Project Update - Approximating the Approximator

Dane Sabo

ME 2046 - Project

April 13th, 2025

```
clear all
close all
```

System Representation and Modeling

For this project, we are modeling a hard disk drive read/write head. This system essentially has two states: head position and velocity. The governing equation for this system is represented by:

$$J\ddot{\theta} + C\dot{\theta} + K\theta = K_i i + W_d$$

Where J, C, and K are physical constants relating to the passive mechanical properties of the read write head. J is the moment of inertia of the read write head, C is viscous damping from the fluid inside the disk enclosure, of the motor movement, and any other non-conservative forces. K is a restorative spring force used such that when the hard drive is turning off or loses power, the read write head will rest on the platter in a special 'landing zone'. Finally, K_i is introduced to convert demanded position into motor torque.

What are some good values for these constants? A MATLAB Documentation example has example has previously used:

```
J = 0.01; %kgm^2
C = 0.004; %Nm/(rad/s)
K = 10; %Nm/rad
K_i = 0.05; %Nm/rad
```

The precision of these values is not critical at this point, as studies here are comparative anyways.

Stabilty, Observability, and Controllability

First, we can create two states θ and $\dot{\theta}$.

$$\dot{\theta} = x_2$$

$$\theta = x_1$$

and then adapt our governing equation to achieve the continuous state space representation where we presume we only measure the position of the read write head:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{-K}{J} & \frac{-C}{J} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{K_i}{J} \end{bmatrix} i + \begin{bmatrix} 0 \\ \frac{1}{J} \end{bmatrix} W_d$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

We include in this formulation a disturbance W_d.

We can put these matrices into a MATLAB state space.

```
F = [0 1; -K/J -C/J];
G = [0; K_i/J];
G_disturb = [0; 1/J];
C = [1 0];
D = [0];
names = ['Angular Position', 'Angular Velocity'];
sys_cont = ss(F,G,C,D)
```

```
sys_cont =

  A =
         x1     x2
   x1     0      1
   x2  -1000   -0.4

  B =
       u1
   x1   0
   x2   5

  C =
       x1  x2
   y1   1   0

  D =
       u1
   y1   0

Continuous-time state-space model.
Model Properties
```

Now, we can convert this system to a discrete time system. First, we assume a sampling rate--for this project I'm assuming 15 ksps (kilosamples per second), typical of an Arduino's analog to digital converter.

```
samples_per_second = 15e3; %samples per second
Ts = 1/samples_per_second; %s
sys = c2d(sys_cont, Ts, 'zoh')
```

```
sys =

  A =
              x1         x2
   x1          1  6.667e-05
   x2   -0.06667          1

  B =
              u1
   x1  1.111e-08
   x2  0.0003333

  C =
       x1  x2
   y1   1   0

  D =
```

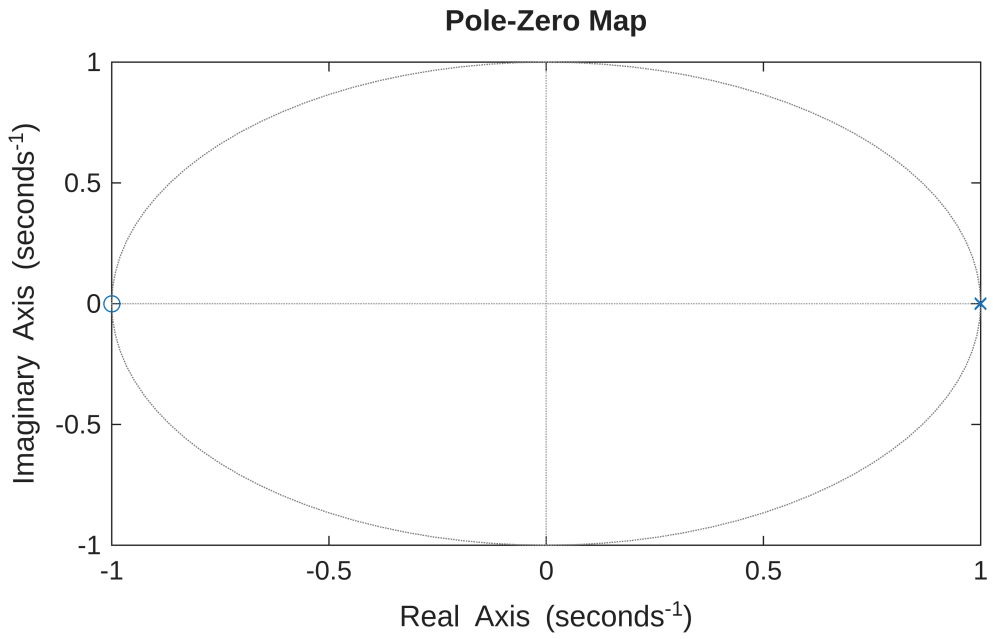
```
        u1
   y1   0

Sample time: 6.6667e-05 seconds
Discrete-time state-space model.
Model Properties
```

Then, we check stability of this system. This can be done by examining the eigenvalues of A:

```
eig(sys)
```

```
ans = 2×1 complex
   1.0000 + 0.0021i
   1.0000 - 0.0021i
```

```
pzmap(sys)
```



This indicates that our system is stable, with poles juuuuust inside the unit circle. This makes sense -- the system incorporates a mechanical spring that forces the system to return to zero angular position, while the damping acts as a non-conservative force creating stability. That being said, the damping is very small compared to the spring constant, so we would anticipate a very 'wobbly' system.

What about controllability? We can use a built in MATLAB function to check.

```
rank(ctrb(sys))
```

```
ans =
2
```

This is good! Our system is controllable, as the controllability matrix has rank 2, the number of states.

What about observability?

```
rank(obsv(sys))
```

```
ans =
2
```

Perfect! With the rank of the observability matrix being full, our system is both controllable and observable.

Model and Controller

Because of the nature of this simulation, I implement this simulation manually as opposed to using a simulink model. This solver incorporates an optional observer and full state feedback controller.

```
function [x_hist, y_hist, u_hist, x_hat_hist] = solve_difference_eq(opts)
    % Solves a discrete-time state-space difference equation iteratively.
    % All inputs are passed in **one structure** so you can build the call
    % incrementally without remembering positional order.
    %
    % Required fields in **opts**
    %   • sys  – discrete-time state-space system (ss object)
    %   • x0   – initial state column vector
    %   • N    – number of discrete simulation steps
    %
    % Optional fields (leave out or set to [])
    %   • L    – observer gain matrix
    %   • K    – state-feedback gain matrix
    %   • r    – reference trajectory [nu × N]
    %   • u    – open-loop input trajectory [nu × N]
    %
    % Outputs
    %   x_hist     – state history  [nx × (N+1)]
    %   y_hist     – output history [ny × N]
    %   u_hist     – input history  [nu × N]
    %   x_hat_hist – observer-state history (empty if no observer)

    arguments
        opts struct
    end

    % ---------- Convenience handles & defaults ----------------------------
    req = {'sys','x0','N'};                                % required fields
    for f = req
        if ~isfield(opts,f{1})
            error('Missing required field opts.%s',f{1});
        end
    end

    sys = opts.sys;
    x0  = opts.x0(:);                % force column
    N   = opts.N;
    L   = opts.L;
    K   = opts.K;
    r   = opts.r;
    u   = opts.u;
```

```
% ----------------- Diagnostics ----------------------------------------
fprintf('\n*** solve_difference_eq called with: ***\n');
vars = struct('x0',x0,'N',N,'L',L,'K',K,'r',r,'u',u);
vars  %#ok<NOPRT>

% ----------------- Extract system matrices ---------------------------
[A,B,C,D] = ssdata(sys);
Ts = sys.Ts;
[nx,nu]  = size(B);
ny       = size(C,1);

% ----------------- Pre-allocate histories ----------------------------
x_hist     = zeros(nx,N+1);
y_hist     = zeros(ny,N);
u_hist     = zeros(nu,N);

x_hist(:,1) = x0;

% Observer bookkeeping
useObserver = ~isempty(L);
if useObserver
    fprintf('Observer enabled.\n');
    x_hat       = [0;0];
    x_hat_hist  = zeros(nx,N+1);
    x_hat_hist(:,1) = x_hat;
else
    x_hat_hist = [];
end

% Use full state feedback?
useFB = ~isempty(K);
if useFB, fprintf('State-feedback enabled.\n'); end

if ~useFB && isempty(u)
    error('Either opts.K or opts.u must be supplied.');
end

% Ensure reference & open-loop inputs are sized correctly if provided
if ~isempty(r) && size(r,2)~=N
    error('opts.r must have N columns.');
end
if ~isempty(u) && size(u,2)~=N
    error('opts.u must have N columns.');
end

% ----------------- Simulation loop -----------------------------------
for k = 1:N
    % Compute input
    if useFB % USE FEEDBACK
```

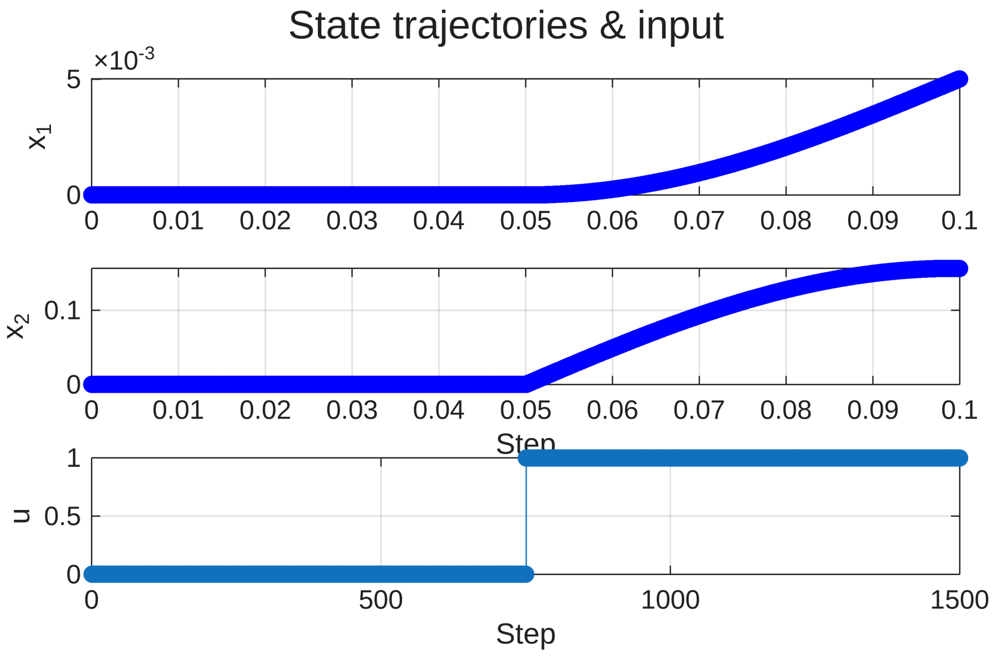
```
            if useObserver % WITH AN OBSERVER
                u_k = K*(-x_hat_hist(:,k) + (isempty(r) * 0 + ~isempty(r) *
r(:,k)));
            else % WITHOUT AN OBSERVER
                u_k = K*(-x_hist(:,k) + (isempty(r) * 0 + ~isempty(r) *
r(:,k)));
            end
        else % NO FEEDBACK
            u_k = u(:,k);
        end

        %D ocument input
        u_hist(k) = u_k;

        % Plant output
        y_hist(:,k) = C*x_hist(:,k) + D*u_k;

        % Propagate state / observer
        if useObserver
            x_hat = A*x_hat + B*u_k + L*(y_hist(:,k) - C*x_hat - D*u_k);
            x_hat_hist(:,k+1) = x_hat; %
        end

        %Calculate Next State
        x_hist(:,k+1) = A*x_hist(:,k) + B*u_k;
    end

    % ----------------- Plot results ----------------------------------------
    figure;
    time = (0:N)*Ts;
    for i = 1:nx
        subplot(nx+1,1,i);
        if useObserver
            plot(time,x_hat_hist(i,:),'-xr', time,x_hist(i,:),'-ob');
            legend('x_{hat}', 'x')
        else
            plot(time,x_hist(i,:),'-ob');
        end
        ylabel(sprintf('x_%d',i)); grid on;
        if i==nx, xlabel('Step'); end
    end
    subplot(nx+1,1,nx+1);
    stairs(1:N,u_hist,'-o'); ylabel('u'); xlabel('Step'); grid on;
    sgtitle('State trajectories & input');

end
```

Now, let's run it!

```
t_max = 0.1; %s
```

```
N = round(t_max/Ts); %num of steps
t = 0:Ts:t_max; % array of real times vs step
step_time = 0.05;

u_traj = zeros(1, N);


% Set up reference trajectory
step_index = find(t >= step_time, 1);
u_traj(1, step_index:end) = 1;

opts      = struct();      % start empty
opts.sys  = sys;
opts.x0   = [0;0];
opts.N    = N;

% Add whichever extras you need
opts.K    = [];             % state-feedback gain
opts.L    = [];             % observer gain
opts.r    = [];             % reference
opts.u    = u_traj;         % reference


[x_hist, y_hist, u_hist, x_hat_hist] = solve_difference_eq(opts);
```

```
*** solve_difference_eq called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: []
     K: []
     r: []
     u: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



Great! The open loop system works, albeit with an incredibly slow response.

Now, let's try it again with some full state feedback. First, we build a K matrix using the place function:

```
Feedback_K = place(sys.A, sys.B, [0.7 0.8])
```

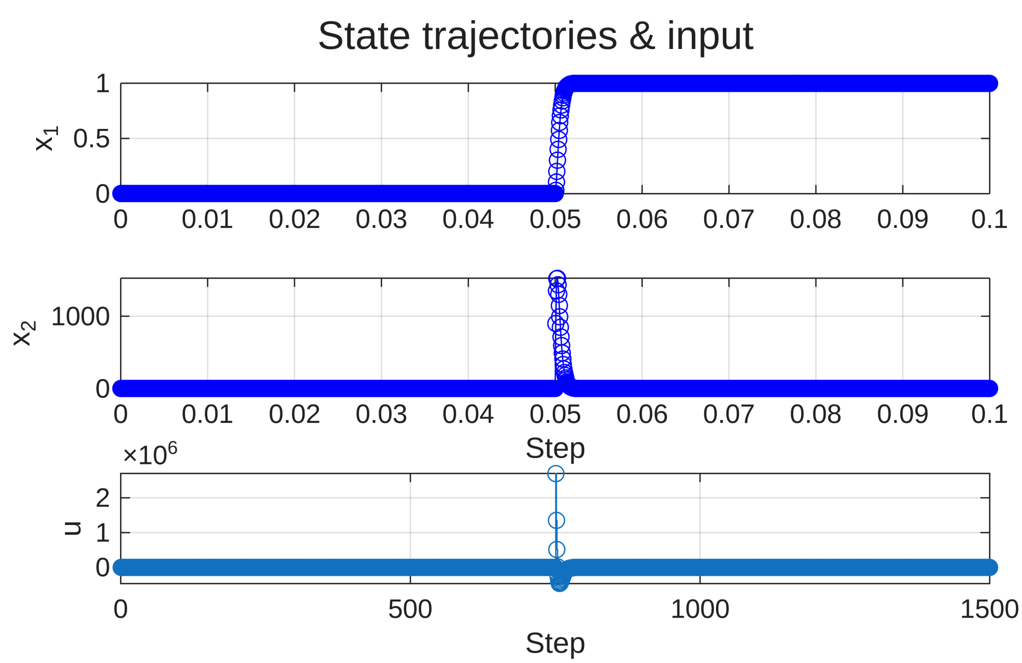
```
Feedback_K = 1×2
10^6 ×
    2.6998    0.0014
```

```
eig(sys.A - sys.B*Feedback_K)
```

```
ans = 2×1
    0.8000
    0.7000
```

```
opts.K = Feedback_K;
opts.r = [u_traj; zeros(size(u_traj))];
[x_hist, y_hist, u_hist, x_hat_hist] = solve_difference_eq(opts);
```

```
*** solve_difference_eq called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: []
     K: [2.6998e+06 1.4099e+03]
     r: [2×1500 double]
     u: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
State-feedback enabled.
```



Wow that's fast!

Now let's implement an observer, also using the place function:

```
L = transpose(place(sys.A', sys.C', [0.1 0.2]))
```

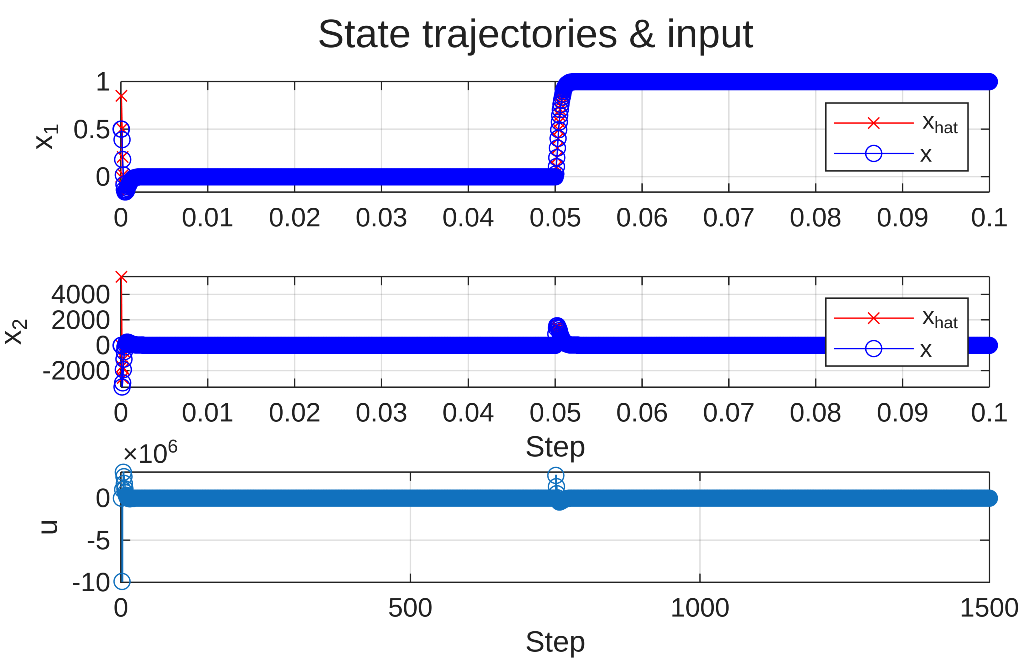
```
L = 2×1
10^4  ×
    0.0002
    1.0799
```

```
eig(sys.A - L*sys.C)
```

```
ans = 2×1
    0.1000
    0.2000
```

```
opts.L = L;
opts.r = [u_traj; zeros(size(u_traj))];
opts.x0 = [0.5; -0.5];
[x_hist, y_hist, u_hist, x_hat_hist] = solve_difference_eq(opts);
```

```
*** solve_difference_eq called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: [2×1 double]
     K: [2.6998e+06 1.4099e+03]
     r: [2×1500 double]
     u: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Observer enabled.
State-feedback enabled.
```



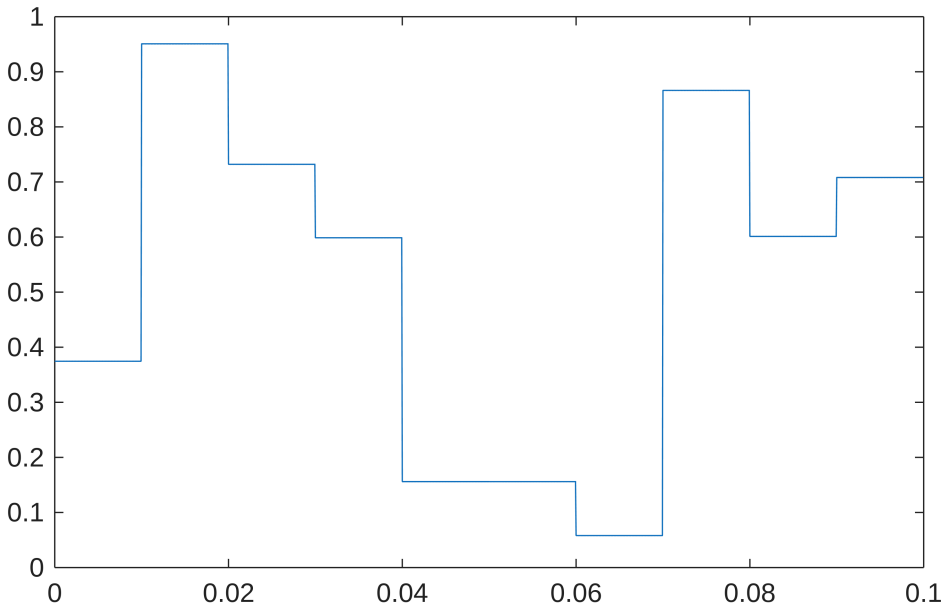
Great. The solver function and model are working properly. We can do a more interesting reference signal than a plain step input however.

Let's make a hard disk drive reference signal. This reference signal demands the read write head to a series of positions of angular displacement, and to hold there for a certain amount of time (presumably while it's reading or writing)! Then, the step changes to another random value, simulating a different, random location.

```
function [ref, t, N] = make_hdd_reference(max_time, Ts, stepDur, seed)
```

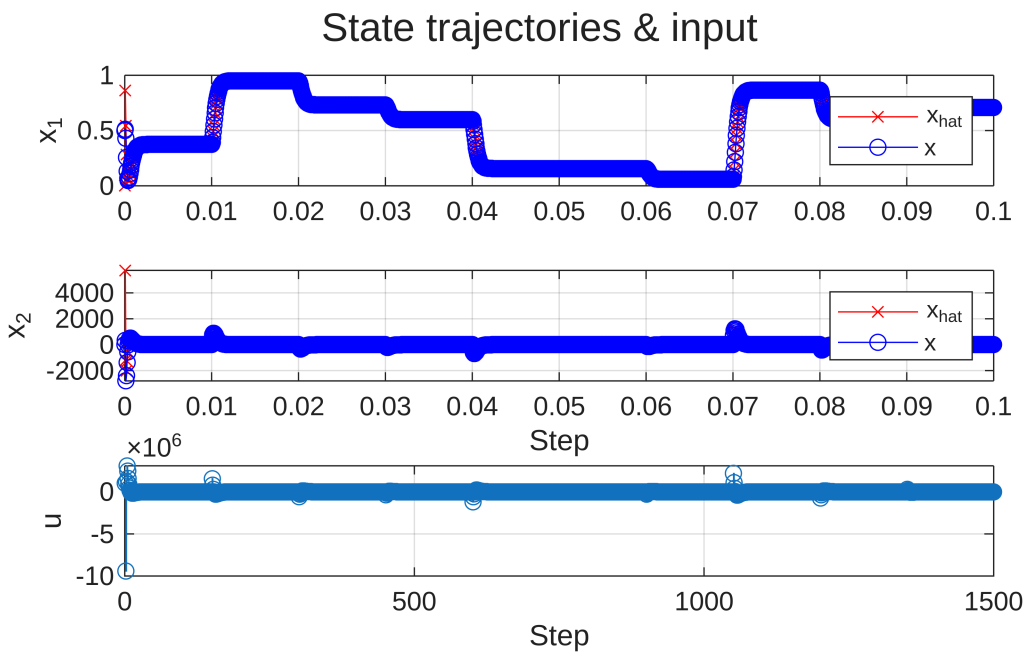
```
    %MAKE_HDD_REFERENCE  Generate a 2-row reference signal for an HDD seek
model.
    %
    %   [REF, t] = MAKE_HDD_REFERENCE(max_time, Ts, stepDur, seed)
    %
    %    max_time  - total simulation time (seconds)
    %    Ts        - sample period (seconds)
    %    stepDur   - dwell time of each demand position (seconds)
[default 5e-3]
    %    seed      - RNG seed for repeatability
[default 1]
    %
    %   Outputs
    %    REF  - 2×N matrix.  Row 1 = demanded angular position [rad, 0–1].
    %                        Row 2 = demanded angular velocity  (always 0).
    %    t    - 1×(N+1) time vector (useful for plotting)
    %
    %   Example
    %   -------
    %   [ref,t] = make_hdd_reference(0.1,1e-4,5e-3,42);
    %   stairs(t(1:end-1),ref(1,:)), xlabel('time (s)'), ylabel('\theta_d
(rad)')
    %
    %   ------------------------------------------------------------------

    % -------- default arguments -----------------------------------------
    if nargin < 4, seed    = 1;      end
    if nargin < 3, stepDur = 5e-3;   end

    % -------- basic sizes -----------------------------------------------
    N = round(max_time / Ts);         % # discrete steps
    t = 0:Ts:max_time-Ts;                % length N+1 (last point is
max_time)

    % -------- set RNG seed for repeatability ----------------------------
    rng(seed);

    % -------- build reference signal ------------------------------------
    ref          = zeros(2, N);       % row 2 (velocity) stays zero
    stepsPerSeg  = max(1, round(stepDur / Ts));

    % indices at which a new position should be chosen
    changeIdx = 1 : stepsPerSeg : N+1;   % +1 so we include final time

    % generate random positions in [0,1] rad
    randPos = rand(1, numel(changeIdx));

    % fill in the piece-wise-constant demand
    for k = 1:numel(changeIdx)-1
        i1 = changeIdx(k);
```

```
        i2 = changeIdx(k+1) - 1;
        ref(1, i1:i2) = randPos(k);
    end
    % final segment
    ref(1, changeIdx(end):N) = randPos(end);

end

[ref, t, N] = make_hdd_reference(0.1, Ts, 1e-2, 42);
figure;
plot(t,ref(1,:))
```




```
opts.u = [];
opts.r = ref;
opts.N = N;
[x_hist, y_hist, u_hist, x_hat_hist] = solve_difference_eq(opts);
```

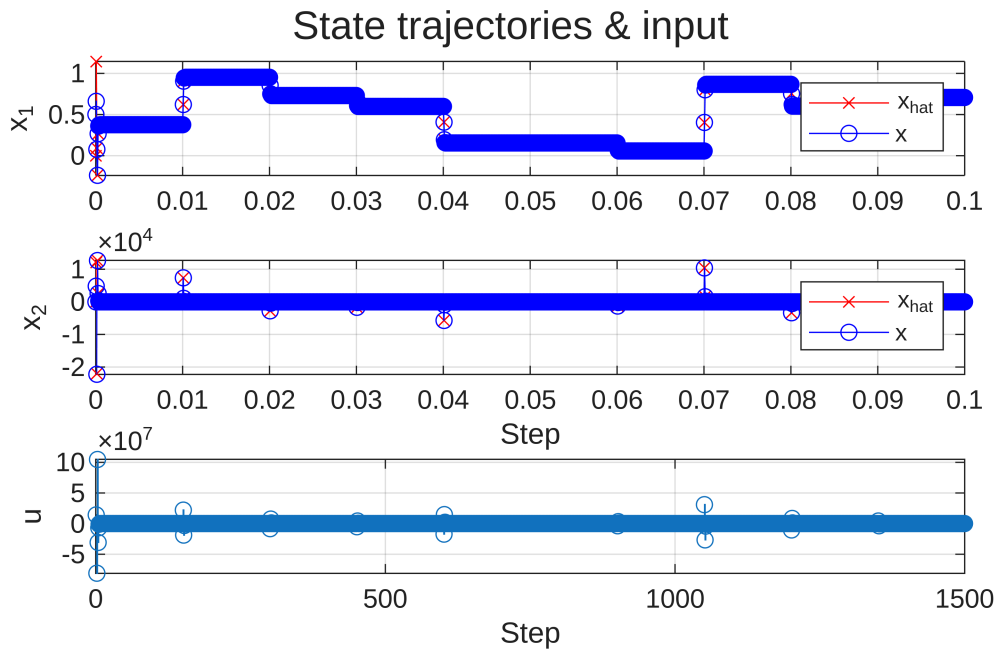
```
*** solve_difference_eq called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: [2×1 double]
     K: [2.6998e+06 1.4099e+03]
     r: [2×1500 double]
     u: []
Observer enabled.
State-feedback enabled.
```



Yikes, that tracking doesn't look very good. Let's make our observer and feedback faster.

```
Feedback_K = place(sys.A, sys.B, [0.07 0.08]);
L = transpose(place(sys.A', sys.C', [0.01 0.02]));
opts.K = Feedback_K;
opts.L = L;
[x_hist, y_hist, u_hist, x_hat_hist] = solve_difference_eq(opts);
```

```
*** solve_difference_eq called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: [2×1 double]
     K: [3.8502e+07 4.2666e+03]
     r: [2×1500 double]
     u: []
Observer enabled.
State-feedback enabled.
```



Now, we implement our code to incorporate an successive approximation (SAR) ADC delay. SAR ADCs will deliver the most recent sensor measurement y *one time step late*. This is the delay caused by the successive approximation process. The following is an implementation of this delay in our simulation:

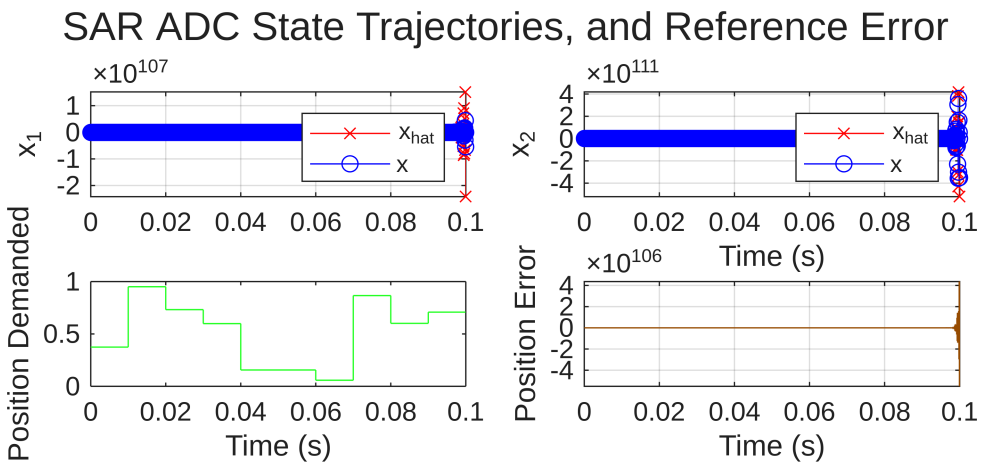
```
function [x_hist, y_hist, u_hist, x_hat_hist] = solve_SAR_ADC(opts)
    % Solves a discrete-time state-space difference equation iteratively
    % including a one time-step delay by an SAR ADC.
    % All inputs are passed in **one structure** so you can build the call
    % incrementally without remembering positional order.
    %
    % Required fields in **opts**
    %   • sys  – discrete-time state-space system (ss object)
    %   • x0   – initial state column vector
    %   • N    – number of discrete simulation steps
    %   • L    – observer gain matrix
    %   • K    – state-feedback gain matrix
    %   • r    – reference trajectory [nu × N]

    % Optional fields in **opts**
    %   • plotting - plot things? or no?
    %
    % Outputs
    %   x_hist     – state history  [nx × (N+1)]
    %   y_hist     – output history [ny × N]
    %   u_hist     – input history  [nu × N]
    %   x_hat_hist – observer-state history (empty if no observer)

    arguments
        opts struct
    end
```

```
% ---------- Convenience handles & defaults ----------------------------
req = {'sys','x0','N'};                                % required fields
for f = req
    if ~isfield(opts,f{1})
        error('Missing required field opts.%s',f{1});
    end
end

sys = opts.sys;
x0  = opts.x0(:);                % force column
N   = opts.N;
L   = opts.L;
K   = opts.K;
r   = opts.r;
plotting = opts.plotting;

% ----------------- Diagnostics -----------------------------------------
fprintf('\n*** solve_SAR_ADC called with: ***\n');
vars = struct('x0',x0,'N',N,'L',L,'K',K,'r',r);
vars  %#ok<NOPRT>

% ----------------- Extract system matrices -----------------------------
[A,B,C,D] = ssdata(sys);
Ts = sys.Ts;
[nx,nu]  = size(B);
ny       = size(C,1);

% ----------------- Pre-allocate histories ------------------------------
x_hist     = zeros(nx,N+1);
y_hist     = zeros(ny,N);
u_hist     = zeros(nu,N);

x_hist(:,1) = x0;

% Observer bookkeeping
useObserver = ~isempty(L);
if useObserver
    fprintf('Observer enabled.\n');
    x_hat       = [0;0];
    x_hat_hist  = zeros(nx,N+1);
    x_hat_hist(:,1) = x_hat;
else
    x_hat_hist = [];
end

% Controller presence
useFB = ~isempty(K);
if useFB, fprintf('State-feedback enabled.\n'); end
```

```
    if ~useFB && isempty(u)
        error('Either opts.K or opts.u must be supplied.');
    end

    % Ensure reference is sized correctly if provided
    if ~isempty(r) && size(r,2)~=N
        error('opts.r must have N columns.');
    end

    % ----------------- Simulation loop ----------------------------------
    for k = 1:N-1
        % Compute input
        u_k = K*(-x_hat_hist(:,k) + (isempty(r) * 0 + ~isempty(r) * r(:,k)));
        u_hist(k) = u_k;

        % Plant output
        y_hist(:,k+1) = C*x_hist(:,k) + D*u_k; %SHIFT RIGHT TO INDUCE DELAY

        % Propagate observer states
        x_hat = A*x_hat + B*u_k + L*(y_hist(:,k) - C*x_hat - D*u_k);
        x_hat_hist(:,k+1) = x_hat;

        %Calculate Next State
        x_hist(:,k+1) = A*x_hist(:,k) + B*u_k;
    end

    % ----------------- Plot results -------------------------------------
    figure;
    time = (0:N)*Ts;
    if plotting
        for i = 1:nx
            subplot(nx+1,2,i);
            plot(time,x_hat_hist(i,:),'-xr', time,x_hist(i,:),'-ob');
            ylabel(sprintf('x_%d',i));
            grid on;
            if i==nx, xlabel('Time (s)'); end
            legend('x_{hat}', 'x')
        end
        subplot(nx+1,2,3);
        stairs(time(1:N), r(1,1:N), "Color", "#22FF22")
        ylabel('Position Demanded');
        xlabel('Time (s)');

        subplot(nx+1,2,4);
        stairs(time(1:N), x_hist(1,1:N) - r(1,1:N), "Color", "#964B00")
        ylabel('Position Error');
        xlabel('Time (s)');
        grid on;
        sgtitle('SAR ADC State Trajectories, and Reference Error');
    end
```

```
end

opts.plotting = true;

[x_hist, y_hist, u_hist, x_hat_hist] = solve_SAR_ADC(opts);
```

```
*** solve_SAR_ADC called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: [2×1 double]
     K: [3.8502e+07 4.2666e+03]
     r: [2×1500 double]
Observer enabled.
State-feedback enabled.
```



It exploded! This is probably because our gains are now too aggressive to accommodate the output delay. Let's adjust to some milder gains:

```
Feedback_K = place(sys.A, sys.B, [0.7+0.0001i 0.7-0.0001i])
```

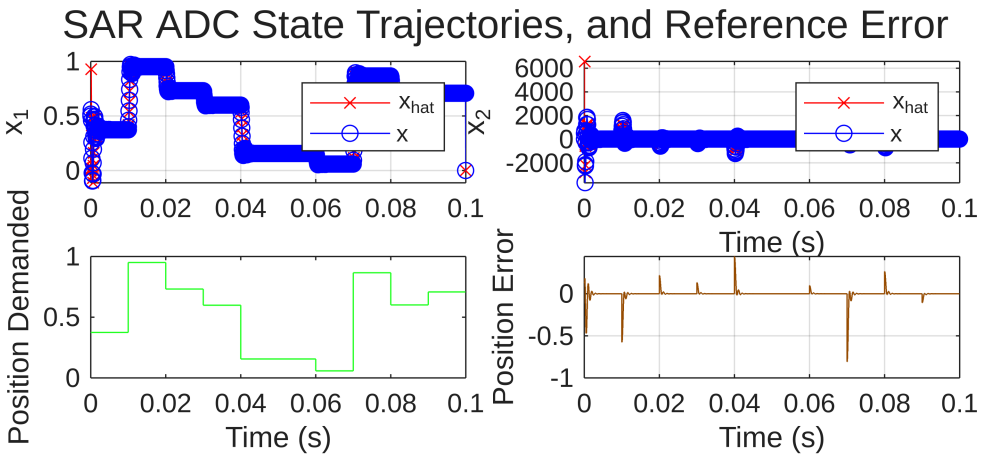
```
Feedback_K = 1×2
10^6 ×
    4.0499    0.0017
```

```
L = transpose(place(sys.A', sys.C', [0.1+0.01i 0.1-0.01i]))
```

```
L = 2×1
10^4 ×
    0.0002
    1.2151
```

```
opts.K = Feedback_K;
opts.L = L;
[x_hist, y_hist, u_hist, x_hat_hist] = solve_SAR_ADC(opts);
```

```
*** solve_SAR_ADC called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 1500
     L: [2×1 double]
     K: [4.0499e+06 1.6649e+03]
     r: [2×1500 double]
Observer enabled.
State-feedback enabled.
```

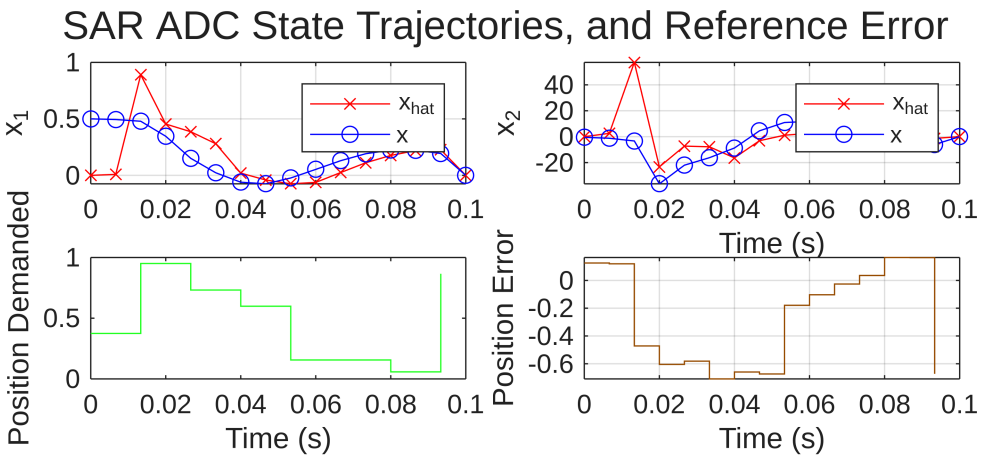


Sampling Period Analysis

In this section, we will adjust the sampling time to see the effects of reduced (or increased!) sampling time on system response.

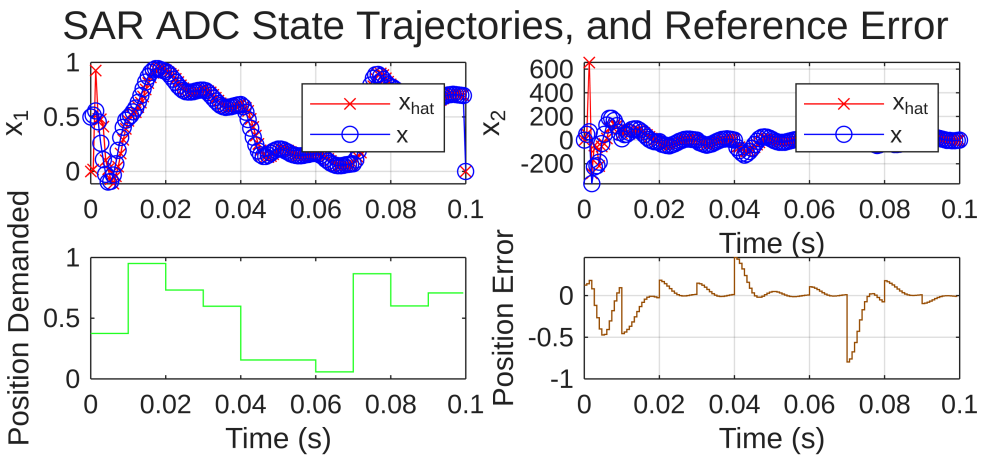
```
Ts = 1/15e1;
opts.sys = c2d(sys_cont,Ts);
Feedback_K = place(opts.sys.A, opts.sys.B, [0.7+0.0001i 0.7-0.0001i]);
L = transpose(place(opts.sys.A', opts.sys.C', [0.1+0.01i 0.1-0.01i]));
opts.K = Feedback_K;
opts.L = L;
[ref, t, N] = make_hdd_reference(0.1, Ts, 1e-2, 42);
opts.r = ref;
opts.N = N;
[x_hist, y_hist, u_hist, x_hat_histdifference_eq] = solve_SAR_ADC(opts);
```

```
*** solve_SAR_ADC called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 15
     L: [2×1 double]
     K: [207.0456 16.0463]
     r: [2×15 double]
Observer enabled.
State-feedback enabled.
```



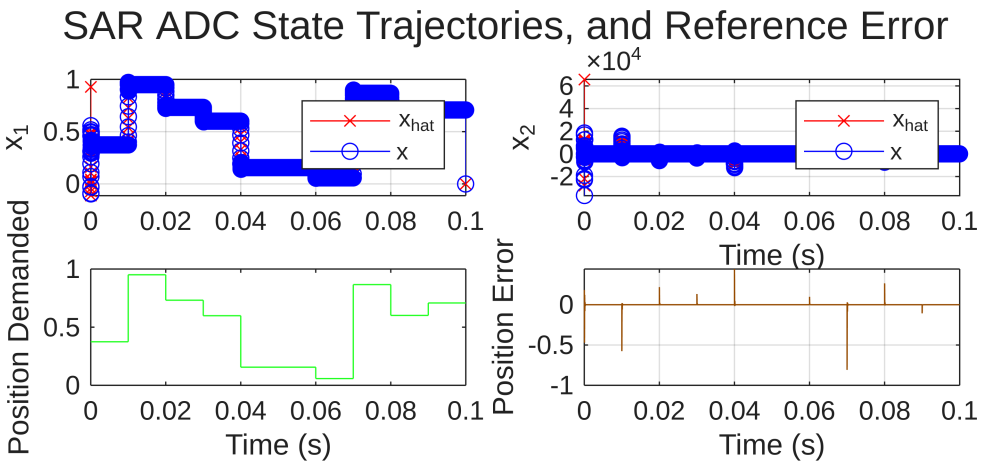

```
Ts = 1/15e2;
opts.sys = c2d(sys_cont,Ts);
Feedback_K = place(opts.sys.A, opts.sys.B, [0.7+0.0001i 0.7-0.0001i]);
L = transpose(place(opts.sys.A', opts.sys.C', [0.1+0.01i 0.1-0.01i]));
opts.K = Feedback_K;
opts.L = L;
[ref, t, N] = make_hdd_reference(0.1, Ts, 1e-2, 42);
opts.r = ref;
opts.N = N;
[x_hist, y_hist, u_hist, x_hat_histdifference_eq] = solve_SAR_ADC(opts);
```

```
*** solve_SAR_ADC called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 150
     L: [2×1 double]
     K: [4.0307e+04 166.3873]
     r: [2×150 double]
Observer enabled.
State-feedback enabled.
```



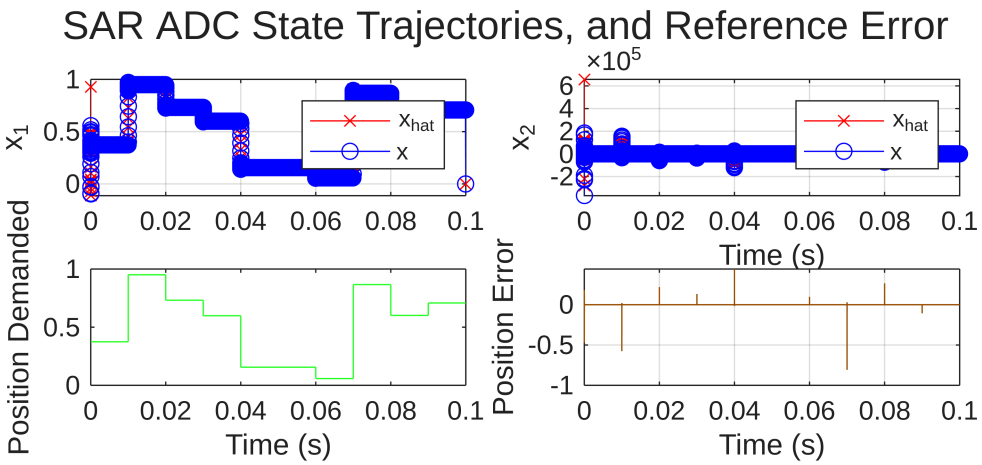

```
Ts = 1/15e4;
opts.sys = c2d(sys_cont,Ts);
Feedback_K = place(opts.sys.A, opts.sys.B, [0.7+0.0001i 0.7-0.0001i]);
L = transpose(place(opts.sys.A', opts.sys.C', [0.1+0.01i 0.1-0.01i]));
opts.K = Feedback_K;
opts.L = L;
[ref, t, N] = make_hdd_reference(0.1, Ts, 1e-2, 42);
opts.r = ref;
opts.N = N;
[x_hist, y_hist, u_hist, x_hat_histdifference_eq] = solve_SAR_ADC(opts);
```

```
*** solve_SAR_ADC called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 15000
     L: [2×1 double]
     K: [4.0500e+08 1.6650e+04]
     r: [2×15000 double]
Observer enabled.
State-feedback enabled.
```




```
Ts = 1/15e5;
opts.sys = c2d(sys_cont,Ts);
Feedback_K = place(opts.sys.A, opts.sys.B, [0.7+0.0001i 0.7-0.0001i]);
L = transpose(place(opts.sys.A', opts.sys.C', [0.1+0.01i 0.1-0.01i]));
opts.K = Feedback_K;
opts.L = L;
[ref, t, N] = make_hdd_reference(0.1, Ts, 1e-2, 42);
opts.r = ref;
opts.N = N;
[x_hist, y_hist, u_hist, x_hat_histdifference_eq] = solve_SAR_ADC(opts);
```

```
*** solve_SAR_ADC called with: ***
vars = struct with fields:
    x0: [2×1 double]
     N: 150000
     L: [2×1 double]
     K: [4.0500e+10 1.6650e+05]
     r: [2×150000 double]
Observer enabled.
State-feedback enabled.
```



These results indicate something somewhat obvious: if the sampling rate is higher, the amount of delay introduced into the system is lower. This then allows the system to have a faster response, and less error.

Milestones

In this update, I have created the following:

1. A working HDD read write head model
2. Converted that model into the discrete time domain
3. A working observer and controller has been implemented
4. A function to simulate hard disk drive movements was created
5. A one-step delay of a SAR ADC was implemented, and sampling time effects were examined.

How does this line up with the outcomes presented in the project proposal? In the proposal, I outlined the following outcomes

1. **Reduced Latency:** Achieve lower control delays by acting on partial ADC data
2. **Improved Accuracy:** Attain a lower average error between the actual and desire head position compared to traditional full-sample control
3. **Validated Simulation:** Develop a realistic simulation of a SAR ADC with sequential register updates and a fast state estimator

To accomplish these outcomes, the following must be performed before the final project submission:

1. The code to simulate ADC register behavior must be written and integrated
2. A way of measuring 'average error' must be determined and implemented
3. A way of measuring 'latency' must be determined and implemented

4. A controller using the in-between ADC sample data must be implemented. At first, just using the rough value as the output and then correcting the early control as more registers come in will be the way to go. The Bayesian way of doing things is a smarter way to do this, but the effort to implement this may be prohibitive before the project deadline, and would make error analysis more challenging.

And finally, some nice to haves that would look great in a paper:

1. Derivations about the stability of observer error when introducing delay. I'm partly done with this for the single delay case: it turns out to be a series solution that is somewhat nasty. The in-between sample case will likely be a nightmare. We shall discuss.
2. Some nice LaTeX graphics using TiKz.
3. Some better sources on HDD read write head parameters. This doesn't really matter, but HDD experts may get upset.