Dynamics and motion control

Workshop B Servo control, code simulation and robustness.

1 Servo Control

There are two reasons why we sometimes have to extend the feedback design with a carefully designed servo controller. The first is when the command/reference signal varies over a large range and it is important that we do not saturate the control signal to the process. The other is when it is important not only to control the position but also the velocity and perhaps even the acceleration of the process. For example, the movement of the gluing or painting robot cannot have large variations in acceleration.

To fulfill the first specification of not saturating the control we have to design a trajectory planner. To fulfill the second specification we have to design a model following feed forward control law.

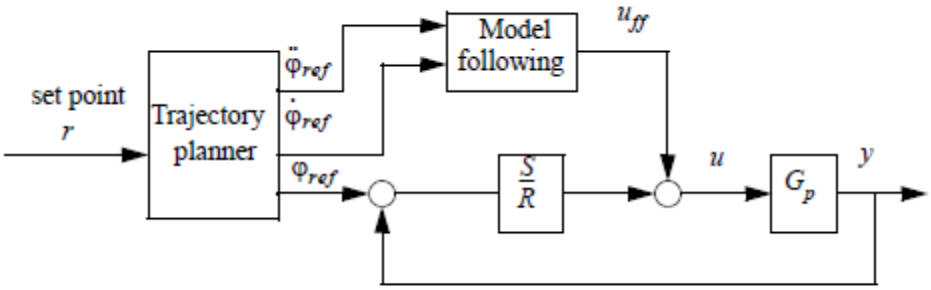
1.1 Trajectory Planner

The output feedback control structure contains both feedforward and feedback parts

$$u(s) = \frac{T(s)}{R(s)}r(s) - \frac{S(s)}{R(s)}y(s)$$

The main disadvantage with the control law above is that it is difficult to prevent the input u from saturation if the reference r has a large sudden change. The controller has either saturated control input for a large reference or very slow response for a small reference input.

One solution to the problem is to use a trajectory planner that generates the reference in a way that it does not saturate the control signal but still generates enough control so that the response is sufficiently fast. The trajectory has to be based on the knowledge of the process that you are controlling, e.g., maximum torque, speed, etc. This is the model following control concept illustrated in the figure below.



1.2 Model Following Control

If the reference signal is designed so that it is as many times differentiable as the order of the process model, then we can design a model following controller by inverting the process model in the time domain. A control structure for the DC-motor is depicted in the illustration figure where both the reference speed and reference acceleration are necessary to invert the second order DC motor model.

Level 1

Design and implement in Simulink the model following control for a position reference for the DC-motor. It should consist of:

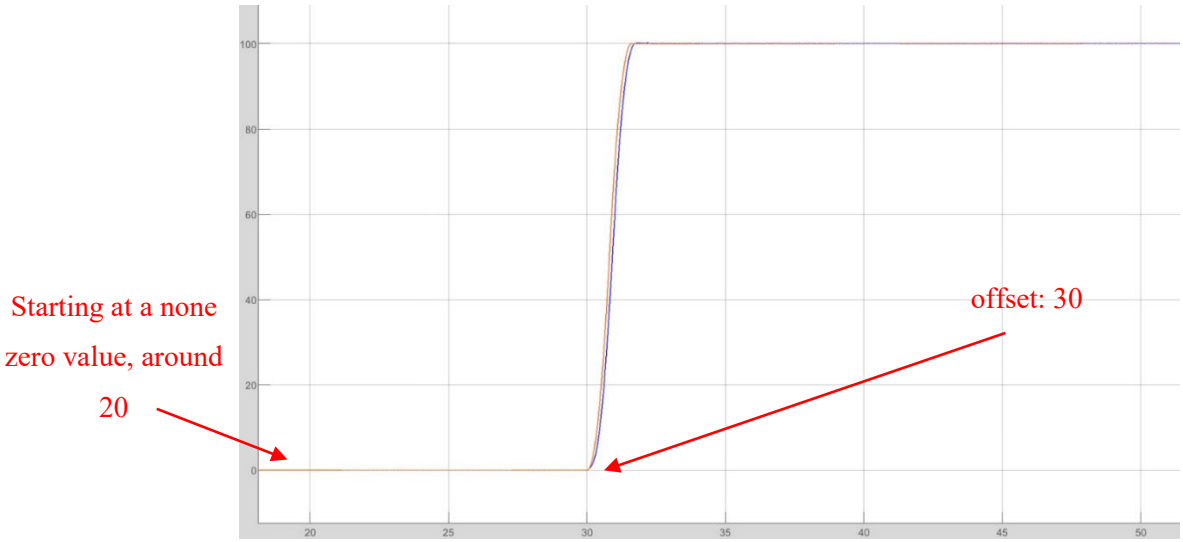
- A PID controller
- Trajectories for two different set points
  - Rs = 10 [rad]
  - Rs = 100 [rad]
- A model following controller that inverts the motor model.

Level 2

Run the model following controller on the real motor. Compare the results from the real motor with those of the simulated motor from level 1 above.

Hints: From Workshop A, you may notice that when you run the simulation in external mode, the simulation time does not start at 0. Furthermore, if you check the scope you may also find that the scope begins recording the data after the simulation time starts to run. The delay does not matter in Workshop A, because you can control the value of reference signal. However, in Workshop B, it is hard to control the reference signal during the simulation. Thus, you have to set an offset of the trajectory planner. When you are doing Workshop A,

you may notice that the simulation time always starts around a certain value. Thus, you have to set the trajectory planner to generate reference signal after this certain time to make sure that the scope can record the data. For example, the simulation time always starts to run around 20 seconds, then you can set the offset of trajectory planner to 30 seconds.



2 Writing controller code

Now you should implement the controller from Section 1.2 on a microprocessor. You therefore need to write the code in C, compile and download it. As always with programming you make mistakes and it is usually very difficult to debug code on an embedded system. However, there is a way of simulating the code in Simulink to verify it before trying to compile it. The code that you will write in Simulink is not C, it is a subset of normal Matlab m-code called Embedded Matlab or Matlab function depending on which Matab version you have. It is however quite similar to C and if you do not use the matrix operations of Matlab, it is actually very similar.

In the Simulink library under User Defined Functions there is a block called Embedded Matlab function.or Matlab function depending on Matlab version. When you double click the block an editor opens where you can write code. A Matlab function is just an m-file that starts with the keyword function. The syntax is

```
function [out_1,out_2,....,out_n] = function_name(in1_, in_2, ..., in_n);
```

If you have only one output argument, you may omit the brackets. Variables inside a function are local, i.e., they are not visible in Matlab command window or in any other m-file.

The opposite is also true; variables defined at the Matlab prompt, directly or from any other m-file, are not visible inside the Matlab function. It is similar to a function in C when you declare a variable inside the function, e.g., **float y**. This type of declaration makes the variables not only local but they also lose their values after the execution of the function since they are volatile. All states in the control code, however, must be saved in some way, i.e., non-volatile, so that they can be used by the function each time it is called, i.e., at each sample instant. In C this is achieved by defining the storage class of the variable as **static,** e.g., **static float y;**.

In Embedded Matlab this is achieved by declaring the variables using the key word **persistent,** e.g. **persistent** y1 y2**.** The command **persistent** initializes the declared variables to the empty matrix. To give the variables an initial value you can perform the following code patch.

```
persistent x;

if isempty(x)

   x = 0;

end
```

Level 1

Write the code for the position controller from Section 1.2 above including the trajectory planner and model following parts. Compare the model following Simulink **simulation** results from Section 1.2 with those of the simulation of the Embedded Matlab implementation.

Include the Embedded Matlab code in the report and show that it gives the same output as when using normal Simulink blocks.

Tips:

- To set the sample time of the Matlab function, right click the block and select “Block parameters”.
- Develop step by step. Do not try to do everything at once.
- Debug your coded controller function by comparing its output with a known Simulink block output of what you want it to do.

- When you have the same signal output from the Embedded Matlab function and the Simulink block, then you can replace the control function in the Simulink blocks by the controller in the (Embedded) Matlab function block.

Please show that the result from the embedded Matlab implementation is equal to the implementation using ordinary Simulink blocks. If there is any (small) difference try to describe what the reason for the difference is.

3 Robustness to parameter uncertainty and sensor noise

The two functions, Sensitivity function and Complementary Sensitivity function, are useful tools for getting help with the difficult design problem of where the closed loop poles should be located. One way (not the only) to select the closed loop poles is to divide the closed loop polynomial into two polynomials $A_m(s)A_o(s)$. The closed loop response with a 2DOF design structure including reference signal r, model error v, and sensor noise n is the following. In the transfer function from r to y, $A_o(s)$ is cancelled by the selection of polynomial $T(s)$.

$$y = \frac{Bt_0}{A_m}r + \frac{AR}{A_mA_0}v - \frac{BS}{A_mA_0}n$$

Where t_0 is calculated to get unit dc-gain for a step response.

Take the DC-motor position control problem as an example and use the second-order model of the motor. Start by selecting $A_m(s)$ from specifications, e.g. how fast should the response be and select the order of $A_m(s)$ the same as the order of the process model. Compute and plot the Sensitivity and Complementary Sensitivity functions.

$$S_e(s) = \frac{A(s)R(s)}{A_m(s)A_0(s)}$$

$$T_e(s) = \frac{B(s)S(s)}{A_m(s)A_0(s)}$$

Then select a couple of faster poles for $A_o(s)$, e.g., 2-5 times faster than $A_m(s)$ and plot $T_e(s)$ and $S_e(s)$ for them and compare.

If we have process parameters, such as inertia and friction, that are not perfectly known or that they even vary during operation, then it is important to use a controller that has as low as possible maximum gain in the Sensitivity function. If the sensor quality is low or it runs in

a noisy environment then it might be more important to have a complementary sensitivity function with a low gain over a wider frequency range, especially at higher frequencies.

Level 2

Design a velocity controller for the valve controlled hydraulic cylinder. You do not have to make a discrete time controller, simulate the nonlinear hydraulic model and the continuous time controller with 0.5 m/s reference velocity (step response) and zero external force.

Try different closed-loop pole locations. For simplicity, start with the same frequency in both $A_m(s)$ and $A_o(s)$. Start with for example 100 rad/s and increase it in steps. Observe the step response and the peak responses of different selections of poles. The overshoot in the step response should decrease with increasing frequency of the desired poles. When you find a closed loop frequency where the improvements are small, select this for $A_m(s)$. Then select the frequency in $A_o(s)$ as a factor of $A_m(s)$, start with 1.0 and increase it until you are satisfied. Plot the Bode diagrams of $T_e(s)$ and $S_e(s)$ for at least two selections of $A_0(s)$.

Deliver the following results in the report.

1. Closed loop step response from Simulink using the nonlinear hydraulic cylinder model and the designed controller. Plot the response for two different selections of $A_o(s)$ (with the same $A_m(s)$) where you also apply a step in external force of 5000 N at a point in time when the velocity has stabilized at constant speed.
2. For the same two $A_o(s)$ as in (1), change the mass in the nonlinear model to 200 kg instead of the 100 kg that you used when you designed the controller. That is, *you should not redesign the controller for the new weight, only change the weight in the simulation model.*
3. Again, for the same two $A_o(s)$ simulate with sensor noise. Simulate the noise as a sine wave with amplitude 0.01 m/s and with a frequency that you select yourself so that you can see the difference in the step responses for the two $A_o(s)$. Plot $T_e(s)$ for the two $A_0(s)$ and mark the amplitudes of both $T_e(s)$ plots at the noise frequency. What is the relationship between the velocity response and the values of $|T_e(s)|$ at the noise frequency?

Give a brief discussion of your results in the report. If you want, you can continue designing a position controller for the cylinder by extending the velocity loop with another

position loop. According to the cascade control structure, this should be very simple since the design model you have to use for the position loop is simply

$$y(s) = \frac{1}{s} v(s)$$

where *y* is the position and *v* the velocity.