        meanpostGHKF5, covpostGHKF5 = GHKF(R, yuse, proc_cov, meas_cov, prior_mean, prior_cov, delta)
        std_devaGHKF5 = get_std(covpostGHKF5) #extract the standard deviations at all the states

    for ii in range(N):
        if ii == 400:
            Nsamples = 100000
            weightsii = weights[:, ii]
            samplesii = samples[:, :, ii]
            s, w = resample(Nsamples, samplesii, weightsii) # resample to obtain equal weights
            xspace = np.linspace(xlim[0], xlim[1],100)
            yspace = np.linspace(ylim[0], ylim[1],100)
            XX, YY = np.meshgrid(xspace, yspace)
            positions = np.vstack([XX.ravel(), YY.ravel()])
            values = np.vstack([s[:, 0], s[:, 1]])
            kernel = stats.gaussian_kde(values) # kernel density estimate to get contours
            f = np.reshape(kernel(positions).T, XX.shape)
            ap9.plot(s[:, 0], s[:, 1], 'x', color='grey', alpha=0.3,zorder=0)


            lambda_EKF, vEKF = np.linalg.eig(covpostEKF[:,:,ii])
            lambda_EKF = np.sqrt(lambda_EKF)
            ellEKF = Ellipse(xy=(meanpostEKF[0,ii], meanpostEKF[1,ii]),
                             width=lambda_EKF[0]*1*2, height=lambda_EKF[1]*1*2,
                             angle=np.rad2deg(np.arccos(vEKF[0, 0])), edgecolor='b', fc='None', lw=2)

            lambda_UKF, vUKF = np.linalg.eig(covpostUKF[:,:,ii])
            lambda_UKF = np.sqrt(lambda_UKF)
            ellUKF = Ellipse(xy=(meanpostUKF[0,ii], meanpostUKF[1,ii]),
                             width=lambda_UKF[0]*1*2, height=lambda_UKF[1]*1*2,
                             angle=np.rad2deg(np.arccos(vUKF[0, 0])), edgecolor='r', fc='None', lw=2)


            lambda_GHKF1, vGHKF1 = np.linalg.eig(covpostGHKF1[:,:,ii])
            lambda_GHKF1 = np.sqrt(lambda_GHKF1)
            ellGHKF1 = Ellipse(xy=(meanpostGHKF1[0,ii], meanpostGHKF1[1,ii]),
                               width=lambda_GHKF1[0]*1*2, height=lambda_GHKF1[1]*1*2,
                               angle=np.rad2deg(np.arccos(vGHKF1[0, 0])), edgecolor='y', fc='None', lw=2)

            lambda_GHKF5, vGHKF5 = np.linalg.eig(covpostGHKF5[:,:,ii])
            lambda_GHKF5 = np.sqrt(lambda_GHKF5)
            ellGHKF5 = Ellipse(xy=(meanpostGHKF5[0,ii], meanpostGHKF5[1,ii]),
                               width=lambda_GHKF5[0]*1*2, height=lambda_GHKF5[1]*1*2,
                               angle=np.rad2deg(np.arccos(vGHKF5[0, 0])), edgecolor='m', fc='None', lw=2)

            ellEKF.set_facecolor('none')
            ellUKF.set_facecolor('none')
            ellGHKF1.set_facecolor('none')
            ellGHKF5.set_facecolor('none')
            ap9.add_patch(ellEKF)
            ap9.add_patch(ellUKF)
            ap9.add_patch(ellGHKF1)
            ap9.add_patch(ellGHKF5)
            ap9.plot(meanpostEKF[0,ii],meanpostEKF[1,ii],'o',color = 'b',markersize=4,label='EKF')
            ap9.plot(meanpostUKF[0,ii],meanpostUKF[1,ii],'o',color = 'r',markersize=4,label='UKF')
            ap9.plot(meanpostGHKF1[0,ii],meanpostGHKF1[1,ii],'o',color = 'y',markersize=4,label='GHKF1')
            ap9.plot(meanpostGHKF5[0,ii],meanpostGHKF5[1,ii],'o',color = 'm',markersize=4,label='GHKF5')


            ap9.set_ylabel('angular rate\nat time = {:5E}'.format(times[ii]))

    ap9[i].legend(bbox_to_anchor=(2,0), loc='lower right',
                  bbox_transform=fig.transFigure, ncol=1)
    ap4[i].legend(bbox_to_anchor=(2,0), loc='lower right',
                  bbox_transform=fig.transFigure, ncol=1)
    fig9.legend(custom_lines, ['EKF', 'UKF', 'GHKF1', 'GHKF5'])

    # fig9.legend()
    fig9.suptitle('Particle Filter',fontsize=12)
    fig9.subplots_adjust(hspace=1.2)
    fig9.suptitle('Joint Posteriors',fontsize=12)
    fig8.suptitle('Joint Posterior and UKF',fontsize=12)
    fig8.suptitle('Joint Posterior and Filters',fontsize=12)
```

Out[36]:

```
Text(0.5, 0.98, 'Joint Posterior and Filters')
```

In [99]:

```
iterator = 1
num_iterations = 100
meas_array = np.zeros(num_iterations)
cov_array = np.zeros(num_iterations)
for i in range(num_iterations):
    nsamples = 10*iterator[-1 + 1
    samples, weights, eff = particle_filter(data, prior_mean, prior_cov,
                                            proposal, proposal_logpdf, likelihood, proposal_logpdf,delta,R,
                                            nsamples=nsamples, resampling_threshold_frac=0.1)
    means, stds = compute_mean_std(samples, weights)
    x, w = resample(nsamples, samples[:,:,499], weights[:,499])
    cov = np.cov(x[:,0],x[:,1])
    mean_array[i] = means[0][0]
    norm_cov = np.linalg.norm(np.cov(samples[0,0,:],weights[0,:]))
    cov_Array[i] = norm_cov
fig10, ap10 = plt.subplots(2,1,figsize=(15,5))
t1 = np.arange(1, iterator-1 + 1, 1)[1:]
ap10[0].plot(t1,mean_array[1:])
ap10[1].plot(t1,cov_array[1:])
```

Out[99]:

```
[<matplotlib.lines.Line2D at 0x2d03bcd0>]
```

In [ ]: