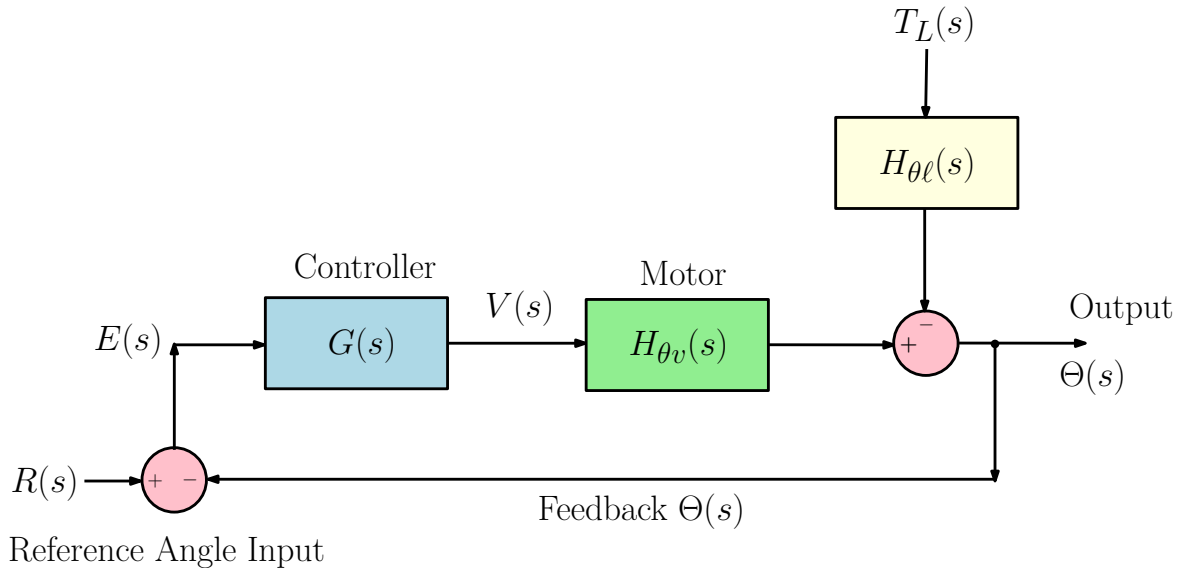
Introduction to Intelligent Mobile Systems Lab II



Lab Manual

Spring Semester

Author:
Prof. Dr. Kaustubh Pathak

Instructors:
Prof. Dr. Kaustubh Pathak and Dr. Fangning Hu

JACOBS UNIVERSITY

Contents

Chapter 1

Part 1: Linear Control Systems Using Matlab & Simulink

Note: Part I of the lab will take 2 to 3 lab sessions. You are expected to stop working at 18:30. After doing each task or sub-task, call the Instructor/TA to show them your result. You will be asked some questions about your task.

Use OpenOffice to create a lab-report simultaneously in which you address each task. You will need to submit the report of what you did during each slot at 18:30 on that day. Your report should also include images of plots that you created in Matlab/Simulink. The report needs to be copied to the USB drive supplied. Do not send it by email. If the task asks you to derive anything, do so on the supplied paper separately and give it to the instructor.

1.1 Objective

In this week's labs, we will learn Simulink and design linear control systems using it. Topics covered include:

- Modeling linear systems using Simulink blocks
- Linear Time-Invariant (LTI) systems analysis using Matlab
- PID controller design for speed and position control of a DC motor
- Discretizing the controller for microcontroller implementation
- Using nested subsystems to use two motors to design the kineamtic model of a differentially driven mobile robot.

1.2 Simulink

After logging in on the lab-room computer, go to Programs/02. Sim&Calc/Matlab2014a to start Matlab: Do not use the older version of Matlab also installed.

Simulink is a widely used tool built on top of Matlab to simulate models of general systems. Underneath the hood, the simulation is done by numerically integrating the system's ODEs: Usually, the user need not know about the lower-level details. The system model (ODE) is specified implicitly by use of interconnected blocks of various types. Due to the use of pictorial blocks the system is easy to understand. You can even design systems with hierarchical nested blocks, where a higher-level block (subsystem) groups together several lower-level blocks. This hierarchical approach keeps the complexity manageable and the system easier to understand and maintain. To view the hierarchical model-browser you can press Ctrl+H in Simulink.

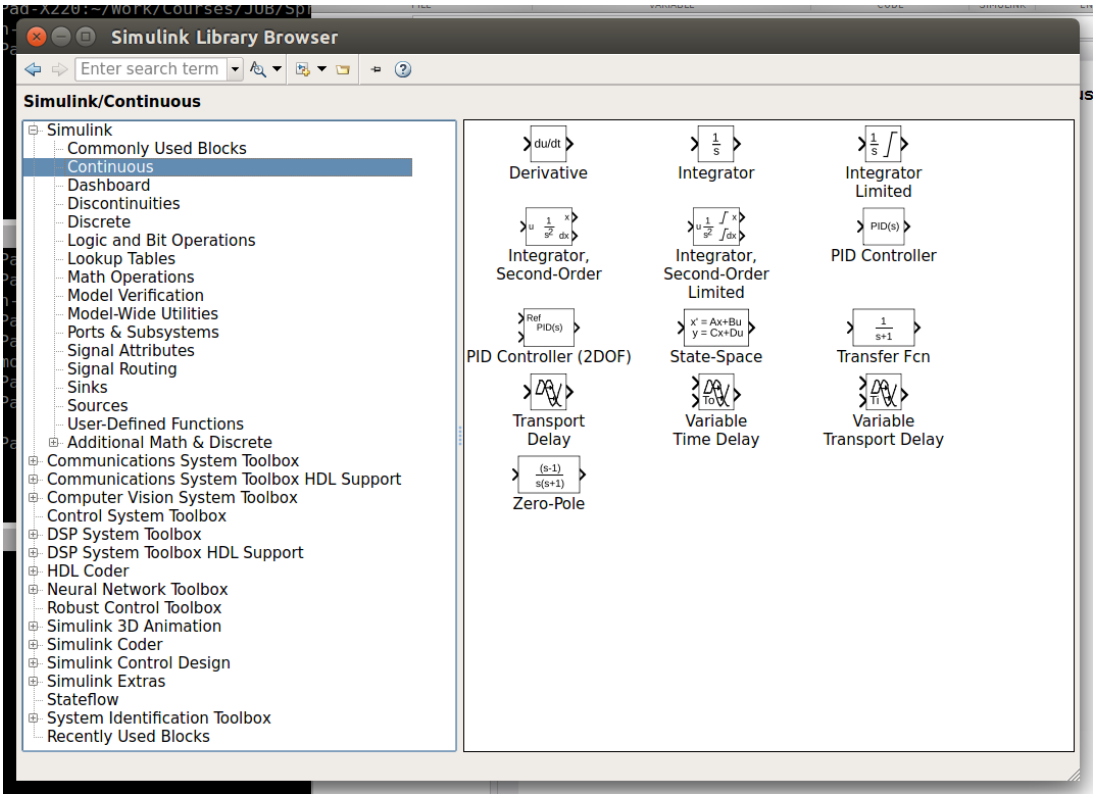


Figure 1.1: The Simulink library browser (SLB).

- Click the *Simulink Library* icon of the *Home* tab to open the SLB (Fig. 1.1).
- The left pane of the SLB has the individual libraries: clicking on any particular library (e.g. *Continuous* for continuous-time models) will show the corresponding block-set in the right pane. A useful block is *Continuous/Transfer-Fcn* in which you can specify the Laplace domain transfer-function of linear system. Double click on the *Transfer-Fcn* block to see a short description and a list of parameters; clicking the *Help* button will show you the detailed description.
- Create a new model by clicking the *New Model* icon in SLB.
- From the SLB drag and drop the *Continuous/Transfer-Fcn* block into your model. You can also select the block in SLB and press Ctrl+I to insert the block into your model.
- In the new model window, double click to create an annotation. This is like adding a comment in code.
- The following keyboard/mouse short-cuts are useful:
  - Use the mouse scroll-wheel to zoom.
  - Press Alt+1 to restore the default zoom.
  - Press space to make the current model fit the window.
  - Press Ctrl+Shift+L to switch to the SLB window.
- Most important blocks have been collected under *Commonly Used Blocks* in SLB for convenience. Some useful libraries and blocks in SLB are:
  - **Sources**: These are blocks which generate different kind of input signals which you can apply to your model. Some useful examples are: Constant, Step, Sine Wave,

and Pulse Generator. There is also the *In1* input port which is useful when you are making hierarchical models. We will use it later.

– **Signal Routing**: Using *Mux* you can multiplex several scalar signals into a vector signal. The reverse action is done by the *Demux* block.

– **Sinks**: Some sinks of interest are:

1. *Scope*: This block is used mainly for visualizing (like in an oscilloscope) various signals within the system. You can also feed a scope with a vector signal (created using Mux): It will then plot all signals together for easy comparison.
2. *Out1*: This is an output port: It is useful when you are making hierarchical models. We will use it later.
3. *To File*: You can pipe the signal to a file for later analysis.

– **Continuous**: This was already explained above.

– **Discrete**: This contains various z-transform blocks useful for modeling discrete systems. In a Simulink model, you can have both continuous and discrete blocks together.

1.2.1 RC Circuit Using Subsystem Blocks

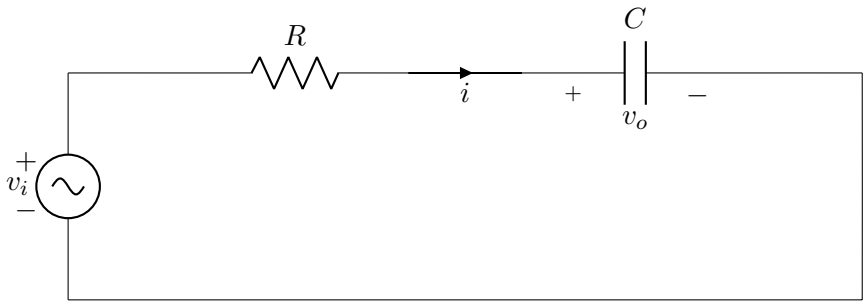


Figure 1.2: RC circuit.

We start off by making an RC circuit (Fig. 1.2) using blocks: In practice, the simplest way to do this is to use the transfer-fn block. We will use it later: First, our objective here is to make you familiar with how to make nested hierarchical subsystems in Simulink.

Each block in Simulink has some inputs and some outputs. Normally, we do not think of resistors and capacitors as input/output systems – but, just for this section, let us model them as follows:

Resistor: has two inputs: the voltages applied to each of its leads. It has one output, namely the current which flows through it.

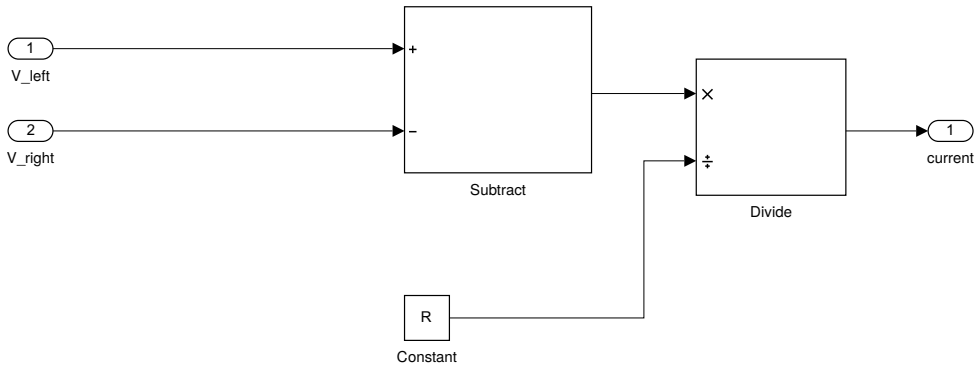


Figure 1.3: The resistor subsytem. The current is positive if $V_{\text{left}} > V_{\text{right}}$.

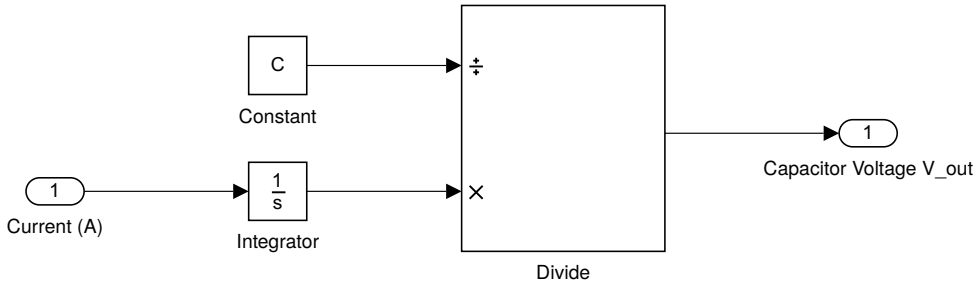


Figure 1.4: The capacitor subsytem.

- In the model window that you opened earlier, create a model as shown in Fig. 1.3.
  - V_{left} and V_{right} are input ports from **Sources**, current is an output port from **Sinks**. You can create V_{left} first and then drag it while pressing Ctrl: a pop-up menu will ask you if you want to paste or duplicate it. Pasting creates a new input port by copying; duplicating just creates a link to the original port – think of this as a symbolic link to a file. Duplicating is very useful in avoiding criss-crossing connections and in keeping the model clean. All ports should have unique names.
  - The subtract and divide blocks can be found under **Math-Operations**. You can double-click on them and change the order of the operators.
  - Constant block is in **Sources**: in the model, double-click the constant block. Now you can enter a value for the resistor. Since we do not want to hard-code any value, we just enter R there. Now, Simulink will take the value of from a variable called R in the Matlab workspace. This way you can re-run the simulation with different values of R. Type `R= 1e6` in the Matlab command-line to create this variable.
- Rename the ports to proper labels with units as shown.
- Now select all the above blocks including ports by left-clicking the mouse on the top-left of a rectangle enclosing all the blocks and dragging the mouse-pointer to enclose all the blocks belonging to our resistor model. Now press Ctrl+G (or right-click and select "Create Subsystem from Selection") to group. Rename this collapsed parent block to resistor. It has now two input ports and one output port. Press Ctrl+H to open the model-browser: here, you will find the resistor block.

Capacitor: In your main model, now create a capacitor-model as shown in Fig. 1.4. Again create a variable in the Matlab workspace by typing `C= 1e-6`.

Task 1.1. *Explain why Fig. 1.4 represents the model of a capacitor by writing down the equation it implements.*

- Group together all these blocks to a capacitor subsytem.

The RC Model: Now in the main model join together the resistor and capacitor subsystems as shown in Fig. 1.5. You may have to delete the automatically created i/o ports of the subsystems and replace them with appropriate connections.

- Check your model by pressing Ctrl-D (or choose *Update Diagram* from the *Simulation* menu).

  Task 1.2. *Proceed as follows:*

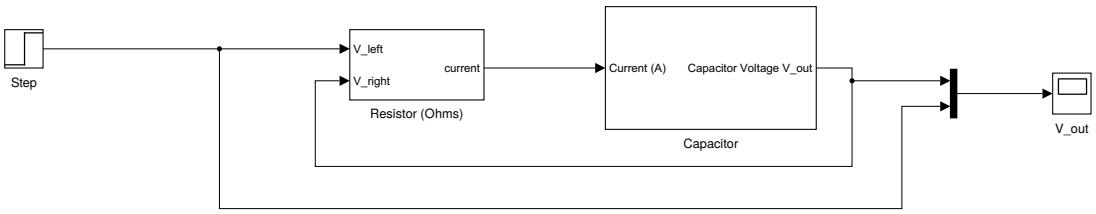


Figure 1.5: The RC circuit.

- *What is the time-constant of this system? How can you see it in the plot? Change the simulation-time (in input field in the tool-ribbon) from the default 10.0 seconds to 5 times this time-constant.*
- *Run the system by pressing Ctrl+T or by clicking the run-button. Look at the output by double-clicking the scope.*
- *Now change the variables R and C in the workspace and re-run the simulation. Does the scope display change as expected?*

1.2.2 RC Circuit Using Transfer Function

For the RC circuit, it is much easier to use the transfer function block.

- Create a new model in SLB and add the *Continuous/Transfer Fcn* block to it. Double click it and then click the help button to read how the transfer function is specified as a ratio of polynomials in the Laplace variable s.
- Referring to Fig. 1.2, the ODE

$$v_i = v_o + RC\frac{dv_o}{dt} \tag{1.1}$$

  can be written in the Laplace domain as the transfer-function

$$H(s) = \frac{V_o(s)}{V_i(s)} = \frac{1}{RCs + 1} \tag{1.2}$$

- Enter the numerator and denominator of this expression in the transfer-function block, taking some suitable values of R and C from the previous task. Connect to step-function input as before, run, and visualize the output in a scope. Is the output same as before?
- Now connect a sinusoidal input of 5V amplitude and of frequency 1 Hz (Note the units of the frequency to be input) and see the output. If the scope curves do not look very smooth, open *Simulation/Model Configuration Parameters* (Ctrl+E) and tweak relative-tolerance to `1e-6` before running again.

  Task 1.3. *Answer the following:*

  1. *How many seconds does it take for the initial transient to die off in the output response?*
  2. *What is the expected gain-ratio (ratio of output to input amplitudes) from theory? You can find this by replacing s by $j\omega$ in the transfer-function and evaluating $|H(j\omega)|$.*
  3. *Zoom in the scope to find the amplitude ratio of the output wave to the input wave. Is it as expected?*

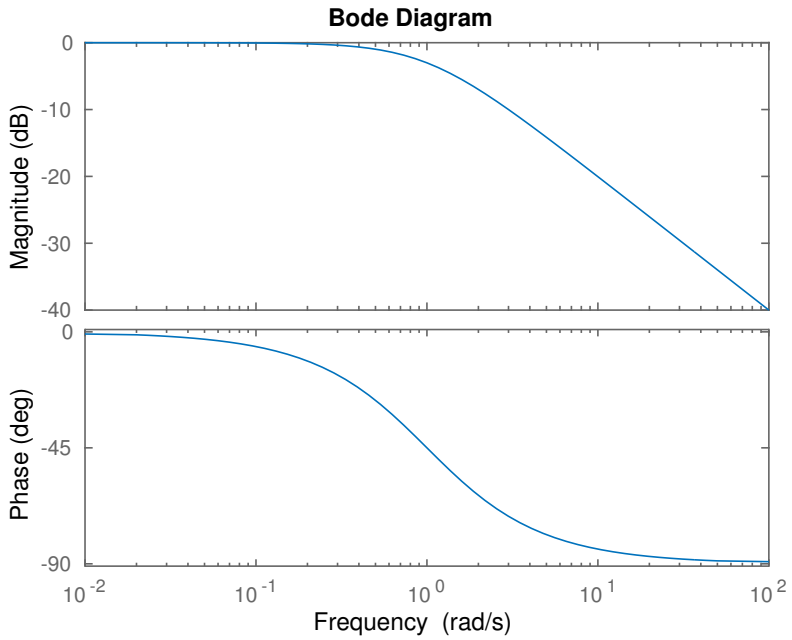


Figure 1.6: The bode plot of $H(s) = 1/(s+1)$.

1.2.3 Analysis of LTI Systems in Matlab

You can also use Matlab directly to analyze LTI systems, e.g. by drawing Bode plots. Put the following code in a Matlab script file and run it. A bode plot for the simple first-order system should pop up.

```
clear all;
k = 1;
s= tf('s'); % The Laplace variable
sysH= 1/(s+1); % The transfer function
bode(sysH);

% Redo the last Simulink part for the input frequency of 1 Hz
w= 2*pi;
[magnitude,phase]= bode(sysH, w)
% Does the magnitude agree with your previous answer?
```

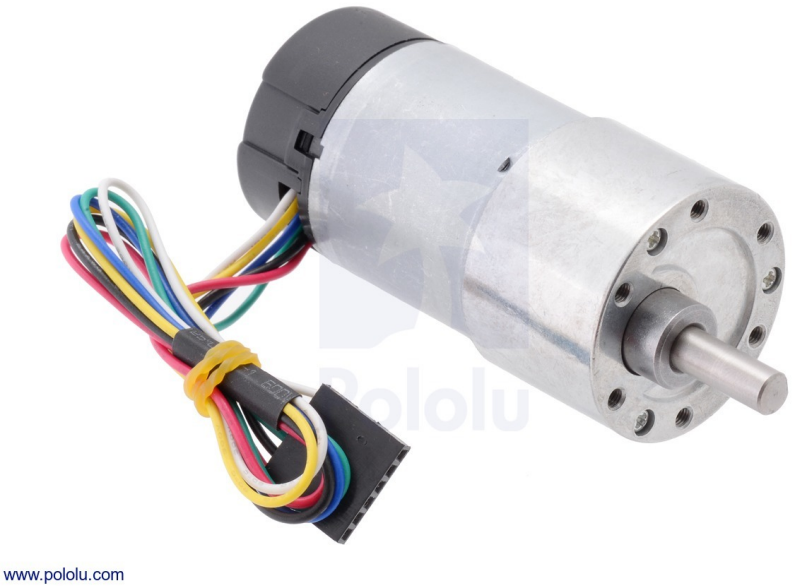


Figure 1.7: The Pololu 19:1 Metal Gearmotor 37Dx68L mm with 64 CPR Encoder. The three main parts of the assembly are clearly visible: The left black part is the encoder, the middle grayish part is the motor, and the rightmost silver part is the enclosure of the gear-box. Note how the output shaft is offset w.r.t. the motor axis due to the intermediate gears.

Table 1.1: Parameters for the Pololu 19:1 Metal Gearmotor at 12V. The values are at the output-shaft which includes the effect of the gear-box.

Parameter	**Value**	**In SI Units**	**Description**
Free-run speed	500 RPM	52.36 rad/s	ω_0, the shaft-speed at no-load
Free-run current	300 mA	0.3 A	i_0, the no-load current
Stall current	5000 mA	5 A	i_s
Stall torque	84 oz-in	0.5932 Nm	τ_s

1.3 A Brushed DC Motor Model

A brushed permanent magnet DC motor has the following main characteristics:

1. Irrespective of the voltage applied, the torque produced by the motor is proportional to the current flowing in the armature.

2. If you keep the external load constant, the shaft speed is proportional to the applied voltage.

As shown in Fig. 1.8, a brushed DC motor has a typical set of characteristic curves at its rated voltage.

Let us look at the vendor specifications of a real motor: Pololu 19:1 Metal Gearmotor 37Dx68L mm with 64 CPR Encoder. At the rated voltage of $v_a = 12V$, the various paramters are listed in Table 1.1.

The torques and rotational speeds are measured at the output shaft which is internally connected to a gear-train. So, *the effect of the gears is already incorporated* in these values. This is also clear looking at the values in Table 1.2.

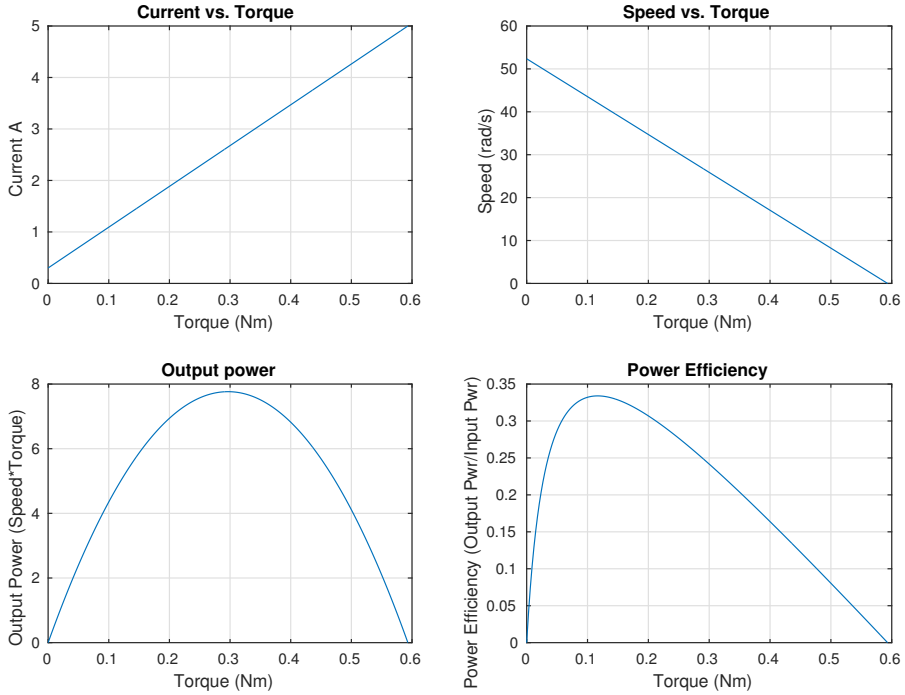


Figure 1.8: The motor charactertic curves. Note that there is one such set of curves for each rated voltage. These curves have been plotted for this Pololu motor using a modified version of this Matlab script. These curves have been plotted for a rated voltage of $12V$.

1.3.1 The Motor Torque Constant

The torque produced by the motor is approximately a linear function of the armature current:

$$\tau_m = \hat{k}_t \, i \tag{1.3}$$

where, $\hat{k}_t$ is called the motor torque constant.

As mentioned, Table 1.1 actually lists the torque τ_g which is measured at the output-shaft, not the motor-shaft torque τ_m directly. The gear-ratio $\eta = 18.75$, so we know that $\tau_g = \eta \, \tau_m$. Since we are only interested in the output-torque τ_g, let us rewrite (1.3) as

$$\tau_g = \eta \, \hat{k}_t \, i \triangleq k_t \, i \tag{1.4}$$

The parameter k_t is our effective motor torque constant.

Table 1.2: Parameter-variation across models, as the gear-ratio is varied. All values are at 12V.

Gear-ratio	ω_0	τ_s	i_s
1:1	11000 RPM	5 oz-in	5 A
19:1	500 RPM	84 oz-in	5 A
30:1	350 RPM	110 oz-in	5 A
50:1	200 RPM	170 oz-in	5 A
70:1	150 RPM	200 oz-in	5 A
100:1	100 RPM	220 oz-in	5 A
131:1	80 RPM	250 oz-in	5 A

This relation is plotted in the top-left subplot in Fig. 1.8. Some part of this produced torque is spent in overcoming the rotational friction (which increases if gears are present): Hence the curve torque/current curve starts at a non-zero current i_0, also called as the no-load current. It is the current drawn by the motor when no load (such as a wheel) is attached to the motor's shaft and it is free to run without any opposing torque.

The stall current i_s is the maximum current drawn by the motor when the shaft is stuck or is prevented from turning – at this maximum current, the motor shaft also applies the maximum torque called the stall-torque. Where are the stall current and the stall torque in the top-left subplot of Fig. 1.8?

Warning: As noted on the Pololu website, the stall condition is dangerous due to the high current drawn and can easily damage the motor if it persists: "*A good general rule of thumb is to keep the continuous load on a DC motor from exceeding approximately 20% to 30% of the stall torque. Stalling gearmotors can greatly decrease their lifetimes, occasionally resulting in immediate damage to the gearbox or thermal damage to the motor windings or brushes. Do not expect to be able to safely operate a brushed DC gearmotor all the way to stall. The safe operating range will depend on the specifics of the gearmotor itself.*"

Task 1.4. *Estimate k_t in SI units for this motor (Table 1.1) from the data provided using (1.4), the stall current, and the stall torque.*

1.3.2 The Back EMF

Let θ be the angle of rotation of the motor rotor: define $\theta(t=0) \triangleq 0$. $\dot{\theta}$ is the rotational speed of the rotor. Since the motor has conductors moving in a magnetic field, it also acts as a generator while turning. This generated voltage or electromotive force (emf) e works against the applied voltage v_a. This is why e is called the *back-emf*. It is the proportional to the rotational speed.

$$e = \hat{k}_e \, \dot{\theta}_m, \tag{1.5}$$

where $\dot{\theta}_m$ is rotation-speed of the motor-shaft. Since we can only measure the output-shaft's speed $\dot{\theta}_g$ which is reduced w.r.t. $\dot{\theta}_m$ due to the gear-ratio η, we rewrite (1.5) as

$$\begin{aligned} e &= \hat{k}_e \, \dot{\theta}_m \\ &= \frac{\hat{k}_e}{\eta} \, \dot{\theta}_g \\ &\triangleq k_e \, \dot{\theta}_g \end{aligned} \tag{1.6}$$

The constant of proportionality k_e is called the *electromotive force constant*. An upper-limit for it can be estimated from the data-sheet as

$$k_e \leq \frac{v_a}{\omega_0} \tag{1.7}$$

This limit is also taken as an estimate of the value sometimes.

Task 1.5. *Estimate k_e in SI units for this motor from the data provided using (1.7). We will get a better estimate later.*

1.3.3 The Mechanical Equations of Motion

The mechanical motion-equations for the motor can be written using Newton-Euler equations. However, since most motors produce low torque and high rotational speed, a gear-train is typically used to increase the output torque and reduce the output speed by the gear-ratio $\eta > 1$. The Pololu motor we have been considering actually has $\eta = 18.75$, not 19. The presence of gears requires us to formulate the equations of motion carefully.

Let us consider two systems:

- A small spur-gear G_m of radius R_m rigidly connected to the *motor-shaft*: Its angle of rotation is the same as that of the motor – let's denote it by θ_m. Let us also denote the torque generated by the motor-shaft as τ_m.
- A bigger gear G_g of radius R_g rigidly attached to the *output-shaft* that we see coming out of the motor-gearbox assembly in Fig. 1.7. This gear is enmeshed to the smaller spur-gear above and serves to step-down the speed and to step-up the torque. Let's denote the angle of rotation of the output shaft by θ_g. Let us also denote the torque produced on the motor-shaft as τ_g.

Note: We typically consider the motor-gearbox assembly as a black-box as we are only interested in the output-shaft's rotation speed $\dot{\theta}_g$ and the torque τ_g which appears on it: the values listed in Table 1.1 are also the ones corresponding to the output-shaft.

As we analyzed in GIMS-I, we have the following relations between the motor and the output shaft:

$$\frac{R_g}{R_m} = \frac{\theta_m}{\theta_g} = \frac{\dot{\theta}_m}{\dot{\theta}_g} = \frac{\ddot{\theta}_m}{\ddot{\theta}_g} = \frac{\tau_g}{\tau_m} = \eta > 1. \tag{1.8}$$

Let f be the force exerted by G_m on G_g at the point on their pitch-circles where they effectively enmesh. By Newton's third law, a reaction force $-f$ will be exerted by G_g on G_m. Additionally, on the output-shaft there is an external load τ_L present.

The Newton-Euler equations of motion can thus be writen based on the free-body diagrams of G_m and G_g.

- For the system consisting of the motor's rotor rigidly connected to G_m:

$$J_m\ddot{\theta}_m + b_m\,\dot{\theta}_m = \tau_m - f\,R_m \tag{1.9}$$

  J_m is the combined rotational intertia of the rotor, motor-shaft, and G_m about their common axis. b_m is the rotational friction coefficient for the motor-shaft. Using (1.8), we can rewrite this in terms of the output-shaft as:

$$J_m\,\eta\,\ddot{\theta}_g + b_m\,\eta\,\dot{\theta}_g = \frac{1}{\eta}\tau_g - \frac{1}{\eta}\,f\,R_g \tag{1.10}$$

  This can be re-written as

$$J_m\,\eta^2\,\ddot{\theta}_g + b_m\,\eta^2\,\dot{\theta}_g = \tau_g - f\,R_g \tag{1.11}$$

- For G_g, we have,

$$J_g\ddot{\theta}_g + b_g\,\dot{\theta}_g = f\,R_g - \tau_L \tag{1.12}$$

Adding together (1.11) and (1.12), we eliminate the unknown f:

$$\underbrace{(J_g + \eta^2 J_m)}_{\triangleq J}\,\ddot{\theta}_g + \underbrace{(b_g + \eta^2 b_m)}_{\triangleq b}\,\dot{\theta}_g = \tau_g - \tau_L. \tag{1.13}$$

This is our final equation of motion which is entirely in terms of the output shaft's rotation and torque. Note that τ_g is proportional to the current due to (1.4).

Estimation of the Equivalent Friction Coefficient

We can estimate b as follows: At no-load ($\tau_L = 0$) after the motor has reached a steady-state speed of $\dot{\theta}_g = \omega_0$, $\ddot{\theta}_g = 0$, the torque τ_g appearing on the output-shaft is exactly balanced by the equivalent friction. Substituting these values in (1.13), we get the estimate

$$b = \frac{k_t\,i_0}{\omega_0} \tag{1.14}$$

Task 1.6. *Estimate b in SI units for the motor in Table 1.1.*

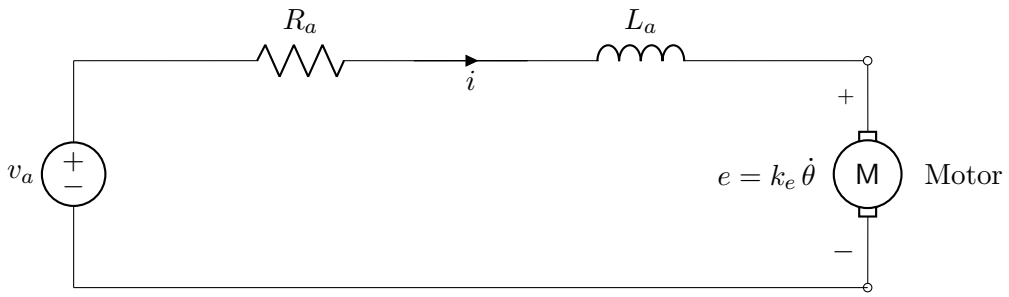


Figure 1.9: DC motor: Simplified electric circuit of the motor armature. The armature inductance L_a and armature resistance R_a are shown separately for analysis – although, of course, they are part of the motor.

Estimation of the Equivalent Moment of Inertia

We will study about the moments of inertia in the second part of GIMS-II. For now, for our model, we can estimate it approximately as follows. From Pololu's web-site, we have the following information:

- Motor size: radius $R = 0.037/2 = 0.0185$ m, length L= 0.068 m.
- Mass: $M = 0.215$ Kg.

For some back of the envelope computations, let us approximate the rotor as a thick hollow cylinder with inner-radius $R_1 = 0.25R$, outer-radius $R_2 = 0.8R$, and mass $m = 0.75M$. The moment of intertia of such a hollow cylinder *about its axis* is independent of its length and can be found from standard tables.

$$J_m = \frac{m}{2}(R_1^2 + R_2^2) \approx 2 \times 10^{-5}\,\mathrm{Kg}\,m^2. \tag{1.15}$$

Ignoring J_g in (1.13), we get

$$J \approx \eta^2\, J_m = (18.75)^2 \times 2 \times 10^{-5} = 0.007\,\mathrm{Kg}\,m^2. \tag{1.16}$$

Later on, we will present how this parameter can be measured experimentally using model identification techniques.

In Laplace Domain

Applying the Laplace transform to (1.13) and taking all initial conditions to be zero, we get,

$$(Js^2 + bs)\,\Theta_g(s) = k_t I(s) - T_L(s) \tag{1.17}$$

1.3.4 The Equivalent Electrical Circuit

The electrical part of the system can be described using the circuit shown in Fig. 1.9. Using KVL,

$$v_a - iR_a - L_a\frac{di}{dt} = e \overset{(1.6)}{=} k_e\dot{\theta}_g. \tag{1.18}$$

The armature resistance R_a leads to the usual ohmic power loss in a motor – this is termed *copper-loss*. To estimate R_a, note that in the stall condition, $\dot{\theta} = 0$ and at steady-state $di/dt = 0$: substitutig these in (1.18) gives us:

$$R_a = \frac{v_a}{i_s} \tag{1.19}$$

Task 1.7. *Compute R_a from the motor data given. Using this value, obtain a better estimate for k_e than the one we obtained earlier in (1.7)? Hint: Consider (1.18) in the no-load steady-state condition.*

Applying the Laplace transform to (1.18),

$$V_a(s) - (R_a + L_a s)\, I(s) = k_e\, s\, \Theta_g(s) \tag{1.20}$$

We are interested in a system-description in which the applied voltage $V_a(s)$ is the *input* (usually denoted as $U(s)$) and the output-shaft angle $\Theta_g(s)$ is the *output* (usually denoted as $Y(s)$). Hence, we combine (1.17) and (1.20) and eliminate $I(s)$. The external load torque $T_L(s)$ can be considered as another input.

Position Transfer Function

Let us first define

$$D(s) \triangleq (R_a + L_a s)(Js + b) + k_e k_t \tag{1.21a}$$

Then, from (1.17) and (1.20), by eliminating $I(s)$, we can derive a relation mapping the inputs to the output.

$$\Theta_g(s) = \frac{k_t}{s\, D(s)} V_a(s) - \frac{(R_a + L_a s)}{s\, D(s)} T_L(s) \tag{1.21b}$$

$$\triangleq H_{\theta v}(s)\, V_a(s) - H_{\theta \ell}(s)\, T_L(s) \tag{1.21c}$$

Task 1.8. *On a sheet of paper, derive (1.21b) from (1.17) and (1.20) as explained. These transfer-functions $H_{\theta v}(s)$ and $H_{\theta \ell}(s)$ are important for position-control (a.k.a. servo mode) of the motor.*

Speed Transfer Function

If instead we define the output as $y(t) = \dot{\theta}_g(t)$, we get another relation which is important for speed-control of the motor. In the context of the car speed control, this is called "cruise control".

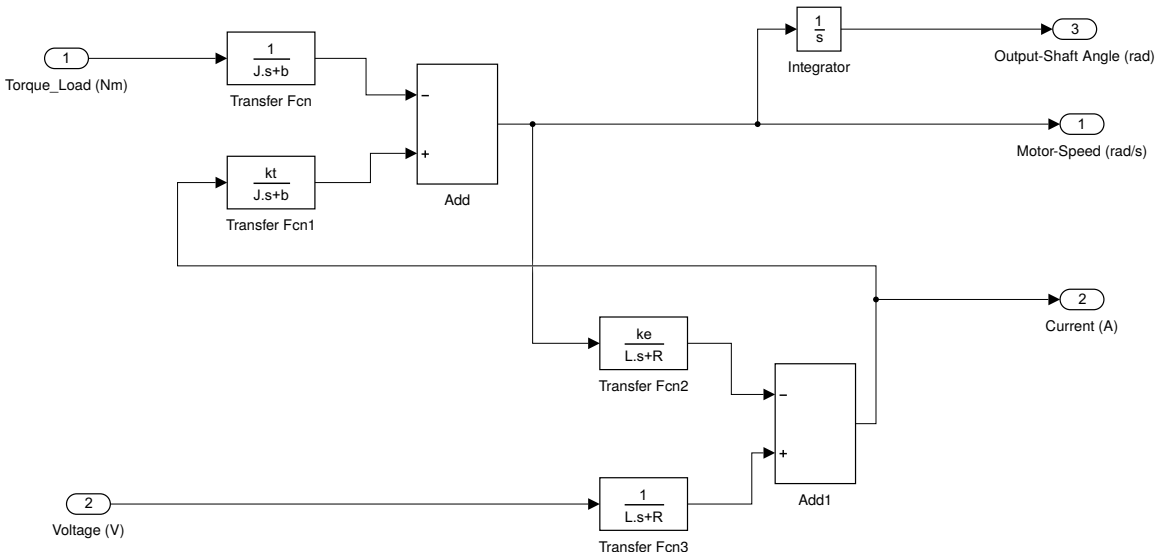
$$\begin{aligned}
\Omega(s) &\triangleq \mathcal{L}(\dot{\theta}_g(t)) \\
&= s\, \mathcal{L}(\theta_g(t)) \\
&= \frac{k_t}{D(s)}\, V_a(s) - \frac{(R_a + L_a s)}{D(s)}\, T_L(s) && (1.22a) \\
&\triangleq H_{\omega v}(s)\, V_a(s) - H_{\omega \ell}(s)\, T_L(s) && (1.22b)
\end{aligned}$$

Task 1.9. *Show that if the inductance L_a is small and can be ignored, $H_{\omega v}(s)$ can be written as a first order system*

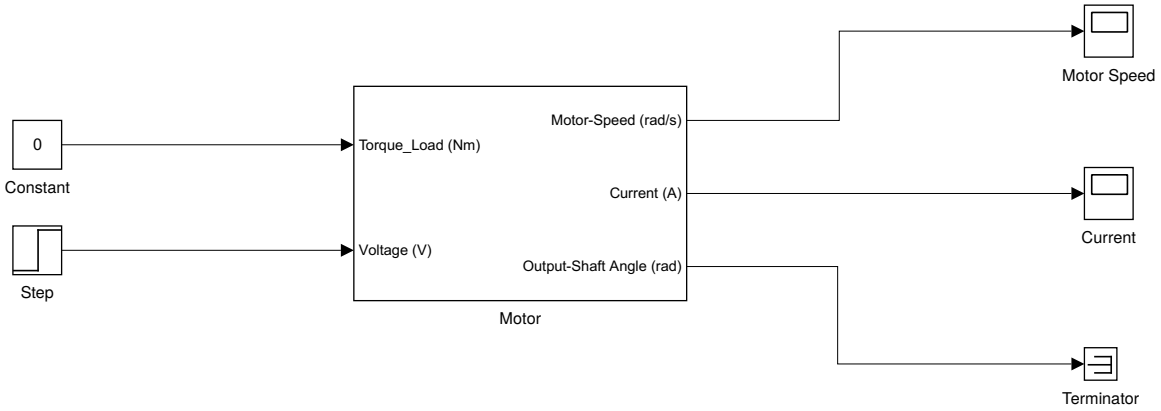
$$H_{\omega v}(s) \approx \frac{K}{s + \alpha} \tag{1.23}$$

What are α and K in terms of the other parameters?

We would now want to implement (1.22a) in Simulink. Most parameters have alreday been computed for the Pololu motor; the sole remaining one is considered next.



(a) The Simulink model.



(b) The above blocks grouped together to form the motor subsystem.

Figure 1.10: The motor model.

The Inductance

The inductance L_a is usually small, and for our simulation we use a typical value of $0.1\,mH$ (milli Henries).

1.4 Simulation of the Motor Model

Although the input/output forms (1.21b) and (1.22a) are useful in analysis of system stability and performance, they hide the intermediate variable $I(s)$. Since we are interested in seeing the behavior of all pertinent system states, we will implement the original equations (1.17) and (1.20) in Simulink using blocks.

For implementation, it is best to rewrite the equations (1.17) and (1.20) in terms of the shaft-speed as follows (verify):

$$\Omega(s) = \frac{k_t}{Js+b} I(s) - \frac{1}{Js+b} T_L(s) \tag{1.24a}$$

$$I(s) = \frac{1}{L_a s + R_a} V_a(s) - \frac{k_e}{L_a s + R_a} \Omega(s) \tag{1.24b}$$

1.4.1 Response to Various Step Inputs

Task 1.10. *Proceed as follows:*

1. *Create the model as shown in Fig. 1.10. Verify that it is indeed implenting the equations (1.24a) and (1.24b).*

2. *In the Matlab workspace create the variables representing the motor-parameters such as k_t, k_e, R_a, L_a, J, b, etc.*

   - *Type* `save('motor_params.mat')` *on the command line to save these values for later use.*
   - *To make sure that these parameter values are automatically loaded when you open this model the next time, navigate to the menu: File/Model Properties/Model Properties, click on the "Callbacks" tab of the dialog-box that pops up, and under the "PreLoad-Fcn" callback, type* `load('motor_params.mat')` *on the right pane. Click OK and then save the model. You can test by closing the model, typing* `clear all` *to clear the workspace, and opening the model again.*

3. *Apply a step of 12 V at t=1. Keep the load-torque to 0. Simulate the system from 0 to 10 seconds using the default solver (ode45).*

4. *If the results shown in the scopes do not appear proprly, press Ctrl+E and switch the solver to ode15s (stiff/NDF). Press the "Autoscale" button of the scope to see the whole curve.*

5. *Explain the behavior of the current (the initial peak) and the speed scopes. Hint: Make use of (1.18). Verify that they settle to the no-load values in Table 1.1? You can use the zoom button in the scope.*

6. *Now at the external load-torque input add a step torque of 0.25 Nm (about half of the stall-torque) at time $t = 6$ seconds. Increase the simulation interval to 15 seconds. Explain what you see.*

7. *Next, keeping the load-torque zero, apply an input voltage which steps from 0V to 12V at t=1 seconds and then drops to 0V again at t=6 seconds. You can achieve this by adding together the signals of two step sources. Explain why you see a negative current spike at t=6 seconds. Hint: Make use of (1.18). The reverse voltage spike which causes this current can damage other components (like transistors) connected to the circuit – this is why, usually a flyback/snubber diode is used.*

1.4.2 Response to a Sinusoidal Input

Task 1.11. *Use the Source/Sine-Wave block to input a sine-wave voltage of amplitude 6V and frequency 1 rad/s to the motor-model. Simulate for about 30 seconds. After the initial transients have died down and the system reaches steady-state, note down the amplitude $\hat{I}$ of the sinusoidal current response from the scope – use the zoom-button. Compute the ratio*

$$h(\omega = 1) = \frac{\hat{I}}{V_a} = \frac{\hat{I}}{6}. \tag{1.25}$$

We will need this value later.

1.4.3 Response to PWM

If we use a motor controller connected to an Arduino to control the motor, we will supply the motor PWM pulses rather than a continuous voltage signal. To build our intuition about how the motor-speed and current respond to PWM signals, let us connect our Simulink model to a source block of type pulse-generator.

Task 1.12. *Set the time-period of the pulse-generator to correspond to that of an Arduino PWM signal, namely 2 milli-seconds. Set the amplitude to be 12 V. Vary the pulse width parameter to vary the PWM duty-cycle value in percentage. See how the motor's speed and current react to such an input.*

Interestingly, running a DC motor with a PWM signal will cause the motor to emit a characteristic audible "whine" (having the same frequency as the signal) while running.

1.4.4 Model Identification

We could estimate most of the motor-parameters from the no-load and stall values given in the data-sheet of the motor. The two exceptions were the moment of inertia J and the inductance L. We estimated the moment of inertia J using its formula, making some assumptions about the geometry. In real life, however, it is preferable to find out these parameters experimentally by subjecting the system to specific inputs and measuring its response. This is called system or model identification.

Estimating the Moment of Inertia

Let us assume we are given the motor-model block that we just created as a black-box. We now want to experimentally estimate the parameter J for this black-box motor. Recall from (1.23) that in the no load condition, if we assume L_a to be negligibly small, we have the following transfer function

$$\frac{\Omega(s)}{V_a(s)} = \frac{K}{s+\alpha}, \text{ where,} \tag{1.26a}$$

$$K = \frac{k_t}{R_a J} \tag{1.26b}$$

$$\alpha = \frac{R_a b + k_t k_e}{R_a J} \tag{1.26c}$$

Now, if we apply a unit-step input $v_a = 12\,u(t)$ to this system, we can estimate J from the system response. In Laplace domain $U_a(s) = 12/s$. Substituting this in (1.26),

$$\Omega(s) = \frac{12K}{s(s+\alpha)} \qquad = \frac{12K}{\alpha}\frac{1}{s} - \frac{12K}{\alpha}\frac{1}{s+\alpha} \tag{1.27}$$

Taking the inverse Laplace transform,

$$\dot{\theta}_g(t) = \frac{12K}{\alpha}(u(t) - e^{-\alpha t}) \tag{1.28}$$

So, the motor-speed increases from 0 to $12K/\alpha$ with a time-constant $T_c = 1/\alpha$. T_c is called the *mechanical time-constant* since it depends primarily on the inertial properties. It is the time in which the motor-speed increases to 63% of its steady-state value. This allows us to measure this quantity. Once, T_c is measured, we can estimate the moment of inertia using (1.26c) as

$$J = \frac{T_c(R_a b + k_t k_e)}{R_a} \tag{1.29}$$

Task 1.13. *In your model, set the load-torque input to zero and apply a step voltage input of 12V at $t = 0$. Measure T_c from the speed-output scope. Now estimate J. Does it match with the value you knew beforehand?*

Note that this method can easily be used in practice with a real motor.

Estimating the Inductance

Since the inductance is small, it is usually ignored. However, if its estimate is desired, we can measure the steady-state response of the no-load system to a sinusoidal input.

Task 1.14. *Find the transfer function $H_I(s) = I(s)/V_a(s)$ from (1.24) by assuming no-load and eliminating $\Omega(s)$ between the two equations.*

Task 1.15. *We would like to compute $|H_I(j\omega)|$ at $\omega = 1$ using the known parameter values in Matlab. $H_I(j\omega)$ is the complex-number which you get by substituting $s = j\omega$ in $H_I(s)$. Compare $|H_I(j\omega)|_{\omega=1}$ with the ratio $h(\omega = 1)$ you found back in (1.25). Are they the same?*

Task 1.16. *(Bonus) How can you use the result above to find L_a if it is the only unknown parameter? It will involve solving a fairly nonlinear equation for L_a, so just give a sketch of the procedure.*

In literature you also find another method for finding L_a using the stall-response. However, as this could possibly damage the motor, that method will not be discussed here.

1.5 The PID Controller For Speed Regulation

We are now ready to attach a controller to our motor as shown conceptually in Fig. 1.11. The input to the controller is a reference speed $R(s)$ which we want the motor to follow. The actual motor speed $\Omega(s)$ is observed by a sensor (e.g. the quadrature encoder we will look at later).

The controller will observe the difference $E(s)$ between the commanded speed $R(s)$ and the current speed $\Omega(s)$ and change the motor input-voltage $V(s)$ so that the difference/error $E(s)$ converges to 0. We want our controller to be robust, so it should work even in the presence of disturbances created by changes in the load-torque $T_L(s)$.

We will design a Proportional-Integral-Derivative (PID) controller $G(s)$ for this task. In time domain, the error signal is $e(t) = r(t) - \omega(t)$, where $\omega(t) = \dot{\theta}_g(t)$. The PID controller will essentially create its output $v(t)$ as follows:

$$v(t) = K_P\, e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{de(t)}{dt}. \tag{1.30}$$

In Laplace domain, the controller's transfer function looks like

$$G(s) = \frac{V(s)}{E(s)} \qquad\qquad = K_P + K_I \frac{1}{s} + K_D s \tag{1.31}$$

1.5.1 The Closed-Loop Transfer-Function

After we connect the controller and attach it to the speed feedback, we'd like to know how the output $\Omega(s)$ responds to the reference input $R(s)$ and how it is affected by the load-torque disturbance $T_L(s)$. To do this let us look at Fig. 1.11. We can immediately write the following

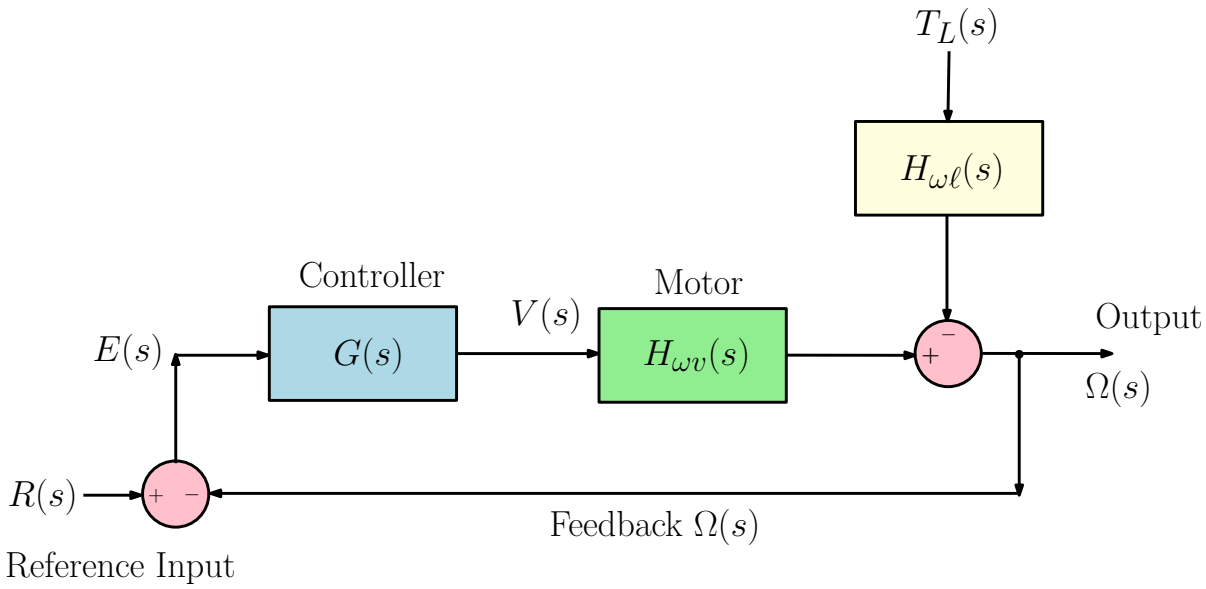


Figure 1.11: The conceptual overview. Refer to Eq. (1.22) for the definitions of the transfer functions $H_{\omega v}(s)$ and $H_{\omega\ell}(s)$.

relations by just following the signal arrows:

$$E(s) = R(s) - \Omega(s) \tag{1.32a}$$
$$V(s) = G(s)\,E(s) \tag{1.32b}$$
$$\Omega(s) = H_{\omega v}(s)\,V(s) - H_{\omega\ell}(s)\,T_L(s) \tag{1.32c}$$
$$\overset{(1.32b)}{=} H_{\omega v}(s)\,G(s)\,E(s) - H_{\omega\ell}(s)\,T_L(s) \tag{1.32d}$$
$$\overset{(1.32a)}{=} H_{\omega v}(s)\,G(s)\,(R(s) - \Omega(s)) - H_{\omega\ell}(s)\,T_L(s) \tag{1.32e}$$

We have the output $\Omega(s)$ on both left and right sides of the above equation. Solving for it gives,

$$\Omega(s) = \frac{H_{\omega v}(s)\,G(s)}{1 + H_{\omega v}(s)\,G(s)} R(s) - \frac{H_{\omega\ell}(s)}{1 + H_{\omega v}(s)\,G(s)} T_L(s) \tag{1.32f}$$

This equations clearly relates the output to the reference input and the disturbance. Perhaps, it is even more illuminating if we look at how the error between the reference-input and output is influenced by the the input and the disturbance. This will help us answer the question if the overall system is stable and the error converges to zero. Substituting (1.32f) in (1.32a),

$$E(s) = \frac{1}{1 + H_{\omega v}(s)\,G(s)} R(s) + \frac{H_{\omega\ell}(s)}{1 + H_{\omega v}(s)\,G(s)} T_L(s) \tag{1.32g}$$
$$\triangleq H_{er}(s)\,R(s) + H_{e\ell}(s)\,T_L(s) \tag{1.32h}$$

Stability

From our theory, we know that for stability, the denominator $1 + H_{\omega v}(s)\,G(s)$ should have its roots (poles) such that their real parts are negative, i.e. they lie in the left-half complex plane.

Recall from (1.22) and (1.21a) that

$$H_{\omega v}(s) = \frac{k_t}{D(s)} \tag{1.33}$$

Substitutig this in (1.32h), let us analyze $H_{er}(s)$ and $H_{e\ell}(s)$ separately.

$$H_{er}(s) = \frac{1}{1 + H_{\omega v}(s)\, G(s)} \qquad = \frac{sD(s)}{sD(s) + k_t(K_D s^2 + K_P s + K_I)} \qquad \triangleq \frac{sD(s)}{P(s)} \tag{1.34}$$

We know that, for the system to be stable, the denominator $P(s)$ should have the roots in the LHP. Let's write $P(s)$ in expanded form using (1.21a).

$$P(s) = s[(R_a + L_a s)(Js + b) + k_e k_t] + k_t(K_D s^2 + K_P s + K_I) \tag{1.35}$$

$$= L_a J s^3 + (R_a J + L_a b + k_t K_D)s^2 + (R_a b + k_e k_t + K_P k_t)s + k_t K_I \tag{1.36}$$

To find the roots of this third-degree polynomial, you would typically give some values to all the constants in Matlab and then define the polynomial as

```
K_P= 0.0084
K_I= 0.02
K_D= 0
>> P= [L*J, (R*J + L*b + kt*K_D),(R*b + ke*kt + K_P*kt), kt*K_I]

P =

    0.0000    0.0168    0.0282    0.0024

>> r= roots(P);
>> r(1)

ans =

  -2.3998e+04

>> r(2)

ans =

   -1.5890

>> r(3)

ans =

   -0.0889
```

Clearly, all the roots are in the LHP and even real (though this is not necessary for stability). One of the roots is much more negative than the other two and hence it is not critical for determining stability.

To ease our analysis, we assume that L_a is quite small and can be ignored. Using this in (1.36), we get

$$P(s) = (R_a J + k_t K_D)s^2 + (R_a b + k_e k_t + K_P k_t)s + k_t K_I \tag{1.37}$$

We thus reduced the polynomial to a second degree one, for finding the roots of which we do not necessarily need Matlab, but let's try it anyway:

```
>> P= [(R*J + kt*K_D),(R*b + ke*kt + K_P*kt), kt*K_I]

P =

    0.0168    0.0282    0.0024

>> roots(P)

ans =

   -1.5889
   -0.0889
```

Comparing these to the roots earlier, we note that we have effectively discarded the one non-critical very negative root.

Steady-State Response

Let's say you've selected the controller parameters such that $P(s)$ is stable. For such a stable system, the Final Value Theorem (FVT) of Laplace-Transform applies. FVT allows us to compute the constant steady-state value of a time-function $y(t)$ given its Laplace transform $Y(s)$. For FVT to apply, the precondition is that $Y(s)$ has only stable poles, i.e. they are in the LHP. The statement of the theorem is as follows.

Final Value Theorem: If all poles of $Y(s)$ are in the LHP, then

$$\lim_{t\to\infty} y(t) = \lim_{s\to 0} sY(s) \tag{1.38}$$

We now want to use this theorem to show that our PID controller takes the error (the discrepancy between the reference input and the actual output) to zero in steady-state. This should at least hold for step reference inputs. Furthermore, this behavior should be robust to unexpected step load-torque changes.

Task 1.17. *First assume that the disturbace $T_L(s) = 0$. Assume that a unit-step reference input $r(t) = k\,u(t)$ is applied to the system, i.e. we want the controller to take the motor to a constant speed k rad/s. Apply the FVT to (1.32g) to find the steady state error $e(t) = r(t) - \omega(t)$.*

Task 1.18. *Next assume that $r(t) = k\,u(t)$ and $T_L(t) = \ell\,u(t)$, i.e. the system is now also disturbed by a load-torque ℓ Nm. Using the FVT, determine the effect of this disturbance on the system steady-state error.*

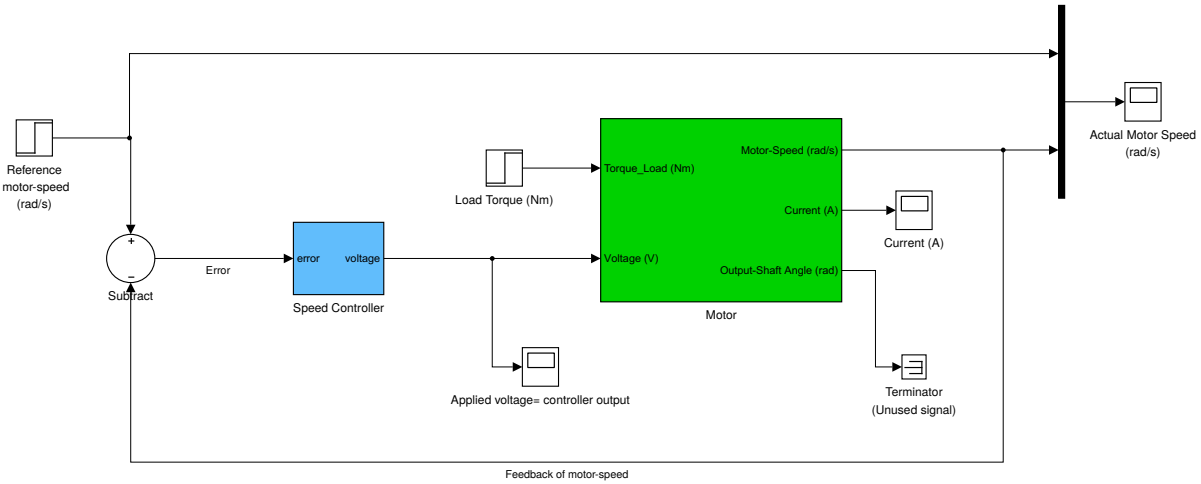
Simulink Motor Model with PID Controller

Task 1.19. *Copy your previous motor-model .slx file to another file and open it in Simulink. In this new model, create the controller subsystem connected to the motor as shown in Fig. 1.12.*

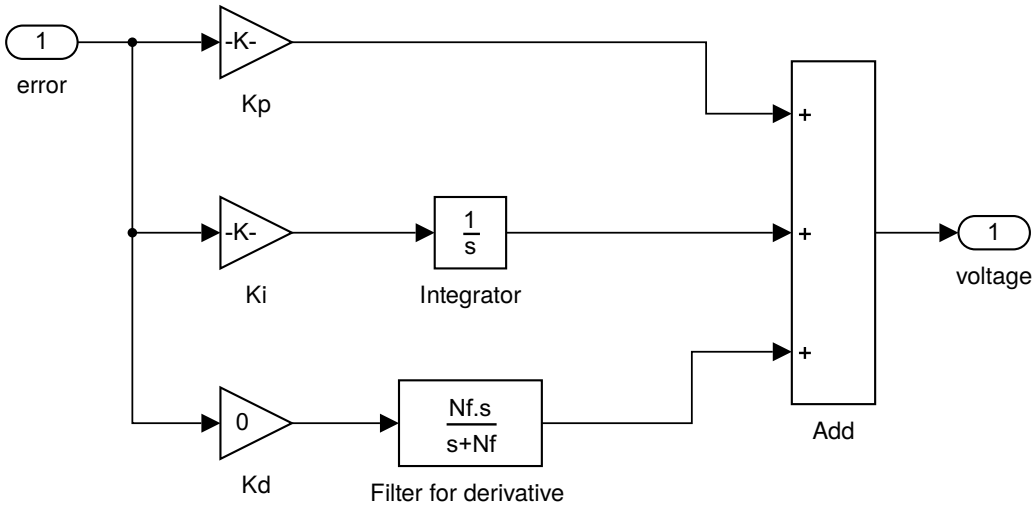
1.5.2 Proportional Control Only

Task 1.20. *Proceed as follows:*

1. *In our controller, let us set $K_I = K_D = 0$ and $K_P = 1$. Let us select a reference speed $r(t) = 300$ RPM (convert to rad/s before applying to the model) which is less than ω_0 and hence doable. Set the load-torque input to 0. Look at the scope: Did $\omega(t)$ reach $r(t)$? If not, measure the steady-state error $e(t)$.*



(a) The overview of the motor and PID speed controller with negative feedback of speed.



(b) The PID controller subsystem (the blue block above). Set the derivative filter $N_f = 100$ as default. The derivative filter is explained later in Sec. 1.5.4.

Figure 1.12: Motor speed control: The controller makes the motor-speed follow the input reference-speed by changing the voltage applied to the motor based on the observed feedback of the current speed.

2. *Since there is some steady-state error, let us increase K_P to 10 and simulate again. What is the steady-state error now? Since it decreased, we could just keep increasing K_P till the steady-state error is smaller than some threshold, right?*

3. *Wrong. Some part of system just went berserk at $K_P = 10$. Look at the scopes to figure out what went wrong. The moral of the story is that if we just use K_P, there will always be a steady-state error. To have it converge to zero, we need a non-zero K_I.*

1.5.3 Proportional and Integral Control

Task 1.21. *Proceed as follows:*

1. *Let us set $K_D = 0$ as before but set $K_P = 0.0084$. Select a value of K_I where the system is stable but just begins to show some oscillations. You can find this by writing an expression for the two roots of (1.37) in terms of K_I and find the value of K_I when the discriminant just turns 0. Simulate the system for 30 seconds with this value. Is there a steady-state error?*

2. *Now select a value of K_I slightly above this threshold, and simulate again. Do you see a ripple before the system settles down?*

3. *Now look at the current values in Amperes. Are they in the safe range? The nice thing about our controller is that it even damps the current-spike when a step-input is applied.*

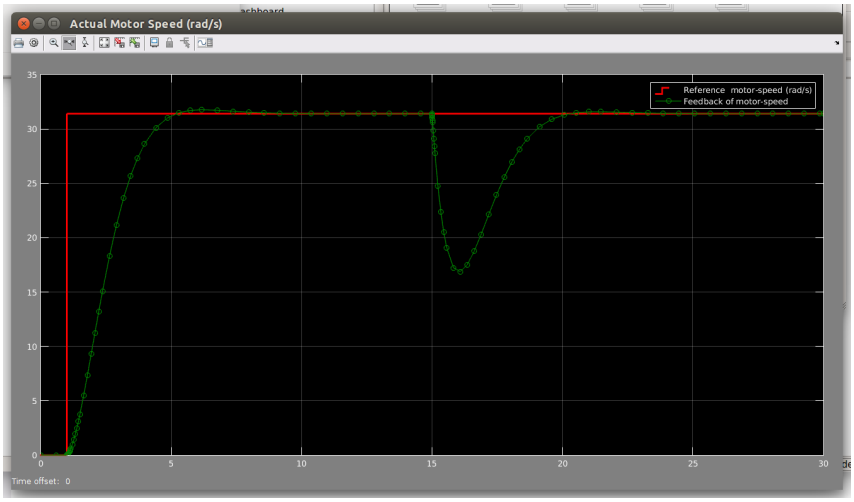
4. *Now add a step load-torque disturbance which is half of the stall torque at $t = 15$ seconds. Did the steady-state error still go to zero? Also look at the other scopes: Are all values within limits? Refer to Fig. 1.13 for an example.*

5. *You are, of course, not restricted to step-inputs. Let us apply a reference-speed signal which is sinusoidal with amplitude of 5 rad/s and a bias of 15 rad/s and the frequency of the sine-wave is 1 rad/s. Keep the step torque-load at $t = 15$ sec. Your scope should look like Fig. 1.14. Are the voltages and currents within limits?*

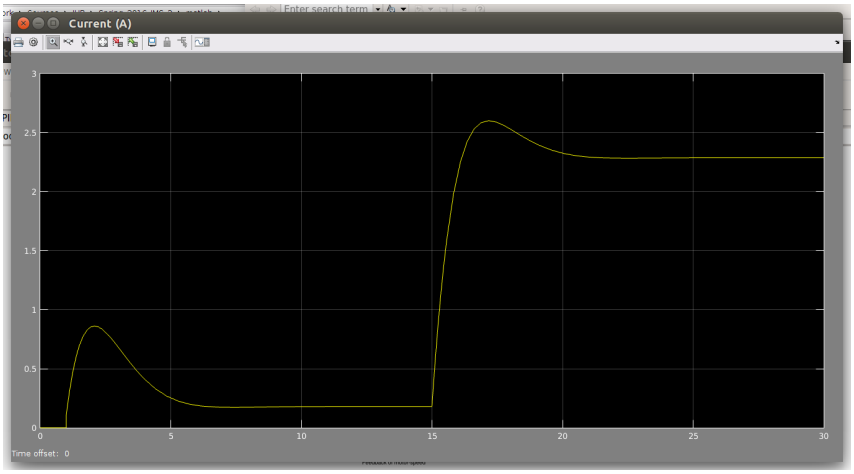
6. *In the previous example, let's try to see if the observed amplitude reduction and phase-difference between the reference and the actual speed sinusoidal signals can be also predicted by the Bode plot. The Bode plot of which transfer function will you have to plot?*

Task 1.22. *To plot the Bode plot proceed as shown in the code listing below. Make sure you use your chosen controller constants rather than the ones shown. Do the magnitude and phase-difference shown by the Bode plot at 1 rad/s match with what you observe in the speed scope?*

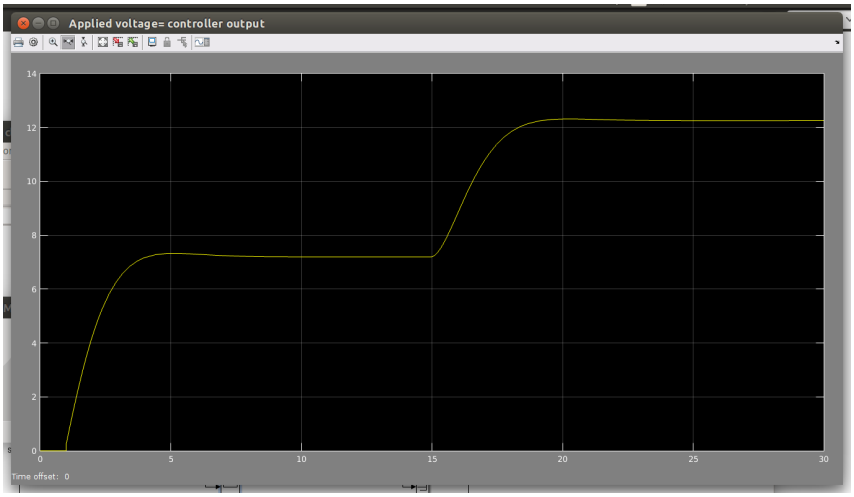
```
>> s= tf('s');

>> Ds= (R + L*s)*(J*s + b) + ke*kt

Ds =

  7e-07 s^2 + 0.0168 s + 0.02719

Continuous-time transfer function.

>> Hwv= kt/Ds

Hwv =

```



(a) The output response to the load-torque step at $t = 15$.



(b) The current response to the load-torque step at $t = 15$.



(c) The voltage output produced by the controller to overcome the load-torque and maintain the reference-speed. Since the voltage is slightly more than the supply-voltage of 12V, we see that in practice, the controller will saturate and will not be able to maintain the reference-speed exactly at this load-torque.

Figure 1.13: Response to a step load-torque of 0.25 Nm at $t = 15$ s.

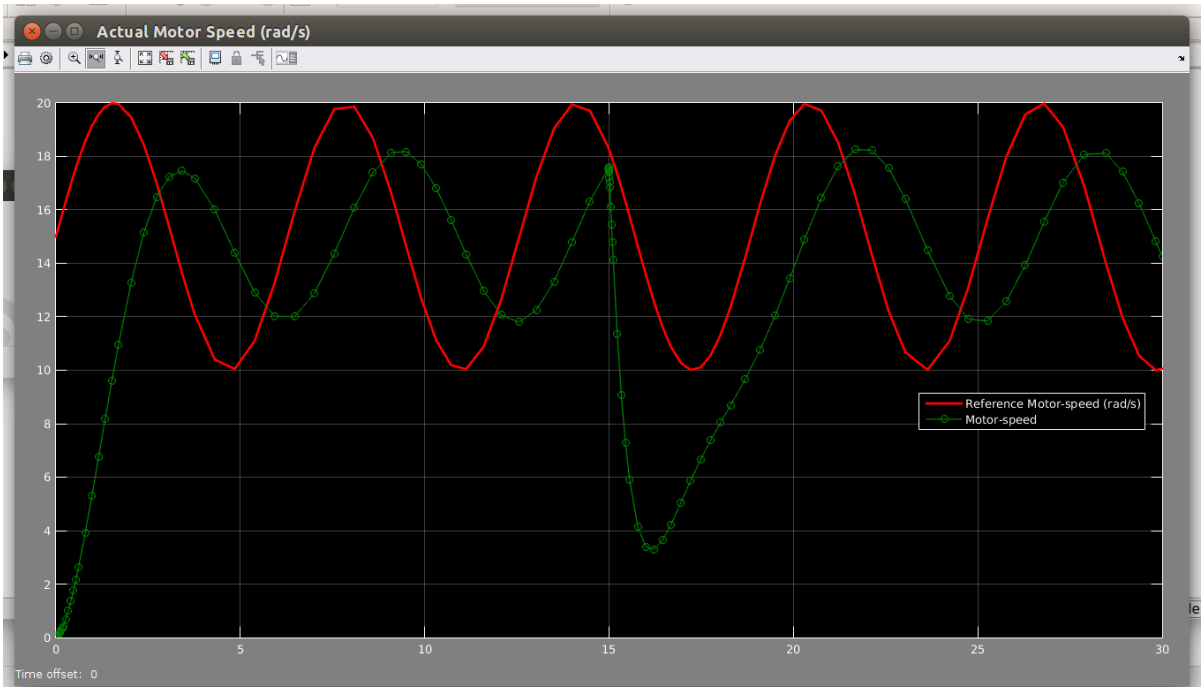


Figure 1.14: Response of the controller to a sinusoidal reference speed.

```
              0.1186
  -----------------------------
  7e-07 s^2 + 0.0168 s + 0.02719

Continuous-time transfer function.

>> K_P= 0.0084; K_I= 0.15; K_D= 0;
>> Gs= K_P + K_I/s + K_D*s

Gs =

  0.0084 s + 0.15
  ---------------
         s

Continuous-time transfer function.

>> T= Hwv*Gs/(1 + Hwv*Gs)

T =

                6.976e-10 s^4 + 1.676e-05 s^3 + 0.0003261 s^2 + 0.0004839 s
  ------------------------------------------------------------------------------------
  4.9e-13 s^6 + 2.352e-08 s^5 + 0.0002823 s^4 + 0.0009303 s^3 + 0.001065 s^2 + 0.0004839 s

Continuous-time transfer function.

>> bode(T) % plot
>> grid on;
>> w=1; % rad/s
>> [mag, phase]= bode(T,w)
% The magnitude and phase of the transfer-fn at a specific frequency.
```

If you do not want to do the algebra to arrive at the closed-loop transfer function in (1.32f), you could also use some functions in Matlab to connect transfer-function blocks in series or feedback arrangements and get the equivalent transfer function. So instead of explicitly supplying the expression for the variable `T` in the previous code-listing, we could have also found it as follows:

```
T1= feedback(series(Gs, Hwv), 1, -1)  % read doc feedback
T1 =

             0.0009966 s + 0.0178
  ------------------------------------------
  7e-07 s^3 + 0.0168 s^2 + 0.02819 s + 0.0178

Continuous-time transfer function.
```

Why is this `T1` different from the transfer function `T`? The mystery is solved when we write both in their pole-zero forms

```
 >> zpk(T1)

ans =

          1423.7 (s+17.86)
  ---------------------------------
  (s+2.4e04) (s^2 + 1.678s + 1.059)

Continuous-time zero/pole/gain model.

>> zpk(T)

ans =

      1423.7 s (s+2.4e04) (s+17.86) (s+1.619)
  -----------------------------------------------
  s (s+2.4e04)^2 (s+1.619) (s^2 + 1.678s + 1.059)
```

They are both clearly the same.

1.5.4 The Derivative Part of the PID Control

You can set K_D to a non-zero value to observe its effect. Normally, it would make the overall system respond faster to the reference signal since the "trend" of the error is being used also by the controller.

However, *taking the derivative of a noisy signal tends to amplify the noise*, so it should not be done unless really necessary. Another problem is that finding the derivative requires future values which is not possible in a causal system. You may be tempted to use the Simulink Continuous/Derivative block but it just estimates the derivative numerically and usually leads to numerical problems in Matlab. This is why, we use the derivative filter as shown in Fig. 1.12(b). It has the net effect of filtering out high-frequencies ($\omega > N_f$) as shown in Fig. 1.15 and at low-frequencies it is a very good approximation of the derivative transfer function s. Having the degree of the denominator polynomial of the transfer-function $\geq$ the degree of the numerator polynomial also makes our derivative block causal, i.e. realizable.

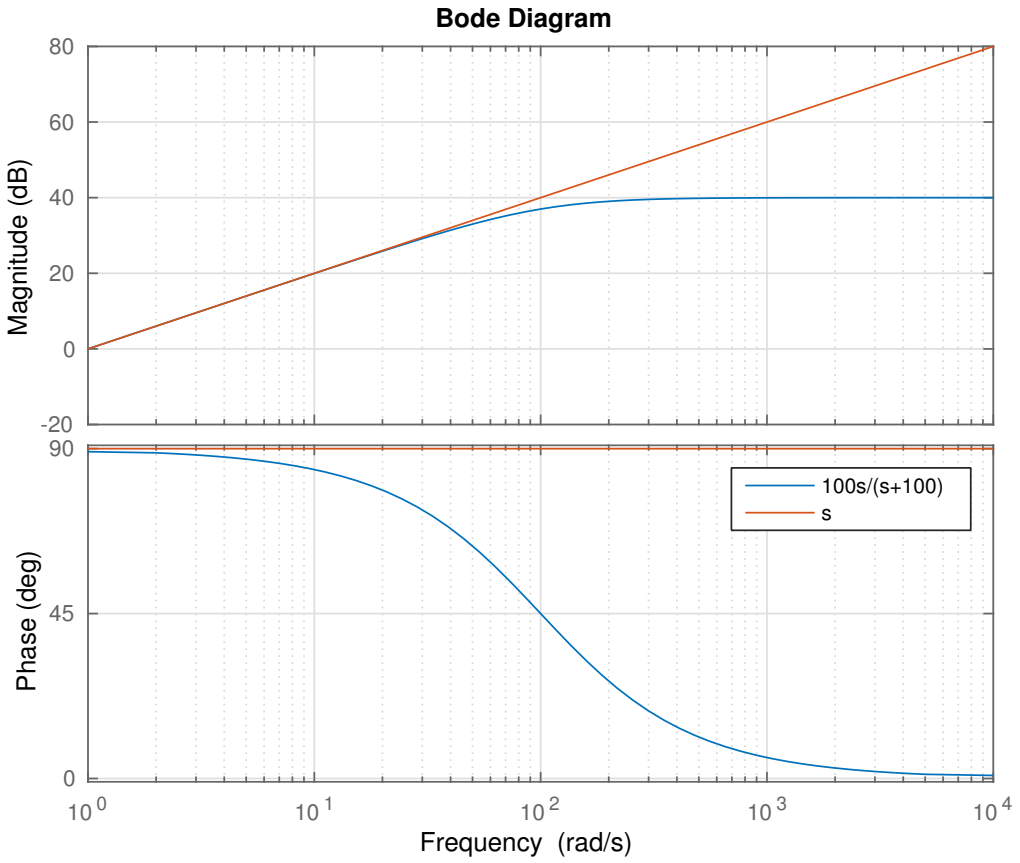


Figure 1.15: Why do we need a derivative-filter?

We saw that using just K_P and K_I was enough to make the steady-state error go to zero and to tune the dynamic response in the case of the motor-model. Hence, we do not really need to pursue the derivative part any further for this model.

Task 1.23. *How will you realize the derivative filter using a gain (multiplication by a constant) block and negative feedback with an integrator block?* ***Hint****: On the Matlab command line try* `Hsp= feedback(Nf, 1/s, -1)`*. Now draw this as a block diagram using only gain, integrator and a negative feedback.*

1.5.5 PID Gains Tuning

You may still be puzzled by the question of how to actually select controller gains/parameters K_P, K_I, K_D. The following are some of the considerations for this selection:

- The closed-loop transfer function should be stable for these values, i.e. the poles should be in LHP.
- Usually, as you increase the controller gains, two things happen:
  - The system responds faster, but,
  - The inputs applied to the system also increase and may exceed allowable limits.

  So, you need to select the controller parameters which satisfy some tradeoff between these two aspects. Usually, as we have done, a K_P is selected, and subsequently K_I and K_D.

There are also heuristic methods such as the Ziegler-Nichols method for gains selection based on step-response of a black-box plant/system: however it is applicable more for systems whose output shows an initial delay to the applied input. This happens, e.g. in systems in chemical engineering. For motor speed control this method is not applicable.

1.6 Discretization Of the Controller

In our design up to now, we have designed the controller in continuous time-domain – hence it can be implemented in an analog circuit, as usually done in the past. Nowadays, most controllers are implemented in microcontrollers (like Arduino): this gives more flexibility in their implementation. So now we're faced with the question of how to implement our PID controller in discrete time. Note that only the controller is discretized, not the plant (in this case, the motor-model), since the plant still responds in continuous time.

1.6.1 Transfer Functions in Discrete Domain

In discrete domain, the equivalent of Laplace-transform is the z-transform (pronounced "zee" transform). The theoretical details will be done in the class – here we just provide a brief summary.

A discrete controller does not run continuously but after every sample-period, say T seconds. In between the sample times, the output is held constant. The continuous time output $u(t)$ is now written as

$$u_k \triangleq u(t = kT), \qquad k = 0, 1, 2, \ldots. \tag{1.39}$$

The z-transform of the discrete signal u_k is defined as

$$\mathcal{Z}(u_k) \equiv U(z) \triangleq \sum_{k=0}^{\infty} u_k \, z^{-k} \tag{1.40}$$

Tables of z-transforms of common discrete signals are available, and just like in the case of Laplace transforms, inverse transforms are ususally found by partial-fractions expansion and table-lookup. One important result is regarding the delay. Let us define a new signal $w_k = u_{k-1}$, so w_k is delayed version of u_k by one sample period.

$$\mathcal{Z}(w_k) = \mathcal{Z}(u_{k-1}) = z^{-1}U(z). \tag{1.41}$$

Hence, premultiplication by z^{-1} represents a delay by 1.

The discrete transfer-function for the controller can be written analogous to the continuous case in (1.31)

$$G(z) = \frac{V(z)}{E(z)} \tag{1.42}$$

Both $V(z)$ and $E(z)$ are usually polynomials of z^{-1}.

Now, the remaining issue is how to convert the PID control law in s to an expression in z. This is where Tustin's method comes in.

1.6.2 Tustin's Method or the Bilinear Approximation

In this method, we first design our controller in continuous domain using Laplace transform. Once this is done, we apply Tustin's method to the control law which involves the following

replacement:

$$s \longleftarrow \frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) \tag{1.43}$$

Let's apply this to the PID controller of (1.31). To keep the expression simpler, let us just consider the PI part. As $G(s) = K_P + K_I/s$,

$$G(z) = K_P + K_I\frac{T}{2}\left(\frac{1+z^{-1}}{1-z^{-1}}\right) \tag{1.44}$$

Although in this example the algebra was simple, for more complex cases, you can take Matlab's help as shown below

```
>> s= tf('s');
>> Gs= K_P + K_I/s

Gs =

  0.0084 s + 0.15
  ---------------
         s

Continuous-time transfer function.

>> Gz= c2d(Gs, Tsample, 'tustin')

Gz =

  0.00915 z - 0.00765
  -------------------
         z - 1

Sample time: 0.01 seconds
Discrete-time transfer function.
```

You can now use $G(z)$ directly in our Simulink model. Simulink has the nice feature that we can mix continuous time and discrete time blocks. Simulink takes care of the sample-and-hold for the output for us.

Task 1.24. *Proceed as follows:*

1. *To start discretizing the controller first make a copy of your work up to now and rename the file. Now open the new Simulink model file.*
2. *Create a variable* `Tsample= 0.01` *(in seconds) in the Matlab workspace.*
3. *Now copy the motor and controller block so that you have two motors and controllers being simulated simultaneosly. We will make one of the controllers discrete. Mux their inputs/output signals before feeding them to the scopes. This way, you can compare the performance of the discrete and continuous controllers easily.*
4. *Now change the PID controller subsystem of one of the two controllers to that shown in Fig. 1.16. You will need the block Discrete/Discrete-Filter from the SLB. Double-click the block and specify the discrete transfer-function. Enter the numerator and denominator of $G(z)$ from (1.44) in the expected format. In the field "Sample Time" enter* `Tsample`*.*

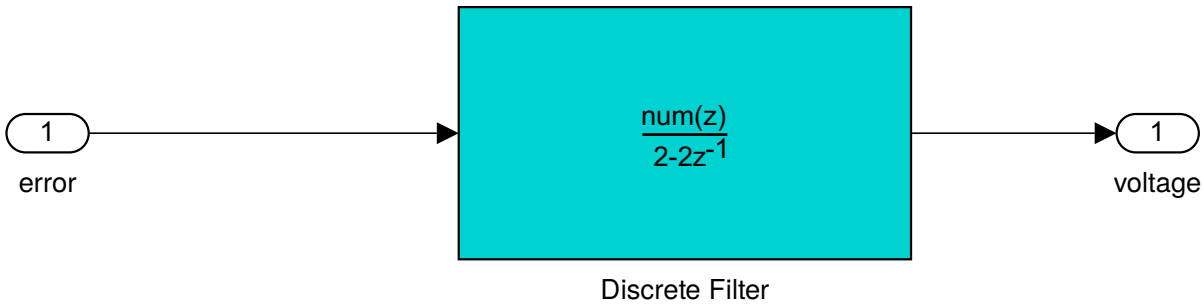


Figure 1.16: Change the PID controller subsystem to make it discrete. Take the sample-time from the workspace variable `Tsample` initialized to 0.1 seconds.

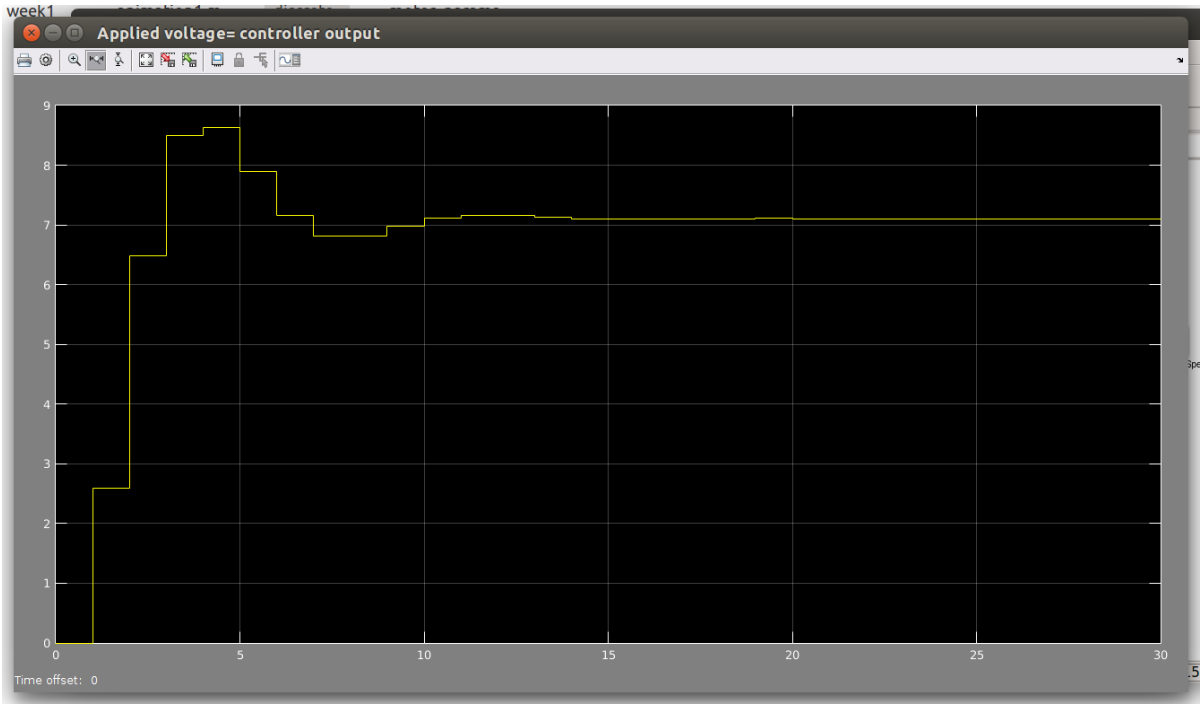


Figure 1.17: The output voltage of the discrete PID controller. `Tsample` was initialized to 1.0 seconds to magnify the steps.

5. *Now do the simulation for a step reference-speed input and look at the response. Zoom in the voltage (controller's output) scope – you will be able to see how the controller changes the voltage only at discrete intervals (refer to Fig. 1.17). What happens when you increase* `Tsample` *to* $0.5s$ *or higher? Also look at the current scope.*

1.7 Microcontroller-Implementable Discrete Controller

The discrete controller model is quite close to the actual implementation in a microcontroller, e.g. as an Arduino sketch.

- The sample time T then corresponds to how often the `loop()` function executes in our sketch.
- The current motor-speed ω_k will be read by Arduino from the encoder outputs. This is the feedback. Using the reference signal r_k, we can compute the current error $e_k = r_k - \omega_k$.

- The controller output voltage v_k will be set by the sketch every time the loop runs by setting the PWM of the output pin.
- But how will the code in the `loop()` compute v_k from e_k and possibly their past values. In other words, how do we implement the discrete transfer function $G(z)$ in (1.44). Let's see this next.

We can rewrite (1.44) as

$$G(z) = \frac{V(z)}{E(z)} = K_P + K_I \frac{T}{2}\left(\frac{1+z^{-1}}{1-z^{-1}}\right) = \frac{(2K_P + K_I T) + (K_I T - 2K_P)\, z^{-1}}{2 - 2\, z^{-1}} \tag{1.45}$$

First, we cross-multiply (1.45) and write

$$2V(z) - 2\, z^{-1} V(z) = (2K_P + K_I T)\, E(z) + (K_I T - 2K_P)\, z^{-1} E(z) \tag{1.46}$$

We now take the inverse z-transform of the above by making use of the delay relation (1.41).

$$2v_k - 2v_{k-1} = (2K_P + K_I T)\, e_k + (K_I T - 2K_P)\, e_{k-1} \tag{1.47}$$

This finally gives us the result which we can implement in code:

$$v_k = v_{k-1} + \frac{(2K_P + K_I T)}{2}\, e_k + \frac{(K_I T - 2K_P)}{2}\, e_{k-1} \tag{1.48}$$

To implement this, our program will have to remember the past values v_{k-1} and e_{k-1} from the previous iteration of loop – this can be done using global state variables. We will implement (1.48) experimentally on a real motor in Sec. 2.7.

1.8 Servo (Angle/Position) Control

In the previous sections, the goal was to control the motor-speed. Now, in the servo control mode, we want to make the motor-shaft angle $\Theta_g(s)$ rotate to a given reference angle value $R(s)$ as shown in Fig. 1.18. The feedback now is $\Theta_g(s)$ measured via an encoder. The behavior of the closed-loop system is now more interesting: it tends to be oscillatory and you need to damp the oscillations using the damping (derivative) part the PID controller.

1.8.1 Closed-Loop Transfer Function

We already derived the motor position/angle transfer function in (1.21). Analogous to the derivations in (1.32), we can derive the relevant transfer functions

$$\Theta_g(s) = \frac{H_{\theta v}(s)G(s)}{1 + H_{\theta v}(s)G(s)} R(s) - \frac{H_{\theta \ell}(s)G(s)}{1 + H_{\theta v}(s)G(s)} T_L(s) \tag{1.49a}$$

$$E(s) = \frac{1}{1 + H_{\theta v}(s)G(s)} R(s) + \frac{H_{\theta \ell}(s)G(s)}{1 + H_{\theta v}(s)G(s)} T_L(s) \tag{1.49b}$$

$$\triangleq H_{er}(s)\, R(s) + H_{e\ell}(s)\, T_L(s) \tag{1.49c}$$

Task 1.25. *Show that under the assumption of negligible L_a,*

$$H_{er}(s) = \frac{s^2 D(s)}{Q(s)}, \tag{1.49d}$$

$$Q(s) = R_a J\, s^3 + s^2(R_a b + k_e k_t + k_t K_D) + k_t K_P\, s + k_t K_I \tag{1.49e}$$

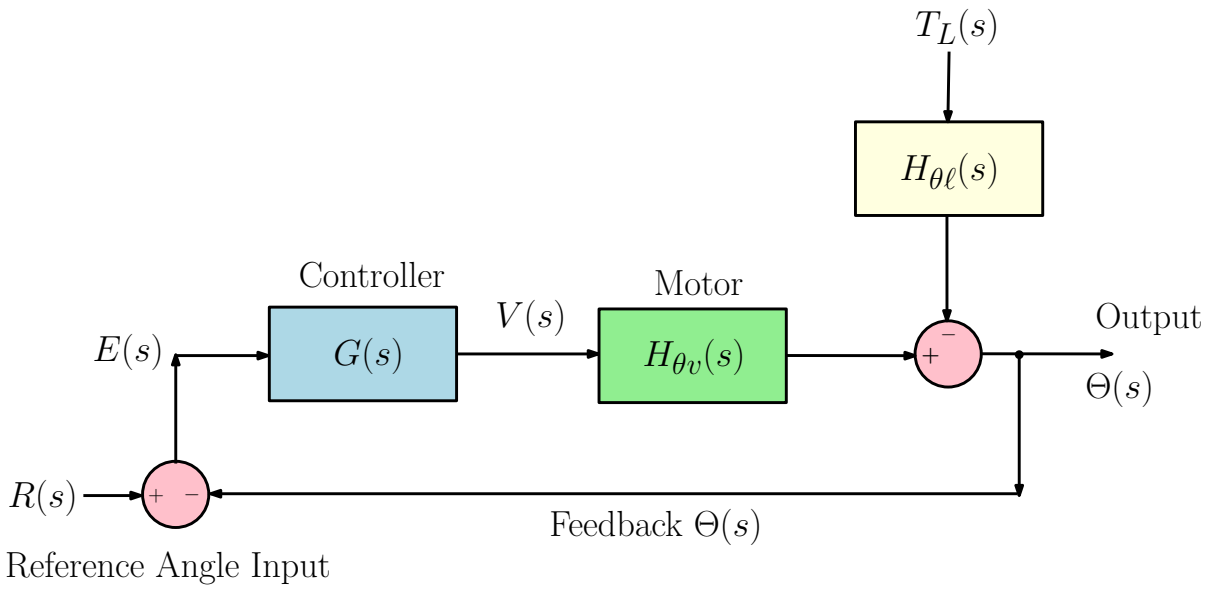


Figure 1.18: The conceptual overview of a servo controller.

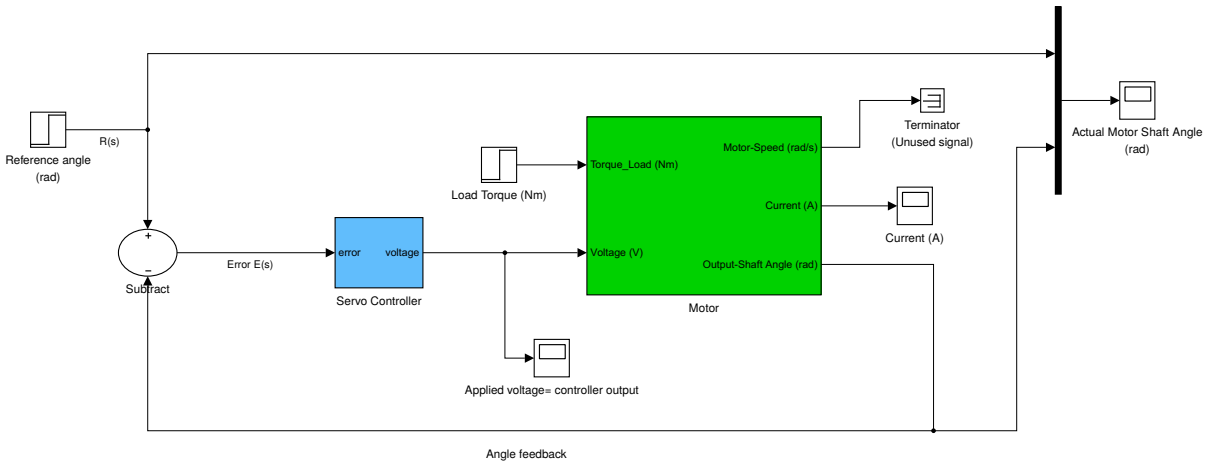


Figure 1.19: The servo controller in Matlab.

1.8.2 PID Parameter-Tuning

Task 1.26. *Copy the model-file for the PID speed-controller to a new file. Now change the speed-controller to a servo-controller as shown in Fig. 1.19.*

Task 1.27. *Proceed as follows:*

1. *Start with some relatively small value of K_P such as 1, keeping K_D and K_I zero. Set the reference input to $\pi/3$ rad and $T_L \equiv 0$. Simulate the sytem for 30 seconds. Create a table with the following columns and jot down your observations: K_P, K_I, K_D, max absolute value of current, max absolute value of voltage, max overshoot of the angle above the reference, settle-time (approx. time it takes for the angle to settle near its steady-state value).*

2. *Now add some damping by setting $K_D = 0.1$. Enter your observations in the table.*

3. *Now add some integral control $K_I = 0.1$. Enter your observations in the table.*

4. *Now we will see what the controller can do if we add a step load torque. First reset $K_I = 0$. Reduce K_P to 0.5 and K_D to 0.001. Make a step load of $T_L = 0.01$ Nm at $t = 10$ seconds. Did the system reach its reference value?*

5. *Now increase K_I to 0.1 and simulate again. Do you see any difference in the steady-state error?*

6. *Although it is now tempting to increase K_I even more, we should be careful. Making K_I too big can easily make the closed-loop system unstable. Let us see how an unstable system reacts: set $K_I = 1$ and simulate.*

7. *Does our model also predict this instability? Find the poles of the system by finding the roots of $Q(s)$ for these controller parameters. Are they in LHP?*

Chapter 2

Part 2: Implementation of a PID Controller on Arduino Mega 2560

2.1 Objective

In this lab, we will use the Arduino Mega 2560 to implement a PID controller for speed and servo control of the actual motor which we modeled in part I. We will also learn some advanced programming concepts of Arduino programming to be able to read motor-encoder inputs. Last but not the least, we will learn to use an oscilloscope.

2.2 Using an Oscilloscope

An oscilloscope is a device on which you can visualize voltage signals in time – the "scopes" which you have been using in Simulink are just the software counterpart of the oscilloscopes. Just like "scopes" you can visualize two signals together for easier visual comparison: you just connect the two voltage signals to the two channels of the oscilloscope.

We will use a Tektronix TDS 220 oscilloscope with two channels. You can find its user-manual on the lab web-page – download it at this point. Your instructor will give you an initial demo on how to do probe-compensation of the oscilloscope before starting. Another demo to familiarize you with the various buttons and dials of the oscilloscope will also be given at your group's table. If at a later time, you need to refresh your memory, please consult the user-manual.

Task 2.1. *Write a program on your Arduino Mega which creates a 50% PWM signal on one of the PWM pins. If you need to refresh your memory, look up* `analogWrite` *or refer to the manual of last semester's lab.*

- *Now visualize this pin's output on your oscilloscope's channel 1. Connect the ground of the probe to the Arduino ground. For starters, press "Autoset" to see the output. Later you can change the time and voltage scales and move the signal up/down till you are satisfied with the display.*
- *Use the horizontal cursor to manually measure one time-period T_{PWM} of the PWM signal. Take the reciprocal to find the frequency.*
- *Now press measure, change the source to channel 1 and look at all the information which is automatically computed for you: this include peak-to-peak voltage, root-mean-square (RMS) value of the signal, its frequency and many others. Does the frequency value tally with what you measured with the cursor?*
- *Move the trigger-level up and down to see its effect.*

Task 2.2. *Now generate another PWM signal with a different duty-cycle on a second pin. Connect it to channel 2. The grounds of both the channels' probes should be connected to the Arduino ground.*

- *Press "CH2 Menu" button to also show the second channel if it was off. Each channel's menu button can be used to toggle its display on or off. Press "Autoset".*
- *Using the horizontal cursors, measure the time during which the second PWM is high. Divide this by the time period T_{PWM} to obtain the duty-cycle. Does this match with what you had set in your code?*

Task 2.3. *Now we'd like to see how often the loop function is called in your Arduino code if all you do within the loop function is to toggle the state of a digital output pin.*

- *In your code (see example below), declare an* `int` *global variable for remembering the toggle state. Also, the pin number should be a global int variable as well. Write and run this code and then visualize the square-wave signal on the scope. Measure its frequency f_1.*
- *Now, change the toggle state variable's type to* `byte`*. Again, measure the frequency f_2 on the scope. How does it compare to f_1?*
- *Now, change the pin-number variable's type from* `int` *to* `const int`*. Again, measure the frequency f_3 on the scope. How does it compare to f_1 and f_2?*

```
bool flip_on= true;
int pinNr= 12;

void setup() {

  pinMode(pinNr, OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:
  if (flip_on){
    digitalWrite(pinNr, HIGH);
  }else{
    digitalWrite(pinNr, LOW);
  }
  flip_on= !flip_on;
}
```

Task 2.4. *Now we'd like to explore, if it is possible for us to speed-up the toggling of the digital output pin some more. This involves some lower-level wizardry which is conveniently encapsulated for us by the library digitalWriteFast. A short description from its web-page is:*
"This library consists of a complex header file that translates digitalWriteFast, pinModeFast, digitalReadFast into the corresponding PORT commands. It provides syntax that is as novice-friendly as the arduino's pin manipulation commands but an order of magnitude faster. It can speed things up when the pin number is known at compile time, so that digitalWrite(9,HIGH); is speeded up. On the other hand a loop with digitalWrite(i,HIGH); or a called function with the pin number as a passed argument will not be faster."

- *Download and install the library. You may have to open the library's header file and comment out the two #includes on the top to make it compile.*
- *Now change your code to use this library's pinModeFast and digitalWriteFast functions. Do not use variables as arguments of functions, e.g. instead of using digitalWriteFast(pin, toggle), use digitalWriteFast(2, HIGH). Measure the frequency f_4 on the scope. What is the speed-up compared to f_3?*

2.3 The Experiment Setup

Your supervisor will now provide you with the motor and driver setup, as shown in Fig. 2.1. Read the instructions given in the image annotations carefully!

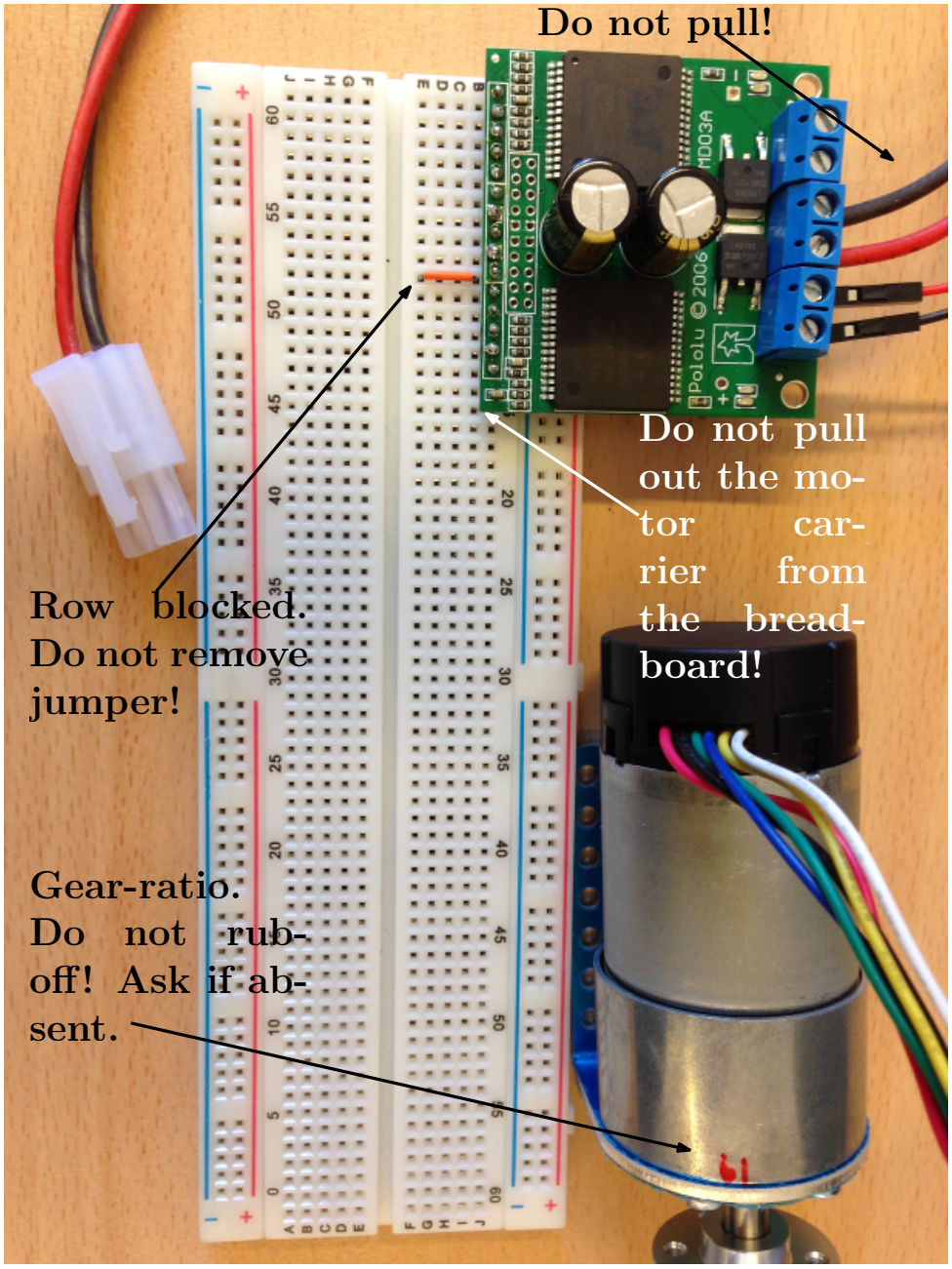


Figure 2.1: The setup.

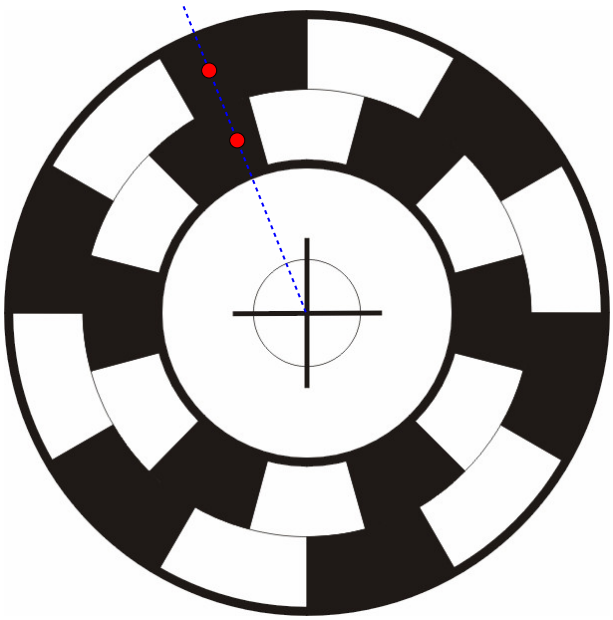


Figure 2.2: Conceptual explanation of a quadrature rotary encoder. Reworked base image from this source. The red-spots represent two sensors which can detect if a white or black patch is over them. In optical encoders, these could be photo-sensors.

2.4 Reading Quadrature Encoders Using Arduino Interrupts

In this section, we aim to understand what quadrature encoders are and how we can read them from our Arduino code.

2.4.1 Rotary Quadrature Encoders

Quadrature encoders are mounted on the motor-shaft and create a series of pulses when the shaft rotates. The frequency of the pulses is proportional to the rotation-speed. By counting the pulses, we can also keep track of the angle by which the shaft rotated thus far since the counting Arduino program started up. More precisely, such encoders are called *incremental* encoders; there are also *absolute* encoders which give the angle directly. However, incremental encoders are more common, and so we will work with them exclusively. In our motor, they are hidden under the black cap: Refer to Fig. 1.7. *The encoders provide the all important feedback signal which our controller needs to take the motor to the desired speed or angle.*

The concept behind a quadrature rotary encoder can be best understood from Fig. 2.2. This disk with the two patten tracks is attached to the motor-shaft and rotates with it. Let's call the outer track, track A and the inner one track B. The two red sensors on the two tracks can detect whether a dark or a bright patch/pulse is over them. Let's assume: if a dark patch is above a sensor, it returns 0 V, and when a bright patch is above the sensor, it returns 5 V. So, the sensors return a train of on/off pulses as the disk rotates with the shaft. Let's consider two cases:

- **The shaft rotates clockwise**: If at this time-instant the situation is as shown in Fig. 2.2 while the shaft rotates CW, which sensor (A or B) will turn on (output changing from 0 to 5V) first? Can you see that it will be sensor A?
- **The shaft rotates counter-clockwise**: If at this time-instant the situation is as shown in Fig. 2.2 while the shaft rotates CCW, which sensor (A or B) will turn on (output changing from 0 to 5V) first? Can you see that it will be sensor B?

As shown in Fig. 2.3, in both cases, the pulse-trains A and B have a phase angle of 90° between them. If we consider a state-variable consisting of the tuple (sensor-A, sensor-B), then it can be in the following 4 (hence the name "quadrature") unique states: (HIGH, HIGH), (HIGH, LOW), (LOW, HIGH), and (LOW, LOW). As usual, LOW represents 0V and HIGH, 5V.

Task 2.5. *For both clockwise and counterclockwise motions list the states in the right order in which they occur, starting from (LOW, LOW).*

2.4.2 Arduino Hardware Interrupts

Within your application, if you need to wait for some event to occur, one way to do it would be to actively check (i.e. poll) some input pin inside the loop function. This is wasteful and if the event occurred really fast, and you are doing other time-consuming things inside the loop function, you may miss the event entirely.

A better option is to use the hardware interrupt mechanism. This allows you to call a function of your choice when an event occurs on the input pin. This function is called an Interrupt Service Routine (ISR) and should not have any input arguments or return values (they should be void). If an event occurs, the execution interrupts whatever it was doing (e.g. it could be doing something in the loop function) and jumps into the associated ISR. Once the ISR is done, the execution resumes whatever it was doing earlier. ISRs typically should return fast and hence should not do anything time-consuming.

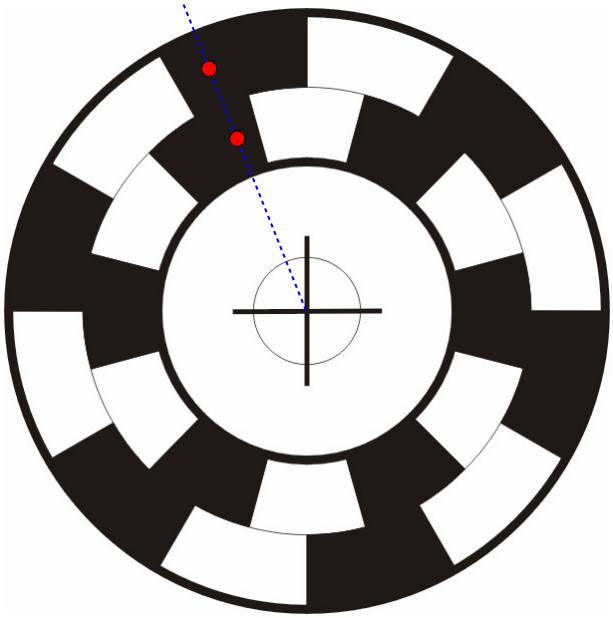
So, which events can you respond to on an input pin? You can associate an ISR on the following events:

1. The voltage on the input pin changes from low to high. This voltage transition is called a RISING edge event.

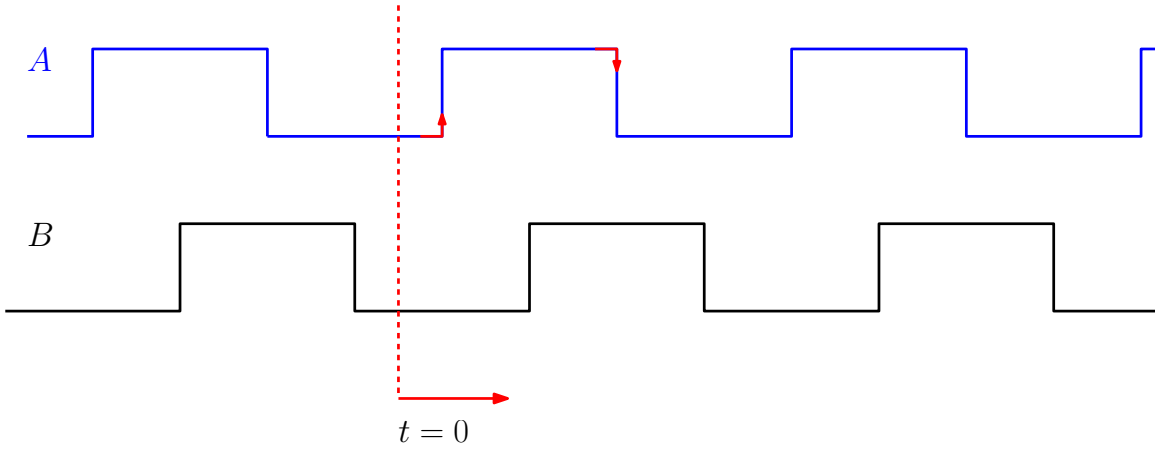
2. The voltage on the input pin changes from high to low. This voltage transition is called a FALLING edge event.

3. BOTH of the above.

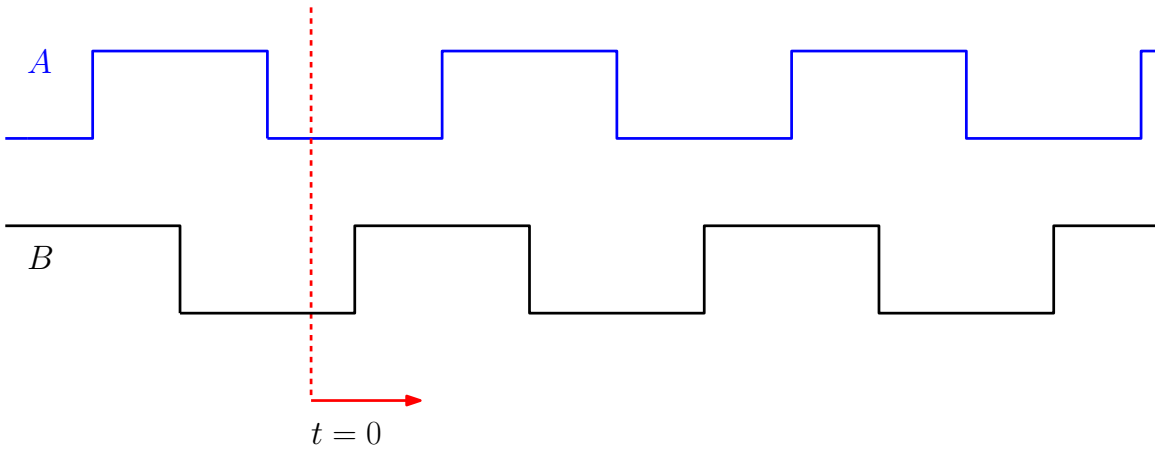
So far so good. However, you cannot use just any input pin for interrupts. Different Arduino boards have different specific pins which only can be used for interrupts on them. Additionally, there is also an interrupt number associated with each pin which is not the same as the pin number. The best way to understand this is to look at an example: Refer to code-listing 2.1. You register your interrupt in the setup function.



(a)



(b) The pulse-trains reported by the sensors for clockwise motion. The interrupts are triggered when at the time-instants when the signal starts rising or falling. These are shown by red-arrows on the signal. The dashed red line marks the situation shown in Fig. 2.3(a) at $t = 0$ when both sensors report 0 V.



(c) The pulse-trains reported by the sensors for counter-clockwise motion. The red line marks the situation shown in Fig. 2.3(a) at $t = 0$ when both sensors report 0 V.

Figure 2.3: The pulse-trains sampled by the sensors on tracks A and B.

```
const int pin_enc_A_rising = 21;
// On Mega, interrupt 2 is on pin 21
const int interrupt_nr_pin_enc_A_rising = 2;

const int pin_enc_A_falling = 19;
// On Mega, interrupt 4 is on pin 19
const int interrupt_nr_pin_enc_A_falling = 4;

// These variables will be updated in interrupts
volatile long enc_counter= 0;
volatile bool enc_A_high= false;
volatile bool enc_B_high= false;

void setup() {
  pinMode(pin_enc_A_rising, INPUT);
  pinMode(pin_enc_A_falling, INPUT);

  attachInterrupt(interrupt_nr_pin_enc_A_rising, handleEncA_Rising, RISING);
  attachInterrupt(interrupt_nr_pin_enc_A_falling, handleEncA_Falling, FALLING);
}

void handleEncA_Rising(){
  enc_A_high= true;
  if (enc_B_high){
    enc_counter -= 1;
  }else{
    enc_counter += 1;
  }
}

void handleEncA_Falling(){
  // Do something in this ISR, e.g. update enc_counter
}
```

Listing 2.1: Using Interrupts.

A couple of restrictions have to be kept in mind while using interrupts.

- If you intend to change a global variable in an ISR, it should be defined `volatile`, as in lines 10-12 of the code-listing 2.1. This tells the compiler that it should not optimize the variable by caching it. The variable should always be kept up-to-date since an interrupt can occur anytime.
- You cannot have two interrupts (say, one on FALLING and the other on RISING) on the same pin. If you need to do it, just duplicate the signal on another pin and attach the other interrupt there. In the case of the motor encoder, this signal could be a pulse-train shown in Fig. 2.3.
- On Arduino Uno, you can only use two pins for interrupts; On Mega 2560, you can use 6 pins. On Arduino Due, you can use all pins for interrupts. The mapping between pin numbers and interrupt numbers is shown in the tables below, where the D before the pin number just stands for "digital".

Uno Interrupt Nr.	Uno Pin Nr.
0	D2
1	D3

Mega Interrupt Nr.	Mega Pin Nr.
0	D2
1	D3
2	D21
3	D20
4	D19
5	D18

- While an ISR is being serviced, other interrupts are turned off. This prevents interrupts from interrupting each other. However, some common functions which use timers and interrupts internally will not work anymore: these include `delay()`, `millis()`, and serial communications. Instead, use `micros()` and `delayMicroseconds()`. Do not expect `Serial.print()` to work reliably inside ISRs.
- If you need to read/write a pin inside an ISR, try using the library digitalWriteFast introduced in Task 2.4. This will reduce the time spent in the ISR.

2.4.3 Angle of Rotation Determination Using 4 Interrupts

We are now ready to write our first interrupts to read the motor-encoder pulses shown in Fig. 2.3. This will allow us to find out how much the motor rotated between any two time instants and later, we can also compute the motor speed. For testing this code, we do not really need to power on the motor: we will just provide power to the encoder and move the motor output shaft by hand. This will also allow us to easily test if our code is working as it should – if we rotate the shaft by approximately 90°, this is what our program should report.

Since Mega 2560 allows us to use 6 interrupts, let us go all out and use two interrupts (RISING and FALLING) for channel A and another two (RISING and FALLING) for channel B of the encoder (Fig. 2.3). This allows us to have the maximum resolution possible: 64 counts per revolution (CPR) of the motor-shaft, if we count all the four events. Although this resolution might sound low, realize that this is for the motor-shaft, not the output-shaft of the gear-box. If the gear-ratio is 18.75, we effectively get a resolution of $18.75 \times 64 = 1200$ CPR for the output-shaft. This corresponds to 0.3° rotation of the output-shaft per count.

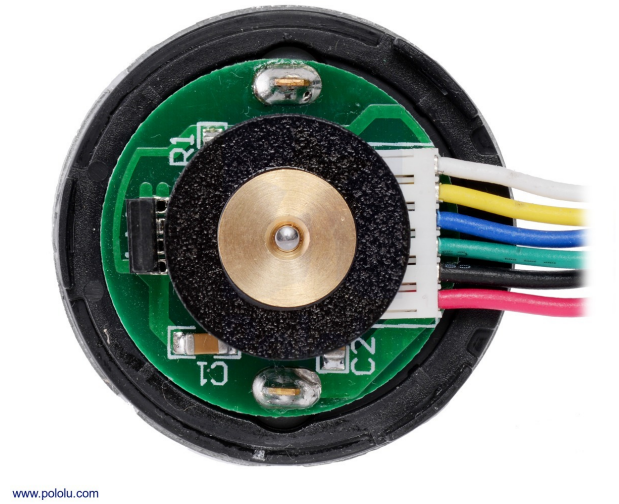


Figure 2.4: Color-coded ribbon for the encoder connections. Source link.

- The basic idea of the code is that within the four ISRs you increment or decrement a global counter based on whether you detect the motor to be rotating clockwise (CW) or counter-clockwise (CCW): Look at lines 23-27 in code-lisiting 2.1. **Note**: The best way to understand it is to look at the encoder pulse-train diagram of Fig. 2.3 and consider what the state of encoder B is at the exact moment when the encoder A rises or falls (i.e. at the very time-instants the interrupt is called) and vice versa.
- Also, since we are now using four interrupts, do not forget to attach two more interrupts after line 19.

Build the Circuit

The motor/encoder has six color-coded, 11" (28 cm) leads terminated by a 1x6 female header with a 0.1" pitch (Fig. 2.4). The Hall-sensor in the encoder draws a maximum of 10 mA, so it is safe to connect it to the 5V pin of the Arduino. Make the following connections.

Color	Function	Connect To
Red	Motor Power	Leave unconnected
Black	Motor Power	Leave unconnected
Green	Encoder GND	Arduino GND
Blue	Encoder Vcc (3.5-20V)	Arduino 5V pin
Yellow	Encoder A output	Any two of the 6 interrupt-capable pins on Mega
White	Encoder B output	Any two interrupt-capable pins different from above

Task 2.6. *Get your circuit checked by your supervisor.*

Write the Code

You should base your code on the code-listing 2.1. You need to register for two more interrupts in the setup function for encoder channel B.

Task 2.7. *Proceed as follows.*

1. *Based on the code of* `handleEncA_Rising()` *fill in the code of the remaining three ISRs.*

2. *In the* `loop()` *function, compute the number of rotations made since the start by dividing the counter by the gearbox CPR. All motors have the same motor-shaft CPR of 64, but the gear-ratio could be different. Motors marked 19 have gear-ratio 18.75, those marked 50 have gear-ratio 50, and those marked 131 have a gear-ratio of 131.25. From these numbers you can compute the gearbox CPR. On the serial monitor, print the number of rotations thus calculated (e.g. 0.25 for 90°).*

3. *Now rotate the gearbox shaft by hand by rotating the wheel mounting hub and read the numbers printed on the serial monitor. Do they make sense? If not, you need to debug your code.*

2.4.4 Angle of Rotation Using 2 Interrupts

If we want to use Mega 2560 to run a differentially driven robot, we would need 4 interrupt pins *each* for the left and right motors. Since the Mega only has 6 interrupts, we would be in a quandary. One way out would be to be frugal and make do with 2 interrupts per motor. The motor-shaft resolution will drop from 64 CPR to 32 CPR, but this could be acceptable, e.g. for a 18.75 gear-ratio, we still get 0.6° per count.

Task 2.8. *Proceed as follows.*

1. *Start working in a copy of your previous sketch.*

2. *Decide which two interrupts out of the four you can sacrifice.*

3. *In the remaining two ISRs, you now need to read the status of one of the channel pins. Use* `digitalReadFast` *to keep things fast.*

4. *Again, test your code by rotating the wheel mounting hub manually.*

2.4.5 Angle of Rotation Using 1 Interrupt

What if we only had an Uno for connecting the two motors of your DDR? We are now down to only two available interrupts. We cut corners even more, and make do with 1 interrupt per motor. The motor-shaft resolution will drop from 64 CPR to 16 CPR, but this could be acceptable, e.g. for a 18.75 gear-ratio, we still get 1.2° per count.

Task 2.9. *Proceed as follows.*

1. *Start working in a copy of your previous sketch.*

2. *In the only remaining ISR, you will still need to use* `digitalReadFast` *to sample the other channel.*

3. *Again, test your code by rotating the wheel mounting hub manually.*

2.4.6 Rotational Speed Using 2 Interrupts

Since we do have a Mega 2560, we do not want to be too frugal, so let us roll-back to the 2 interrupts case of Sec. 2.4.4. Our aim is now to compute the rotational speed of the motor. We do this every $T_s = 0.1$ seconds. In the loop function, approximately every $\Delta t \approx T_s$ seconds, we subtract the encoder count before Δt from the encoder-count now, and divide the result by Δt to get an average counts per seconds, which we can convert to rotations per seconds or radians per second by applying appropriate factors.

Task 2.10. *Proceed as follows.*

1. *Implement the above mentioned strategy in your code from Sec. 2.4.4. Avoid using* `delay` *and* `millis`*. Instead use* `micros` *to get the number of microseconds since the code started running. This is a number of type* `unsigned long` *and will overflow every 70 minutes. Compute the speed only if the elapsed time since the last computation is more than T_s. Use the actual elapsed time Δt, not T_s to compute the speed. Needless to say, you need to cache the last recorded encoder count in a global variable.*

2. *Print the computer angular-speed in RPM and rad/s every time it is refreshed. Again, do a sanity-check of your code by rotating the wheel mounting hub manually. Does the sign change when the rotation direction changes? Does the value remain zero when no rotation takes place?*

2.5 The Motor Driver Carrier VNH2SP30

It is now finally time to connect our motor directly to power. As you know, the Arduino board cannot by itself provide enough power to a typical motor – we need to use an external motor-driver board with its own high-power input supply which is output to the motor. The Arduino then controls this board using its low-power digital and PWM pins. One of the main tasks of the driver board is to be able to run the motor in both directions using a circuit called the H-bridge. Improper use of an H-bridge can lead to a short-circuit, so usually professionally designed boards are used.

For running less powerful motors using Arduino, it is common to use the motor-shield which fits on top of the Arduino. However, this shield can only provide a maximum of 2A per channel and hence is not sufficient for our motors which have a stall current of 5A.

Hence, we decided to use the Pololu Dual VNH2SP30 Motor Driver Carrier MD03A (Fig. 2.5) with the maximum current rating of 30A and a current for infinite time of 14A. Additionally, it has a current-sensing (CS) capability and one can use `analogRead` to read a voltage proportional to the current. This allows us to put in safety checks: If the current is near 50% of the stall current for a few seconds, we can switch off the motor to avoid any damage to it.

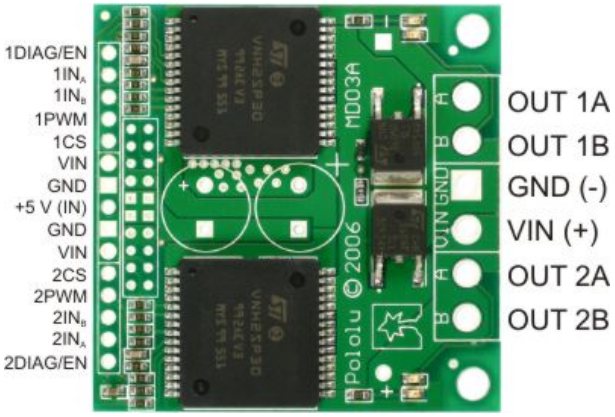


Figure 2.5: The VNH2SP30 motor driver carrier: Source link. On the left side, we have the control pins which connect to the Arduino and on the right there are the input power pins connecting to an AC adapter or battery and the output power pins connecting to the motors. Two motors can be controlled by the same board.

Referring to Fig. 2.5, a breakaway male header strip has been soldered to the control connections on the left side. It has already been put on the bread-board. **Do not pull** the motor-driver out from the bread-board, even after the experiment. Depending on the direction of the strip, this may cause the labels of the pins to be hidden. However, the labels are symmetric from the two ends: They correspond to the two motors you can connect to the board. You can get the hidden labels either from Fig. 2.5 or from the table below.

Nr.	Label	Function	Connect To
1	1DIAG/EN	Diagnostic pin of motor M1	Leave unused or read as an input. In case of error conditions, it would read low, else high.
2	1IN-A	Controls the direction of M1	Set as digital output. See truth-table below.
3	1IN-B	Controls the direction of M1	Set as digital output. See truth-table below.
4	1PWM	Controls the speed of M1	A PWM pin of the Arduino.
5	1CS	Current-sensing of M1	An analog input of the Arduino.
6	VIN	Repeated input 12V from the other side.	**Leave unconnected.** This row of the bread-board is already blocked with a jumper.
7	GND	GND	GND of the Arduino.
8	+5 V (IN)	Input of the board's logic circuit	5V pin of the Arduino. **Don't forget!**
7	GND	GND	Leave unconnected, if using only one motor.
6	VIN	Repeated input 12V from the other side.	**Leave unconnected.**
5	2CS	Current-sensing of M2	Leave unconnected, if using only one motor.
4	2PWM	Controls the speed of M2	Leave unconnected, if using only one motor.
3	2IN-B	Controls the direction of M2	Leave unconnected, if using only one motor.
2	2IN-A	Controls the direction of M2	Leave unconnected, if using only one motor.
1	2DIAG/EN	Diagnostic pin of motor M2	Leave unconnected, if using only one motor.

Warning: On the right hand side of Fig. 2.5, the pins VIN(+) and GND(-) should be connected to the right corresponding polarity of the input voltage from the power-adapter or battery. This is because, for the attached capacitors, the polarity is important. Connecting the wrong polarity will damage them. These connections have already been made for you.

Warning: Do not connect any of the power pins on the right-hand side of Fig. 2.5 to the oscilloscope. Use the oscilloscope only for the left-side control pins which are connected to the Arduino.

Warning: Keep your hands away from any naked metal surfaces. While measuring any voltages using the multimeter or the oscilloscope, make sure that the two leads are not dangling

and accidentally touch each other to cause a short-circuit. In case of sparks or smell of burning plastic, disconnect the Arduino USB cable and the battery immediately.

Note: Referring to Fig. 2.5, the outputs OUT 1A and 1B go to motor-1, and the outputs OUT 2A and 2B go to motor-2. The top and bottom set of input pins correspond to motors-1 and 2 respectively. One pair out of OUT 1A/B and OUT 2A/B has already been screwed to cables going to the motor — hence on the input side, attach your jumpers only to the corresponding pins.

Task 2.11. *Build the circuit by making the connections as shown in the table above and get your circuit checked by your supervisor.*

The direction of the motor i.e. the OUT-A and OUT-B pins on the right hand side of Fig. 2.5 can be set by the following truth-table.

IN-A	IN-B	OUT-A	OUT-B	CS	Operation Mode
HIGH	HIGH	HIGH	HIGH	High impedance	Brake to Vcc
HIGH	LOW	HIGH	LOW	Output voltage = 0.13 Current	Clockwise (CW) motion
LOW	HIGH	LOW	HIGH	Output voltage = 0.13 Current	Counter-clockwise (CCW) motion
LOW	LOW	LOW	LOW	High impedance	Brake to GND

Task 2.12. *Now modify your code from Sec. 2.4.6 to turn on the motor and run it at different speeds in both directions by modifying the digital output on pins IN-A and IN-B according to the truth-table above. The PWM pin should be supplied a PWM (0-255) from an Arduino PWM pin. The best place to do this is in the* `setup` *function after setting up the interrupts. Get your code checked by the supervisor. Your supervisor will now give you a 12V battery and will help you make your first run.*

A PWM of 255 will run the motor at full speed. Make a table of rotation-speed vs. PWM for PWM values varying between 255 and 0 in steps of 25, and plot these values. You will note that this relation is not totally linear.

Task 2.13. *Look at the encoder connections table from way back in Sec. 2.4.3. Connect the channels A and B of the encoder (* ***Warning:*** ***not*** *the motor-driver) to the two channels of the oscilloscope. Get your connections checked by your supervisor before turning on the power. On running the motor, are you able to see a display similar to Figs. 2.3 and 2.6?*

Task 2.14. *In the* `loop` *function, we are computing the angular-speed. Now, in addition, we'd like to read the CS pin using* `analogRead` *and convert the read voltage into a current value in Amperes (see the formula in the truth-table in the CS column). What is the range of values of voltages that* `analogRead` *will map to the range 0-1023, if the motor-current is expected to vary between 0 and 5A? Read the documentation of analogReference to find out how you can increase the resolution of this reading by using the INTERNAL1V1 option. Explain your reasoning to your supervisor. Also print the current value in Amperes within your* `loop` *function. You can even visualize the output on the CS pin on the oscilloscope to see its fast cyclical variation.*

2.6 Determination Of Rotor Moment of Intertia

This is an optional task. In case of time-constraints, skip ahead to Sec. 2.7 on PID speed control.

Task 2.15. *(Bonus) Proceed as follows:*

1. *Within* `loop` *print the current time along with the speed.*

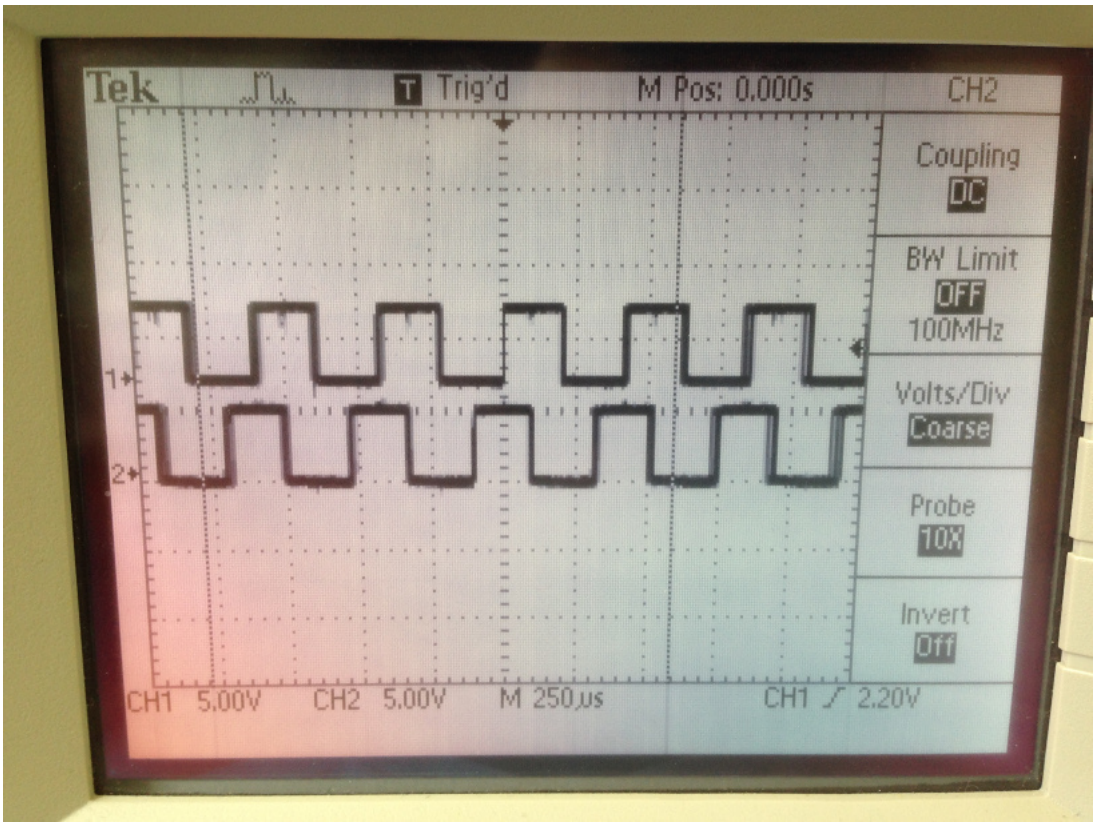


Figure 2.6: The two encoder signals visualized on an oscilloscope while the motor is rotating at a constant speed.

2. *Apply a PWM of 255 and from the Serial monitor cut and paste these two columns in a script in Matlab and plot it. If any initial readings are missing, assume that the curve starts at* $t = 0$, $\omega = 0$.

3. *From the 63% rule, use* (1.29) *to estimate the motor moment of inertia. For groups with motors marked 50 or 131, the motor parameter values such as* k_t, k_e, R, b *will have to be re-estimated since we only estimated them for the motor marked 19. Use the no-load and stall data from the links: motor with gear-ratio 50; motor with gear-ratio 131.*

2.7 PID Speed Controller

We are now ready to implement the PID speed-controller which we designed and simulated in Sec. 1.5. For implementing on Arduino, you will need to use the expression we derived in Sec. 1.7, viz. (1.48).

At the beginning of your code include the following.

```
float rpm_ref= 200.0; // The reference RPM value

long Tsample_us= 100000; // micro-seconds
float Tsample_s= float(Tsample_us)*1e-6; // seconds

// PI speed controller parameters: They assume that the error is in rad/sec not RPM
float Kp= 0.0084;
float Ki= 0.15;

```

```
bool first_controller_output= true;
float e_km1= 0.0; // The error in rad/s from last sample instant (k minus 1)
float v_km1= 0.0; // The voltage set in the last sample instant (k minus 1)
```

Task 2.16. *Now write code for speed controller which takes the reference RPM, the current RPM and returns the voltage v_k. Keep all error computations e_k and e_{k-1} in rad/s (not in RPMs) to be able to use our previous values of K_P and K_I. The sample time T in (1.48) should be taken as* `Tsample_s`.

```
float speed_controller(float reference_RPM, float actual_RPM)
```

Task 2.17. *Now call* `speed_controller` *at the right place in* `loop`. *You should call it only every* `Tsample_s` *seconds. The output voltage $|v_k|$ should be converted to PWM by scaling it with* $255.0/12.0$ *(ignore the nonlinear relationship for now – the PID controller will automatically compensate for this). If this value lies outside the range* $[0, 255]$ *(this can only happen if your reference speed is more than the no-load speed or if something else is wrong), threshold it to the range* $[0, 255]$. *Then you should apply this PWM to the PWM pin connected to the driver board. Remember to change the direction of the motor using IN-A and IN-B if the voltage v_k is negative.*

Task 2.18. *Ask your supervisor to check your code. Running your code now should print the reference RPM and the actual RPM in* `loop`. *If everything was correct, you should see that the speed controller works astonishingly well and the actual RPM converges to the reference RPM quite fast. Change the reference RPM* `rpm_ref` *a few times (also negative values) and retry. Of course, the reference value should be below the no-load speed of the motor for it to work.*

Task 2.19. *Ask your supervisor to show you the effect of applying a disturbance torque $T_L(t)$. Your supervisor will try to lightly squeeze the motor wheel mounting hub with his/her fingers to try to reduce its speed by about 10%* ***without completely stopping (stalling)*** *it. The disturbance torque applied by the fingers is the load-torque $T_L(t)$ on the motor which you modeled earlier in Simulink. After an initial reduction of speed below the reference speed, is the controller able to converge the speed back to the reference-speed by modifying v_k? Now the superviser will let go of the hub. After an initial phase of above-reference speed, is the controller able to to converge the speed back to the reference-speed again? This vividly shows the effectiveness of the PID speed controller.*

2.8 PID Servo/Angle Controller

We are now ready to implement the servo (angle) controller from Sec. 1.8. **Note:** A DC motor with a servo controller is not exactly the same as a stepper motor, although both are used to rotate the shaft to the specified angle. The internals of a stepper motor are more complex and will be covered in the IMS "Automation" course.

Task 2.20. *Proceed as follows.*

1. *Reduce the sampling time.*

```
long Tsample_us= 50000; // in micro-seconds
```

2. *Use PID controller function as shown in code-listing 2.2. This is now a second order system since we're also using the D part of the PID control, hence, we need to remember two past values of the voltage and the angular error.*

```
1   float servo_controller(float reference_rotation, float actual_rotation){
2
3   float e_k= (reference_rotation - actual_rotation)*2*PI;
4   // convert rotations to radians
5
6   if (first_controller_output){
7     first_controller_output= false;
8     e_km2= e_km1= e_k;
9     v_km2= v_km1= 0.0;
10  }
11
12  float v_k= 0.5714*v_km1 + 0.4286*v_km2 +
13             1.935*e_k - 3.134*e_km1 + 1.217*e_km2;
14
15  v_km2= v_km1;
16  v_km1= v_k;
17  e_km2= e_km1;
18  e_km1= e_k;
19
20  return v_k;
21
22 }
```

Listing 2.2: Servo controller. The expression in line 12 corresponds to certain values of K_P, K_I, and K_D.

3. *Call this function appropriately from* `loop`. *Does the motor rotate to the right angle? If it behaves unexpectedly, quickly switch off the motor-power (red switch above the table).*

A geared motor only turns when the PWM is above a certain threshold (for gear-ratio 19, this is PWM 15) – this is called a dead-zone. Hence when the servo controller is produces small voltages, the motor does not turn at all and stops before reaching the reference angle. Interestingly, this causes the integral of the error to build up and the integral (I) part of the PID controller kicks in after some time and increases the output voltage to the point that the motor starts moving again. In my experiments, the controller was able to converge the shaft to within $\pm 3^\circ$ of the reference value, though it moves choppily due to the dead-zone. However, this experiment clearly shows how the integral part of the PID controller "winds up" if the error persists due to a dead-zone or due to output saturation. In our case, it helped us, but it may not always be desired. To avoid this winding up, "anti-windup" techniques are employed, but they are beyond the scope of this lab.

Chapter 3

Part 3 (Bonus): The Differential-Drive Robot Model

Now that we've looked at an individual DC motor and its controllers in detail, it is time to use two of these motors to make a differentially-driven mobile robot that we studied in GIMS-I. In particular, we will implement the homework-10 problem from that course. Make a copy of your discrete speed-controller model from Sec. 1.6: We will extend it in this section.

Task 3.1. *Implement the overall system shown in Fig. 3.1: Its individual blocks are shown in detail in Figs. 3.2, 3.3, and 3.4. The "Reference Time Trajectory" subsystem generates the reference heading and turning speeds from GIMS-I/homework-10 problem. The robot then tries to follow this using its motor speed-controllers despite the disturbance torque-load acting on the wheels. The plot of the XY motion of the robot and the maximum current flowing through either of its motors is available in the scopes.*

- *In the "Reference Time Trajectory" subsystem, we use two User-Defined Functions/Fcn blocks to compute the reference v and $\dot{\theta}$ from the given parameters. See the solution of the homework-10 problem.*

$$\dot{\theta} = \frac{b\,a\,\alpha}{a^2\cos^2(\alpha t) + b^2\sin^2(\alpha t)} \tag{3.1}$$

$$v = \alpha\sqrt{a^2\cos^2(\alpha t) + b^2\sin^2(\alpha t)} \tag{3.2}$$

  In Matlab add the suffix `_ref` *to variables $R = 0.15$ (wheel radius), $L = 0.3$ (distance between the wheels), $a = 3$, $b = 9$, $\alpha = \pi/10$ so that they do not clash with the motor-parameters.*

- *Set the limits in the XY-Graph block to ± 20 m.*

- *In the "Load-Torque Band-Limited White Noise" block, set the "Noise-Power" to* 0.001 *with "Sample Time"* 0.5.

- *Simulate the system for $t_f = 2\pi/\alpha$ seconds. Does the robot follow the reference trajectory? Plot the two XY trajectories together by piping the relevant signals to the workspace using Sinks/To Workspace and then using the usual Matlab plotting commands.*

- *At t_f what is the residual error to the start position? Also look at the current scope: Are values within the feasible range for the motor?*

- *You may be tempted to increase α to speed up the reference trajectory. What happens if you do this? What seems to be causing this?*

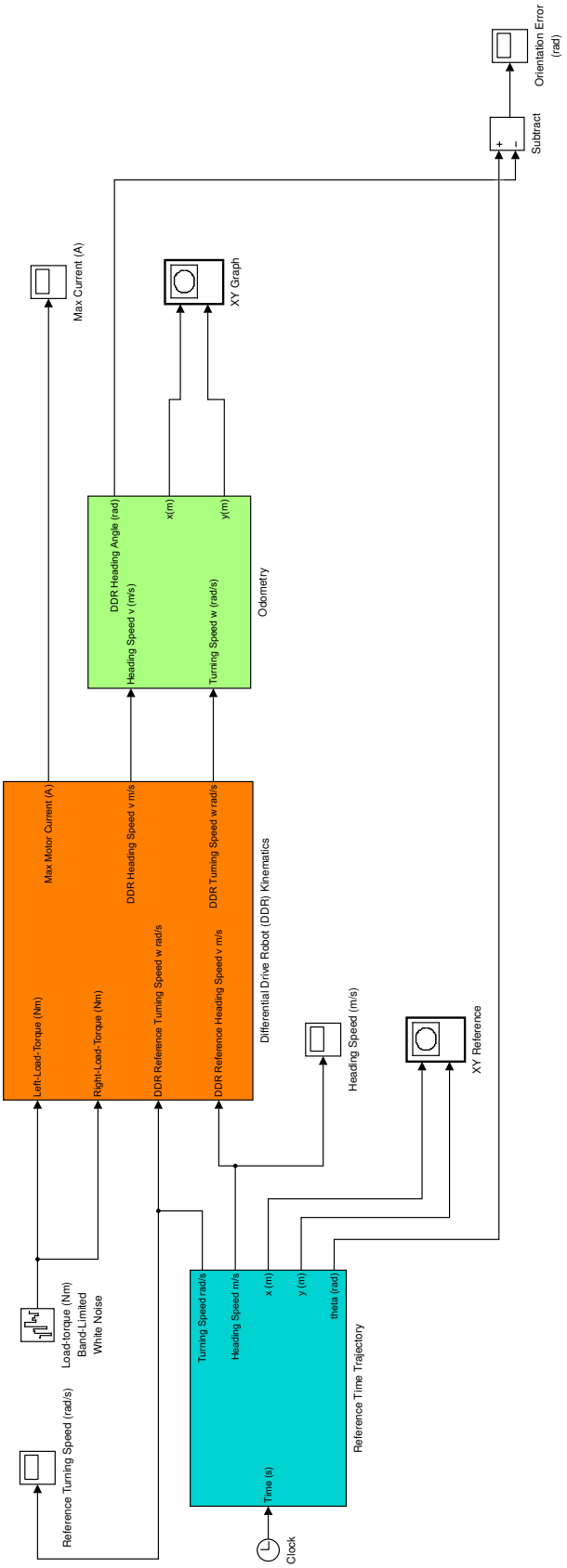


Figure 3.1: The overview of the differential-drive mobile robot. The individual subsytems/blocks are shown in the following figures.

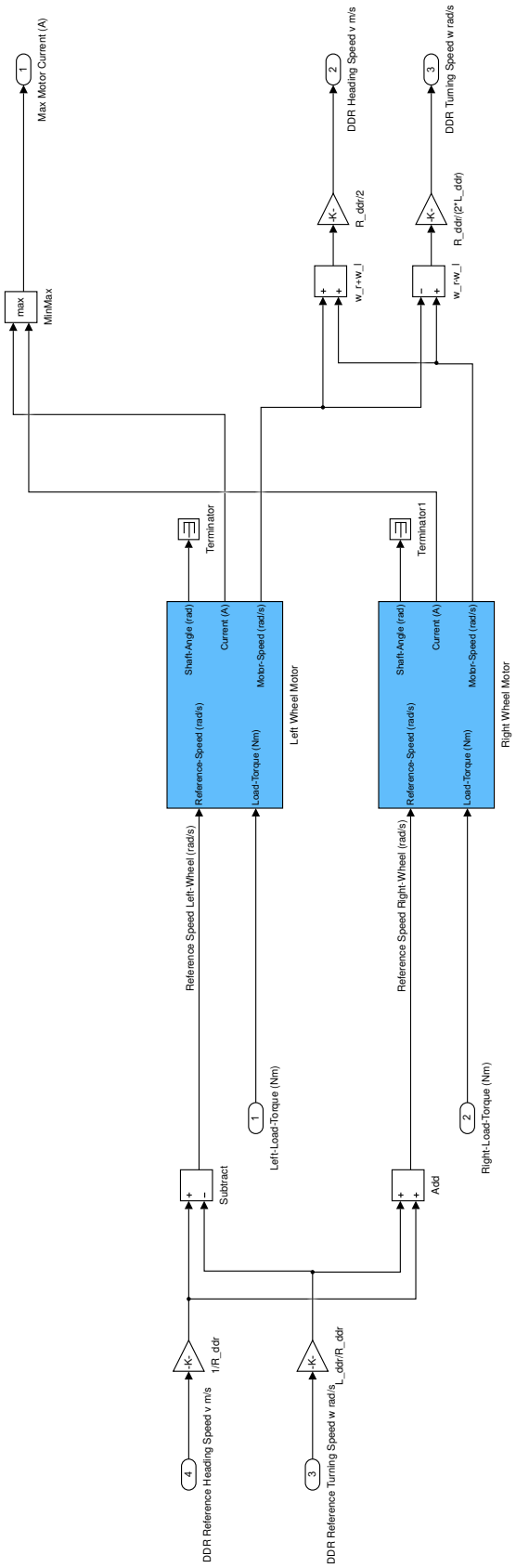


Figure 3.2: The kinematics subsystem of the differential-drive robot consisting of two motored wheels. Reference heading and turning speeds are specified; the subsytem outputs the actual heading and turning speeds at any point of time.

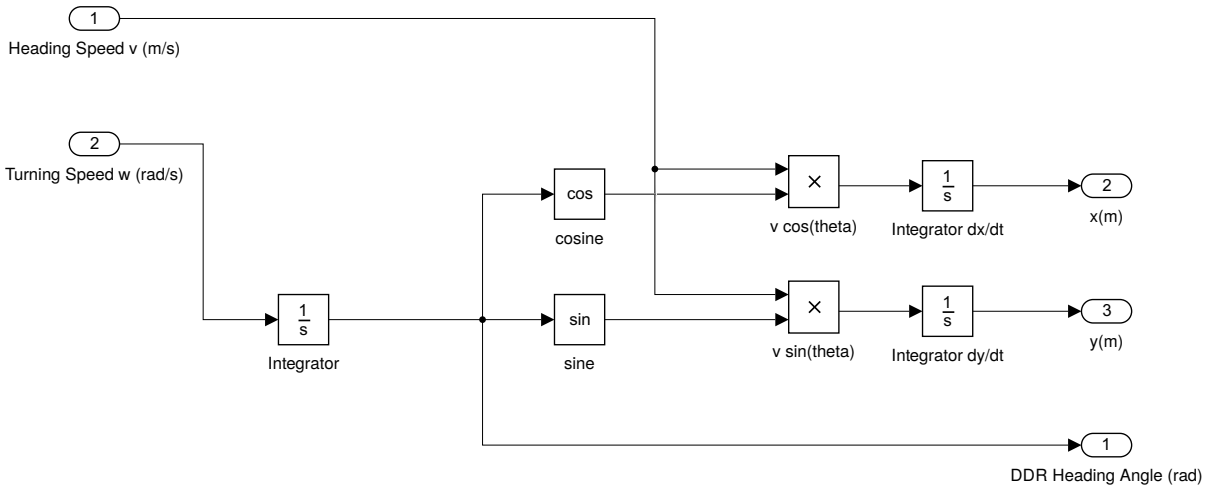


Figure 3.3: The odometry subsytem which implements the kinematics equations that we studied in GIMS-I.

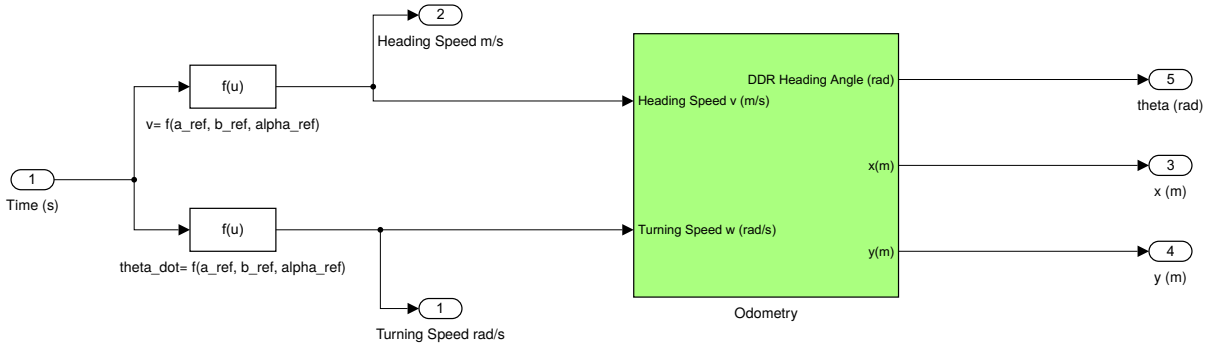


Figure 3.4: The subsytem which generates the reference-trajectory for the robot. This is based on the the homework-10 problem of GIMS-I. Here we reuse the "Odometry" block from Fig. 3.3: You can use the usual keyboard shortcuts Ctrl+C and Ctrl+V to copy and paste this block.

3.1 Way-Point Following

During exploration, a mobile robot creates a map of the environment using its sensors. In its most basic form, a map is essentially a representation of where the obstacles are, and where the free-space is, in which the robot can move without collision.

A path-planning algorithm then runs on this map and creates a collision-free path to the next goal of the robot. This path is usually a set of way-points: assume that the robot is at (x_0, y_0) and the goal point is (x_n, y_n). The path-planning algorithm will create a list of way-points: $(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$ which, when followed successively be the robot, will safely guide it around the obstacles to the goal-point.

In this project, we'll assume that $(x_0 = 0, y_0 = 0)$ and the robot needs to go to the next given way-point (x_1, y_1). Let's assume a differentially-driven robot which can only move forward in the direction θ it is pointing to, although it can turn on a dime (in place).

A simple path-following controller can be as follows:

1. Given (x_1, y_1), it will first turn the robot in-place at $(0, 0)$ such that it is pointing towards (x_1, y_1).

2. Then it simply drives forward till (x_1, y_1) is reached.

You can see that doing this for each pair of way-points, the robot can traverse the planned path. Here the time-trajectory is not important, only that the robot traverses the path sufficiently accurately. Of course, typically we want the robot to move as fast as possible while not exceeding its limits (such as current-limits etc).

Since conceptually steps 1 and 2 above are similar, you will only implement step 1 (rotation in place).

Task 3.2. *Proceed as follows.*

1. *Assume that initially, the robot is at the origin with zero heading angle. Given (x_1, y_1), your model should compute the angle θ by which the robot should rotate. The robot can rotate two different ways to point to (x_1, y_1): select the shorter angle.*

2. *Modify the model in Fig. 3.1 (remove the reference time-trajectory block) and use only proportional control on the angular error to compute the DDR reference turning-speed. Do you notice any overshoots? Choose K_P such that:*

   - *The overshooting is small (30° or less for the maximum initial angular-error possible).*
   - *Time to convergence is less than 20 seconds even for the maximum initial angular-error possible.*
   - *The currents are safe for maximum initial angular-error possible.*

   You can also try additing the derivative part of the controller as in Task 1.23 to damp out the oscillations.