Simulation and Estimation Handbook

Draft 1 - Fall 2022

Izaac Facundo
Joseph Flores
Nicholas Franken
Neel Pandey
Preston Thomas
William Wang

Project Overview

The overall need of this project is to develop drones capable of sensing and avoiding obstacles with a minimized risk of collision or damage to property in GPS-denied environments. The system that will be obtained will be one that takes real time inputs from actively flying drones and outputs a state estimation with a visualized three dimensional error ellipsoid. The purpose of this system is collision avoidance concerning other flying drones as well as obstacles within the environment whether they are static or dynamically moving. The real world application of a drone collision avoidance system would be for the increase of commercial drone delivery operations such as the developing amazon drone delivery service. It is important to increase the time efficiency it takes for the drones to complete their deliveries while minimizing the risk of damage towards private property. Having an accurate error ellipsoid will allow the drones to fly closer together with greater confidence that no collision will occur.

As the USRC estimation and simulation team, an Extended Kalman Filter (EKF) will be used to turn error into ellipsoid visualization. Implementing an EKF on multiple drones will aid in tracking drones in a highly dense drone cluster. Physical data taken from the sensors on board the drone will be used in a powerful three dimensional simulation created through software called Gazebo. The collision avoidance system will ensure that no two error ellipsoids around their respective drones will not intersect.

Goals

The USRC estimation and simulation team has the following 6 main goals:

- Implement an extended Kalman filter (EKF) on drone positions
- Develop flight software to control drones precisely
- Visualize drones and drone behavior using simulation software
- Develop drones' ability to successfully avoid obstacles
- Develop architecture for communication between drones
- Design system for software-software and software-hardware integration

Although these goals are in no particular order, they do lay out a general framework of the process required to achieve our overall goal of avoiding obstacles in GPS-denied environments.

Implementing an EKF on a dynamic system has already been achieved by our pendulum and potential field simulations in Simulink. We need to use this knowledge of the EKF to estimate the drone's state and error ellipsoid within an actual simulation environment and ultimately in the real world. To ensure that our EKF accurately measures the drone's state, we will need to complete the third goal of visualizing and modeling drone behavior within simulation software.

To control the drone behavior within the simulation software, the flight software needs to be developed. This goal goes hand-in-hand with the goal of developing the drone's ability to successfully avoid obstacles, which will involve an algorithm we will write ourselves. All the software (flight, sensor fusion, and algorithms) will need to be integrated together both with the simulation software as well as the real-life hardware, fulfilling the sixth goal.

We eventually will be developing this collision avoidance software to be run on multiple drones at once, all communicating with each other. This will require scalability to be a factor when determining how we want to structure our systems to interact with each other.

Professional Responsibilities

Our professional responsibility revolves around ensuring the safety of the areas we operate our drones in. The collision avoidance system employed must be able to mitigate any sort of risk of colliding with people, animals, or other inanimate/animate obstacles. This will affect the maximum velocity, altitude, and paths of avoidance our drone is able to use. Furthermore, the sound level of our drone will also play a part in determining where and when we can fly. Finally, the end of life of the system must also be considered to ensure unnecessary waste is not created from disposal of our system.

Software Systems

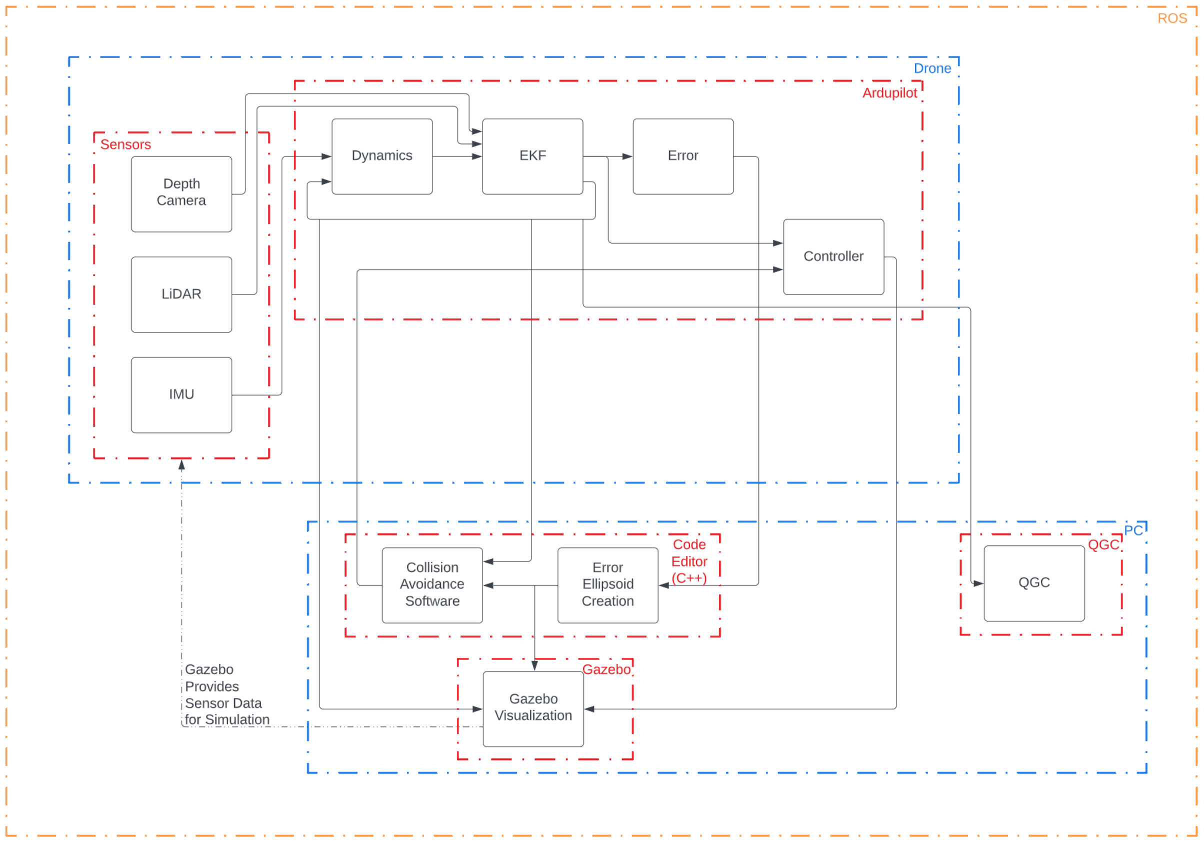

Figure 1: Block diagram overview of the softwares we will use and develop

Block Diagram

The explanations for the hierarchy and uses of each block in the block diagram above in Fig. XX are explained below:

- Robot Operating System (ROS)
  - Drone
    - Sensors
      - Include depth camera, LIDAR, IMU and any additional sensors necessary for stable flight of the drone and error ellipsoid generation
    - Ardupilot
      - This is the backbone of our code, provides the basis for general flight through integrated EKF and actuator control [source]
      - Will be flashed to the flight controller as our main flight software
  - Computer
    - Code Editor
      - Languages used will include C++, Simulink, MATLAB, etc.

- Possible editors include Eclipse, VSCode, etc.
    - Will be used for error ellipsoid generation, data analysis
- QGroundControl (QGC)
    - Software to be run on the ground station computer during flight experiments
    - Provides real time data on the base systems of the drone (IMU, GPS, etc.)
    - Provides link for manual override during flight
    - Base that allows Ardupilot specific parameters to be tuned
- Gazebo [source]
    - Simulation software that is linked with ROS
    - Able to accurately simulate the physics of a 3D system

ROS

The Robot Operating System serves as a manager for all of the systems that we will develop. It allows us to integrate simulation with our error ellipsoid generation and similarly the drone with the error ellipsoid generation and ground station. ROS works on the premise of different nodes that publish and/or subscribe to specific topics from other nodes. By running all the nodes through a single ROS Master, each node can communicate with the others via their configuration (pub/sub). The ROS Master also allows for the communication of multiple systems working over a shared WiFi connection so that data can be sent and received wirelessly between a computer and a flying drone. ROS also provides us with the ability to use additional analysis tools in the form of packages like RVIS and RQT in addition to being able to mass record data form all or specific nodes using ROS Bags.

ArduPilot

ArduPilot will provide the majority of the code used to control the drone and perform sensor fusion.

Gazebo

Gazebo is the software we will use to model and simulate the drone's behavior. Gazebo will also provide simulated inputs for the sensors to our code for estimating the state of the drone, thus allowing us to test our code without risk of damaging the drone or our environment. The parameters of the drone and sensors can be modified as needed to model our drone as accurately as possible.

Kalman Filter

A Kalman Filter is a process that takes an imperfect dynamic system and imperfect measurements to estimate an incredibly accurate state of a system. The Kalman Filter works by estimating (predicting) the state and then correcting (updating) that estimation based on the data it has available to it. The algorithm behind the filter is somewhat complicated, and can be found here. Understanding the exact algorithm is not as important for the purposes of the project as understanding what it needs and its limitations. A phenomenal resource to become familiar with and understand what a Kalman filter does can be found on the MATLAB file exchange here. It is an interactive lesson that teaches the user the step by step construction and variation of components of the filter.

To construct a Kalman filter, a few things are needed. First a roughly accurate state space model of the desired system must be created. The desired observed state equation (C and D matrices) must be calculated as well. For use in the Simulink Kalman Filter block, the A, B, C, and D matrices are needed. Process noise and measurement noise covariance matrices are required as well, and are explained below.

To learn what covariance is and what a covariance matrix represents, click here to watch this video. The process noise covariance matrix (Q) is a measure of the uncertainty or error there is in the dynamic state space model. An example of process noise is air resistance or random turbulent wind in a pendulum system where only the force of gravity has been taken into account in the model. The measurement noise covariance matrix (R) is a measure of the uncertainty or error there is in the measurements of the system. This can be understood as the noise in the sensors used to measure variables in the system. The covariance matrix N is a measure of how Q and R interact and influence each other. For most simple cases, this can be set to 0.

In summary, this is what is needed to construct a Kalman Filter:

- A state space model with A, B, C, and D matrices
- Q and R covariance matrices (and N, N usually = 0)
- An initial state estimate (does not have to be accurate, will get adjusted over time)

The biggest limitation of this filter is that it is only accurate for linear systems. An Extended Kalman Filter (EKF) is required to estimate the state of a nonlinear system. An EKF works just about the same as a regular Kalman Filter, except one more step is required.

Once the dynamic model is obtained from the system, the states must be discretized into new equations using this formula from the Mathworks lesson:

$$\dot{x} \approx \frac{x_{n+1} - x_n}{T_s}$$

Ts is the timestep of the system. The State Transition Function that MATLAB requires is the set of updated state variables in this form:

$$x_1[k+1] =$$
$$x_2[k+1] =$$

The measurement function is more simple. All you need is a function of the form (this is a function where only the first state variable is observed):

$$y = x_1$$

Once these extra steps are taken, they can be inputted into an Extended Kalman Filter block in Simulink. Make sure the data going into the filter is discrete for the EKF. You can do this by inserting Zero Order Hold blocks before the signals enter the EKF.

In summary, a Kalman Filter estimates a very accurate state with not very accurate data. The filter needs a dynamic model and some noise covariance matrices. An Extended Kalman Filter can be used to estimate a nonlinear system, and only a few extra steps are required to convert from a regular KF to an EKF.

Error Ellipsoids

As is known, the ultimate error visualization method to be used in this project is the three dimensional error ellipsoid. However, the effort necessary to develop an algorithm and graphical visualization for a 3D error ellipsoid of a drone is quite excessive for an initial demonstration. Because of this, early work for this project involved creating and visualizing 2D error ellipses instead, an example of which is shown in Fig. [2]. The boundaries of the ellipse encompass all possible combinations of the two parameters on the x and y axes, which can be positions or any other state variables. If used for a 2D position visualization in an xy plane, the origin on the error ellipse (which is in a body fixed frame) represents the estimated position of an object moving around in an inertial frame.

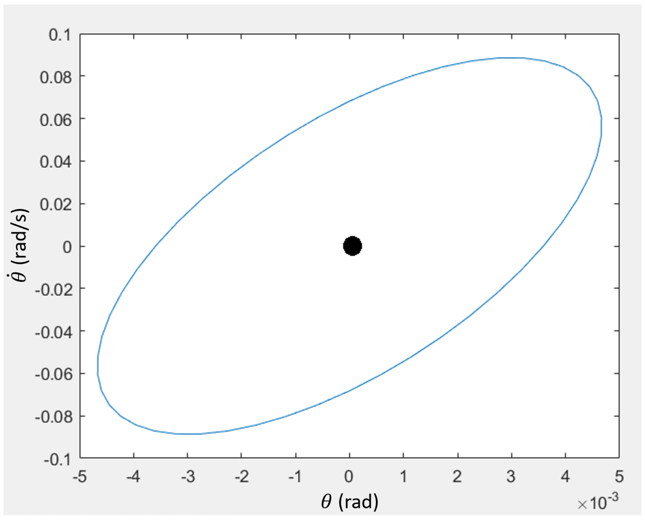

Figure 2: An error ellipse from the pendulum simulation

The 2D error ellipses used in early demos were computed using instructions from lecture notes obtained online from the University of Utah School of Computing, which can be found here. A summary of those instructions follows:

1. From the Kalman filter, there should be an output of a covariance matrix similar to the one in the equation below. If the matrix is 2 x 2, then the error ellipse will be for the 2 variables represented in the matrix. If the matrix is larger than 2 x 2, then isolate the elements relevant to the two variables that the error ellipse will represent to create a 2 x 2 matrix.

$$covmat = \begin{pmatrix} (\sigma x)^2 & \sigma xy \\ \sigma xy & (\sigma y)^2 \end{pmatrix}$$

2. Once the covariance matrix is ready, calculate the length of the ellipse axes, which are obtained by taking the square root of the covariance matrix's eigenvalues.
3. Afterwards, the counter-clockwise rotation of the ellipse is found using the equation below.

$$\theta = \frac{1}{2} \cdot \mathrm{Tan}^{-1}\left[\left(\frac{1}{aspectratio}\right) \cdot \left(\frac{2 \cdot \sigma xy}{(\sigma x)^2 - (\sigma y)^2}\right)\right]$$

4. To give the error ellipse a confidence interval of 95%, it is enlarged by a factor of 2.4477.
5. Plot the ellipse. Use the following convention to see which axis the ellipse lengths will be parallel to before they are rotated:
    a. If σx > σy, the semi-major axis length sqrt(max(eigenvalue)) is parallel to x.
    b. If σy > σx, the semi-major axis length sqrt(max(eigenvalue)) is parallel to y.
6. Rotate the ellipse coordinates counterclockwise with θ.

The above instructions were used to write a MATLAB script for visualizing a 2D error ellipsoid in both the pendulum and potential field simulations that will be discussed next. The code for the script can be found in the appendix under “Error Ellipse Generation”.

Models

Pendulum

The Simulink Pendulum Model was created to understand and explore an initial implementation of a Kalman Filter. This model was based on an incredibly helpful Mathworks virtual lesson which we highly recommend everyone does to learn the basic concepts. The model consists of a simple pendulum system whose dynamics only only include the physical properties of the pendulum and the force of gravity.

The only complexity added to this system is the addition of process and measurement noise. These were added as Band-Limited White Noise blocks with respective magnitudes of 1e-3 and 1e-4. Here is the block diagram of this system:

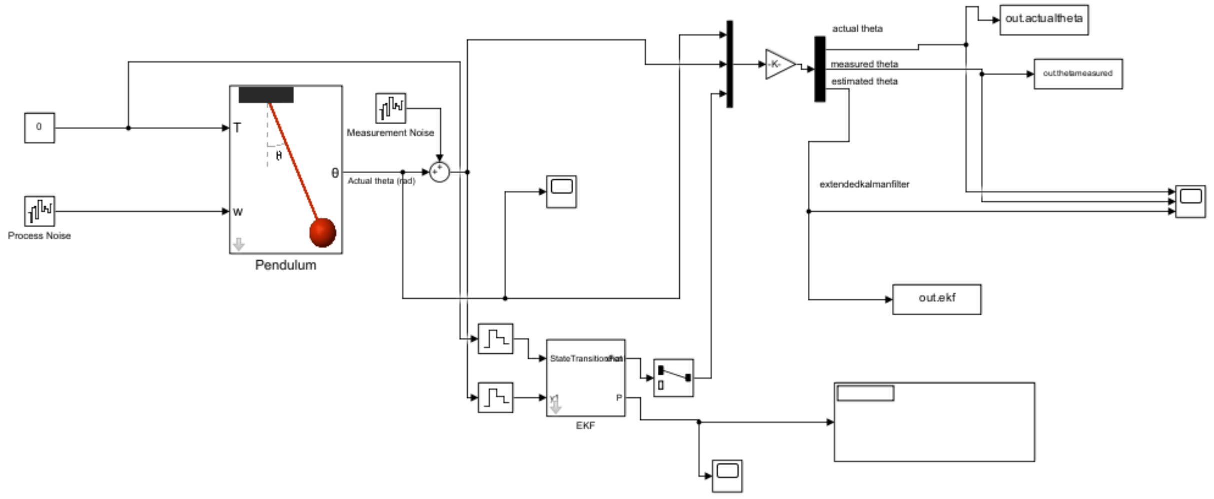

Figure 3: General Structure of Pendulum Simulation

The pendulum block is where all the dynamics of the system occur. The inputs are the external torque (T) and process noise (w). The output is the true/actual theta of the system. The true theta is carried through the end and exported to the workspace of the MATLAB window being used to run Simulink. Measurement noise is added to the actual theta to simulate measurement of the theta using an imperfect sensor. This measured theta is exported to the workspace and fed into the Extended Kalman Filter at the bottom.

Before the torque and the measured theta go into the filter, their signals are discretized by the zero order hold blocks. The outputs of the EKF are the estimated state (xhat) and the updated covariance matrix P of the EKF. The first component of the estimated state (theta) is extracted and exported to the workspace. The P covariance matrix is transmitted to scope and display block for observation.

To use this simulation and edit it, the Aerospace Toolbox and Simscape Multibody Toolbox must be installed on your MATLAB. All files of the project must be downloaded and added to the path. There is an initialization script that needs to be run before the model to initialize parameters like the Q and R covariance matrices of the EKF. Once that is done, the model can be run and edited.

2D Potential Field

The main motivation in creating a potential field model was to have a two dimensional model that a Kalman filter could be applied to. This model will have two degrees of freedom, which allows an error ellipse to be constructed from the covariance matrix that relates the covariances of the x and y positions. An error ellipse constructed from this type of covariance matrix will have a physical meaning because it shows the error in the x and y position.

The potential field model was created by selecting accelerations for the particle in the x and y direction that are based off of the x and y position. The accelerations chosen for this specific model were Ax=-20x+10y and Ay=4x-20y. Because both of these accelerations are reliant on the position of the particle in both directions the system is coupled.

Once the accelerations were selected, a Simulink model was created. The overview of the model is as follows:

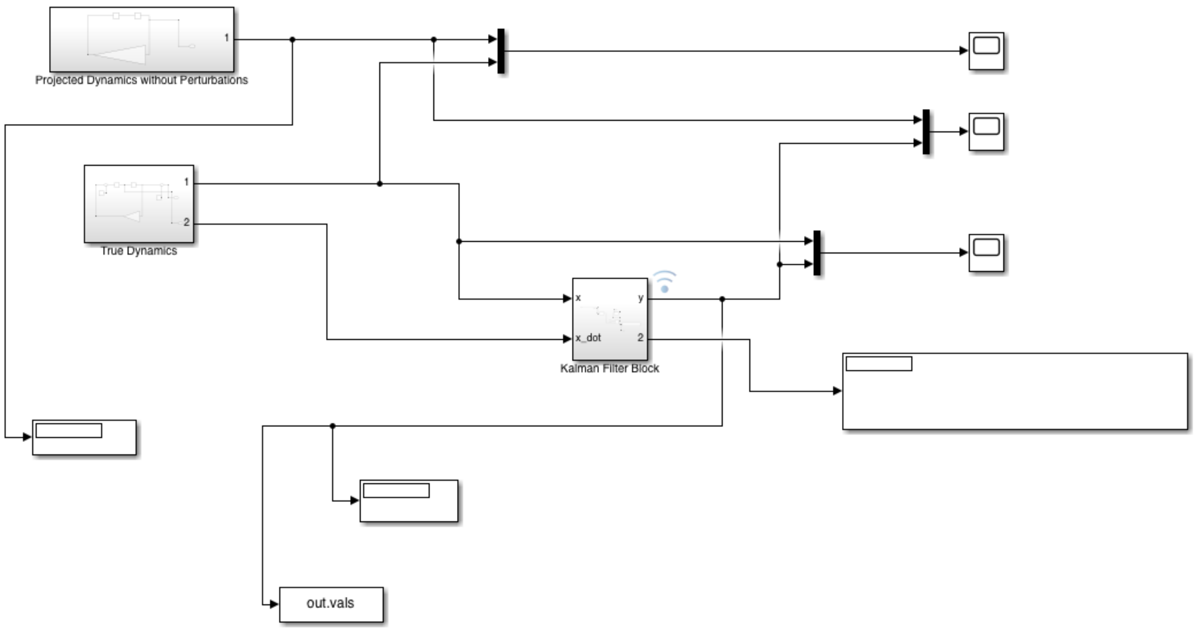

Figure 4: Overall Simulink Model for the Potential Flow Model

In order to fully understand how the accelerations were implemented in the above model, it is important to examine the Projected Dynamics without Perturbations Block:

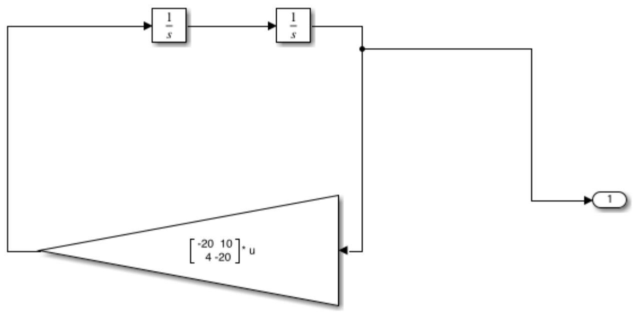

Figure 5: Projected Dynamics without Perturbations Block

As can be seen in the Projected Dynamics without Perturbations Block, the accelerations were integrated twice in order to get position, which was then fed back to the model to get the new acceleration values. This block outputs the perfect dynamics that would exist without any perturbations.

The next block to understand in the model is the True Dynamics Block. The True Dynamics Block is shown below:

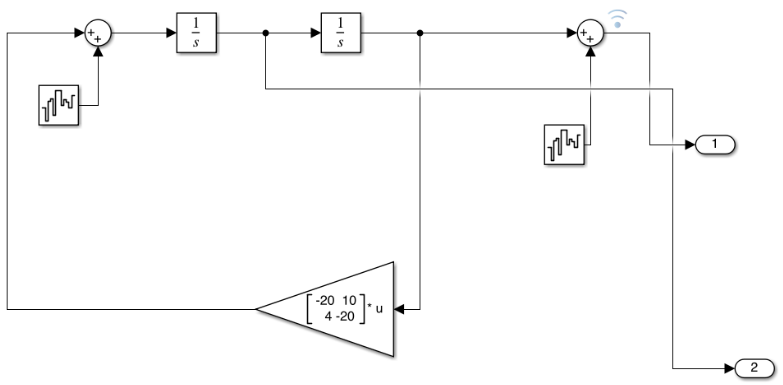

Figure 6: True Dynamics Block

As can be seen, the True Dynamics Block is extremely similar to the Projected Dynamics without Perturbations Block, however, the True Dynamics Block has noise added into the integration, and the result of the integration. These noise blocks simulate process and measurement noise respectively. By allowing for noise using the noise blocks, the True Dynamics Block can simulate if the dynamics were found using sensors or other methods. The noises show that the state can not be perfectly known. This block ultimately represents what the actual dynamics of the particle is as seen by the sensors. The process noise was selected to have a noise power of 0.01 and a sample time of 0.0001. The measurement noise was selected to have a noise power of 0.0000001 and a sample time of 0.001 as these values produced a reasonable amount of noise in the model.

The final block that should be examined is the Kalman filter block. The Kalman filter block is as follows:

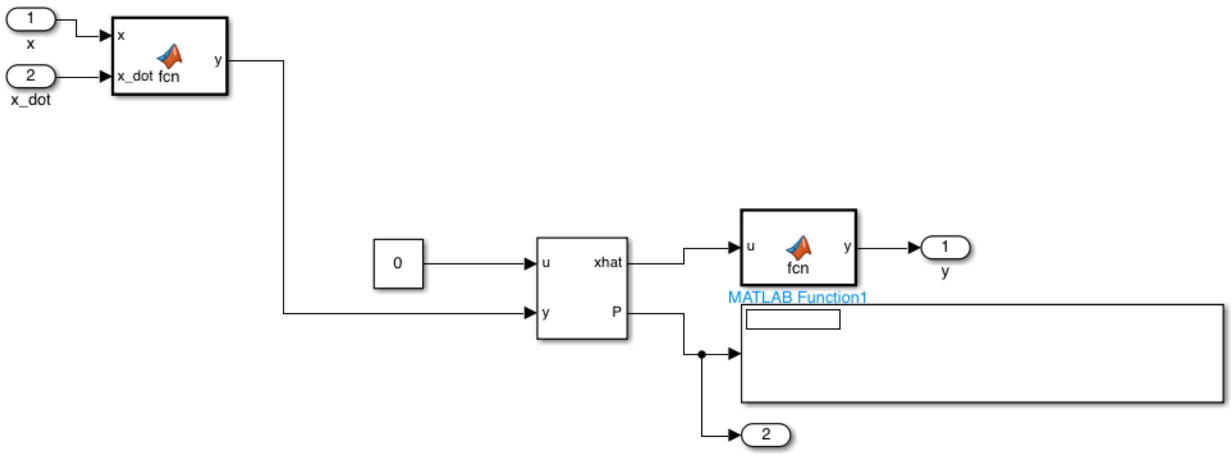

Figure 7: Kalman Filter Block

As can be seen in the Kalman filter block, the block inputs the u (which is 0 in this case as the dynamics are affected by the states not by an input as the accelerations are based on the current state), and the true dynamics as the y. The MATLAB function to the left in the Kalman Filter Block is used to sum the vector to include the positions and velocities of x and y as this is the full state because the full state is required for the Kalman filter block. The Kalman filter block also requires the Projected Dynamics without Perturbations. This, however, is not fed via a port. This is rather imputed into the Kalman filter by inputting the A,B,C, and D matrices for the state space model that represents the system.

Using the accelerations above, a state space model was created where the first state was x position, 2nd state was x velocity, 3rd state was y position, and 4th state was y velocity. The A matrix of the model based on the above dynamics is [0 1 0 0; -20 0 10 0; 0 0 0 1; 4 0 -20 0]. The B matrix of the model is [0;0;0;0]. The C matrix of the model is [1 0 0 0; 0 0 0 0; 0 0 1 0; 0 0 0 0] as the objective was to only output the x and y position. The D value was 0.

The Kalman filter also requires an input for process and measurement noise covariances. In this case the process noise covariance was selected to be [0.01 0.01 0.01 0.01] and the measurement noise covariance was selected to be 0.1. These were the values that were inputted into the Q and R prompts in the Kalman filter block respectively.

The key outputs of the Kalman filter are the new estimated state based on the Kalman filter algorithm and the covariance matrix. Because there are 4 states, the covariance matrix is a four by four matrix. The values of interest in this matrix are the 1st and 3rd column and row combinations as these values are the correlations between the x position and y position and x and y position are the first and third state. By isolating these values (1,1),(1,3),(3,1),(3,3) from the matrix, a new covariance matrix can be made that just signifies the x and y position covariance. This matrix is then hardcoded in a MATLAB script where it is manipulated to create an error ellipse. The state as outputted by the Kalman filter is outputted to MATLAB via the to workspace block.

In a MATLAB script, the two by two covariance matrix is inputted. Using the error ellipse algorithm discussed above [1], the error ellipse is created. Since the noises were selected to be constant at all times, the covariance matrix is constant, and the error ellipse is, therefore, constant for this model. Using the state outputted by Simulink, the position of the particle is plotted for all times on an x-y plane using the plot command. The plot is run in a for loop with the drawnow and clear figure commands to show a dynamically moving state (more details are shown in the MATLAB script in the appendix). Then, the error ellipse, with the center at the particle position, is plotted for all times with the drawnow and clear figure commands. This creates a moving ellipse. The end result is a particle moving with an ellipse moving around it. This is shown at one time below:

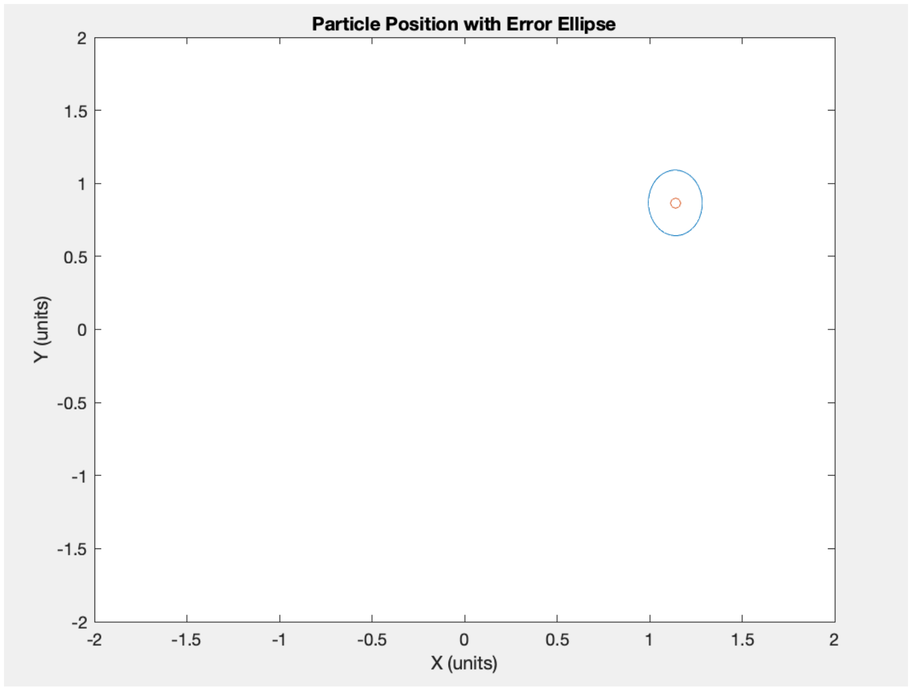

Figure 8: Error Ellipse Visualization

Overall, this model was created to demonstrate how to create a dynamic error ellipse for a 2-degree of freedom system as a proof of concept for error ellipse creation.

3D Potential Field

The main motivation for this model was to have a three dimensional model that a Kalman filter could be applied to. This model will have three degrees of freedom, which allows an error ellipsoid to be constructed from the covariance matrix that relates the covariances of the x,y, and z positions. An error ellipsoid constructed from this type of covariance matrix will have a physical meaning because it shows the error in the x,y, and z position.

The potential field model was created by selecting accelerations for the particle in the x,y, and z directions. The accelerations chosen for this specific model were Ax=-2x+1y-2z, Ay=4x-50y+2z, and Az=-2x-5y-4z. Because all of these accelerations are reliant on the position of the particle in all three directions the system is coupled.

Once the accelerations were selected, a Simulink model was created. The overview of the model is as follows:

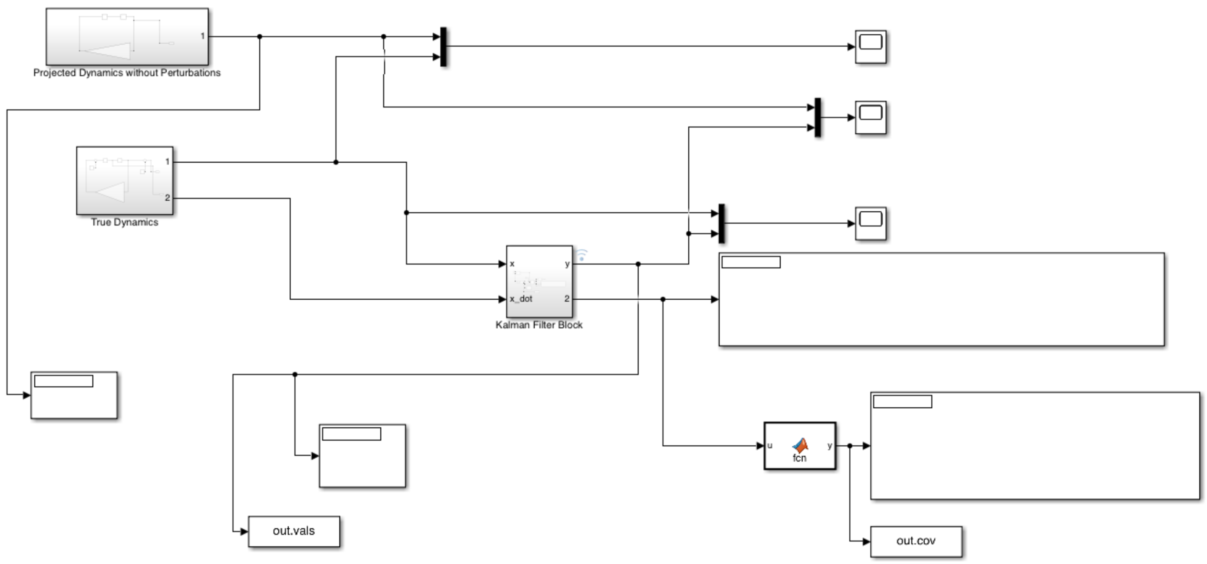

Figure 9: Overall Simulink Model for the Potential Flow Model

In order to fully understand how the accelerations were implemented in the above model, it is important to examine the Projected Dynamics without Perturbations Block:

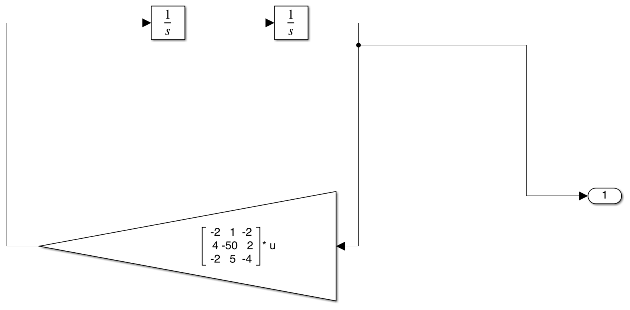

Figure 10: Projected Dynamics without Perturbations Block

As can be seen in the Projected Dynamics without Perturbations Block, the accelerations were integrated twice in order to get position, which was then fed back to the model to get the new acceleration values. This block outputs the perfect dynamics that would exist without any perturbations.

The next block to understand in the model is the True Dynamics Block. The True Dynamics Block is shown below:

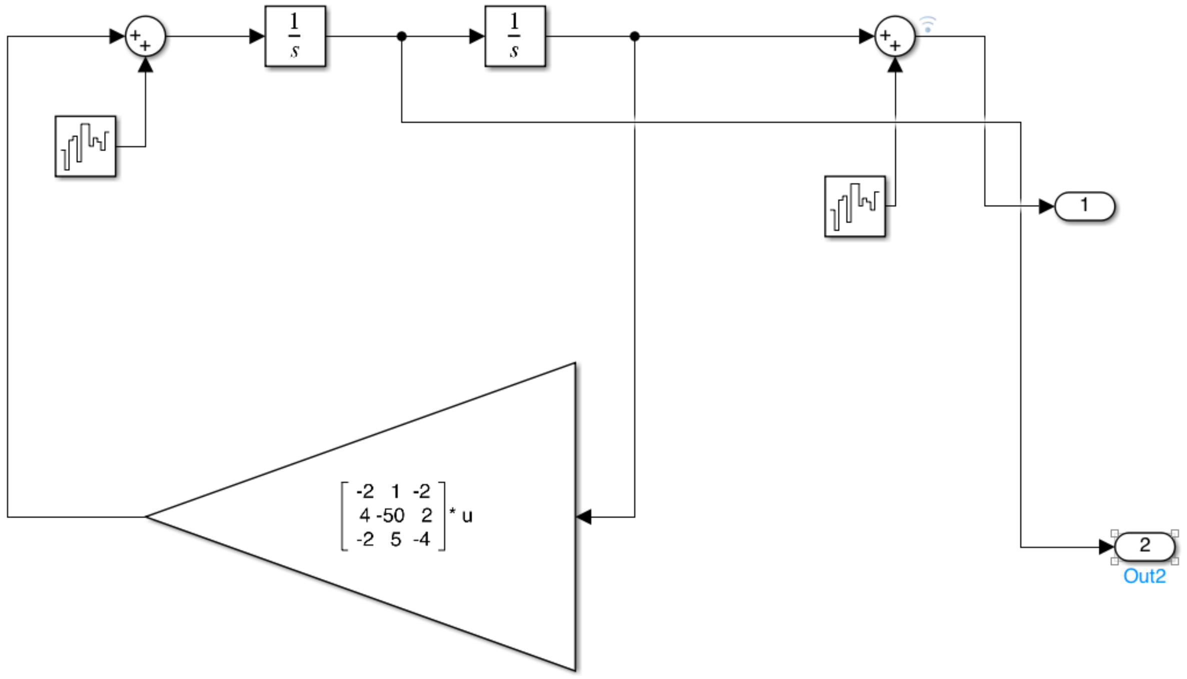

Figure 11: True Dynamics Block

As can be seen, the True Dynamics Block is extremely similar to the Projected Dynamics without Perturbations Block, however, the True Dynamics Block has noise added into the integration, and the result of the integration. These noise blocks simulate process and measurement noise respectively. By allowing for noise using the noise blocks, the True Dynamics Block can simulate if the dynamics were found using sensors or other methods. The noises show that the state can not be perfectly known. This block ultimately represents what the actual dynamics of the particle is as seen by the sensors. The process noise was selected to have a noise power of 0.01 and a sample time of 0.0001. The measurement noise was selected to have a noise power of 0.0000001 and a sample time of 0.001 as these values produced a reasonable amount of noise in the model.

The final block that should be examined is the Kalman filter block. The Kalman filter block is as follows:

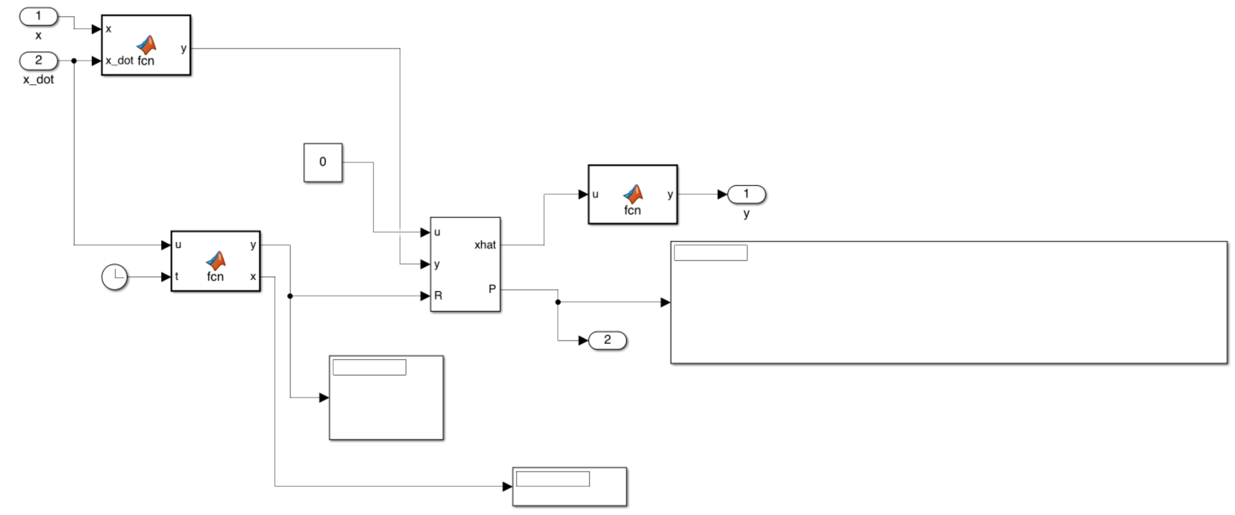

Figure 12: Kalman Filter Block

As can be seen in the Kalman filter block, the block inputs the u (which is 0 in this case as the dynamics are affected by the states not by an input as the accelerations are based on the current state), and the true dynamics as the y. The MATLAB function to the left in the Kalman Filter Block is used to sum the vector to include the positions and velocities of x and y as this is the full state because the full state is required for the Kalman filter block. The Kalman filter block also requires the Projected Dynamics without Perturbations. This, however, is not fed via a port. This is rather imputed into the Kalman filter by inputting the A,B,C, and D matrices for the state space model that represents the system.

Using the accelerations above, a state space model was created where the first state was x position, 2nd state was x velocity, 3rd state was y position, 4th state was y velocity, 5th state was z position, and 6th state was z velocity. The A matrix of the model based on the above dynamics is [0 1 0 0 0 0; -2 0 10 0 -2 0; 0 0 0 1 0 0; 4 0 -50 0 2 0; 0 0 0 0 0 1; -2 0 5 0 -4 0]. The B matrix of the model is [0;0;0;0;0;0]. The C matrix of the model is [1 0 0 0 0 0; 0 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 0 0;0 0 0 0 1 0; 0 0 0 0 0 0] as the objective was to only output the x and y position. The D value was 0.

The Kalman filter also requires an input for process and measurement noise covariances. In this case, the process noise covariance was selected to be [1 1 1 1] and the measurement noise covariance was determined by a function of velocity. The objective was to have the velocity greatly impact the value of the covariance so that the error ellipsoid shrinks with lower velocity and grows with higher velocity. For this to occur, the function must change the noise greatly based on velocity changes. The following matlab function of velocity was used to create this effect:

```
function [y]= fcn(u,t)
x=norm(u);
```

```
if t>0.5

y=0.0001*9999999999999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999999999999999999999^
(x^25/10^21);
else
   y=0.01;
end
```
This dynamically changing measurement noise was inputted into the Kalman filter.

The key outputs of the Kalman filter are the new estimated state based on the Kalman filter algorithm and the covariance matrix. Because there are 6 states, the covariance matrix is a six by six matrix. The values of interest in this matrix are the 1st,3rd, and 5th column and row combinations as these values are the correlations between the x position, y position, and z position, and x position, y position, and z position are the first, third, and fifth state. By isolating these values (1,1),(1,3),(1,5),(3,1),(3,3),(3,5),(5,1),(5,3), and (5,5) from the matrix, a new covariance matrix can be made that just signifies the x,y, and z position covariance. This matrix is then fed to a MATLAB script where it is manipulated to create an error ellipse. The state as outputted by the Kalman filter is outputted to MATLAB via the to workspace block.

In a MATLAB script, the 3 by 3 covariance matrix is taken from the Simulink model. Using the error ellipsoid software derived from source code [2] , the error ellipsoid is created. The plot is run in a for loop with the drawnow command to show a dynamically moving state (more details are shown in the MATLAB script in the appendix). Then, the error ellipsoid, with the center at the particle position, is plotted for all times with the drawnow command. This creates a moving ellipse. The end result is a particle moving with an ellipse moving around it. This is shown at one time below:

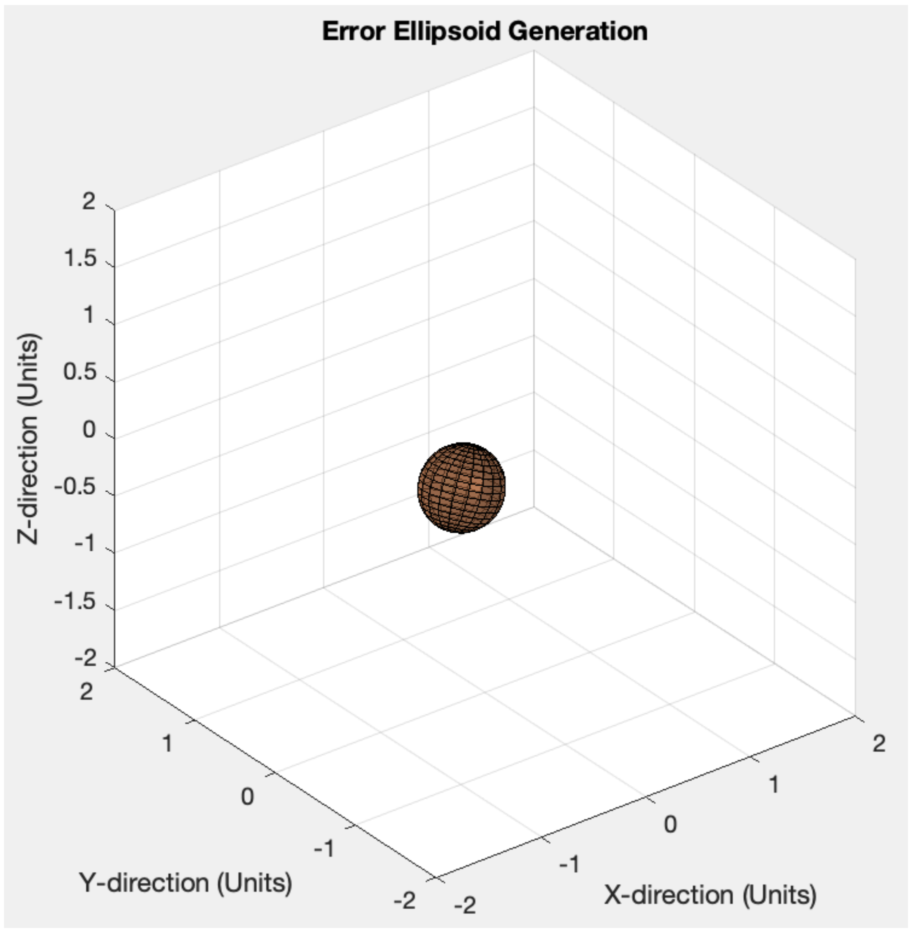

Figure 13: Error Ellipsoid Visualization

Overall, this model was created to demonstrate how to create a dynamic error ellipse for a 3-degree of freedom system as a proof of concept for error ellipsoid creation.

Current Next Steps

As of right now, we have figured out how to implement an extended Kalman filter (EKF) on a dynamic system in 2-D. Our next steps will involve figuring out how to model and simulate a drone with the required sensors in Gazebo so we can get simulated sensor inputs. This will involve implementing flight software for the drone in the simulation as well. Based on the sensor inputs from the simulation, we can then write the algorithm to determine the drone's estimated state and the error ellipsoid surrounding the drone. After figuring out how to compute an error ellipsoid, we will then need to visualize the ellipsoid around our drone within the simulation.

After determining the estimated drone state and the error ellipsoid surrounding the drone, we can begin determining a method to compute a collision avoidance trajectory. This will start with developing the software to make the drone follow set waypoints. Eventually, obstacles will be placed in the path of the drone and we will need to implement a collision avoidance algorithm.

Once we confirm that our drone has the ability to avoid collisions at a specified TBD success rate, then we can begin testing our software on the actual drone the USRC Hardware team builds. Our testing location will most likely be at the top of the parking garage at the ASE building, inside the batting cage.

Appendix

Error Ellipse Generation

```
P = [2.182e-5 0.000265; 0.000265 0.007868];

[V, D] = eig(P);

eig1 = D(1,1);
eig2 = D(2,2);

a = sqrt(eig1);
b = sqrt(eig2);

theta = .5*atan( 2*P(1,2) /  (P(1,1) - P(2,2)));

scalefactor = 2.4477;

t = 0:2*pi/50:2*pi;

x = a*cos(t);
y = b*sin(t);

points = [x;y];
R = [cos(theta) -sin(theta); sin(theta) cos(theta)];
```

```
points = R*points;
Points = points*scalefactor;

figure(1)
plot(points(1,:), points(2,:))
```

Error Ellipse Animation Code

```
p=animatedline;
C=[0,0];
a=2;
th=linspace(0,2*pi);
xe=C(1)+a*cos(th);
ye=C(2)+a*sin(th);
f=[0.05033 0 0.000326;1 1 1; 0.000326 1 0.02119];
xc=out.vals(1,:,:);
xc=xc(:,:);
yc=out.vals(2,:,:);
yc=yc(:,:);
for i=1:100:length(xc)
   u=[f(1,1) f(1,3);f(3,1) f(3,3)];
   [V, D] = eig(u);
   eig1 = D(1,1);
   eig2 = D(2,2);
   a = sqrt(eig1);
   b = sqrt(eig2);
   theta = .5*atan( 2*u(1,2) /  (u(1,1) - u(2,2)));
   scalefactor = 2.4477;
   t = 0:2*pi/50:2*pi;
   x = a*cos(t);
   y = b*sin(t);
   points = [x;y];
   R = [cos(theta) -sin(theta); sin(theta) cos(theta)];
   points = R*points;
   plot(xc(i)+points(1,:), yc(i)+points(2,:))
   hold on
   plot(xc(i),yc(i),'-o')
   hold on
   p=animatedline;
   addpoints(p,xc(i),yc(i));
   xlim([-2 2])
   ylim([-2 2])
   xlabel('X (units)')
   ylabel('Y (units)')
   title('Particle Position with Error Ellipse')
   drawnow
   clf
end
```

Error Ellipse Animation Code

```
p=animatedline;
xc=out.vals(1,:,:);
xce=xc(:,:);
yc=out.vals(2,:,:);
yce=yc(:,:);
zc=out.vals(3,:,:);
zce=yc(:,:);
for i=7000:50:length(xc)
      Cov=[out.cov(:,:,i)];
      mu=[xce(i);yce(i);zce(i)];
      [U,L] = eig(Cov);
      % L: eigenvalue diagonal matrix
      % U: eigen vector matrix, each column is an eigenvector
      % For N standard deviations spread of data, the radii of the eliipsoid
      will
      % be given by N*SQRT(eigenvalues).
      N = 1; % choose your own N
      radii = N*sqrt(diag(L));
      % generate data for "unrotated" ellipsoid
      [xc,yc,zc] = ellipsoid(0,0,0,radii(1),radii(2),radii(3));
      % rotate data with orientation matrix U and center mu
      a = kron(U(:,1),xc);
      b = kron(U(:,2),yc);
      c = kron(U(:,3),zc);
      data = a+b+c; n = size(data,2);
      x = data(1:n,:)+mu(1);
      y = data(n+1:2*n,:)+mu(2);
      z = data(2*n+1:end,:)+mu(3);
      % now plot the rotated ellipse
      % sc = surf(x,y,z); shading interp; colormap copper
      figure(5)
      h = surfl(x, y, z); colormap copper
      title('Error Ellipsoid Generation')
      axis equal
      alpha(0.7)
      xlim([-2 2])
      ylim([-2 2])
      zlim([-2 2])
      xlabel('X-direction (Units)')
      ylabel('Y-direction (Units)')
      zlabel('Z-direction (Units)')
      drawnow
end
```

References

[1] “How to Draw a Covariance Error Ellipse?,” https://visiondummy.com https://www.visiondummy.com/2014/04/draw-error-ellipse-representing-covariance-matrix/

[2] “Plot 3D Error Ellipsoid,” https://kittipatkampa.wordpress.com. https://kittipatkampa.wordpress.com/2011/08/04/plot-3d-ellipsoid/