Project report: PID velocity control for a DC motor

This report focuses on digital implementation of a PID controller for velocity control of a DC motor. The goals of this project are to create a detailed simulation of a DC motor with PID control, to experiment with different parameters, and to investigate how the choice of the control loop sample time affects the controller performance.

The report will start with brief overview of PID control and its digital implementation. Next, a mathematical model of a DC motor is formulated, and its dynamic behaviour analysed briefly. This is followed by an in-depth review of the methods and algorithms used in the simulation.

1. Digital PID control

A controller is any device that is used to manipulate the inputs of a system to achieve a desired output. If the input-output relations of the system, or in other words, the mathematical model of the system is known, it should be possible to control the system by sending input commands according to this model. This type of control is called open-loop control, and can be represented with the following block diagram:

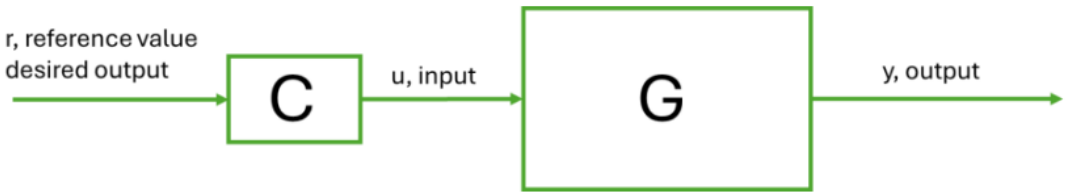

Figure 1: Block diagram of open-loop control.

In the above figure, G is the system that is being controlled, also referred to as the plant. C is the controller, which in this case is just a conversion between the input and output given by the mathematical model of the plant. The problem when working with real physical systems is, that it is not always possible to perfectly model the system, and the system might be exposed to varying operating conditions with external disturbances and noise. This will cause an error between the desired and actual output. Luckily, this can be solved by measuring the error term, and controlling the system based on these error values. This concept is called a closed-loop controller, with the following block diagram:

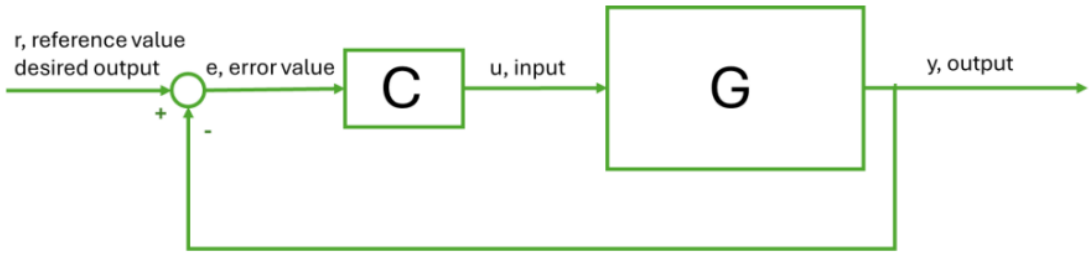

Figure 2: Block diagram of closed-loop control.

The simplest controller of this form is the proportional controller. This is achieved by using a proportional gain Kp as the control block. It takes the error value and multiplies it with the gain Kp and feeds the resulting signal to the system. The problem with proportional gain is that it will leave a steady state error, as the error signal, and therefore the controller output, gets smaller the closer the system output reaches to the reference value. This can be minimised by aggressive proportional gain, but it will lead to overshoot and eventually an unstable system. The steady state error can be fixed by taking the integral of the error values and feeding this to the system through the integral gain Ki. By summing up the past errors, it is possible to remove the accumulated steady state error. However, increasing the integral gain too much will lead to oscillations. From the following graph, it is easy to see why this happens:

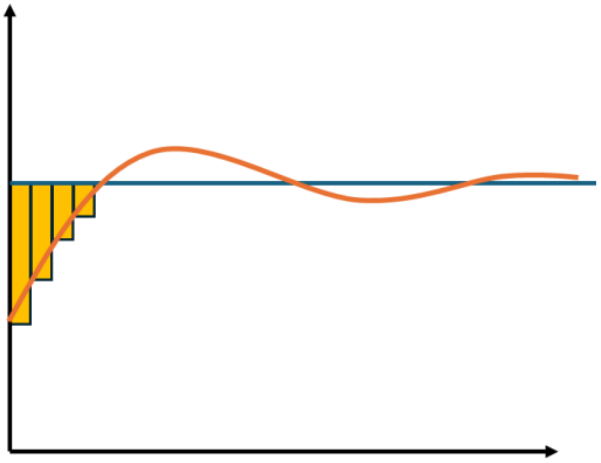

Figure 3: Error accumulation in a controller.

By the time the signal reaches the reference value, the controller has accumulated a lot of errors. If the integral gain is then much greater than the proportional gain, the dominant dynamics are determined by the accumulated error instead of the instantaneous error. This is why the controller gain Kp and Ki must be chosen carefully. This type of controller is called a PI controller, short for proportional-integral controller. To make it a PID controller, we must calculate the rate of change of the error and feed it through the derivative gain Kd. The derivative term can predict the error trajectory and

therefore smooth out the response. However, the derivative term is susceptible to high frequency noise and in practical implementations it is always paired with a low-pass filter or not implemented at all. It is also possible to use the output values to calculate the derivative term instead of the errors.

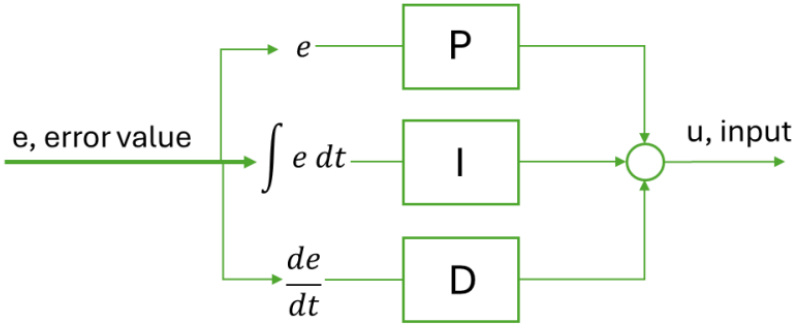

Figure 4: Because of its simplicity and ease of implementation, PID is a popular choice of a controller.

When implementing a digital PID controller, apart from the controller gains, another important parameter to consider is the control loop sample time. The control loop sample time refers to the time it takes for the controller to read the error signals, process them and send them to the controlled system. The control loop sample time should be considerably faster than the dominant time constant of the process to ensure that the controller can capture the essential dynamics of the controlled system and is able to respond to changes quickly. Even though the availability of microcontrollers and high computational power makes it possible to reach high sampling frequencies, approaching this limit might still become an issue, especially when running multiple tasks in parallel.

2. Modelling a DC motor

The dynamic behaviour of a DC motor can be separated in electrical and mechanical parts, both described by a first order differential equation. The electrical dynamics can be obtained by Kirchhoff's Voltage Law, which states that the applied voltage equals the voltage drop around a closed loop. For the DC motor armature circuit, this can be written as follows:

$$V = L\frac{dI}{dt} + RI - e_b,$$

where eb is the back electromotive force (EMF) in volts. The back EMF is proportional to the velocity of the motor, ω, and thus the equation can be written as:

$$V = L\frac{dI}{dt} + RI + K_e\omega.$$

The mechanical dynamics can be obtained from Newton's second law, which states that the torques acting on the system must equal the rate of change of the angular momentum. If the rotor has a moment of inertia J, we can then write:

$$J\frac{d\omega}{dt} = T_m - T_L,$$

where Tm is the motor torque and TL is the load torque. The amount of torque produced by the motor is directly proportional to the current flowing through the armature circuit, and it relates to the armature current with the motor torque constant KT. In SI units, the motor torque constant [Nm/A] and back EMF constant [V/(rad/s)] have the same numerical value, and for the rest of this report, K will be used to refer to both. The load torque represents the external torques opposing the motion of the rotor. This can be due to attached loads, gears, pulleys, gravitational forces or external disturbances to name a few. Often a friction model is added separately to the mechanical description of a DC motor. A simple way to describe the frictional effects is to relate the friction to the angular velocity with the damping coefficient B. Finally, this gives us the following differential equations for a DC motor:

$$J\frac{d\omega}{dt} + B\omega = KI - T_L$$
$$V = L\frac{dI}{dt} + RI + K\omega.$$

And with little re-arranging we obtain:

$$\frac{d\omega}{dt} = (KI - B\omega - T_L)/J$$
$$\frac{dI}{dt} = (V - RI - K\omega)/L.$$

3. Methods and algorithms

The primary goal of the project was to create an interactive plot that shows the effect of different controller parameters on the system output. For this, the differential equations describing the dynamics of the motor had to be solved. Firstly, the motor parameters were initialised as global constant parameters. This was a conscious decision to avoid passing the motor parameters through the functions in addition to the controller parameters, and in that way making the code more exhaustive to read. The motor parameters do not need to be changed when running the interactive plot. Secondly, a motor model function was created that takes in the plant outputs ω and I, inputs V and TL, and returns the derivatives $d\omega/dt$ and dI/dt. Time was added as a redundant input to make the code more general and reusable as explained later.

Two methods were used to solve the differential equations described by the motor model function: the forward Euler method and a fourth order Runge-Kutta method. Both were implemented as functions that solve for a single timestep forward. The Euler method is described as follows:

$$y_{i+1} = y_i + hf(y_i, u_i, t_i),$$

where y is the output vector [ω, I], u is the input vector [V, TL] and f is the motor model function, or in mathematical sense, the respective derivatives of the two outputs. The step size h is equal to the simulation time step dt. The index i refers to the time step number, and therefore ti = i · dt. The time variable is not used in the motor model, but it is used to make the differential equation solvers reusable in other code. Simply put, the forward Euler method works by calculating the rate of change of the function y and then multiplying it with the timesteps. This returns the change in y, and it is then added to the current value of y. The forward Euler method is computationally efficient way of numerically solving differential equations but will introduce some errors due to approximating the derivative with finite difference. The global error will be in the order of the step size h.

The second method used was a fourth order Runge-Kutta method described by the following equations:

$$\begin{gathered} y_{i+1} = y_i + h(k_1, 2k_2, 2k_3, k_4)/6 \\ k_1 = f(y_i, u_i, t_i) \\ k_2 = f(y_i + hk_1/2, u_i, t_i + h/2) \\ k_3 = f(y_i + hk_2/2, u_i, t_i + h/2) \\ k_2 = f(y_i + hk_3, u_i, t_i + h) \end{gathered}$$

Again, time t is not actually used in the simulation, as there is no explicit dependency on time in the motor model. As we can see, the first k parameter in the Runge-Kutta method is just the rate of change used in the Euler method. The next three rates of change are calculated at different points, and in the end a weighted average is taken to obtain the final estimate. Generally, this gives a more accurate estimate for the function than the Euler method, with the global error in the order of h^4. However, when using the interactive plot, it was noted that the difference between the two solvers is not significant, and both give good results for the purpose of demonstrating the effect of controller parameters. Even though the RK4 adds more computation steps, it does not slow down the drawing of the functions and therefore it is used as the default solver in the simulation.

One input parameter to the motor is the control signal u. The controller calculates a voltage sent to the motor every control loop sample time Ts ≥ dt. The main simulation solves the motor equations using the RK4 or Euler algorithm every simulation step, and then a cascaded for loop is added to calculate the new control voltage values every f step, where f is the control loop sample time divided by the simulation step size cast as an integer value. The inter sample behaviour used in the simulation is zero-order hold, which means that the value is held constant until the next update. The integral is implemented as a cumulative sum of the error value multiplied by the apparent control loop sample time f · dt. The derivative is approximated as the difference between the previous error and the current error divided by the same apparent sample time. A low-pass filter has been implemented to be used for the derivative error by simply taking a weighted sum of the previous and current values. It is implemented as a separate function for reusability and to make the code more readable.

There are two kinds of noise introduced in the system model: process noise and measurement noise. The measurement noise is used to simulate any noise that a position encoder would have. It is simulated as zero mean Gaussian noise, with adjustable magnitude, meaning there is no bias in the sensor. The measurement noise is added to the actual velocity before using it in any of the calculations, as would be the case in a real system also. The process noise in comparison describes the uncertainty in the dynamic model of the system, and the chosen injection point is the load torque. The process noise is calculated using a separate function called load disturbance. It models the load torque disturbances as low-pass filtered zero mean Gaussian noise with adjustable magnitude. This is because mechanical vibrations are typically lower in frequency than other noise sources such as electromagnetic interference. Another typical disturbance could be a static torque offset, which is not implemented in this model.

The interactive plot itself is made using the matplotlib plotting library, and it relies heavily on the slider widget. A 2x2 subplot grid is created and each one is initialised with a different plot, highlighting different aspects of the controller. The initialisation is done by running the simulation with the initial parameters and then creating a line object for all the signals in each plot. A slider object is created for each of the parameters that the user should be able to change. A function is defined that runs the simulation with the values defined by the sliders and updates the lines with the new signal values. Finally, the widgets have an associated method that can call any function when a slider value is changed, which is used to call the plot update function every time a slider is touched. The interactive plot uses 0.001 as the default simulation step but it can be adjusted for slower computers.