EECE 212
Project

OBJECTIVES

- Use MATLAB and Linear Algebra to explore Dynamical Systems.

EQUIPMENT

MATLAB

Project Rules

- You can work by yourself, or partner with one of your classmates: Group size ≤ 2
- Either way, please sign-up for your group on BrightSpace. If you don't sign-up for a group, you won't be able to see the group assignment.
- Students in one group get the same grade.

Report Details

Please submit your report and executable m-files via BrightSpace by the due date/time:

- **Report: the submitted report for the project should be in a more formal format than previous lab reports**. You should submit a report describing how the functions you have written work (more detail than is included in the comments).
    - Also explains what you did to test them (e.g., which code can be used to regenerate your Figure 1 plot), and explains the characteristics of the results (e.g., comments about the dynamical systems and linear algebra aspects).

- **Matlab codes**: You should also submit the m-files you created for the lab. Codes include:
    - The functions you have created. Functions are expected to be written using the methods and styles discussed throughout this course. Functions will be judged on their use of efficient means of implementation and clarity of their operation and explanation.
    - Code that you used for generating the plots in the report. TAs will use these codes to regenerate your results. If the code cannot be executed (due to error), you will not get any points on that problem.

You can submit a combined "Report + Code" in the Live Script format. If you chose to do so, submit:

- **Report:** same requirement as mentioned above, include the formal report and the Figures in a .pdf format file.
- **Codes:** the same requirement as mentioned above, submit your .mlx file and all functions. TA will run your live script.

Background

In Lab 9 and 10, we have looked into using matrix to represent a linear dynamic system. In this project, we will extend the idea from several perspectives:

- Add some physical disturbance to our dynamic equation, which is to simulate the effect of wind blowing our system (e.g., the wind blowing on an airplane).
- Add measurement noise to our sensors, so what we have "seen" from the sensor is not the real state of the system. These noises will make least squares estimation very difficult to use.
- We will use a more advanced method called "recursive least squares" to handle the noise.

I. Dynamic systems with disturbances

For more complicated dynamical systems, engineers are more likely to develop the state space model directly. Then it can be used to iteratively solve for the dynamical system's behavior as we described above. But, it is also possible to use the model to understand why the system behaves the way it does. It is also possible to use the state-space model to develop engineering techniques that allow you to *control* the system's behavior (e.g., an auto-pilot control system) or aid in the *measurement* of information about the system's state using sensors (e.g., an airplane's navigation system uses sensors to measure the plane's velocity and acceleration). Note that it is not possible to *control* a system's behavior without having accurate *measurements* of how it is responding! Thus, here we will focus on using state space models to aid in the measurement of the system's behavior. In this section, we will describe a more general state-space model than what was developed before. This more general state-space model will characterize the dynamics of the system being modeled. In the next section, we will introduce a measurement model that can be used together with the state-space model to extract information about the system's state behavior.

The previous section developed a simple state-space model of $\mathbf{q}[n+1] = \mathbf{A}\mathbf{q}[n]$ with specific forms for the state vector and the state transition matrix (e.g., $\mathbf{A}$ was upper diagonal with 1s on the diagonal). Let the state vector $\mathbf{q}[n]$ be $N \times 1$ – that is, we have N state elements; then we say we have an N-state system or the system has an order of N. Then the state transition matrix $\mathbf{A}$ is $N \times N$. We can then write a more general form of the state space dynamics model as

$$\mathbf{q}[n+1] = \mathbf{A}\mathbf{q}[n] + \mathbf{B}\mathbf{x}[n], \tag{1}$$

where the $q \times 1$ vector $\mathbf{x}[n]$ is a vector of inputs to the system (e.g., stimuli to the system – such as voltages, forces, etc. – that affect the system's behavior), and the $N \times q$ matrix $\mathbf{B}$ is the so-called input matrix that forms N linear combinations of the inputs and adds one to each state. Sometimes the inputs are modeled as known functions of time but it is common to model them as random to mimic the randomness of the effect of things like wind forces, etc. It is common to model the effect of wind as causing a random change in the velocity state as we will explore in the lab work below.

In MATLAB we can create a sequence of random $\mathbf{x}[n]$ vectors from a Gaussian distribution (which is most commonly used) as follows[1]:

[1] Note that the "n" at the end of randn means that it generates random numbers from the "normal" distribution – "normal" is another term for "Gaussian".

X = randn(q,M);

where q is the dimension of the **x**[*n*] vector and M is the number of vectors to create. This puts one **x**[*n*] vector in each column of the q by M matrix X.

Lab Work

Write a MATLAB function called gen_state.m to implement a 2D aircraft trajectory (x & y axes) under the following conditions:

- **The state transition matrix gives a constant-velocity model for the dynamics**
- **The dimension of the random input vectors is $q = 2$ and these are created using the randn command.**
- **The input matrix models:**
  - **No random input applied to either of the position states**
  - **The first element of the random input vector is applied to the x velocity state with a multiplier of σ_x**
  - **The second element of the random input vector is applied to the y velocity state with a multiplier of σ_y**
  - **From these conditions, you should be able to deduce the proper form for B**
- **Provide as output the time-varying state vector**
  - **The function itself should NOT make any plots**

Write a MATLAB script called project_1.m to perform the following experiment:

- **Pick the initial positions as (0,0) and the initial velocities as (100,100) m/s**
- **Let the time spacing be $T = 1$ second and consider a time range from 0 to 60 seconds.**
- **Explore the effect of the values selected in the B matrix on the trajectory**
  - **Plot the y position versus the x position to display the trajectory**
  - **By selecting the B matrix values small you can get trajectories with small deviations from a true constant velocity path**
    - **If you set them both to 0 your aircraft's path will be a perfect constant-velocity path!**
  - **By selecting the B matrix values fairly large you can get trajectories with large deviations from a true constant velocity path that are not realistic with the effect of wind but can instead model a maneuvering aircraft (interesting!!).**
    - **Don't start real large… work your way up and see how the nature of the trajectory changes**

II. Using Sensors to Measure States

Now suppose we have p sensors we wish to use to measure the state behavior of our system (e.g., speed sensors and/or accelerometers, etc.). Because any measurement has error in it we must include something in the measurement model that models errors. Also, our sensors may not be able to measure the state elements directly – instead each sensor may only be able to measure a linear combination of the state values at a given time. This then leads to the following measurement model[2]

$$\mathbf{y}[n] = \mathbf{C}\mathbf{q}[n] + \mathbf{D}\mathbf{w}[n], \tag{2}$$

where $\mathbf{y}[n]$ is a $p \times 1$ vector of the sensor measurements made at time nT, $\mathbf{C}$ is the $p \times N$ sensor matrix that forms linear combinations of the state elements, $\mathbf{D}$ is a $p \times p$ diagonal matrix that holds the standard deviation of each measurement noise, and $\mathbf{w}[n]$ is the $p \times 1$ "noise" vector that contains random values (with a standard deviation of 1) to model the measurement errors made by the sensors. In MATLAB we can create a sequence of random $\mathbf{w}[n]$ vectors from a Gaussian distribution with a standard deviation of 1 as follows:

```
W = randn(p,M);
```

where p is the dimension of the $\mathbf{y}[n]$ vector and M is the number of vectors to create. This puts one $\mathbf{w}[n]$ vector in each column of the p by M matrix W. Then we pick the elements in $\mathbf{D}$ to provide the desired level of measurement error for each element in $\mathbf{y}$[n]: let σ_i be the standard deviation (i.e., noise level) of the i^{th} element in $\mathbf{y}$[n] then

$$D = diagonal(\sigma_1, \sigma_2, \ldots, \sigma_p).$$

Note that the MATLAB command "diag" is convenient for creating a diagonal matrix like $\mathbf{D}$.

Lab Work

Write a MATLAB function called measure_state.m that will implement the measurement model discussed in this section. Your function should:

- **Accept a general sensor matrix C from the user**
- **Accept a vector of noise levels that will be used for the diagonal elements of the matrix D**
  - **These values must be greater than 0**
  - **Your program will put those into a diagonal matrix D**
- **Accept a matrix that holds the sequence of state vectors to be measured**
- **Provide as output the time-varying measurement vector**
  - **The function itself should NOT make any plots**

Now write a script function "project_2.m" that uses your measure_state.m together with your gen_state.m to yield sensor measurements of a 2D aircraft trajectory (x & y axes) under the following conditions:

[2] Note that this is not the most general measurement model but it is sufficient for our purposes here.

- **The state transition matrix gives a constant-velocity model for the dynamics**
- **The dimension of the random input vectors is $q = 2$ and these are created using the randn command.**
- **The input matrix models:**
  - **No random input applied to either of the position states**
  - **The first element of the random input vector is applied to the x velocity state with a multiplier of σ_x**
  - **The second element of the random input vector is applied to the y velocity state with a multiplier of σ_y**
  - **From this you should be able to deduce the proper form for B**
- **Pick some reasonable values for σ_x and σ_y (e.g., $\sigma_x = 1$, $\sigma_y = 10$)**
- **Pick the initial positions as (0,0) and the initial velocities as (100,100) m/s.**
- **Let the time spacing be $T = 1$ second and consider a time range from 0 to 60 seconds.**
- **Assume that you have sensors that can measure the states directly – that is you have one sensor that measures x position, one that measures y position, one that measures x velocity, and one that measures y velocity**
  - **Start using measure noise levels (standard deviations) that are 25m for the position measurements and 10m/s for the velocity measurements.**
  - **Explore the effect of setting the measurement noise levels to different values**
    - **Suggestion: you can plot the distance between real states and measurements under different noise levels**
    - **Suggestion: search "data representation methods"**

III. Exploiting the Dynamic Model to Improve Sensor Measurements

This section will discuss a famous algorithm called "recursive least squares"(RLS) that uses the dynamics model to process the measurements and give an accurate estimate of the system's state. Here we will not delve into how or why the recursive least squares filter works but rather focus on a simple demonstration of its capability for a very simple case.

First, a description of a fairly general version of the recursive least squares filter for a dynamic system described by the state model in (1) and a sensor system described by the measurement model in (2), which are repeated here for ease:

$$\mathbf{q}[n+1] = \mathbf{A}\mathbf{q}[n] + \mathbf{B}\mathbf{x}[n], \tag{3}$$

$$\mathbf{y}[n] = \mathbf{C}\mathbf{q}[n] + \mathbf{D}\mathbf{w}[n]. \tag{4}$$

We've set these up here to have the dynamic system's inputs $\mathbf{x}[n]$ be random Gaussian with a standard deviation of 1 and the noise levels controlled by setting values in the **B** matrix. Similarly, we've set up the measurement noises $\mathbf{w}[n]$ to be random Gaussian with a standard deviation of 1, and the noise levels are controlled by setting values in the **D** matrix.

The filter is "built" using information about these two models. In practice, this knowledge is not perfectly known, but we will assume we know it perfectly. Here are the things the recursive least squares method needs from the models:

- A matrix
- All measurements: y
- A matrix R which has forgetting factors

If you like to know more about the recursive least squares method, read the wiki page[3]

Once the recursive least squares is coded using that model information it will accept the sensor's measurements $\mathbf{y}[n]$ and produce an estimate of the system states. In practice the sensor measurements are provide to the recursive least squares one at a time – but when simulating this in MATLAB it is common to provide all of the measurements at once as an input to the m-file. A version of the recursive least squares algorithm is given in the following equations:

- Input and initialization:
  - A matrix, R matrix, and all measurements y
  - P(:,:,1) = I; % initialize the first covariance matrix
- Assume our system has the following format:
  - $q(n+1) = Aq(n) + noise$
  - We can use measurements (y, output from measurement_state.m) to estimate $\hat{q}$
- Each step, do the following for $n = 1,2,3, \ldots$? (?: think about when the loop ends):
  - Kalman Gain: $K(n) = P(n-1)A^T(AP(n-1)A^T + R)^{-1}$
  - Covariance matrix: $P(n) = R^{-1} - K(n)AR^{-1}P(n-1)$
  - Estimated states: $\hat{q}(n) = \hat{q}(n-1) + K(n)\big(y(n) - A\hat{q}(n-1)\big)$

Note: $y(n) - A\hat{q}(n-1)$ is the estimation error. The RLS method will update its estimation gain (estimation strategy) over time. Note that this version has been specialized to the setting we are considering here (googling "recursive least squares" + "matlab" won't give you a direct answer)!

In MATLAB it is typical to input all of these measurement vectors at once in a $p \times (k+1)$ matrix called Y. Once those variables are provided to the recursive least squares routine the algorithm sets up some initial variables and then iterates through the steps in the box labeled "Do the following for….". Once the iterations are done the routine outputs the results computed for the sequence vectors $\hat{\mathbf{q}}[0], \hat{\mathbf{q}}[1], \ldots, \hat{\mathbf{q}}[k]$which are typically output as the columns of a matrix called Q_hat.

Looking at the box that contains the main computations (the box labeled "Do the following for….") we see that there are a handful of things computed in five steps. The fourth step is the one where we compute what we want: an estimate of the n^{th} state vector. This is the thing that we must store for output from the m-file: $\hat{\mathbf{q}}[0], \hat{\mathbf{q}}[1], \ldots, \hat{\mathbf{q}}[k]$– note that in the initialization step we do create $\hat{\mathbf{q}}[-1]$ but we do not output that value (it is only used internally). The rest of the things that are computed are used only within the loop so we need not store all their past values[4].

Lab Work

Write a MATLAB function called RLS_estimation.m that will implement the recursive least squares estimation discussed in this section. Your function should:

- **Accept matrices A,R from the user for a specific state model and forgetting factors**

[3] https://en.wikipedia.org/wiki/Recursive_least_squares_filter#RLS_algorithm_summary

[4] If really want to store paste values, matrices, … you may need store K, P in 3D matrices: https://www.mathworks.com/help/matlab/math/multidimensional-arrays.html

- **Accept a sequence of output vectors y[0], y[1], … , y[*k*] for an arbitrary *k* number of points**
- **Provide as output the sequence of state estimates q_hat[0], q_hat[1], … , q_hat[*k*]**

Note: the "R" matrix in the algorithm is called "forgetting factors", search/Google what that is and what number you should pick for the forgetting factors.

Now write a script function "project_3.m" that uses your gen_state.m, measure_state.m, and RLS_estimation.m as follows:

- **Generate a sequence of states for a 2-D constant-velocity model of an aircraft with an initial position of (0,0) in meters and an initial velocity of (100,100) m/s.**
  - **The dynamic model should be driven with noises that are input to the velocity states with a standard deviation level of 5 m/s for each x and y velocity.**
- **Measure the states using sensors that measure the states directly**
  - **The standard deviation level for the position measurements should be 25m**
  - **The standard deviation level for the velocity measurements should be 10m/s**
- **Use the RLS Filter to compute estimates of the states (which are the x & y positions and x & y velocities)**
- **Make an y-position vs x-position plot that shows three curves:**
  - **The true positions**
  - **The measured positions**
  - **The estimated positions**
  - **Use a different line style, color, and symbol for each curve (see the information in Figure 1)**
  - **Use the "legend" command to add a legend that identifies the three curves (see the help for the "legend" command).**

Experiment to explore the ability of the RLS Filter to provide better estimates of the aircraft position than is possible from the sensors alone – try increasing the noise levels on the measurements. Discuss your results. Again, you can plot the distance between real states and measurements under different noise levels.

Various line types, plot symbols and colors may be obtained with plot(X,Y,S) where S is a character string made from one element from any or all the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

For example, plot(X,Y,'c+:') plots a cyan dotted line with a plus at each data point; plot(X,Y,'bd') plots blue diamond at each data point but does not draw any line.

Figure 1: From the help entry for "plot" command.