System identification
Spring 2025

CE1 Report : Nonparametric Methods

Risse Maxime (345491), Jules Streit (327732), Yo-Shiun Cheng (386249)
Group 25

Professor Alireza KARIMI

1 Step response

We created a Simulink model of a third-order transfer function, which is used for all simulations in CE-1. In this model, a `Transfer Function` block with the following parameters:

$$G(s) = \frac{1.2}{s^3 + 2s^2 + 1.35s + 1.2}$$

, a `Random Number` block, a `Saturation` block, a `From Workspace` source, and a `To Workspace` sink are used to simulate a system with measurement noise and upper / lower limits. The Simulink model is shown in Figure 1. A step input with magnitude equal to the upper saturation limit and an impulse are applied to the system. The responses of the two input signals are shown in Figure 2.

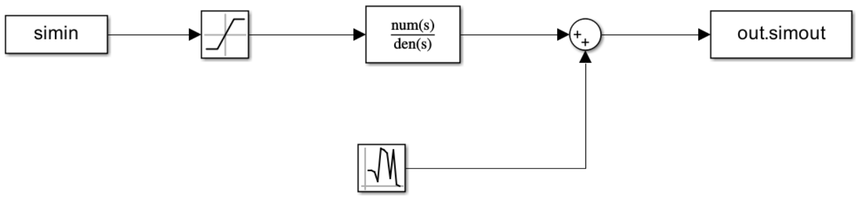
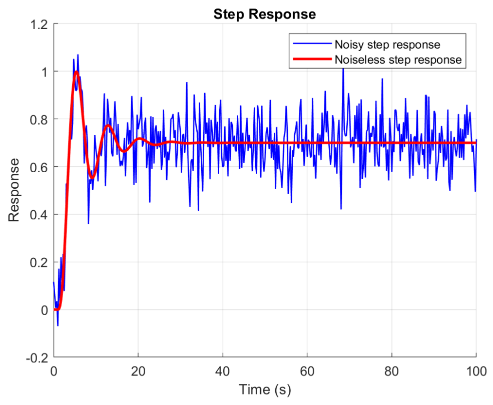
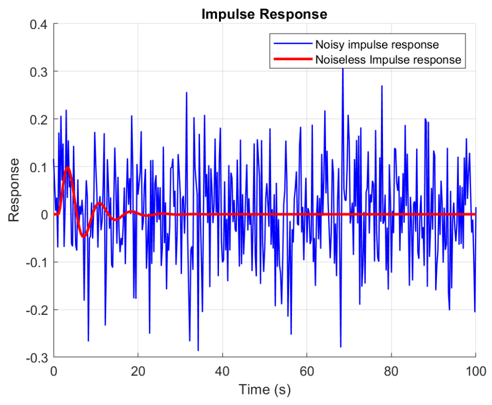


Figure 1: Simulink Model



(a) Step response of the system.



(b) Impulse response of the system.

Figure 2: Step and impulse response of the Simulink model system.

By using the `pole()` command in MATLAB, the poles of $G(s)$ are found to be: $s_1 = -1.6237$, $s_2 = -0.1881 + 0.8388i$, and $s_3 = -0.1881 - 0.8388i$.The resonant frequency ω_r is approximately the imaginary part of the complex pole, which is 0.8388 rad/s.

Our sampling frequency is $\omega_s = \frac{2\pi}{T_s} = \frac{2\pi}{0.25} = 8\pi \approx 25.13 rad/s$, which is approximately 30 times the system's highest frequency component, and is well beyond the Nyquist rate and even the conservative 10 times rule-of-thumb used in control and signal processing. Moreover, the bode diagram of the transfer function G shows that the gain of the system is already below -80 dB, which means that signals at or beyond that frequency are strongly attenuated. As a result, we believe that $T_s = 0.25s$ is a reasonable and valid choice for sample time.

```
%% **Exercise 1: Step Response**
% Define simulation time
T_end = 100; % Total simulation time in seconds
t = (0:Ts:(N-1)*Ts)'; % Generates exactly 400 elements

```

```matlab
% Define step input
u_step = zeros(size(t)); % Initialize with zeros
u_step(t >= 1) = 1; % Step occurs at t = 1s

% Create struct simin for step response
simin.signals.values = u_step;
simin.time = t;
simin1.signals.values = u_step;
simin1.time = t;

% Simulate step response
out_step = sim("CE1.slx");

% Plot step response
figure;
hold on;
plot(out_step.simout.Time, out_step.simout.Data, 'b', 'LineWidth', 1);
plot(out_step.simout1.Time, out_step.simout1.Data, 'r', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Response');
title('Step Response');
legend('Noisy step response', 'Noiseless step response');
```

```matlab
%% **Exercise 1: Impulse Response**
% Define impulse input (delta function)
u_impulse = zeros(size(t)); % Initialize with zeros
u_impulse(t == 1) = 1; % Impulse at t = 1

% Create struct simin for impulse response
simin.signals.values = u_impulse;
simin.time = t;
simin1.signals.values = u_impulse;
simin1.time = t;

% Simulate impulse response
out_impulse = sim("CE1.slx");

% Extract impulse response as a vector y
y = out_impulse.simout.Data;
y2 = out_impulse.simout1.Data;


% Plot impulse response
figure;
hold on;
plot(out_impulse.simout.Time, y, 'b', 'LineWidth', 1);
plot(out_impulse.simout1.Time, y2, 'r', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Response');
title('Impulse Response');
legend('Noisy impulse response', 'Noiseless Impulse response');
```

2 Auto Correlation of a PRBS signal

The following equation calculates the cross-correlation of two discrete-time signals $u(k)$ and $y(k)$:

$$R_{uy}(h) = \frac{1}{M}\sum_{k=0}^{M-1} u(k)y(k-h)$$

Assume that signal u and signal y have the same length M, we choose the following interval for h

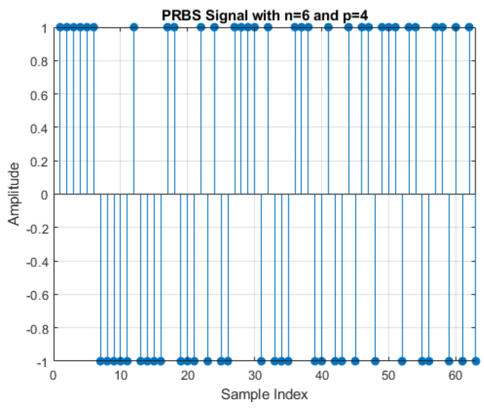
$$h \in \{-\left\lceil \frac{M}{2} \right\rceil \ldots, -1, 0, 1, \ldots, \left\lceil \frac{M}{2} \right\rceil\}.$$

To validate our function, we use the `prbs` matlab function to make the desired PRBS signal (shift register length $n = 6$ and number of periods $p = 4$), , as shown in Figure 3a). We then compute the autocorrelation of this PRBS signal, the result is shown in Figure 3b.

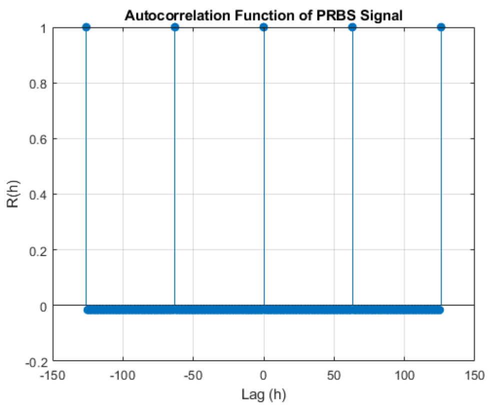
```
n = 6;   % Shift register length
p = 4;   % Number of periods

% Run the PRBS function
u = prbs(n, p);

% Compute autocorrelation using intcor function
[R, h] = intcor(u, u);
```



(a) The PRBS signal



(b) Autocorrelation of the PRBS signal.

Figure 3: The PRBS signal and its autocorrelation.

The function `intcor(u, y)` computes the circular cross-correlation between two periodic signals u and y, both assumed to be of length M. The output $R_{uy}(h)$ represents the correlation values evaluated over the lag vector $h \in [-\lfloor M/2 \rfloor, \lfloor M/2 \rfloor]$. For each lag $h(i)$, the function shifts the signal y circularly (modulo M) so that even an aperiodic signal is treated like a periodic one.

```matlab
function [R,h] = intcor(u,y)

M = length(u); % M is the period length of the signal
h = -floor(M/2):floor(M/2);
R = zeros(1, length(h));


for i = 1:length(h)
    sum_value = 0;

    for k = 1:M
        y_shifted = y(mod(k - h(i) - 1, M) + 1);
        sum_value = sum_value + u(k) * y_shifted;
    end

    R(i) = sum_value / M;
end
```

3 Impulse response by deconvolution method

The objective of this section is to compute the impulse response of the simulink system using the numerical deconvolution method. In order to achieve this, we first generate a random white noise signal with `rand()` command in MATLAB as the input to the model and then record the system's response. Then, we can reconstruct the discrete impulse response with the convolution relation by computing $g(k)$ recursively

$$y(k) = g(k) * u(k) = \sum_{j=0}^{k} g(j)u(k-j),$$

$$g(k) = \frac{1}{u(0)}[y(k) - \sum_{j=0}^{k-1} g(j)u(k-j)], \text{ where } k = 0, 1, 2, .., N-1$$

or in a matrix form

$$Y = U\Theta \Rightarrow \Theta = U^{-1}Y.$$

We start by running the simulink again, excited by a random signal between -0.7 and 0.7 (zero mean input), to have our output vector Y.

```matlab
Ts = 0.25;
T_sim = 100;
N = T_sim / Ts + 1;

u = (rand(N, 1) - 0.5) * 1.4;     % random signal between -0.7 and 0.7

t = 0 : Ts: (N - 1) * Ts; % time vector

simin.signals.values = u;
simin.time = t';

out = sim("CE1.slx");
y = out.simout.Data;
```

Notice that U is an asymmetric Toeplitz matrix which is ill-conditioned, so Θ cannot be directly computed. To solve this problem, we truncate the impulse response by keeping its first K values ($K << N$) and by assuming that its length is finite. The number of unknowns, $g(k)$, can be reduced and so the problem can be solved in the least squares sense by computing the pseudo inverse of U_K, where U_K and Θ_K are the truncated versions of U and Θ:

$$Y = U_K\Theta_K \Rightarrow \Theta_K = U_K^{\dagger}Y = (U_K^T U_K)^{-1}U_K^T Y$$

In our case, we assume that the length of the finite impulse response is 100 and take $K = 100$.

```matlab
L = 100; % assumed length of the impulse response
U = toeplitz(u, [u(1); zeros(L - 1, 1)]);
Theta_K = inv(U'*U) * U' * y(1:size(U, 1));
```

Another method of computing the impulse response is regularization (trade-off between the bias and the variance). Let us consider a modified least square criterion as:

$$J(\Theta) = \epsilon^T\epsilon + \lambda\Theta^T\Theta,$$

where $\epsilon = Y - U\Theta$, and λ is the bias-variance trade-off factor. By making a partial differential of Θ on both sides of the equation, we obtain the following.

$$\Theta = (U^TU + \lambda I)^{-1}U^TY.$$

In our case, we take λ as 0.1 by trial and error.

```matlab
lambda = 0.1; % Regularization parameter (adjust as needed)
I = eye(N); % Identity matrix of size K

% Compute regularized impulse response
```

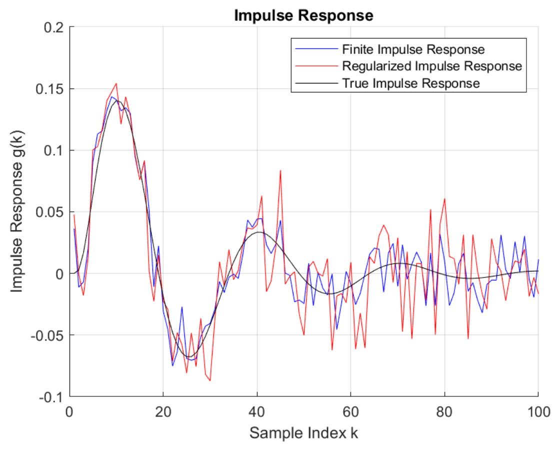


Figure 4: Comparison of the impulse responses computed by numerical deconvolution and the real one computed with `impulse()` function in MATLAB

```
U = toeplitz(u, [u(1); zeros(N - 1, 1)]);
Theta_K_reg = inv(U' * U + lambda * I) * U' * y(1:size(U, 1));
```

```
% Define the continuous-time transfer function
Gs = tf(1.2, [1 2 1.35 1.2]);

% Discretize with zero-order hold (ZOH)
Gz = c2d(Gs, Ts, 'zoh');
true_impulse_response = impulse(Gz, t)*Ts;
```

Figure 4 shows the comparison of impulse response computed by both the finite impulse response method and the regularization method, and the true impulse response of the system computed with `impulse()` function in MATLAB. We notice that the regularized impulse response is less precise, and therefore, the error should be greater than the error of the finite impulse response.

We then compute the 2-norm of the error for both impulse responses to check our assumption.

```
% 2-norm of the error for the finite impulse response
error_finite = Theta_K - true_impulse_response(1:L);
norm_error_finite = norm(error_finite, 2);

% 2-norm of the error for the regularized impulse response
error_regularized = Theta_K_reg(1:L) - true_impulse_response(1:L);
norm_error_regularized = norm(error_regularized, 2);
```

We obtain the following results, which are consistent with the graph we plotted.

$$\text{err}_{\text{fir}} = 0.14545 \quad < \quad \text{err}_{\text{reg}} = 0.2429$$

We also tried to repeat this process with an input signal u between 0 and 0.7 (instead of [-0.7; 0.7]) and we noticed that both error values (2-norm) were higher.

4 Impulse response by correlation approach

The objective of this section is to estimate the impulse response of the discrete-time system using the correlation approach. To achieve this, a pseudorandom binary sequence will be applied as input to the system. Two different correlation functions will be used. A custom one, `intcor`, and a Matlab's built-in function, `xcorr`. Finally, estimated impulse responses will be compared with the true impulse response. The 2-norm of the error will then be analyzed to evaluate the accuracy of both methods.

The first step is to generate the PRBS signal (line 2) and set up simulation parameters. Then, a simulation is done (line 15).

```
1  Ts = 0.25;
2  u = prbs(7, 2);
3  N = size(u,1);
4  T_sim = N * Ts;
5  t = (0:Ts:(N-1)*Ts)';
6
7  % Create struct simin for impulse response
8  simin.signals.values = u;
9  simin.time = t;
10
11 % Simulate impulse response
12 out_impulse = sim("CE1.slx");
13
14 % Extract impulse response as a vector y
15 y = out_impulse.simout.Data;
```

Then, we want to use the correlation approach. This technic is based on the following relationship that gives the output of an noisy LTI system:

$$y(k) = g(k) * u(k) + d(k)$$

Then, as long as we take a $u(k)$ signal independant of the zero-mean $d(k)$, the following relationship can be proven :

$$R_{yu}(h) = g(h) * R_{uu}(h)$$

In the next snippet of code, we're doing a numerical deconvolution to get $g(h)$.

```
Ruu = intcor(u, u);
Ryu = intcor(y, u);

% Resize the vectors
Ruu = Ruu(1:N);
Ryu = Ryu(1:N);

% Construct Toeplitz input matrix
row = [Ruu(1), zeros(1, N-1)];
col = Ruu(1:N);
RUU = toeplitz(col, row);

K = 80; % 20*4, estimation of the response time

% Troncation
RUUk = RUU(1:N, 1:K);

% Impulse response computation
imp_res = inv(RUUk'*RUUk) * RUUk' *Ryu';
```

In a very similar process, the impulse response using the Matlab built-in function is computed in the next snippet of code. Note that we are doing a biased correlation. This will normalize the function by the number of samples, regardless of the lag. This is what we are already doing in our custom function *intcor*.

```
Ruu_matlab = xcorr(u,u, 'biased');
Ryu_matlab = xcorr(y,u, 'biased');
```

```

Ruu_matlab = Ruu_matlab(1:N);
Ryu_matlab = Ryu_matlab(1:N);

% Construct Toeplitz input matrix
row = [Ruu_matlab(1), zeros(1, N-1)];
col = Ruu_matlab(1:N);
RUU_matlab = toeplitz(col, row);

K = 80; % 20*4, estimation of the response time

% Troncation
RUUk_matlab = RUU_matlab(1:N, 1:K);

% Impulse response computation
imp_res_matlab = inv(RUUk_matlab'*RUUk_matlab) * RUUk_matlab' *Ryu_matlab;
```

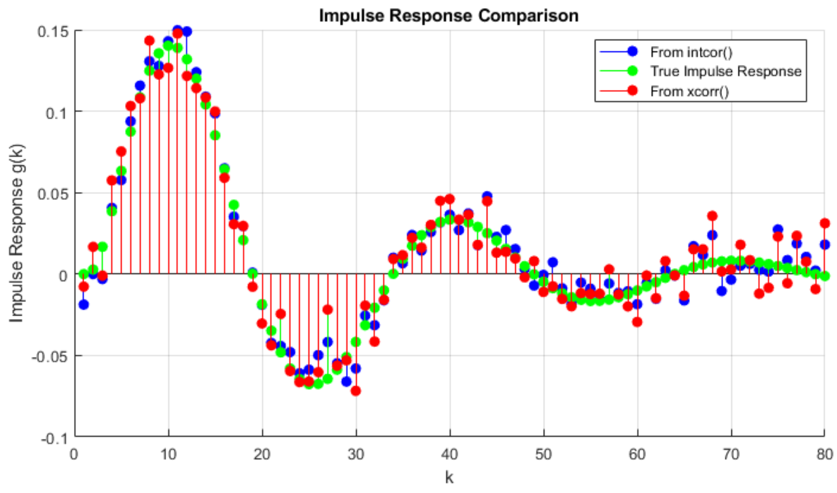


Figure 5: Comparison between the impulse responses computed and the True Impulse Response

The 2-norm error is given in the next snippet :

```
2-norm error (intcor): 0.0899
2-norm error (xcorr):  0.1155
```

Discussion

According to the 2-norm error, a better estimate is obtained using our custom function, *intcorr*. This may be due to how we handled periodicity. In our custom function, we are treating the input functions as they repeat infinitely following the same pattern, as described earlier in this report. However, the PRBS signal given in input of the system is a periodic signal. This could explain how the 2-norm error was slightly lower than from the build-in function.

The parameter K represents the assumed duration of the system's impulse response. In this case, since we know the true impulse response, this parameter was set to an approximate value of 80 which corresponds to a response duration of 20 seconds. This is when the system visually reached a settling point. In an unknown system, setting a large K would be a wise choice as it would capture most of the impulse response. However, it could also induce an unnecessary complexity of computation.

5 Frequency domain Identification (Periodic signal)

In this part, the goal is to use the Fourier analysis method to identify the frequency response of a model excited by a PRBS signal. In order to obtain a signal of length $N \approx 2000$, we chose to generate an 8-bit register ($n = 8$) over a period of 8 using the function `prbs(n, p)`, which yields an exact length of $N = 2040$. We then ran the Simulink model again to obtain the output vector y corresponding to the input signal described above.

```matlab
Ts = 0.25;
u = prbs(8, 8) *0.7; % *0.7 in order to use the post-saturation values
N = size(u,1);
T_sim = N * Ts;
t = (0:Ts:(N-1)*Ts)';

% Create struct simin for impulse response
simin.signals.values = u;
simin.time = t;

% Simulate impulse response
out_impulse = sim("CE1.slx");

% Extract impulse response as a vector y
y = out_impulse.simout.Data;
```

Then we divided the input and output vectors into 8 equal segments, each corresponding to one of the 8 periods. We applied FFT on each of these segments and then computed the average on each periods in order to reduce the effect of noise. With the resulting average array we are able to compute the average gain G_{avg}. Notice that we choosed not to consider the first period of both input and output to avoid the transient regime and wait the stable regime of the step response (otherwise it leads to an offset on the Bode magnitude [dB] plot).

```matlab
y_segments = cell(8,1);
u_segments = cell(8,1);  % Preallocate cell array to hold the segments
segment_length = N / 8;

for i = 1:8
    idx_start = (i-1)*segment_length + 1;
    idx_end = i*segment_length;
    u_segments{i} = u(idx_start:idx_end);
    y_segments{i} = y(idx_start:idx_end);
end

u_fft = cell(8,1);
y_fft = cell(8,1);

for i = 1:8
    u_fft{i} = fft(u_segments{i});
    y_fft{i} = fft(y_segments{i});
end


% Initialize average vector with zeros of the same size as one segment
U_avg = zeros(segment_length, 1);
Y_avg = zeros(segment_length, 1);

% Sum over all segments
for i = 2:8
    U_avg = U_avg + u_fft{i};
    Y_avg = Y_avg + y_fft{i};
end

% Divide by 7 to get the average
U_avg = U_avg / 7;
Y_avg = Y_avg / 7;
```

```
G_avg = Y_avg ./ U_avg;
```

We then plot the bode plot by associating each value of G_{avg} to the corresponding value of the frequency vector that we can compute with the value of T_s. We can see the difference between the estimated gain G_{avg} and real one in Figure 9.The estimation proves to be accurate in the low-frequency range, whereas at higher frequencies, the noise becomes increasingly dominant.

$$\text{freq\_vect} = \left[0, \frac{\omega_s}{L}, \frac{2\omega_s}{L}, \dots, \frac{(L-1)\omega_s}{L}\right]$$

where $\text{L} = \text{segment\_length}$

```
ws = 2*pi/(Ts);
freq_vect = zeros(segment_length, 1);

for i=1:(segment_length-1)
    freq_vect(i+1) = ws * i / segment_length;
end

sys = frd(G_avg, freq_vect);

bode(sys);
hold on

G_true = tf([1.2], [1 2 1.35, 1.2])
bode(G_true, 'r--');

hold off
legend('Identified', 'True');
```

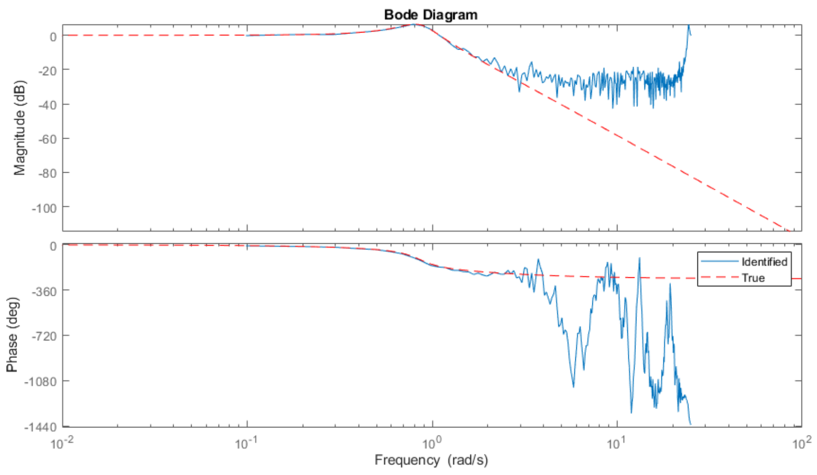


Figure 6: Comparison of the Bode plots: estimated gain (blue) vs real gain (red).

6 Frequency domain Identification (Random signal)

This question aim to use the spectral analysis method to identify the frequency response of a model excited by random signal. We were asked firstly to talk about the richness of the chosen random signal. In our case, a binary signal was chosen as it has the maximal energy. Recall that the energy of a discrete signal given from 0 to $N = 2000$ is given by this formula :

$$E = \sum_{k=1}^{N} u(k)^2$$

Empirical variance of the signal is given by this formula :

$$Var(u) = \frac{1}{N}\sum_{k=1}^{N}(u(k) - \mu)^2$$

As our random signal is centered around 0, $\mu = 0$. This implies the following relationship :

$$Var(u) = \frac{1}{N}\sum_{k=1}^{N}(u(k) - 0)^2 = \frac{1}{N}\sum_{k=1}^{N}(u(k))^2 = \frac{E}{N}$$

This implies that the energy of a signal is directly proportional to the variance. By choosing a binary random signal between -0.7 and 0.7, we maximize this variance and are more likely to get better results. The generation of the input signal is done in the following snippet of code.

```
%% Data preparation
N = 2000;
Ts = 0.25;
u = 0.7 * sign(rand(N, 1) - 0.5); % Binary signal between -0.7 and 0.7
T_sim = N * Ts;
t = (0:Ts:(N-1)*Ts)';

% Create struct simin for impulse response
simin.signals.values = u;
simin.time = t;

% Simulate impulse response
out_impulse = sim("CE1.slx");

% Extract impulse response as a vector y
y = out_impulse.simout.Data;

% Already prepare the segment length
segment_length = N / 8;
```

Whole data

The next code shows how we computed the frequency response without windowing and without averaging. We keep only the positive lags of the correlation because they represent how the past and present inputs affects the ouptut of the real system. Our function, "intcor", computes the cross-correlation in both negative and positive time shift (because we take it between -floor(M/2) and + floor(M/2). By selecting only the positive part (from the middle of the output of the function), we isolate the meaningful part of the response from the system. Better results were obtain this way.

```
%% Whole data, no windowing

R_yu_full = intcor(y, u);
R_uu_full = intcor(u, u);

mid = floor(N/2) + 1;

% Only take the positive lag
```

```
R_yu = R_yu_full(mid:end-1);
R_uu = R_uu_full(mid:end-1);

% FFT without windowing
Y_whole = fft(R_yu);
U_whole = fft(R_uu);
G_whole = Y_whole ./ U_whole;
```

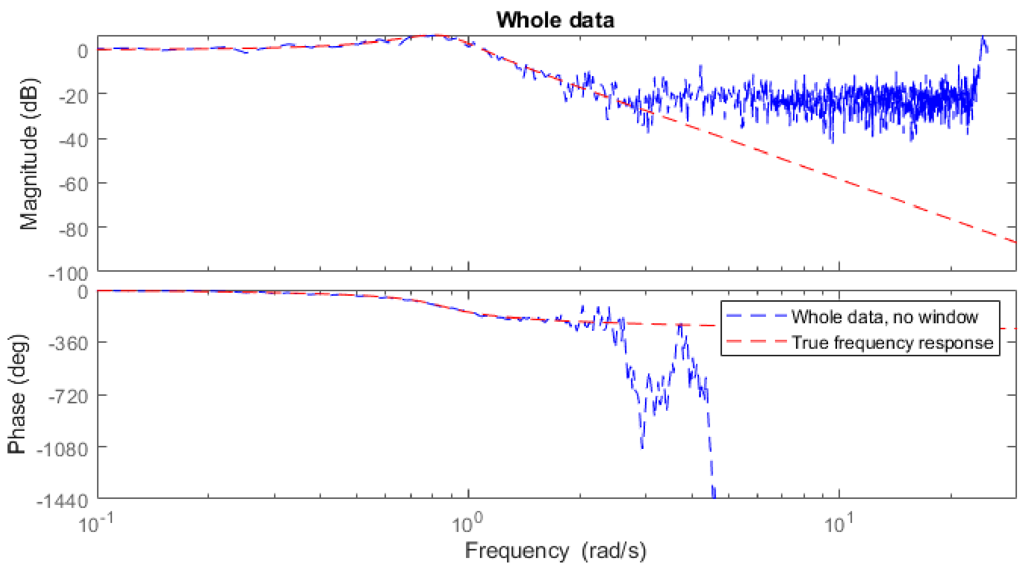


Figure 7: Bode plot using no windowing using the whole data

Comparison

The result shows a strong match between 0.1 and 2.5 rad/s. After this, the magnitude predicted just becomes random around -20db. At 2.5 rad/s, the phase begin to be instable and deep down starting 4.5 rad/s, showing incoherent behavior.

Windowing only

For the windowing, we use a hann window. Since we use only positive lags of the cross-correlation, we apply only the second half of the Hann window from 251 to 500. This minimize the high-frequencies artifacts in the FFT and results in a stronger frequency response estimation. The size of the hann window was decided by visually looking at the true system's impulse response duration.

```
%% Windowing only

R_yu_full = intcor(y, u);
R_uu_full = intcor(u, u);

mid = floor(N/2) + 1;

% Only take the positive lag
R_yu = R_yu_full(mid:mid+segment_length-1);
R_uu = R_uu_full(mid:mid+segment_length-1);

% Apply Hann window
h = hann(500);
h_shifted = h(251:end)';  % use second half
R_yu_win = R_yu .* h_shifted;
R_uu_win = R_uu .* h_shifted;

% FFT
```

```
Y_win_whole = fft(R_yu_win);
U_win_whole = fft(R_uu_win);
G_win_whole = Y_win_whole ./ U_win_whole;
```

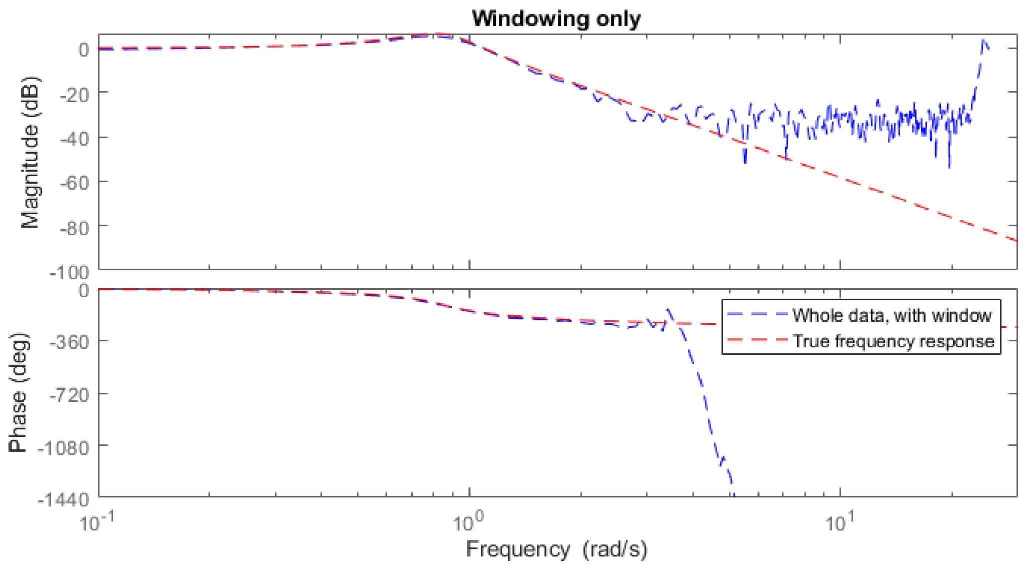


Figure 8: Bode plot using windowing

Comparison

The result is slightly better than the one without windowing. First of all, removing some frequencies is drawing a smoother line as less noise is being taken into account. The prediction is robust from 0.1 to 3.5 rad/s, after which it becomes impredictible. Phase is going decreasing as well starting 3.5 rad/s.

Windowing and averaging

Windowing and averaging is realized in the next snippet of code. Since the duration of the true system's impulse response is approximately $250 \cdot T_s$, we choose to separate the 2000 data points into 8 chunks of 250 samples.

```
%% Averaging + Windowing

y_segments = cell(8,1);
u_segments = cell(8,1);  % Preallocate cell array to hold the segments

for i = 1:8
    idx_start = (i-1)*segment_length + 1;
    idx_end = i*segment_length;
    u_segments{i} = u(idx_start:idx_end);
    y_segments{i} = y(idx_start:idx_end);
end

u_fft = cell(8,1);
y_fft = cell(8,1);

num_lags = 500;
h = hann(num_lags);
mid = floor(num_lags / 2);
h_shifted = h(mid + 1:end)';

U_avg = zeros(1, segment_length);
Y_avg = zeros(1, segment_length);

```

```
for i = 1:8
    y_seg = y_segments{i};
    u_seg = u_segments{i};

    R_yu = xcorr(y_seg, u_seg, 'unbiased');
    R_uu = xcorr(u_seg, u_seg, 'unbiased');

    R_yu_tr = R_yu(mid : end)';
    R_uu_tr = R_uu(mid : end)';

    y_fft = fft(R_yu_tr .* h_shifted);
    u_fft = fft(R_uu_tr .* h_shifted);

    % Accumulate directly
    Y_avg = Y_avg + y_fft;
    U_avg = U_avg + u_fft;
end

Y_avg = Y_avg / 8;
U_avg = U_avg / 8;

% Final frequency response
G_avg = Y_avg ./ U_avg;
```

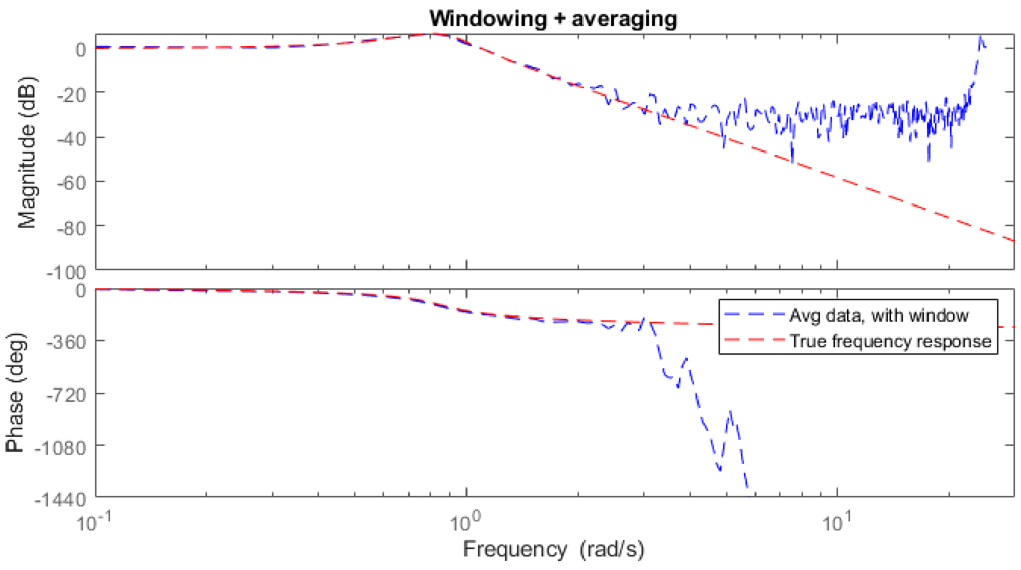


Figure 9: Bode plot using windowing and averaging

Comparison

The result shows similar behavior as windowing only because the prediciton is solid from 0.1 to 3.5 rad/s, but uncertain afterward.

The plots were given by this next code :

```
%% Plot

ws = 2*pi/(Ts);

segment_length = 1000;
freq_vect = (0:segment_length-1)' * (ws / segment_length);

% True system
G_true = tf([1.2], [1 2 1.35, 1.2]);
```

```

% Whole system
figure;
set(gcf, 'Position', [100, 100, 800, 400]);
sys_whole = frd(G_whole, freq_vect);
bode(sys_whole, 'b--');
hold on;
bode(G_true, 'r--');
xlim([0.1 30]);
ylim([-1440 0]);
title("Whole data");
legend('Whole data, no window', "True frequency response");

segment_length = 250;
freq_vect = (0:segment_length-1)' * (ws / segment_length);

% Windowing only
figure;
set(gcf, 'Position', [100, 100, 800, 400]);
sys_win_whole = frd(G_win_whole, freq_vect);
bode(sys_win_whole, 'b--');
hold on;
bode(G_true, 'r--');
xlim([0.1 30]);
ylim([-1440 0]);
title("Windowing only");
legend('Whole data, with window', "True frequency response");


% Windowing + averaging
figure;
set(gcf, 'Position', [100, 100, 800, 400]);
sys_win_avg = frd(G_avg, freq_vect);
bode(sys_win_avg, 'b--');
hold on;
bode(G_true, 'r--');
xlim([0.1 30]);
ylim([-1440 0]);
title("Windowing + averaging");
legend('Avg data, with window', "True frequency response");
```