# Project 3: Filtering

In this project, we are asked to run Gaussian filters on the dynamics of a time-discretized simple pendulum defined as follows:

$$\begin{pmatrix} x_1^{k+1} \\ x_2^{k+1} \end{pmatrix} = \begin{pmatrix} x_1^{k+1} + x_2^k \Delta t \\ x_2^{k+1} - g sin(x_1^k) \Delta t \end{pmatrix} + \mathbf{q}^{\mathrm{k}}$$

$$y^{\delta k} = \sin\left(x_1^{\delta k}\right) + r^{\delta k}$$

Where, $x_1$ is the angle and $x_2$ is the angular rate and $y$ is actual data observed at every $\delta k$ timestep. Here $\mathbf{q}^{\mathrm{k}}$ is the process noise at every timestep k and $r^{\delta k}$ is the measurement noise at every $\delta k$ timestep.

Given that $\mathbf{q}^{\mathrm{k}}$ is sampled from a Gaussian $\mathbf{q}^{\mathrm{k}} \sim \mathcal{N}(0, \mathrm{Q})$ and $r^{\delta k} \sim \mathcal{N}(0, \mathrm{R})$ where Q is the covariance matrix for process covariance and R is the noise variance, defined as follows:

$$\mathbf{Q} = \begin{pmatrix} \frac{\mathrm{q^c} \Delta t^3}{3} & \frac{\mathrm{q^c} \Delta t^2}{2} \\ \frac{\mathrm{q^c} \Delta t^2}{2} & q^c \Delta t \end{pmatrix}$$

And $R \in \{1, 0.1, 0.01, 0.001\}$, $\mathrm{q^c} = 0.1$. The question asks to use a standard Gaussian centered at initial condition with standard deviation as the prior. This means the prior mean and prior covariance are defined as follows:

$$prior\ mean = \begin{pmatrix} x_1^0 \\ x_2^0 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 0 \end{pmatrix}$$

$$prior\ covariance = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

## 1. Model

Questions:

1. Linearized extended Kalman filtering equations for the given dynamic system.
2. Integrals required for Gaussian filtering.

Handwritten answers to these two questions are found on the next page.

1. LINEARIZED EXTENDED KALMAN FILTERING EQUATIONS:

The general form of the time-discrete nonlinear systems to be,

$$X_{k+1} = \phi(X_k ; \theta_1) + \xi \qquad \xi \sim N(0, \Sigma)$$
$$Y_k = h(X_k ; \theta_2) + \eta \qquad \eta \sim N(0, \Gamma)$$

Comparing the given system with these equations:

$$\phi(X_k ; \theta_1) = \begin{pmatrix} x_1^k + x_2^k \Delta t \\ x_2^k - g \sin x_1^k \Delta t \end{pmatrix}$$

$$h(X_k ; \theta_2) = \sin(x_1^{\delta k})$$

$$\Sigma = Q \quad \text{and} \quad \Gamma = R$$

Using these, the gradients required to make a linear approximation of the model using Taylor series are found to be as follows:

$$A_k = \nabla\phi(X_k)\big|_{x_k = m_{k-1}} = \begin{bmatrix} 1 & \Delta t \\ -g\cos(x_1^k)\Delta t & 1 \end{bmatrix}_{x_k = m_{k-1}}$$

$$H_k = \nabla h(x_k)\big|_{x_k = m_{k-1}} = \begin{bmatrix} \cos(x_1, \theta_k) \\ 0 \end{bmatrix}_{x_k = m_{k-1}}$$

Substitute these gradients to obtain the linear approximation of the model using Taylor's Series as:

$$\phi(x_{k-1}) = \phi(m_{k-1}) + A_k(x_k - m_{k-1}) + H.O.T$$

$$h(x_k) = h(m_k^-) + H_k(x_k - m_k^-) + H.O.T$$

The prediction for this system can now be determined as,

$$m_k^- \equiv E[x_k | y_{k-1}] = E[\phi(x_{k-1}) + \xi \mid y_{k-1}]$$
$$\approx \phi(m_{k-1})$$

$$C_k^- \equiv \text{cov}[x_k, x_k | y_{k-1}] = \text{cov}[\phi(x_{k-1}) + \xi, \phi(x_{k-1}) + \xi \mid y_{k-1}]$$

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

Now, for the update,

$$\mu = E[Y_k] = E[h(X_k) + \eta] \approx h(m_k^-)$$

$$U \equiv \mathrm{Cov}[X_k, Y_k] = \mathrm{Cov}[X_k, h(X_k) + \eta]$$

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

$$S \equiv \mathrm{Cov}[Y_k, Y_k] = \mathrm{Cov}[h(X_k) + \eta, h(X_k) + \eta]$$

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

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

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

## 2. Integrals For Gaussian Filtering

The equation for the prediction step -

$$m_k^- = \int \phi(x_{k-1}) N(x_{k-1}, m_{k-1}, C_{k-1}) \, dx_{k-1}$$

Given -

$$\phi(x_k) = \begin{bmatrix} x_1^k + x_2^k \Delta t \\ x_2^k - g \sin x_1^k \Delta t \end{bmatrix} \quad \text{and}$$

$$E[x_1^k] = \int x_1^{k-1} N(x_{k-1}, m_{k-1}, C_{k-1}) \, dx_{k-1}$$

Plugging this into the above equation -

$$m_k^- = \begin{bmatrix} E[x_1^{k-1}] + E[x_2^{k-1}] \Delta t \\ E[x_2^{k-1}] - g \Delta t \int \sin(x_1^{k-1}) N(x_{k-1}, m_{k-1}, C_{k-1}) \, dx_{k-1} \end{bmatrix}$$

For covariance -

$$C_k^- = \int (\phi(x_{k-1}) - m_k^-)(\phi(x_{k-1}) - m_k^-)^T N(x_{k-1}; m_{k-1}, C_{k-1}) \, dx_{k-1} + \Sigma$$

Here, $\phi(X_{k-1}) - m_k^-$ is computed as,

$$= \begin{bmatrix} x_1^{k-1} + x_2^{k-1}\Delta t - E[x_1^{k-1}] - E[x_2^{k-1}]\Delta t \\ x_2^{k-1} - g\sin x_1^{k-1}\Delta t - E[x_2^{k-1}] + g\Delta t \int \sin(x_1^{k-1}) \\ \qquad N(X_{k-1}, m_{k-1}, C_{k-1}) \\ \qquad dX_{k-1} \end{bmatrix}$$

For update -

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

$$= \int \sin(x_1^k) N(X_k; m_k^-, C_k^-) dX_k$$

$$U' = \int (X_k - m_k^-)(h(X_k) - \mu)^T N(X_k; m_k^-, C_k^-) dX_k$$

Here - $X_k - m_k^-$

$$= \begin{bmatrix} x_1^k - E[x_1^{k-1}] - E[x_2^{k-1}]\Delta t \\ x_2^k - E[x_2^{k-1}] + g\Delta t \int \sin(x_1^{k-1}) \\ \qquad N(X_{k-1}, m_{k-1}, C_{k-1}) \\ \qquad dX_{k-1} \end{bmatrix}$$

Finally -

$$S = \int (h(x_k) - \mu)(h(x_k) - \mu)^T N(x_k; m_k^-, C_k^-) \, dx_k + \Gamma$$

$$= \int (\sin(x_1^k) - \mu)^2 N(x_k; m_k^-, C_k^-) \, dx_k + \Gamma$$

The equations obtained for $m_k^-$, $C_k^-$, $\mu$, $U$ and $S$ are the integrals required for Gaussian filtering.

## 2. Gaussian Filtering

1. **Extended Kalman Filter**: A Kalman filter is used to filter linear models. An extended Kalman filter is used to filter nonlinear models by approximation. The ExKF choses to make a linear approximation via linearization using a Taylor's series approximation around the mean of the weight being integrated against. Using this idea, we linearize the dynamics model around $m_{k-1}$ (mean at state k-1) and the measurement model around $m_k^-$ (prior mean). This idea is used to implement this filter on the given model.

   **Programming approach and implementation:**
   To implement the extended Kalman filter I constructed the following functions based on the functions given in the kalman_filter.ipynb file:

   - generate_truth (): This function essentially constructs a truth model for the dynamics and measurement models. It takes as input the N timesteps (here N = 500), initial condition, delta timestep for the measurement model, the process covariance, and the noise variance. This function uses a for loop to obtain the true values for 500 iterations of the dynamics model as well as the those of the measurement model. This function returns the values $\begin{pmatrix} x_1^{k+1} \\ x_2^{k+1} \end{pmatrix}$ and $y^{\delta k}$ values. These values are used to compare the filtered values to the true values graphically.

   - ekf_prediction_step (): This function takes as arguments the gradient of the dynamics model, the transpose matrix of the gradient of the dynamics model, the prior mean and covariance and the process covariance matrix. It returns the predicted mean and predicted covariance which will later be passed into the update function. The following equations were used to obtain the predicted mean and covariance:

   $$Predicted\ Mean: m_k^- = \mathbf{\Phi}(m_{k-1}) = \begin{pmatrix} m_1^{k+1} + m_2^k \Delta t \\ m_2^{k+1} - gsin(m_1^k)\Delta t \end{pmatrix}$$

   $$Predicted\ Covariance: C_k^- = \mathrm{A_k} C_{k-1} A_k^T + \Sigma$$

   Here $\mathrm{A_k}$ is the gradient of the dynamics model at $X_k = m_{k-1}$ given by:

   $$\mathrm{A_k} = \nabla \mathbf{\Phi}(X_k)|_{X_k = m_{k-1}} = \begin{pmatrix} 1 & \Delta t \\ -gcos(m_1^k)\Delta t & 1 \end{pmatrix}$$

   - ekf_update_step (): This function takes as arguments true data values of the measurement model constructed in the generate_truth () function, the gradient of the measurement model, the transpose of the gradient of the measurement model, the predicted mean and predicted covariance from ekf_prediction_step () and the noise

variance. It returns the updated mean and updated covariance using the following equations:

$$Updated\ mean: m_k = m_k^- + US^{-1}(y_k - \mu)$$
$$Updated\ Covariance: C_k = C_k^- - US^{-1}U^T$$

Here $U$ and $S$ are defined as:

$$U = C_k^-\ H_k^T$$
$$S = H_k C_k^-\ H_k^T + \Gamma$$

And $H_k$ is the gradient of the measurement model at $X_k = m_k^-$ given by:

$$H_k = \nabla \mathbf{h}(X_k)|_{X_k = m_k^-} = \begin{pmatrix} \cos\left(m_1^{\delta k}\right) \\ 0 \end{pmatrix}$$

- extended_kalman_filter (): This function obtains and stores the mean and covariance after implementing the filter on the given model. This function takes as arguments the prior mean and prior covariance, truth data from the measurement model and noise variance. The Q matrix which is the process covariance matrix is constructed within the function as specified in the question. 500 iterations were run to obtain the required number of samples. The function ekf_prediction_step () is used to obtain the predicted mean and covariance. Then an if statement is used to verify if the measurement values obtained are numbers or not. This is because, the measurement data is collected at every $\delta$ timestep only and the values in between may or may not be defined as a number. On satisfying the condition that the measurement data is a number, the function ekf_update_step () is called to obtain the updated mean and covariance which are then stored as the filtered mean and covariance. In the case where the if condition is not satisfied, the previous mean and covariance are stored as the filtered mean and covariance.

The extended Kalman filter is run for 500 iterations for 16 combinations of R and $\delta$ where, $R \in \{1, 0.1, 0.01, 0.001\}$ and $\delta \in \{5, 10, 20, 40\}$. Figure 1 represents the state estimates $\pm 2\sigma$ for the extended Kalman Filter.

In Figure 1, the dotted red and blue lines represent the true values of the two states and the solid lines represent the filtered values. The black dots are the measurement values. The solid line filtered values seem to be coinciding with the dotted line truth values very closely. As R value decreases, the filtered values are becoming less smooth. Changes in $\delta$ values do not seem to make much of a difference in the results. The variance is also decreasing for decreasing value of R. The filtered values are very close to the true values here. So, it makes sense that variance is also significantly decreasing.

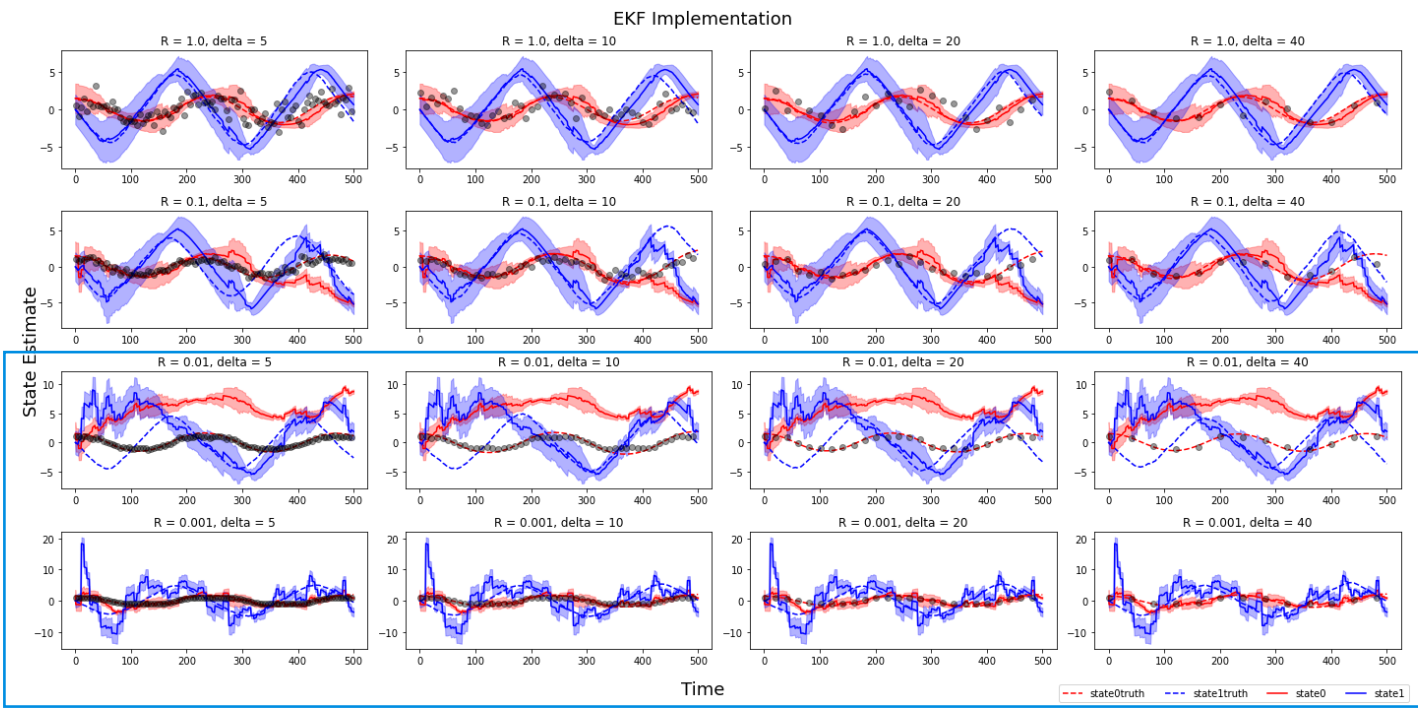



Figure 1: Extended Kalman Filter Implementation on a simple pendulum dynamics model

**Results from code:**

R = 1.0, delta = 5

Time Elapsed for Extended Kalman Filter = 0.015621662139892578

Mean squared error for state 1= 0.01047236529255327

Mean squared error for state 2= 0.0

R = 1.0, delta = 10

Time Elapsed for Extended Kalman Filter = 0.0

Mean squared error for state 1= 0.012755623576003444

Mean squared error for state 2= 0.0

R = 1.0, delta = 20

Time Elapsed for Extended Kalman Filter = 0.0

Mean squared error for state 1= 0.005156982866565686

Mean squared error for state 2= 0.0

R = 1.0, delta = 40

Time Elapsed for Extended Kalman Filter = 0.015627145767211914

Mean squared error for state 1= 0.0038847408434432578

Mean squared error for state 2= 0.0

R = 0.1, delta = 5

Time Elapsed for Extended Kalman Filter = 0.0

Mean squared error for state 1= 0.0033897958086123114

Mean squared error for state 2= 0.0

R = 0.1, delta = 10

Time Elapsed for Extended Kalman Filter = 0.015619516372680664

Mean squared error for state 1= 0.09430869465130533

Mean squared error for state 2= 0.0

R = 0.1, delta = 20

Time Elapsed for Extended Kalman Filter = 0.01562666893005371

Mean squared error for state 1= 0.08563863196358196

Mean squared error for state 2= 0.0

R = 0.1, delta = 40

Time Elapsed for Extended Kalman Filter = 0.0

Mean squared error for state 1= 0.018662275870644143

Mean squared error for state 2= 0.0

R = 0.01, delta = 5

Time Elapsed for Extended Kalman Filter = 0.013962030410766602

Mean squared error for state 1= 0.04092492760273882

Mean squared error for state 2= 0.0

R = 0.01, delta = 10

Time Elapsed for Extended Kalman Filter = 0.008542537689208984

Mean squared error for state 1= 0.008952048474999136

Mean squared error for state 2= 0.0

R = 0.01, delta = 20

Time Elapsed for Extended Kalman Filter = 0.015592098236083984

Mean squared error for state 1= 0.053514733980213315

Mean squared error for state 2= 0.0

R = 0.01, delta = 40

Time Elapsed for Extended Kalman Filter = 0.0

Mean squared error for state 1= 0.06157829431090943

Mean squared error for state 2= 0.0

R = 0.001, delta = 5

Time Elapsed for Extended Kalman Filter = 0.015621185302734375

Mean squared error for state 1= 0.019226596640168667

Mean squared error for state 2= 0.0

R = 0.001, delta = 10

Time Elapsed for Extended Kalman Filter = 0.015625715255737305

Mean squared error for state 1= 0.029872779777932238

Mean squared error for state 2= 0.0

R = 0.001, delta = 20

Time Elapsed for Extended Kalman Filter = 0.0

Mean squared error for state 1= 0.013592005106762079

Mean squared error for state 2= 0.0

R = 0.001, delta = 40

Time Elapsed for Extended Kalman Filter = 0.011966943740844727

Mean squared error for state 1= 0.024712625374151177

Mean squared error for state 2= 0.0

2. **Unscented Kalman Filter**: The unscented Kalman filter uses a 2d+1 point quadrature rule where d is the dimension of the dynamics model. To implement this filter, first a set of five unscented points is defined. These points use a scaled version of the elements of the matrix M which are the spherical quadrature points. An integral approximation is performed using these points and their corresponding weights in order to perform the filter process.

   **Programming approach and implementation:**
   To implement the unscented Kalman filter I constructed the following functions based on the functions in the file integration_rules.ipynb and Kalman_filter.ipynb:

   - unscented_points (): This function obtains a set of 5 scaled points that need to be run through the filter. Corresponding weights for these points are also calculated within this function. This function accepts as arguments the prior mean and prior covariance. The points and weights are calculated using the following algorithm:
     The quadrature points are defined by:

$$u^{(i)} = \sqrt{d+\lambda}\hat{u}^{(i)}, \quad \forall \hat{u}^{(i)} \in [\mathbf{M}].$$

     Where,

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

UKF also uses slightly different weights for the mean and covariance, defined as follows:

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

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

Here, $\alpha, \beta, \kappa$ are free parameters set to the values:

$$\alpha = 1, \qquad \beta = 0, \qquad \kappa = 0$$

And the integral approximation is given by:

$$\int h(X)\mathbb{N}(X|\mu, \Sigma)dX \approx \sum_{i=0}^{2d} h(\mu + \sqrt{\Sigma}\sqrt{d+\lambda}\hat{u}^{(i)})w^{(i)} \quad \hat{u}^{(i)} \in [\mathbf{M}] \cup \{u^{(0)}$$

This function gives the quadrature points and corresponding weights which are then passed into the predict and update functions.

- ukf_prediction_step (): This function takes as arguments the unscented points and weights, the prior mean and covariance and the process covariance matrix. It returns the predicted mean and predicted covariance which will later be passed into the update function. The following equations were used to obtain the predicted mean and covariance:

$$m_k^- = \mathbb{E}\left[X_k \mid \mathcal{Y}_{k-1}\right] = \int \Phi(X_{k-1})\mathcal{N}\left(X_{k-1}; m_{k-1}, C_{k-1}\right) dX_{k-1} \tag{21.7}$$

$$C_k^- = \mathbb{C}\text{ov}\left[X_k, X_k \mid \mathcal{Y}_{k-1}\right] = \int \left(\Phi(X_{k-1}) - m_k^-\right)\left(\Phi(X_{k-1}) - m_k^-\right)^T \mathcal{N}(X_{k-1}; m_{k-1}, C_{k-1})dX_{k-1} + \Sigma \tag{21.8}$$

On performing integral approximation, the predicted mean and covariance can be written as:

$$m_k^- = \sum \boldsymbol{\Phi}(X_{k-1}) \,.\, w_m$$

$$C_k^- = \sum (\boldsymbol{\Phi}(X_{k-1}) - m_k^-).\,(\boldsymbol{\Phi}(X_{k-1}) - m_k^-)^T \,.\, w_c$$

Note that, here the weight for mean and covariance is different as specified in the previous function. Additionally, for both calculations of mean and covariance, the first point uses $w_m^0$ and $w_C^0$ as the weights and remaining four points use $w_m^i$ and $w_c^i$ as defined previously.

- ukf_update_step (): This function takes as arguments the measurement data, unscented points, and weights, the prior mean and covariance and the noise variance. It returns the updated mean and updated covariance. The following equations were used to obtain the updated mean and covariance:

$$\mu \equiv \mathbb{E}[Y_k] = \int h(X_k)\mathcal{N}(X_k; m_k^-, C_k^-)dX_k$$
$$U \equiv \mathbb{C}\text{ov}[X_k, Y_k] = \int (X_k - m_k^-)(h(X_k) - \mu)^T \mathcal{N}(X_k; m_k^-, C_k^-)dX_k$$
$$S \equiv \mathbb{C}\text{ov}[Y_k, Y_k] = \int (h(X_k) - \mu)(h(X_k) - \mu)^T \mathcal{N}(X_k; m_k^-, C_k^-)dX_k + \Gamma$$
$$m_k = m_k^- + US^{-1}(y_k - \mu)$$
$$C_k = C_k^- - US^{-1}U^T$$

  On performing integral approximation, these can be rewritten as:

$$\mu = \sum \mathbf{h}(X_k) \,.\, w_m$$
$$U = \sum (\mathrm{X_k} - m_k^-).(\boldsymbol{h}(\mathrm{X_k}) - \mu)^T \,.\, w_c$$
$$S = \sum (\boldsymbol{h}(\mathrm{X_k}) - \mu).(\boldsymbol{h}(\mathrm{X_k}) - \mu)^T \,.\, w_c$$

- unscented_kalman_filter (): This function obtains and stores the mean and covariance after implementing the filter on the given model. This function takes as arguments the prior mean and prior covariance, truth data from the measurement model and noise variance. The Q matrix which is the process covariance matrix is constructed within the function as specified in the question. 500 iterations were run to obtain the required number of samples. The function ukf_prediction_step () is used to obtain the predicted mean and covariance. Before calling the prediction function, the unscented points and weights are obtained using the prior mean and covariance. Then an if statement is used to verify if the measurement values obtained are numbers or not. This is because, the measurement data is collected at every $\delta$ timestep only and the values in between may or may not be defined as a number. On satisfying the condition that the measurement data is a number, the function ukf_update_step () is called to obtain the updated mean and covariance which are then stored as the filtered mean and covariance. For the update function as well, a new set of unscented points and weights are obtained before passing it into the update function. In the case where the if condition is not satisfied, the previous mean and covariance are stored as the filtered mean and covariance.

The unscented Kalman filter is run for 500 iterations for 16 combinations of R and $\delta$ where, $R \in \{1, 0.1, 0.01, 0.001\}$ and $\delta \in \{5, 10, 20, 40\}$. Figure 2 represents the state estimates $\pm 2\sigma$ for the extended Kalman Filter.

In Figure 2, the dotted red and blue lines represent the true values of the two states and the solid lines represent the filtered values. The black dots are the measurement values. From the figure, it is evident that the values obtained from the unscented Kalman filter are quite off from the true values. As the value of R decreases, filtered values of state 1 seem to be trailing off further away from the true values. This may be an issue with the points being passed into the predict and update function. In the code, I am calling the unscented_points () function twice, once before the predict function and once before the update function. So, the unscented points being used at predict and update may be different. However, the predict mean and covariance passed into the update function is corresponding to the same timestep. I did not have the time debug this part of the code. I believe if I call the unscented_points () function inside the predict and update functions then the filter would give better results. There could be an inconsistency with the timesteps here that snowballed, and the filtered values are significantly different from the truth values. The filtered values for both states do not seem to change with change in $\delta$ values. Another point to notice is that the variance decreased with decreasing R values. This does not make much sense either because the filter values are significantly off from the true values.

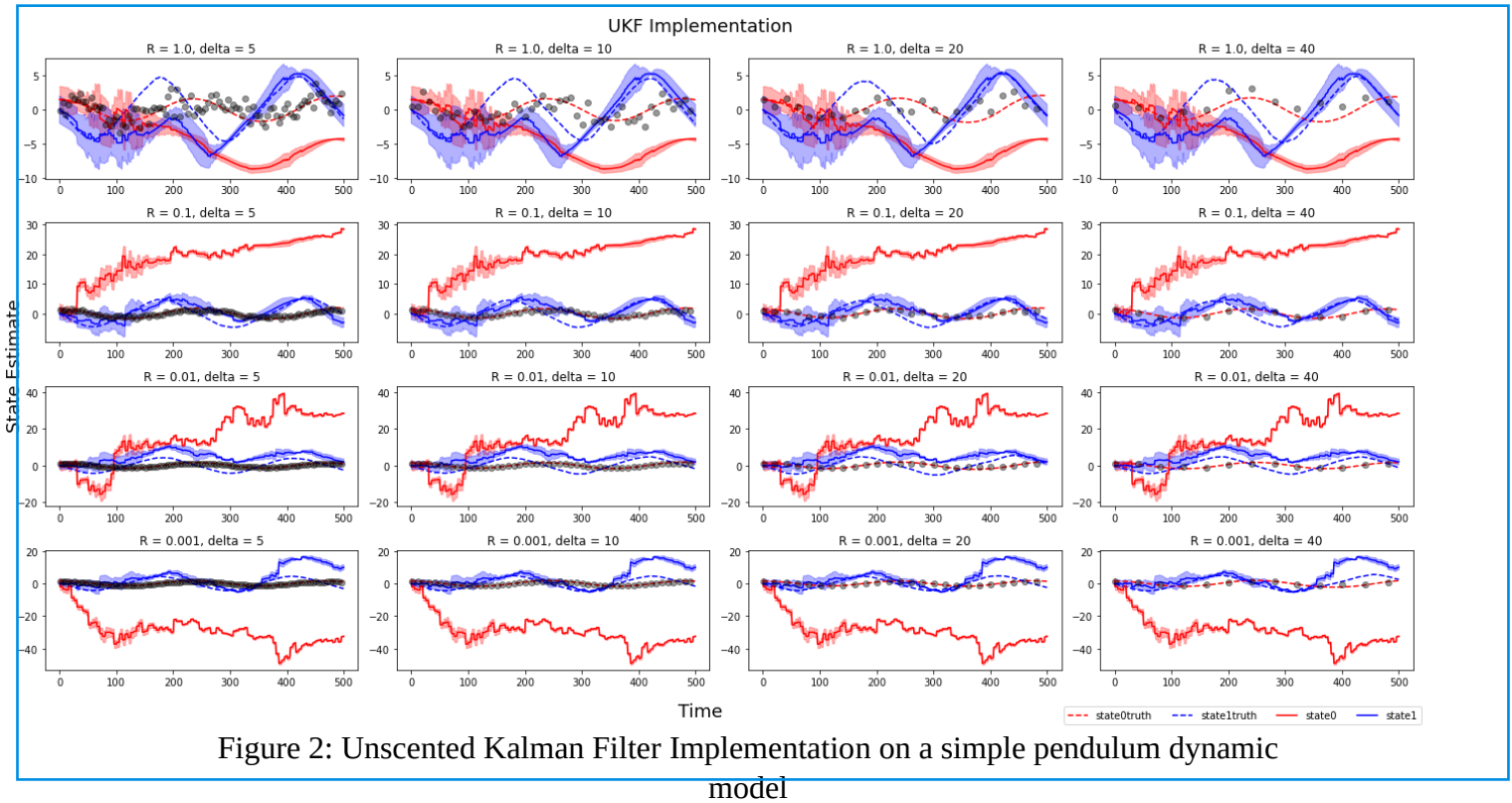



Figure 2: Unscented Kalman Filter Implementation on a simple pendulum dynamic model

**Results from code:**

R = 1.0, delta = 5

Time Elapsed for Unscented Kalman Filter = 0.0377

Mean squared error for state 1= 1.7844e-05

Mean squared error for state 2= 0.0

R = 1.0, delta = 10

Time Elapsed for Unscented Kalman Filter = 4.286331470656497e-05

Mean squared error for state 1= 0.0503

Mean squared error for state 2= 0.0

R = 1.0, delta = 20

Time Elapsed for Unscented Kalman Filter = 0.0378

Mean squared error for state 1= 0.0021

Mean squared error for state 2= 0.0

R = 1.0, delta = 40

Time Elapsed for Unscented Kalman Filter = 0.03125

Mean squared error for state 1= 0.0455

Mean squared error for state 2= 0.0

R = 0.1, delta = 5

Time Elapsed for Unscented Kalman Filter = 0.0534

Mean squared error for state 1= 1.784470829770008e-05

Mean squared error for state 2= 0.0

R = 0.1, delta = 10

Time Elapsed for Unscented Kalman Filter = 0.0429

Mean squared error for state 1= 4.286e-05

Mean squared error for state 2= 0.0

R = 0.1, delta = 20

Time Elapsed for Unscented Kalman Filter = 0.0469

Mean squared error for state 1= 0.0006

Mean squared error for state 2= 0.0

R = 0.1, delta = 40

Time Elapsed for Unscented Kalman Filter = 0.0534

Mean squared error for state 1= 0.0013251467989248177

Mean squared error for state 2= 0.0

R = 0.01, delta = 5

Time Elapsed for Unscented Kalman Filter = 0.04698038101196289

Mean squared error for state 1= 0.017242953788859264

Mean squared error for state 2= 0.0

R = 0.01, delta = 10

Time Elapsed for Unscented Kalman Filter = 0.036418914794921875

Mean squared error for state 1= 0.018561706581219247

Mean squared error for state 2= 0.0

R = 0.01, delta = 20

Time Elapsed for Unscented Kalman Filter = 0.031244516372680664

Mean squared error for state 1= 0.02357999560839938

Mean squared error for state 2= 0.0

R = 0.01, delta = 40

Time Elapsed for Unscented Kalman Filter = 0.03782391548156738

Mean squared error for state 1= 0.045863951224181514

Mean squared error for state 2= 0.0

R = 0.001, delta = 5

Time Elapsed for Unscented Kalman Filter = 0.0540776252746582

Mean squared error for state 1= 0.0038370655276743003

Mean squared error for state 2= 0.0

R = 0.001, delta = 10

Time Elapsed for Unscented Kalman Filter = 0.04090714454650879

Mean squared error for state 1= 0.021370764415764423

Mean squared error for state 2= 0.0

R = 0.001, delta = 20

Time Elapsed for Unscented Kalman Filter = 0.04690074920654297

Mean squared error for state 1= 0.014309895465236767

Mean squared error for state 2= 0.0

R = 0.001, delta = 40

Time Elapsed for Unscented Kalman Filter with 0.045380353927612305

Mean squared error for state 1= 0.09619131779670849

Mean squared error for state 2= 0.0

In [28]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import time
LW=5 # linewidth
MS=10 # markersize
```

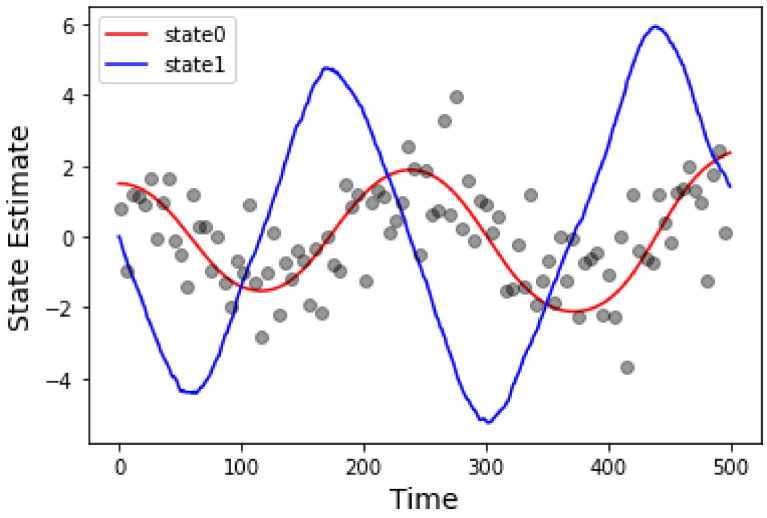
In [29]:
```
def generate_truth(N, x0, H, del_step, Q, R):
    """Generate the truth for the second-order system"""
    del_t = 0.01
    g = 9.8 #m/s
    xout = np.zeros((N+1, 2))
    m = H.shape[0]
    yout = np.empty((N, 1))
    yout[:] = np.nan
    xout[0, :] = x0
    for ii in range(N):
        xout[ii+1, 0] = xout[ii, 0] + (xout[ii, 1]*del_t)
        xout[ii+1, 1] = xout[ii, 1] - (g * np.sin(xout[ii,0]) * del_t)
        q_k = np.dot(np.linalg.cholesky(Q), np.random.randn(2))
        xout[ii+1, :] = xout[ii+1, :] + q_k

    for ii in range(0, N, del_step):
        r_delk = np.sqrt(R)*np.random.randn(1)
        yout[ii] = np.sin(xout[ii, 0]) + r_delk

    return xout, yout
```

In [30]:
```
x_init = [1.5, 0]
del_t = 0.01
del_step = 5
R = 1
q_c = 0.1
N_steps = 500
H = np.eye(2)
Q_cov = [[(q_c*(del_t**3))/3, (q_c*(del_t**2))/2], [(q_c*(del_t**2))/2, (q_c*del_
print(Q_cov)
xout, yout = generate_truth(N_steps, x_init, H, del_step, Q_cov, R)
print(xout.shape)
print(yout.shape)
```

```
[[3.333333333333334e-08, 5e-06], [5e-06, 0.001]]
(501, 2)
(500, 1)
```

In [31]:
```
DT = 1
T = 500
Nsims = int(T / DT)
t = np.linspace(0, T, Nsims+1)
```

In [32]:
```
plt.figure()
plt.plot(t, xout[:, 0], color='red', label='state0')
plt.plot(t, xout[:, 1], color='blue', label='state1')
plt.plot(t[1:], yout[:, 0], 'ko', alpha=0.4)
#plt.plot(t[1:], yout[:, 1], 'kx', alpha=0.4)
plt.legend()
plt.xlabel('Time', fontsize=14)
plt.ylabel('State Estimate',fontsize=14)
plt.show()
```




In [33]:
```
def ekf_update_step(data, H, H_T, mean, cov, noise_cov):

    mu = np.sin(mean[0])
    delta = data - mu
    S = np.dot(H, np.dot(cov, H_T)) + noise_cov
    U = np.dot(cov, H_T)

    update_mean = mean.reshape(2,1) + (U*delta/S).reshape(2,1)
    update_mean = update_mean.reshape(2,)

    update_cov = cov - np.dot(U, (U.T)/S)
    return update_mean, update_cov
```

In [34]:
```
def ekf_prediction_step(A, A_T, mean, cov, noise_cov):

    pred_mean = np.zeros((2,1))
    del_t = 0.01
    g = 9.8
    pred_mean[0] = mean[0] + (mean[1]*del_t)
    pred_mean[1] = mean[1] - (g*np.sin(mean[0])*del_t)
    pred_mean = np.reshape(pred_mean,(2,))
    pred_cov = np.dot(A, np.dot(cov, A_T)) + noise_cov
    return pred_mean, pred_cov
```

In [35]:
```
def extended_kalman_filter(data, prior_mean, prior_cov, R):

    del_t = 0.01
    g = 9.8 #m/s
    qc = 0.1
    d = prior_mean.shape[0]
    N = data.shape[0]
    m = data.shape[1]

    mean_store = np.zeros((N+1, d))
    mean_store[0, :] = np.copy(prior_mean)
    cov_store = np.zeros((d, d, N+1))
    cov_store[:, :, 0] = np.copy(prior_cov)
    Q = [[(qc*(del_t**3))/3, (qc*(del_t**2))/2], [(qc*(del_t**2))/2, (qc*del_t)]]
    #R = 1

    #Loop over all data
    for ii in range(N):
        # Prediction
        A_k = np.array([[1, del_t], [-g*np.cos(mean_store[ii, 0])*del_t, 1]])
        A_k_trans = A_k.T
        #print(A_k.T)
        H_k =np.array([[np.cos(mean_store[ii, 0]) ,0]])
        #print(H_k)
        H_k_trans = H_k.T
        pred_mean, pred_cov = ekf_prediction_step(A_k, A_k_trans, mean_store[ii,
         # Update

        if not np.isnan(data[ii]):
            mean_store[ii+1, :], cov_store[:, :, ii+1] = ekf_update_step(data[ii,
        else:
            mean_store[ii+1, :] = pred_mean
            cov_store[:, :, ii+1] = pred_cov

    return mean_store, cov_store
```

In [36]:
```
def get_std(cov):
    """Get square root of diagonals (standard deviations) from covariances """

    d, d, N = cov.shape
    std_devs = np.zeros((N, d))
    for ii in range(N):
        std_devs[ii, :] = np.sqrt(np.diag(cov[:, :, ii]))
    return std_devs
```

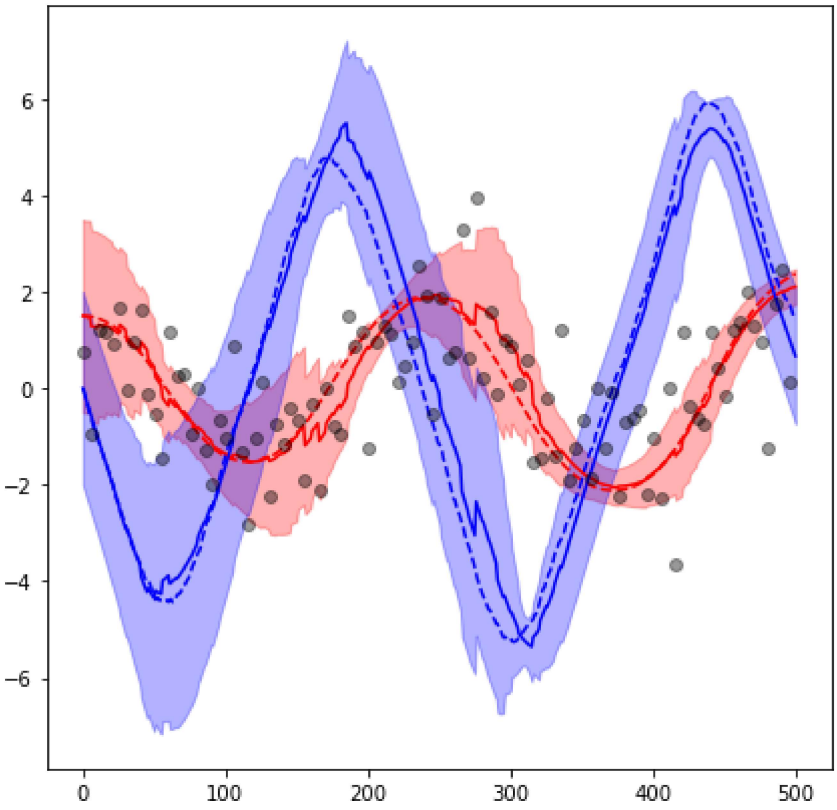
In [37]:
```
data = yout
R = 1
prior_mean = np.array([1.5, 0])
print(prior_mean)
prior_cov = np.eye(2)
print(prior_cov.shape)

mean_ekf, cov_ekf = extended_kalman_filter(data, prior_mean, prior_cov, R)
```

```
[1.5 0. ]
(2, 2)
```

In [38]:
```
K = 5 # time discretization error, only use a model that integrates at 5 times la
DTsim = 1
#Asim = DTsim * Asim + np.eye(2)
T = 500
Nsims_sim = int(T / DTsim)
tsim = np.linspace(0, T, Nsims_sim+1)
# yuse = yout[np.arange(K-1, yout.shape[0], K), :]
```

In [39]:
```
std_devs = get_std(cov_ekf) #extract the standard deviations at all the states
plt.figure(figsize=(7,7))
plt.plot(t, xout[:, 0],'--', color='red', label='state0truth')
plt.plot(t, xout[:, 1],'--', color='blue', label='state1truth')

plt.plot(t, mean_ekf[:, 0], color='red', label='state0')
plt.plot(t, mean_ekf[:, 1], color='blue', label='state1')
plt.fill_between(t, mean_ekf[:, 0] - 2 * std_devs[:, 0],  mean_ekf[:, 0] + 2*std_
plt.fill_between(t, mean_ekf[:, 1] - 2 * std_devs[:, 1],  mean_ekf[:, 1] + 2*std_
plt.plot(t[1:], yout[:], 'ko', alpha=0.4)
# #plt.plot(t[1:], yout[:, 1], 'kx', alpha=0.4)
# #plt.plot(tsim[1:], yuse[:, 0], 'ko', alpha=0.4)
# #plt.plot(tsim[1:], yuse[:, 1], 'kx', alpha=0.4)
# plt.legend()
# plt.xlabel('Time', fontsize=14)
# plt.ylabel('State Estimate',fontsize=14)
# plt.show()
```

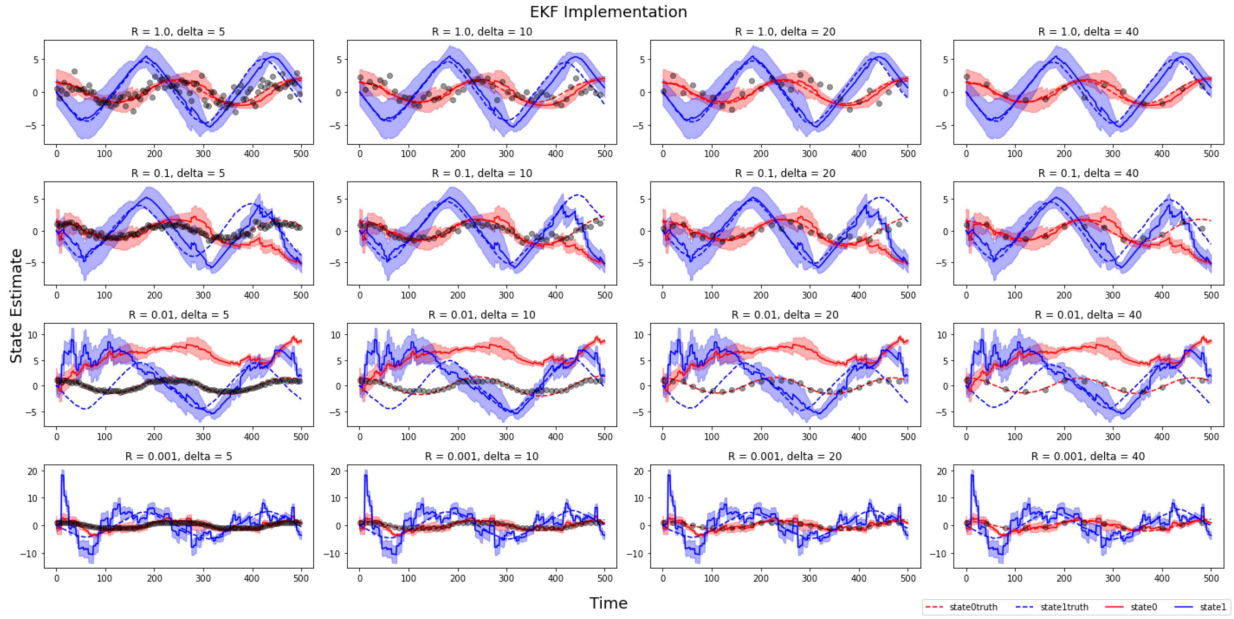
Out[39]:
```
[<matplotlib.lines.Line2D at 0x28175af1708>]
```



In [41]:
```
N = 500
x_init = np.array([1.5, 0])
g = 9.8 #m/s
prior_cov = np.eye(2)
R_set = np.array([1, 0.1, 0.01, 0.001])
delta_set = np.array([5, 10, 20, 40])
fig, subplt = plt.subplots(4, 4, figsize = (20,10))

for ii in range(4):
    R = R_set[ii]
    for jj in range(4):
        del_step = delta_set[jj]
        tsim = np.linspace(0, T, N+1)

        xout, yout = generate_truth(N_steps, x_init, H, del_step, Q_cov, R)
        start_time = time.time()
        mean_EKF, cov_EKF = extended_kalman_filter(data, x_init, prior_cov, R)

        print("Time Elapsed for Extended Kalman Filter with R = " + str(R) + ", d

        for kk in range(0, xout.shape[0]):
            sq_err_s0 = np.zeros((xout.shape[0],1))
            sq_err_s1 = np.zeros((xout.shape[0],1))
            sq_err_s0[kk,:] = (xout[kk,0] - mean_EKF[kk,0])**2
            sq_err_s0[kk,:] = (xout[kk,1] - mean_EKF[kk,1])**2
        mean_sq_err_s0 = np.mean(sq_err_s0)
        print('Mean squared error for state 1=', mean_sq_err_s0)

        mean_sq_err_s1 = np.mean(sq_err_s1)
        print('Mean squared error for state 2=', mean_sq_err_s1)

        std_devs = get_std(cov_ekf)
        ax = subplt[ii,jj]
        ax.plot(t, xout[:, 0],'--', color='red', label='state0truth')
        ax.plot(t, xout[:, 1],'--', color='blue', label='state1truth')
        ax.plot(tsim, mean_EKF[:, 0], color='red', label='state0')
        ax.plot(tsim, mean_EKF[:, 1], color='blue', label='state1')
        ax.plot(t[1:], yout[:], 'ko', alpha=0.4)
        ax.fill_between(t, mean_EKF[:, 0] - 2 * std_devs[:, 0],  mean_EKF[:, 0] +
        ax.fill_between(t, mean_EKF[:, 1] - 2 * std_devs[:, 1],  mean_EKF[:, 1] +
        ax.set_title("R = " + str(R) + ", delta = " + str(del_step))
fig.suptitle('EKF Implementation', fontsize = 18)
fig.supxlabel('Time', fontsize = 18)
fig.supylabel('State Estimate', fontsize = 18)
fig.legend(["state0truth", "state1truth", "state0", "state1"], loc = 'lower right
fig.tight_layout()
plt.show()
```

```
Time Elapsed for Extended Kalman Filter with R = 1.0, delta = 5 0.0156216621398
92578
Mean squared error for state 1= 0.01047236529255327
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 1.0, delta = 10 0.0
Mean squared error for state 1= 0.012755623576003444
Mean squared error for state 2= 0.0
```

```
Time Elapsed for Extended Kalman Filter with R = 1.0, delta = 20 0.0
Mean squared error for state 1= 0.005156982866565686
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 1.0, delta = 40 0.015627145767
211914
Mean squared error for state 1= 0.0038847408434432578
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.1, delta = 5 0.0
Mean squared error for state 1= 0.0033897958086123114
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.1, delta = 10 0.015619516372
680664
Mean squared error for state 1= 0.09430869465130533
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.1, delta = 20 0.015626668930
05371
Mean squared error for state 1= 0.08563863196358196
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.1, delta = 40 0.0
Mean squared error for state 1= 0.018662275870644143
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.01, delta = 5 0.013962030410
766602
Mean squared error for state 1= 0.04092492760273882
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.01, delta = 10 0.00854253768
9208984
Mean squared error for state 1= 0.008952048474999136
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.01, delta = 20 0.01559209823
6083984
Mean squared error for state 1= 0.053514733980213315
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.01, delta = 40 0.0
Mean squared error for state 1= 0.06157829431090943
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.001, delta = 5 0.01562118530
2734375
Mean squared error for state 1= 0.019226596640168667
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.001, delta = 10 0.0156257152
55737305
Mean squared error for state 1= 0.029872779777932238
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.001, delta = 20 0.0
Mean squared error for state 1= 0.013592005106762079
Mean squared error for state 2= 0.0
Time Elapsed for Extended Kalman Filter with R = 0.001, delta = 40 0.0119669437
40844727
Mean squared error for state 1= 0.024712625374151177
Mean squared error for state 2= 0.0
```


EKF Implementation
R = 1.0, delta = 5
R = 1.0, delta = 10
R = 1.0, delta = 20
R = 1.0, delta = 40
R = 0.1, delta = 5
R = 0.1, delta = 10
R = 0.1, delta = 20
R = 0.1, delta = 40
R = 0.01, delta = 5
R = 0.01, delta = 10
R = 0.01, delta = 20
R = 0.01, delta = 40
R = 0.001, delta = 5
R = 0.001, delta = 10
R = 0.001, delta = 20
R = 0.001, delta = 40
State Estimate
Time
state0truth
state1truth
state0
state1

In [ ]:

In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import time
LW=5 # linewidth
MS=10 # markersize
```

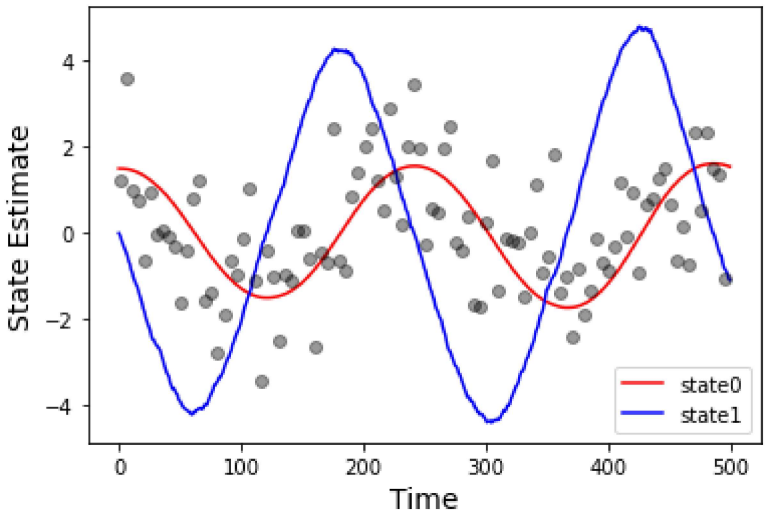
In [2]:
```
def generate_truth(N, x0, H, del_step, Q, R):
    """Generate the truth for the second-order system"""
    del_t = 0.01
    g = 9.8 #m/s
    xout = np.zeros((N+1, 2))
    m = H.shape[0]
    yout = np.empty((N, 1))
    yout[:] = np.nan
    xout[0, :] = x0
    for ii in range(N):
        xout[ii+1, 0] = xout[ii, 0] + (xout[ii, 1]*del_t)
        xout[ii+1, 1] = xout[ii, 1] - (g * np.sin(xout[ii,0]) * del_t)
        q_k = np.dot(np.linalg.cholesky(Q), np.random.randn(2))
        xout[ii+1, :] = xout[ii+1, :] + q_k

    for ii in range(0, N, del_step):
        r_delk = np.sqrt(R)*np.random.randn(1)
        yout[ii] = np.sin(xout[ii, 0]) + r_delk

    return xout, yout
```

In [3]:
```
x_init = [1.5, 0]
del_t = 0.01
del_step = 5
R = 1
q_c = 0.1
N_steps = 500
H = np.eye(2)
Q_cov = [[(q_c*(del_t**3))/3, (q_c*(del_t**2))/2], [(q_c*(del_t**2))/2, (q_c*del_
print(Q_cov)
xout, yout = generate_truth(N_steps, x_init, H, del_step, Q_cov, R)
print(xout.shape)
print(yout.shape)
```

```
[[3.333333333333334e-08, 5e-06], [5e-06, 0.001]]
(501, 2)
(500, 1)
```

In [4]:
```
DT = 1
T = 500
Nsims = int(T / DT)
t = np.linspace(0, T, Nsims+1)
```

In [5]:
```
plt.figure()
plt.plot(t, xout[:, 0], color='red', label='state0')
plt.plot(t, xout[:, 1], color='blue', label='state1')
plt.plot(t[1:], yout[:, 0], 'ko', alpha=0.4)
#plt.plot(t[1:], yout[:, 1], 'kx', alpha=0.4)
plt.legend()
plt.xlabel('Time', fontsize=14)
plt.ylabel('State Estimate',fontsize=14)
plt.show()
```




In [6]:
```
def unscented_points(mean, cov, alg='svd', alpha=1, beta=0, kappa=0):
    """Generate unscented points"""
    dim = cov.shape[0]
    lam = alpha*alpha*(dim + kappa) - dim
    if alg == "chol":
        L = np.linalg.cholesky(cov)
    elif alg == "svd":
        u, s, v = np.linalg.svd(cov)
        L = np.dot(u, np.sqrt(np.diag(s)))
    pts = np.zeros((2*dim+1, 2))
    pts[0, :] = mean.squeeze()
    for ii in range(1, dim+1):
        pts[ii, :] = mean.squeeze() + np.sqrt(dim + lam)*L[:, ii-1]
        pts[ii+dim,:] = mean.squeeze() - np.sqrt(dim + lam)*L[:, ii-1]

    W0m = lam / (dim + lam)
    W0C = lam / (dim + lam) + (1 - alpha*alpha + beta)
    Wim = 1/2 / (dim + lam)
    Wic = 1/2 / (dim + lam)
    return pts, (W0m, Wim, W0C, Wic)
```

In [7]:
```
# #check
# data =yout[0]
# H  =np.array([np.cos(0.5) , 0])
# H_T = H.T
# mean = np.array([1.5, 0])
# cov = np.eye(2)
# noise_cov= 1
# update_mean, update_cov= ekf_update_step(data, H, H_T, mean, cov, noise_cov)
```

In [8]:
```
def ukf_prediction_step( points_phi, wts_phi, mean, cov, noise_cov):

    pred_mean = np.zeros((2,1))
    func_for_mean = np.zeros((2,5))
    func_for_cov_p2to5 = np.zeros((2,2))
    del_t = 0.01
    g = 9.8
    num_points = 5

    func_for_mean[0, :] = points_phi[:, 0] + (points_phi[:, 1]*del_t)
    func_for_mean[1, :] = points_phi[:, 1] - (g*np.sin(points_phi[:, 0])*del_t)

    func_for_mean = func_for_mean.T
    pred_mean = (func_for_mean[0, :]*wts_phi[0] + wts_phi[1]*(func_for_mean[1, :]
#     pred_mean = np.reshape(pred_mean,(2,))
#     print(pred_mean)
    #print(func_for_mean[0,:])


    ####covariance for point 1
    func_for_cov_p1 = np.dot((func_for_mean[0, :].reshape(-1,1) - pred_mean), (fu
    #print(func_for_cov_p1)
    for ii in range(1, num_points):
        ####covariance for points 2 to 5
        func_for_cov_p2to5 = func_for_cov_p2to5 + np.dot((func_for_mean[ii, :].re
        #print(func_for_cov_p2to5)

    pred_cov = (func_for_cov_p1*wts_phi[2]) + (wts_phi[3]*func_for_cov_p2to5) + n

    return pred_mean, pred_cov
```

In [9]:
```
# mean = np.array([1.5, 0])
# cov = np.eye(2)
# p, w = unscented_points(mean, cov, alg='chol', alpha=1, beta=0, kappa=0)
# print(p)
# print(w[0])
```

In [10]:
```
# del_t = 0.01
# g = 9.8
# mean_store = np.array([2, 3])
# A = np.array([[1, del_t], [-g*np.cos(mean_store[0])*del_t, 1]])
# A_T = A.T
# mean = np.array([1.5, 0])
# cov = np.eye(2)
# noise_cov = 1
# pred_mean, pred_cov = ukf_prediction_step(A, A_T, p, w, mean, cov, noise_cov)
```

In [11]:
```
def ukf_update_step(data, points_phi, wts_phi, mean, cov, noise_cov):

    num_points = 5
    h_p1 = np.sin(points_phi[0, 0])
    mu_p1 =  h_p1*wts_phi[0]
    mean = mean.squeeze()

    h_p2to5 = np.zeros((5,1))
    mu_p2to5 = 0
    U_p2to5 = 0
    S_p2to5 = 0
    for ii in range (1, num_points):
        h_p2to5[ii] = np.sin(points_phi[ii, 0])
    mu_p2to5 = np.sum(h_p2to5)*wts_phi[1]
    mu_tot = mu_p1 + mu_p2to5

    #U_p1 = np.matmul((points_phi[0, :] - mean), (h_p1 -  mu_p1))*wts_phi[2]
    U_p1 = np.dot((points_phi[0, :] - mean), (h_p1 -  mu_tot))*wts_phi[2]
    #S_p1 = np.matmul((h_p1 - mu_p1),(h_p1 - mu_p1).T)*wts_phi[2]
    S_p1 = np.dot((h_p1 - mu_tot),(h_p1 - mu_tot).T)*wts_phi[2]

    for ii in range(1, num_points):
        #U_p2to5 = U_p2to5 + np.matmul((points_phi[ii, :] - mean), (h_p2to5[ii] -
        U_p2to5 += np.dot(((points_phi[ii, :] - mean)).reshape(2,1), (h_p2to5[ii]
        #S_p2to5 = S_p2to5 + np.matmul((h_p2to5[ii] -  mu_tot),(h_p2to5[ii] -  mu
        S_p2to5 += np.dot((h_p2to5[ii] -  mu_tot),(h_p2to5[ii] -  mu_tot).T)

    U_p2to5_weighted = U_p2to5*wts_phi[3]
    S_p2to5_weighted = S_p2to5*wts_phi[3]

    U_tot = U_p1 + U_p2to5_weighted
    S_tot = S_p1 + S_p2to5_weighted + noise_cov

    delta = data - [mu_tot, 0]

    #print(U.shape)
    update_mean = (mean + (U_tot*delta/S_tot)).reshape(2,1)
    update_mean = update_mean.reshape(2,)
    #print(update_mean.shape)
    #update_mean = mean + np.dot(cov, np.dot(H_T, np.linalg.solve(S, delta)))
    update_cov = cov - np.dot(U_tot, (U_tot.T)/(S_tot))
    #print(update_cov.shape)
    #update_cov = cov - np.dot(ch, np.linalg.solve(S, ch.T))
    return update_mean, update_cov
```

In [12]:
```
# #check
# data = yout[0]
# H  = np.array([np.cos(0.5) , 0])
# H_T = H.T
# mean = np.array([1.5, 0])
# cov = np.eye(2)
# noise_cov= 1
# update_mean, update_cov = ukf_update_step(data, H, H_T, p, w, mean, cov, noise_
# print(update_mean)
# print(update_cov)
```

In [19]:
```
def unscented_kalman_filter(data, prior_mean, prior_cov, R):

    del_t = 0.01
    g = 9.8 #m/s
    qc = 0.1
    d = prior_mean.shape[0]
    N = data.shape[0]
    m = data.shape[1]

    mean_store = np.zeros((N+1, d))
    mean_store[0, :] = np.copy(prior_mean)
    cov_store = np.zeros((d, d, N+1))
    cov_store[:, :, 0] = np.copy(prior_cov)
    Q = [[(qc*(del_t**3))/3, (qc*(del_t**2))/2], [(qc*(del_t**2))/2, (qc*del_t)]]
    #R = 1

    #Loop over all data
    for ii in range(N):
        # Prediction
#         A_k = np.array([[1, del_t], [-g*np.cos(mean_store[ii, 0])*del_t, 1]])
#         A_k_trans = A_k.T
        H_k =np.array([[np.cos(mean_store[ii, 0]) ,0]])
        H_k_trans = H_k.T
        p_1, w_1 = unscented_points(mean_store[ii, :], cov_store[:, :, ii])
        #print('shape of p_1',p_1.shape)
        pred_mean, pred_cov = ukf_prediction_step( p_1, w_1, mean_store[ii, :], c
#         Update
        #print(mean_store.shape)
        if not np.isnan(data[ii]):
            p_2, w_2 = unscented_points(pred_mean, pred_cov)
            mean_store[ii+1, :], cov_store[:, :, ii+1] = ukf_update_step(data[ii,
        else:
            mean_store[ii+1, :] = pred_mean.squeeze()
            cov_store[:, :, ii+1] = pred_cov

    return mean_store, cov_store
```

In [20]:
```
def get_std(cov):
    """Get square root of diagonals (standard deviations) from covariances """

    d, d, N = cov.shape
    std_devs = np.zeros((N, d))
    for ii in range(N):
        std_devs[ii, :] = np.sqrt(np.diag(cov[:, :, ii]))
    return std_devs
```

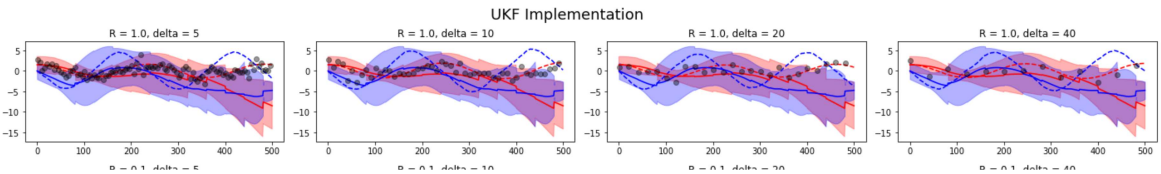
In [21]:
```
data = yout
# R = 1
# prior_mean = np.array([1.5, 0])
# prior_cov = np.eye(2)

# mean_ekf, cov_ekf = unscented_kalman_filter(data, prior_mean, prior_cov, R)
```

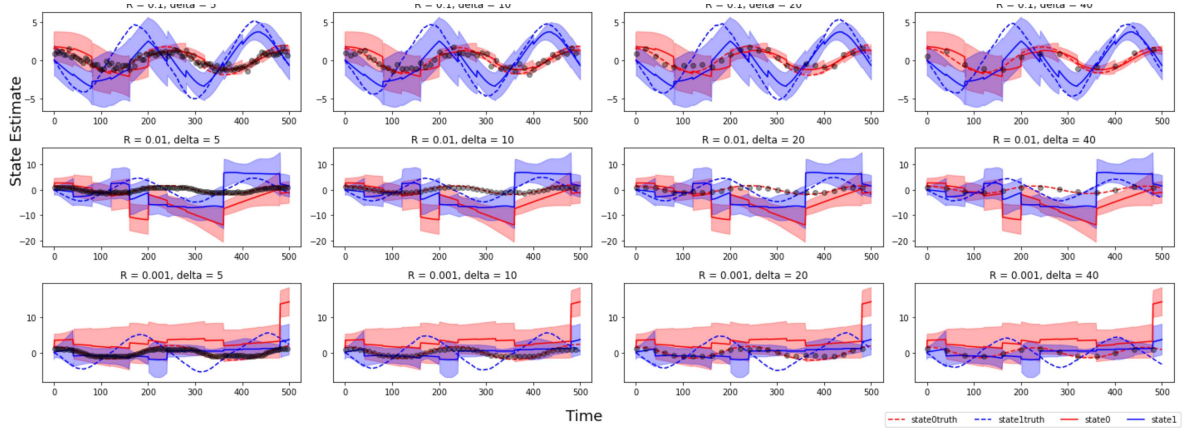
In [22]:
```
K = 5 # time discretization error, only use a model that integrates at 5 times la
DTsim = 1
#Asim = DTsim * Asim + np.eye(2)
T = 500
Nsims_sim = int(T / DTsim)
tsim = np.linspace(0, T, Nsims_sim+1)
# yuse = yout[np.arange(K-1, yout.shape[0], K), :]
```

In [23]:
```
# std_devs = get_std(cov_ekf) #extract the standard deviations at all the states
# plt.figure(figsize=(7,7))
# plt.plot(t, xout[:, 0],'--', color='red', label='state0truth')
# plt.plot(t, xout[:, 1],'--', color='blue', label='state1truth')

# plt.plot(t, mean_ekf[:, 0], color='red', label='state0')
# plt.plot(t, mean_ekf[:, 1], color='blue', label='state1')
# plt.fill_between(t, mean_ekf[:, 0] - 2 * std_devs[:, 0],  mean_ekf[:, 0] + 2*st
# plt.fill_between(t, mean_ekf[:, 1] - 2 * std_devs[:, 1],  mean_ekf[:, 1] + 2*st
# plt.plot(t[1:], yout[:], 'ko', alpha=0.4)
# # #plt.plot(t[1:], yout[:, 1], 'kx', alpha=0.4)
# #plt.plot(tsim[1:], yuse[:, 0], 'ko', alpha=0.4)
# #plt.plot(tsim[1:], yuse[:, 1], 'kx', alpha=0.4)
# plt.legend()
# plt.xlabel('Time', fontsize=14)
# plt.ylabel('State Estimate',fontsize=14)
# plt.show()
```

In [24]:
```
N = 500
x_init = np.array([1.5, 0])
g = 9.8 #m/s
prior_cov = np.eye(2)
R_set = np.array([1, 0.1, 0.01, 0.001])
delta_set = np.array([5, 10, 20, 40])
fig, subplt = plt.subplots(4, 4, figsize = (20,10))
data = yout

for ii in range(4):
    R = R_set[ii]
    for jj in range(4):
        del_step = delta_set[jj]
        tsim = np.linspace(0, T, N+1)

        xout, yout = generate_truth(N_steps, x_init, H, del_step, Q_cov, R)
        start_time = time.time()
        mean_UKF, cov_UKF = unscented_kalman_filter(data, x_init, prior_cov, R)
        print("Time Elapsed for Unscented Kalman Filter with R = " + str(R) + ",

        for kk in range(0, xout.shape[0]):
            sq_err_s0 = np.zeros((xout.shape[0],1))
            sq_err_s1 = np.zeros((xout.shape[0],1))
            sq_err_s0[kk,:] = (xout[kk,0] - mean_UKF[kk,0])**2
            sq_err_s0[kk,:] = (xout[kk,1] - mean_UKF[kk,1])**2
        mean_sq_err_s0 = np.mean(sq_err_s0)
        print('Mean squared error for state 1=', mean_sq_err_s0)

        mean_sq_err_s1 = np.mean(sq_err_s1)
        print('Mean squared error for state 2=', mean_sq_err_s1)


        std_devs = get_std(cov_UKF)
        ax = subplt[ii,jj]
        ax.plot(t, xout[:, 0],'--', color='red', label='state0truth')
        ax.plot(t, xout[:, 1],'--', color='blue', label='state1truth')
        ax.plot(tsim, mean_UKF[:, 0], color='red', label='state0')
        ax.plot(tsim, mean_UKF[:, 1], color='blue', label='state1')
        ax.plot(t[1:], yout[:], 'ko', alpha=0.4)
        ax.fill_between(t, mean_UKF[:, 0] - 2 * std_devs[:, 0],  mean_UKF[:, 0] +
        ax.fill_between(t, mean_UKF[:, 1] - 2 * std_devs[:, 1],  mean_UKF[:, 1] +
        ax.set_title("R = " + str(R) + ", delta = " + str(del_step))
fig.suptitle('UKF Implementation', fontsize = 18)
fig.supxlabel('Time', fontsize = 18)
fig.supylabel('State Estimate', fontsize = 18)
fig.legend(["state0truth", "state1truth", "state0", "state1"], loc = 'lower right
fig.tight_layout()
plt.show()
```

```
  import sys
```




State Estimate
R = 0.01, delta = 5
R = 0.01, delta = 10
R = 0.01, delta = 20
R = 0.01, delta = 40
R = 0.001, delta = 5
R = 0.001, delta = 10
R = 0.001, delta = 20
R = 0.001, delta = 40
Time
state0truth
state1truth
state0
state1

In [ ]: