LABORATORY EXERCISE 2: MOTION CONTROL

OBJECTIVES

The objectives of this laboratory are:

- To give practical experience of how servomotors are used in closed loop mode in order to achieve accurate motion control. To give experience of the practical meaning of proportional and derivative control in the context of a simple motion control system
- To run different ramping techniques for a stepper motor, observe how ramping of step rates can be achieved without the use of division and identify advantages and disadvantages of the different methods.

SAFETY

In practice this is a low hazard laboratory, but it does involve the rotating shafts on two small motors. To avoid any risk of entrapment, **sleeves should be rolled up and long hair tied back**. No personal protective equipment is required.

INTRODUCTION: THE CONTROL LOOP IN THE CONTEXT OF A SERVO MECHANISM

Fundamental to the operation of a servomotor is the concept of closed loop control – this was covered mathematically in an elementary manner in MMME2046, and forms the basis of more advanced study in MMME3063.

In brief, the control loop when applied to a servomotor consists of the following:

- Decide on a position we want the motion control system to move to
- Measure the current position
- Compare the position we want with the current position (to give *error*)
- Decide upon corrective action to bring system closer to its desired position (this decision process is known as the *control algorithm*)
- Take the corrective action

This can be represented diagrammatically as shown in Figure 1 .

During this lab you will be using programs which go from “open loop control” (where you vary the motor signal directly, with no attempt at controlling position) to “closed loop control”, where an extremely simple control algorithm is implemented (proportional control). Finally a library is used which implements a more sophisticated PID (proportional integral derivative) algorithm.

You will also be examining the control of a stepper motor. Unlike a servo motor, a stepper motor normally operates in open-loop mode as no feedback is required to achieve a given position.

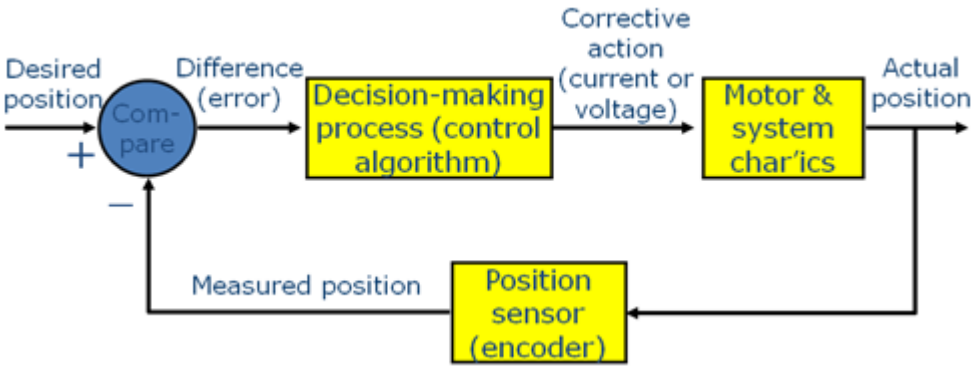


Figure 1 Closed loop motor control

UNDERSTAND THE LAB KIT

A labelled picture of the lab exercise setup is shown in Figure 2. The Lab Kit has been developed to make it very easy for you to set it up, and the focus is on understanding the methodology and process. Consequently, all connections between different sensors/buttons/peripherals and the Arduino are facilitated via a custom wiring hereness with mating plugs and sockets. This is different from the "take-home" kits where the skill to develop your own electrical circuit using breadboard and jumper wires is important.

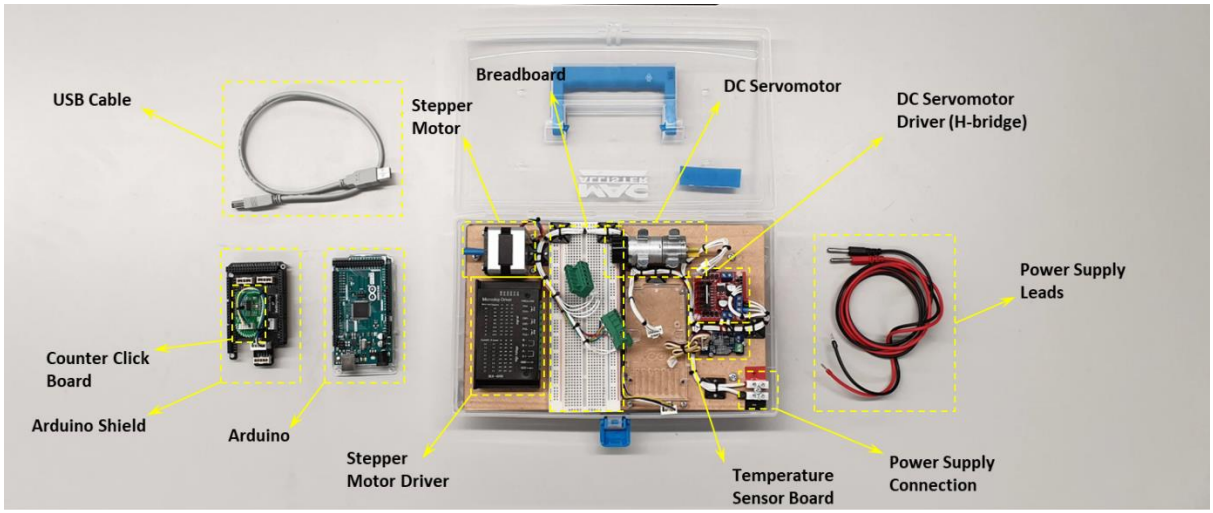


Figure 2 All components in the Lab-Kit

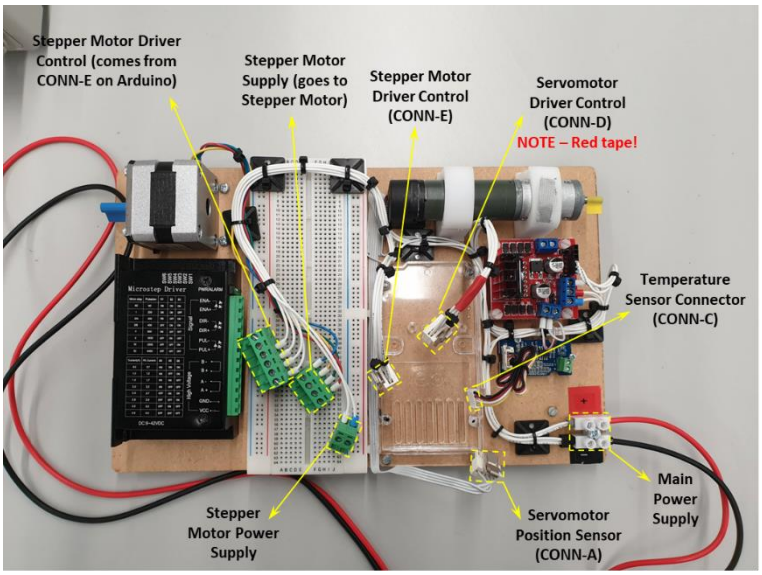
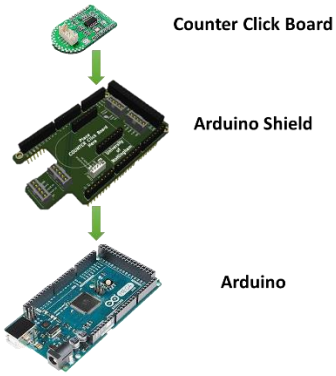


Figure 3 Description of Connectors in the Lab Kit

1. **USB Cable** – Used to connect the Arduino with your laptop or PC.
2. **Arduino** – Microcontroller device that you can program to do various things.
3. **Arduino Shield** – Hardware interface that has been specifically developed to work for these experiments. All connections you require to make with the Arduino (for these experiments) can be made via connectors (no bread board and jumper wires!). The Shield slots on top of the Arduino.
4. **Counter Click Board** – Daughter board required to measure shaft rotation speed. This board slots on top of the Shield.
5. **Stepper Motor** – Motor for applications where precise control of shaft angle position is required.
6. **Stepper Motor Driver** – Produces the signals to drive the stepper motor.
7. **Breadboard** – Prototyping board using jumper wires. You will not require this for your experiments.
8. **DC Servomotor** – General purpose motor for cheap and low power applications where position of the shaft is controlled via closed-loop feedback mechanism.
9. **DC Servomotor Driver (H-Bridge)** – Controls the DC Servomotor. A constant DC voltage across the motor's terminals would also spin the motor, but you cannot control direction or torque produced. The H-Bridge allows that.
10. **Temperature Sensor Board** – Daughter board with a thermistor and connection for a thermocouple.
11. **Power Supply Connection** – DC power supply leads should be screwed in these terminals. Care must be exercised with the polarity (i.e., red lead goes with the red connection point).
12. **Power Supply Leads** – The leads to power the lab kit (using the programmable power supply unit available for the experiments).



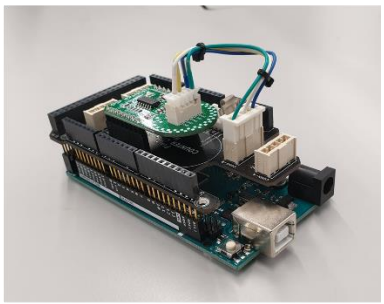


Figure 4 Three boards (Arduino + Arduino Shield + Counter Click Board) together

UNDERSTAND AN OSCILLOSCOPE

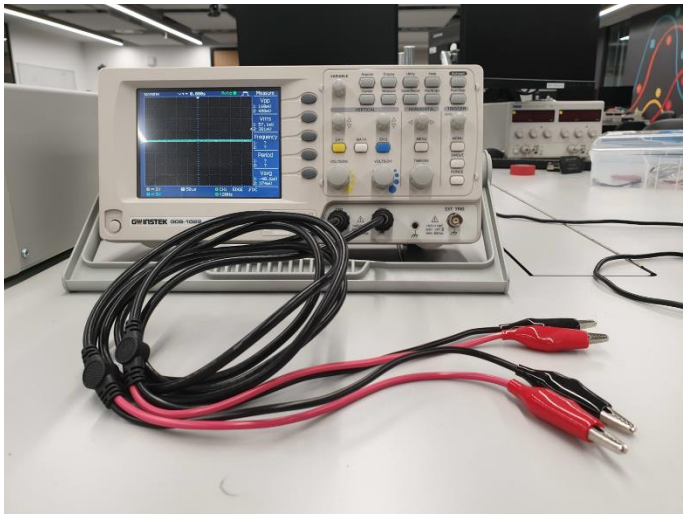


Figure 5 GW-INSTEK GDS-1022 Digital Storage Oscilloscope

WHAT IS IT?

An oscilloscope is an instrument to measure, visualise and record a voltage signal. It has a probe that is required to be connected to the source of voltage (it may be the output of an electronic circuit you have designed or something as simple as an AA-size battery!) and the oscilloscope "set" to read the signal. Usually there are multiple channels (or signals to probe) that can be visualised simultaneously, and some simple mathematical functions done live. The speciality of an oscilloscope (compared to other similar voltage measurement devices like a digital multimeter) is its ability to visualise the signal over a time duration (not just an instantaneous measurement) and record it on a USB stick.

WHY DO YOU NEED IT?

Unlike engineers from other disciplines, electrical and electronic engineers cannot "see" the output of their physical circuits and systems (needless to say, its crucial to be able to see what you have created, is it behaving how you expect it to and if not, what bit needs modifying). An oscilloscope usually goes hand in hand with the design and build of any circuit as it allows you to visualise and record the input and output signals.

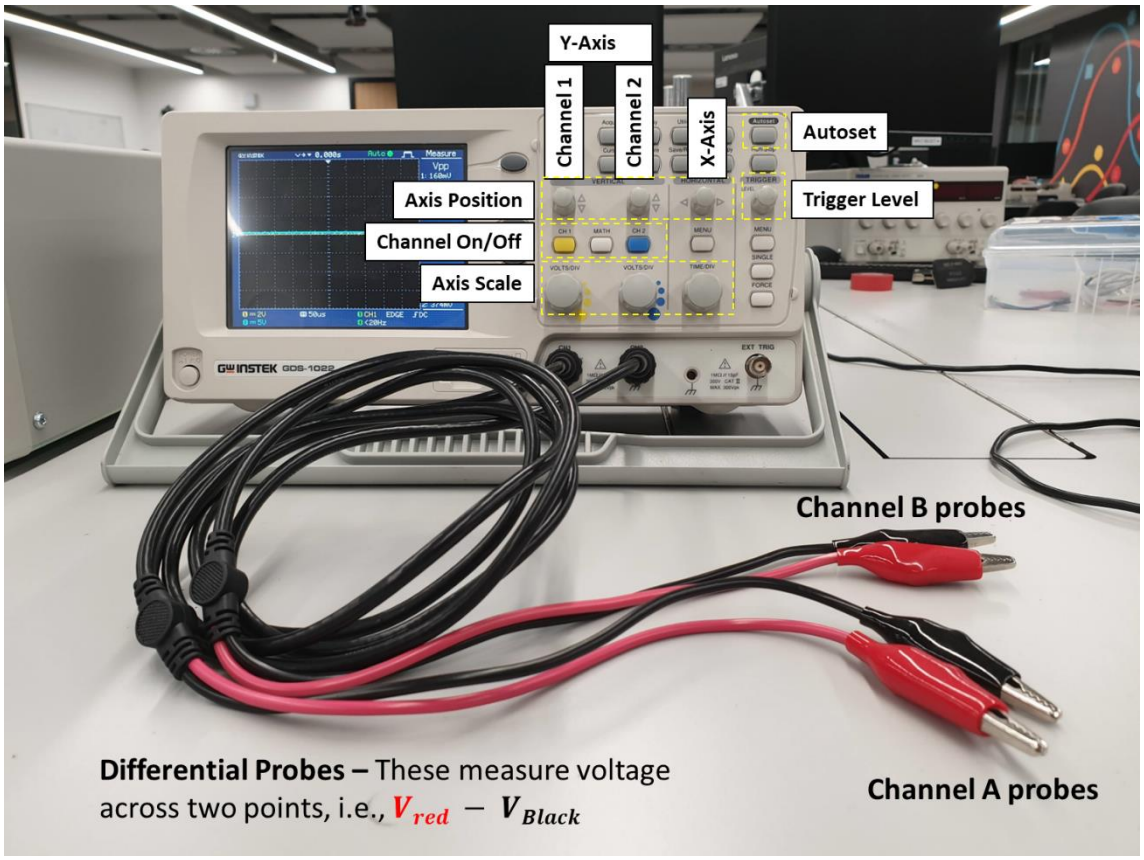


Figure 6 Details of an oscilloscope

THE AXES – HORIZONTAL (TIME) AND VERTICAL (VOLTAGE AMPLITUDE)

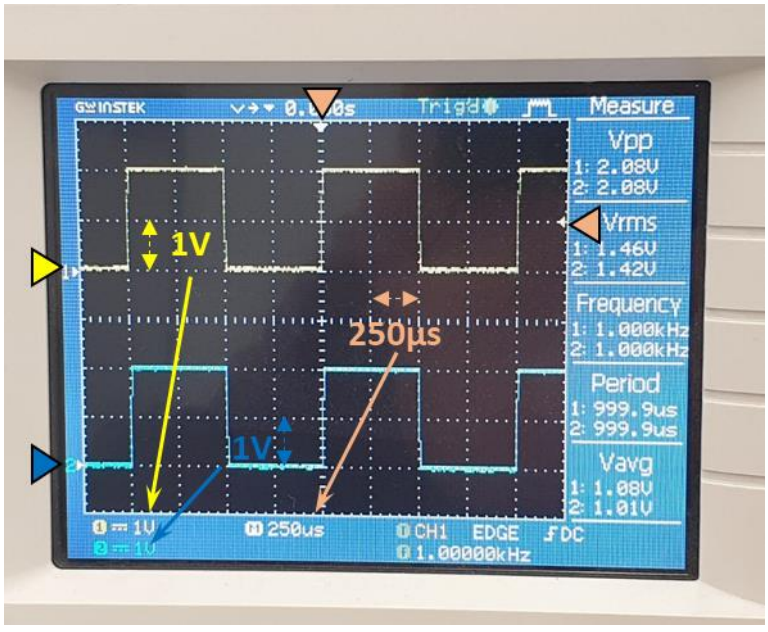


Figure 7 Standard square wave shown on both channels

The information bar at the bottom gives essential information to understand the magnitudes of the visualised signal (Figure 7). The screen is divided in 10 divisions on both axes (illustration not accurate) and the 1st number depicts the signal voltage per sub-division. The second number depicts the time per sub-division, i.e.,

the entire axis is 10x of this value. The third number is the frequency of triggering (read more about triggering in the next section) of the signal and can be interpreted as the frequency of the signal itself.

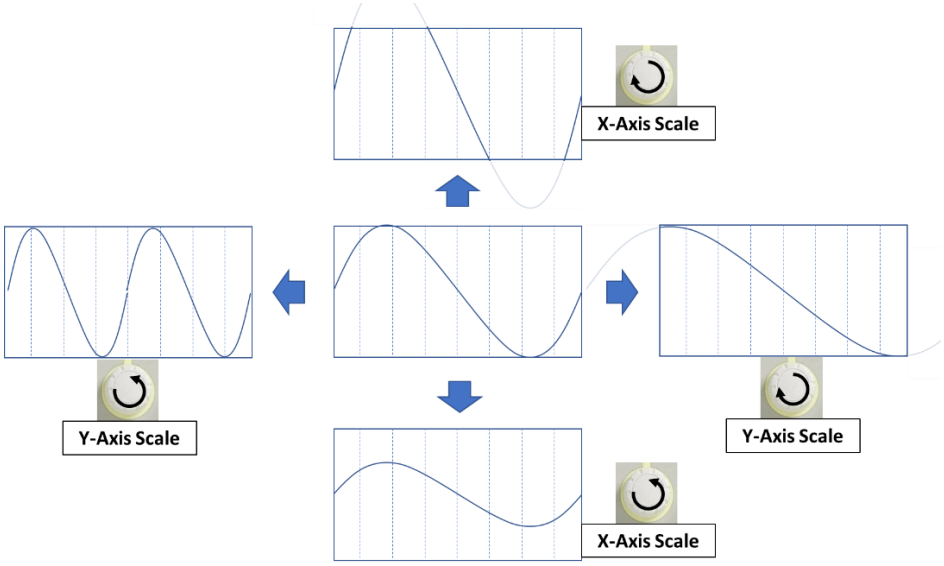


Figure 8 "Scale" functionality (Large knobs) for two channels (Y Axes) and time (X Axis)

The dial knobs called SCALE (both axes) are used to make the visualised signal smaller or larger on the display area (Figure 8). The two channels have their individual Y-Axis scaling dials, but they are both on the same X-Axis, i.e., time axis, and hence there is only one X-Axis scale dial. This means both channels would scale on the time-axis.

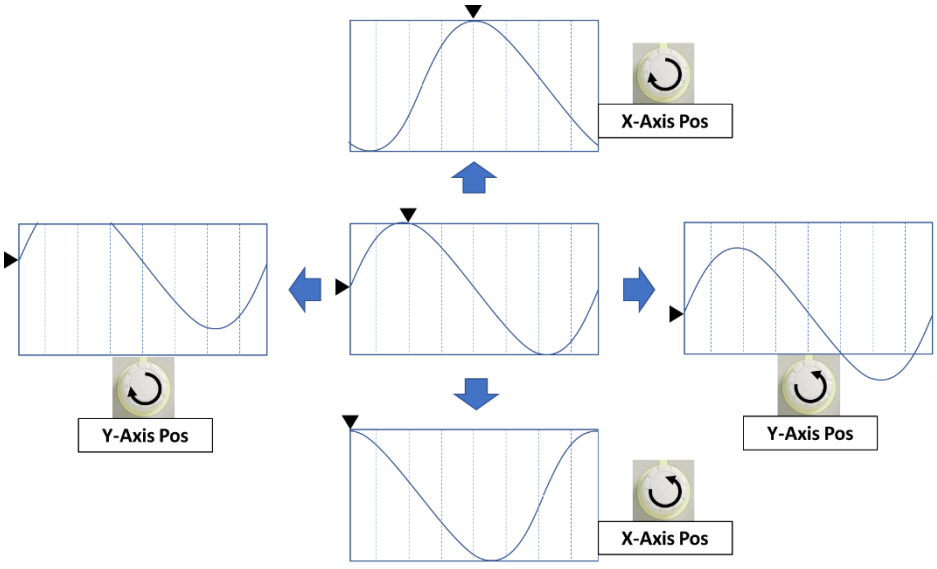


Figure 9 "Position" functionality (Small knobs) for two channels (Y Axes) and time (X Axis)

POSITION dials can be used the move the visualised signal up, down, left, or right. This is very useful when you want to compare two channels visually, you can align them as you want or overlap them on top of each other.

TRIGGERING

This is a bit complicated to understand in the first go, but once you have grasped the concept it is very straightforward. The way an oscilloscope works, it rapidly clicks “snapshots” of the signal (for the entire duration as you have set up using the X-Axes). Now we need to tell the oscilloscope at what instances should it click these snapshots. This is done using the TRIGGER Level and Menu functions. You can tell the oscilloscope to continually detect for “Falling Edge” (selected from the Trigger Menu) at a particular trigger voltage “Level” (set using the dial), e.g., 0V.

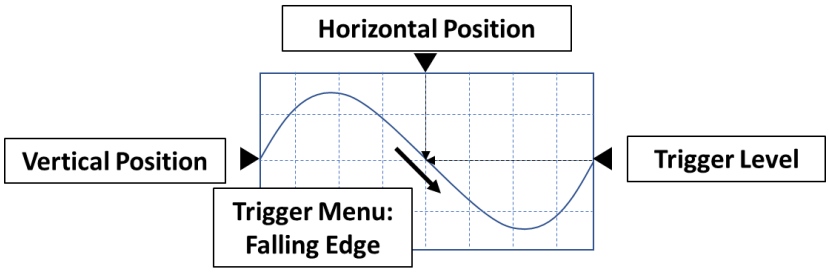


Figure 10 "Triggering" functionality

AUTOSET BUTTON

This is a very useful function; it automatically sets all the above settings to produce a usable display of the input signals.

EXERCISE 1: PROPORTIONAL CONTROL AND PID CONTROL OF DC SERVOMOTOR

OPEN LOOP CONTROL PROGRAM

1. Download the program **SimpleControlNoFeedback.ino** from Moodle. This is essentially the same program that was used in Lab 1 but with the encoder state machine and interrupt codes stripped out.

This program drives the motor using an open loop PWM signal, allowing the duty cycle to be varied by input in the serial monitor. The L298 motor driver is used to process these signals and the H-bridge incorporated on the chip allows for direction control using the positive and negative polarity of the input. There is a nice explanation of the functioning of the L298 motor driver here: https://lastminuteengineers.com/l298n-dc-stepper-driver-arduino-tutorial/ . It will help you to understand the program used to control the motor (note that the pins used are numbered differently as the article uses an Arduiono Uno not a Mega as we use here).

2. Test the program before you go to the lab using the Arduino Mega in your take-home kit:

- Compile the program and upload (you may need to check and set the correct COM port).
- Set the Serial Monitor to 9600 baud rate and to send "Both NL & CR" on pressing a number.
- Type in a number between -100 and +100 to represent the percentage PWM value. You should see the LED next to pin 13 glow brightly or dimly accordingly.

OPEN LOOP CONTROL EXPERIMENT

1. Ensure the large bench power supply unit (PSU) is connected to the mains.
2. Switch the PSU on and ensure the ON/OFF button is in OFF. When it is OFF, the screen displays the set-point voltage and current limits (see Figure 11).

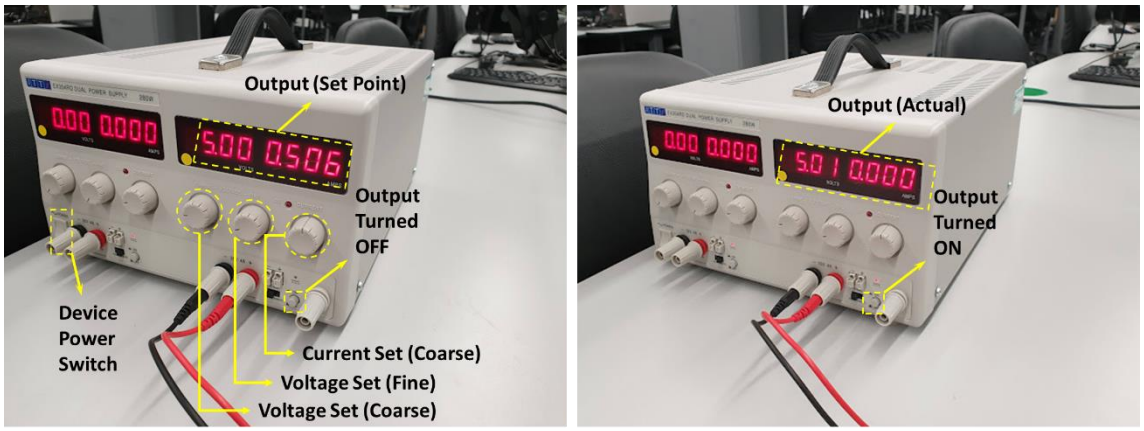


Figure 11 Using the Power Supply Device

3. Adjust the voltage to read 5V and the current limiter to read ~0.5 A. Use the three knobs:
   a. Voltage Set Point – Coarse (Left)
   b. Voltage Set Point – Fine (Middle)
   c. Current Limit Set Point – Coarse (Right)
4. Turn ON the supply. The displays now read the actual voltage and current. The voltage should remain around 5V, but the current should be approximately zero as no load is connected.
5. Turn OFF the output and switch off the device.
6. Ensure the Arduino is disconnected from the PC.
7. Make the following connections:

a. **CONN-D** (on Arduino Shield) with the "**Servomotor Driver Control**" connector (see Figure 3 a few pages above – this is the plug/connector with a red tape wrapped around the cables)
b. **CONN-A** (on Arduino Shield) with "**Servomotor Position Sensor**" connector (see Figure 3 a few pages above)
c. **CONN-B** (on Arduino Shield) with "**Counter Click Board**" connector (see Figure 4 a few pages above)
d. **Main Power Supply** (bottom right of the lab kit base board) using the **Power Supply Leads** (see Figure 2 and Figure 3 a few pages above – ensure polarity is correct, i.e., **Red** with **Red**, and **Black** with **Black** – see Figure 12)

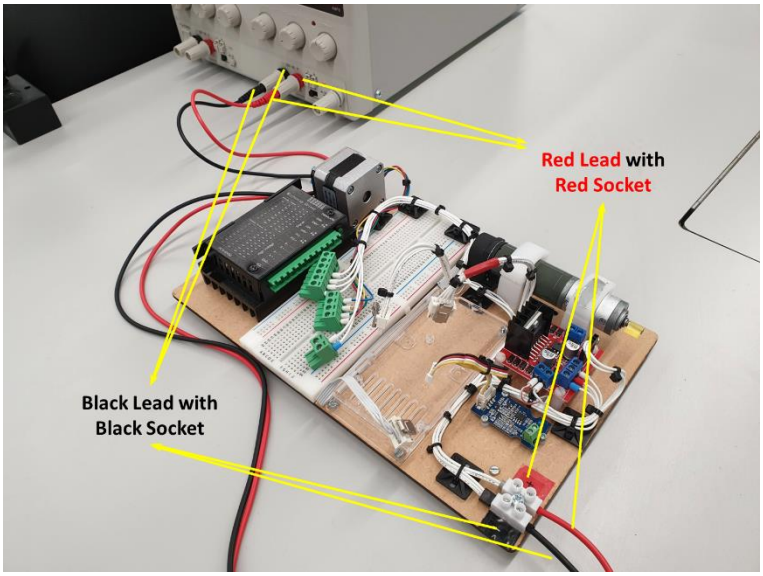


Figure 12 Polarity matching when connecting Power Supply

8. **Get a demonstrator to check your wiring!** It should look like Figure 13.

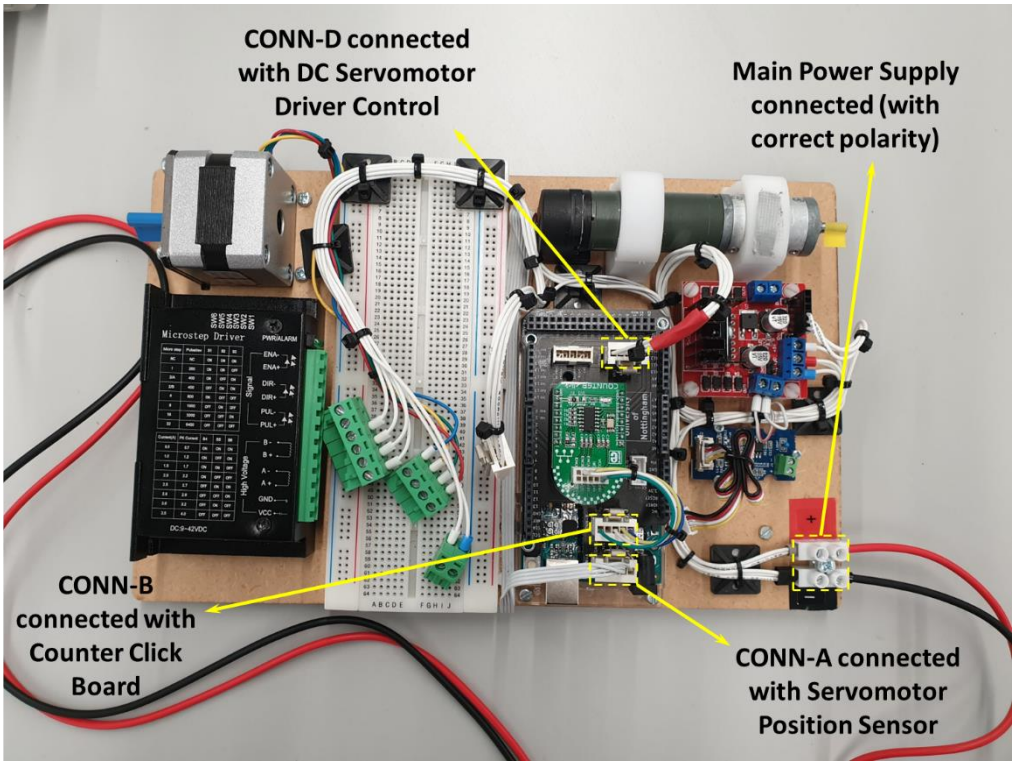


Figure 13 Hardware setup for Lab-1 Exercise-2

9. Connect the Arduino to your PC/Laptop using the USB cable. You should see the light appear on the Arduino. This indicates there is 5V power from the USB.
10. Turn ON the PSU and turn ON its output. Ensure you start at 5V.
11. Load the **SimpleControlNoFeedback.ino** into the Arduino IDE.
12. Ensure that the correct port and board type are set by checking **Tool | Board** and **Tools | Port**.
13. Now compile and upload your program. Open the Serial Monitor and check the box in the bottom right-hand corner of the monitor reads "Both NL and CR" to ensure that your program will understand when a line of text has been sent.
14. Type different values of PWM duty cycle percentage in the range -100 to 100 into the input field. The motor should run forward or backwards for positive and negative values. **You now have a working DC Motor with variable torque and interchangeable direction!**

You must have realised that this is just a repeat of exercise 2 and 3 from the previous practical session. The outcome from this is simply to implement an "open-loop" control of the DC Servomotor. Open-Loop control simply means that the controller (Arduino controlling the H-Bridge that drives the motor) is sending commands without getting any feedback from the motor itself, i.e., it is operating blind. The H-Bridge can only control the torque output (-100% to +100% – this is what your program basically commands) in open-loop. In order to have the H-Bridge also control the exact position of the motor shaft,

- it needs to know where its current position is,
- drive the motor towards the desired position by demanding a torque output,
- continuously monitor the motor position while the motor is driving,
- slowly reduce the torque demand as it approaches the desired position,
- and finally stop the motor when it has arrived at the target position.

The above series of steps is what constitutes "closed-loop" control. It requires the controller to "read" the live position of the motor shaft; it is doing so via the quadrature encoder, or "counter click board" that you experimented with in lab 1 exercise 3. We shall now proceed to employ this working "open-loop" motor control setup for "closed-loop" position control using both a simple proportional and a "PID Control".

CLOSED LOOP CONTROL PROGRAMS

Download the programs **ProportionalClosedLoop.ino** and **PIDClosedLoop.ino** from Moodle. These are both based on the open loop program used in the previous section.

In both programs an extra loop has been set up in the loop() function for executing a controlLoop() function at intervals specified by the "controlInterval" parameter. The controlLoop function adjusts the output to the motor in order to control the position as follows:

- Proportional Control: The control loop is shown in Figure 14.

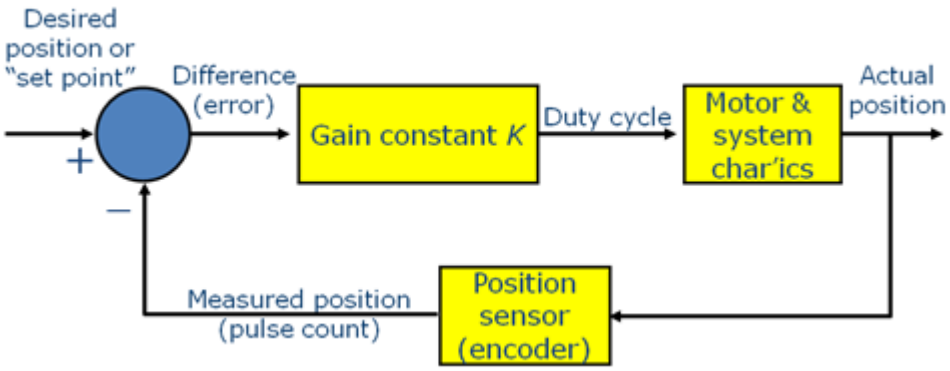


Figure 14 Closed loop proportional control

In the program the proportional gain is given by the global constant Kp and is initialised to 0.02. The control loop reads in the current position using the LS7366R and calculates an error between this and the requested value. The error is multiplied by the gain to give the new output value to be sent to the momtor.

- PID Control: The PID control is more complicated, as shown in Figure 15, and we are using an existing library to implement this.

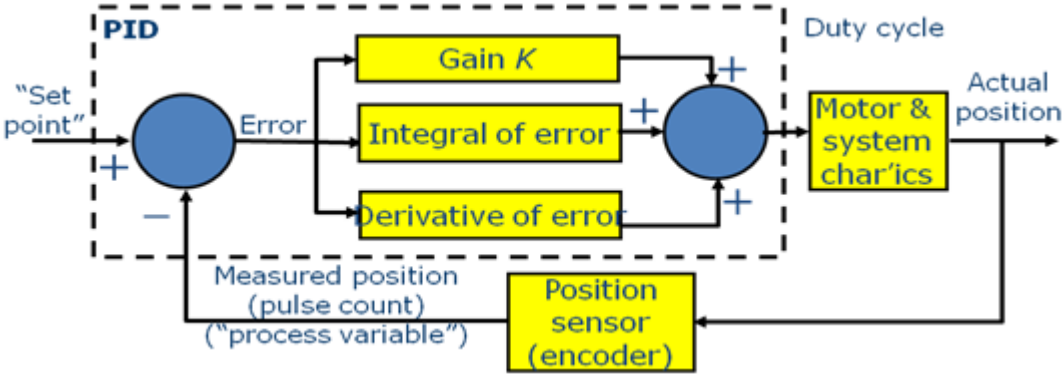


Figure 15 PID Control Loop

In order to run the program supplied on Moodle you will need to install the PID library:

- From the Arduino menu bar, look in the **Sketch | Include Library** sub-menu to see if the PID library is installed (if it is, it will be in the bottom section of the sub-menu). If it isn't, look at the top of the sub-menu for Library Manager and click on it to start it. You can then search for **PID**, but be careful to highlight **PID by Brett Beauregard** not any of the other PID libraries (there are several) or the following instructions won't be applicable. Install the V1.2 of the PID library.
- From the Arduino menu bar, load the example sketch from **Examples | PID | Basic** (you may need to scroll down to near the bottom of the Examples sub-menu) and examine the example closely. You will see that:
  a. The definition of the PID class is contained within the file **PID_v1.h** which is included in the sketch via the directive **`#include<PID_v1.h>`**

  b. Global variables for the values of proportional, integral and derivative gain K_p, K_i and K_d are defined and initialised.

  c. A global object called **`myPID`**, of class **`PID`**, is declared, linking the object to the measured value named here as the input, the control action named here as the output, and the setpoint. Note that these values are all passed by pointer so the PID object has the ability to change their values. The variables **`Kp`**, **`Ki`** and **`Kd`** are also linked to the object here. The value **`DIRECT`** is not important; it defines that the positive directions for input and output are the same.

  d. In the function **`setup()`** the **`Setpoint`** variable is initialised, also the value of **`Input`** is initialised by reading the value of from an analogue input pin, and the PID controller is made active (set to automatic control).

e. Now everything is set up for the PID controller to control the output by taking decisions on what the input is doing compared with the setpoint. This takes place in the function **`loop()`**, where for each pass through the loop:
   - the value of **`Input`** is read from the analogue input
   - the PID calculations are calculated by the member function **`Compute()`**
   - the value of **`Output`** is used to make something happen via an **`analogWrite()`** function call.

- The **PIDClosedLoop.ino** program implements the motor control in the same way as the example program. The three gains Kp, Ki and Kd are set up as global variables. The global PID object is defined. Look at the variables which are used as input to this function in order to understand how the PID control will work. The myPID.Compute() function is called in the control loop in order to calculate the new output value to be sent to the motor.

CLOSED LOOP CONTROL EXPERIMENT

Watch the video demonstration for Lab 2. You can find this on Moodle under "Lab and Programming Exercises" > "Laboratory 2 – Motion Control".

15. Now load **ProportionalClosedLoop.ino** and set **`Kp`** to 0.001. Compile and upload it to the Arduino. You should be able to enter a target position and watch the motor move approximately to that position.
16. What happens as you make the proportional gain **`Kp`** take different values in the range 0.001 to 0.1? If it oscillates wildly, de-activate the PSU, insert a smaller gain, and try again.
17. De-activate the power supply and load **PIDClosedLoop.ino** Initially try **`Kp`**=**`0.02, Ki=0.0`** and **`Kd=0.0.`** Re-activate the power supply confirm that the result is the same as for your proportional gain example.
18. What happens when you add some **`Ki`** value?
19. What happens when you add some **`Kd`** value?
20. What happens when all three parameters are non-zero? Are they competing with each other?
21. When you have completed the experiment, please proceed with the shutdown procedure:
    a. Disconnect the Arduino USB cable to the PC/laptop (this removes 5V supply and immediately mitigates against any accidental shorting).
    b. Disconnect all the connectors that you plugged in on the Arduino Shield (CONN-A to CONN-E).

EXERCISE 2: OPEN LOOP POSITION CONTROL – ONE STEP AT A TIME

OVERVIEW OF STEPPER MOTORS AND RAMPING

This part of the laboratory exercise involves controlling a stepper motor. These devices are motors which operate under (usually) open-loop control and produce a fixed angle of rotation for each increment (step) of their movement. The basics of stepper motors were covered in Electromechancial Devices and should be revised before the laboratory.

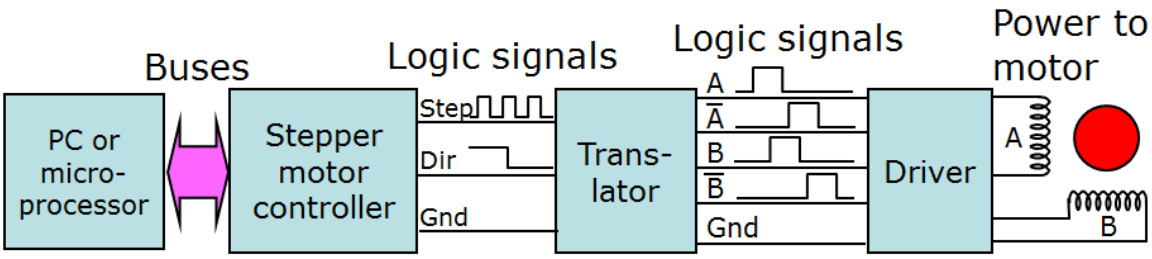


Figure 16 Stepper motor driver and controller

In the present case, the stepper motor controller is a combination of a timer and various input-output lines rather than a separate device. Stepper motors are driven by circuitry (translator and driver) which normally take in signals on two lines:

- Step signal: each complete pulse on the step line corresponds to a single increment of angular rotation
- Direction signal: if the signal is TRUE, the step takes place in one direction e.g. clockwise, if it is FALSE, the step is in the opposite direction

In practice it is usual to make a movement by ramping up the motor from a low speed (which may be zero) up to full running speed and back towards rest via a "profile" of uniform acceleration, constant speed and uniform deceleration. This smooth profile is needed so that there are no sudden changes in speed or very rapid accelerations or decelerations, in order to avoid "desynchronising" the rotor of the motor from the magnetic field which is stepping around the stator.

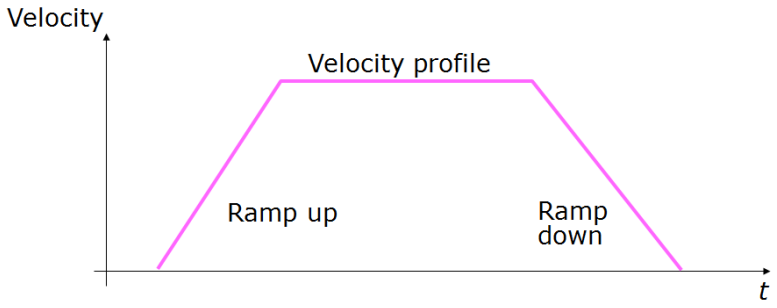


Figure 17 Ideal velocity profile

This profile involves generating trains of pulses which are initially widely spaced in time, then gradually become more closely spaced as the motor is accelerated, and eventually become more widely spaced again and cease altogether as the motor slows down to a halt.

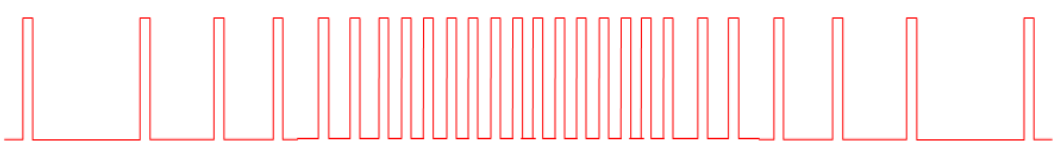


Figure 18 Spacing of pulses to generate velocity profile

The generation of these pulses under computer control is simple in principle but in practice is non-trivial, not least because it is desirable to avoid computationally intensive calculations (such as division operations and mathematical functions) which will take longer to complete than the time available for each step. Some ingenious approaches are therefore used to perform the calculations using only addition and multiplication operations, for example by making a Taylor series expansion of the "difficult" functions.

The pulses are typically timed using an interrupt service routine which is repeatedly called at intervals determined by a timer counter. For simplicity however simple timed loops are used for the initial version of the sketch. It could be naively assumed that incrementally changing the timer counter intervals (so that they ramp down linearly from long intervals to short intervals and back up again) would provide the required acceleration and deceleration profile, and your starting point is a sketch which implements this. However, as you will see, this results in a poor profile, which starts very slowly then builds up to an unrealistically large acceleration as the desired speed is approached.

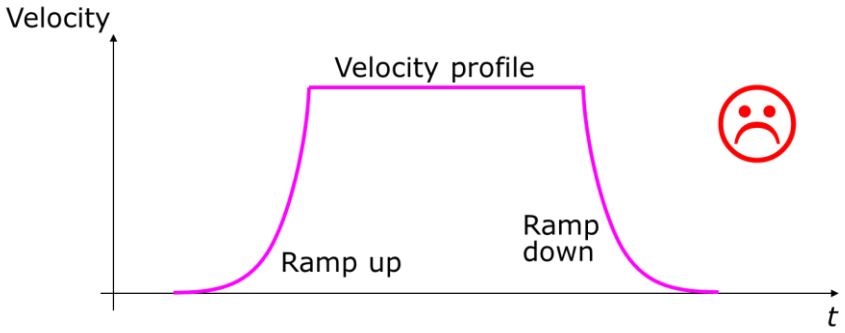


Figure 19 Simplistic velocity profile

The approach you will use is the "Leib Ramp Algorithm" as presented by Eiderman[1], one of several approximate methods available for performing a good approximation to the desired uniform acceleration/deceleration profile.

OPEN LOOP CONTROL PROGRAMS

A SIMPLISTIC APPROACH THAT NEARLY WORKS

A simplistic algorithm has been implemented in two programs which can be downloaded from Moodle. They both implement a ramping profile which works, but not very effectively. The first, SimplisticRampStepper.ino, is an Arduino sketch which will be used in the next experiment in the lab. To give you an appreciation of how this algorithm performs the second, SimplisticRampStepper.c, is a C program which acts as a simulation of the real sketch behaviour. (Health warning: so that the functions in the C program are largely the same as those in the Arduino sketch, the C program uses several global variables. **Don't do this in other C programs!)** As set

[1] Aryeh Eiderman, "Real Time Stepper Motor Linear Ramping Just by Addition and Multiplication". http://hwml.com/LeibRamp.pdf

up, this uses a timed loop (like the one in the "Blink Without Delay" example, but working in microseconds not milliseconds) to time the steps.

Firstly, try running **SimplisticRampStepper.c** (or **SimplisticRampStepperMac.c** if using a Mac) and see that it displays (in real time) a table simulating the individual steps, the time relative to starting for each step, the speed and acceleration at each step, the duration of each step etc. You can specify a position (in steps, relative to the initial starting position) for the motor to move to, so you can see the effects of running the motor short distances involving only acceleration and deceleration, or longer distances involving a period of running at constant speed.

Run the program for the case of starting at position 0 and running to position 50. Use the default values of minimum and maximum speed and maximum acceleration (1 step/s, 20 steps/s and 10 steps/s^2 respectively). Cut and paste the text from the command prompt window into a text file (call it "SimplisticProfile.csv") and import it into Excel. Plot a graph of velocity vs. time and acceleration vs. time and save your Excel file as "SimplisticProfile.xlsx". It is desirable for the velocity profile to involve constant acceleration and deceleration and (if required) a period of constant velocity. Is this achieved? If not, how badly does it deviate from the ideal?

The current version of the program (and Arduino sketch) uses a crude and simplistic ramping algorithm which increments or decrements the loop interval by a fixed value calculated as:

$$\Delta p = \frac{p_{\max} - p_{\min}}{S} \qquad (1)$$

where S is the number of steps over which acceleration takes place, so that for each step the new step interval p_{new} is given in terms of the previous step interval p_{old} as follows:

$$p_{new} = p_{old} - \Delta p \text{ (during acceleration)}, \quad p_{new} = p_{old} + \Delta p \text{ (during deceleration)},$$
$$p_{new} = p_{old} \text{ (during constant speed phase)} \qquad (2)$$

Note that in the code the new value of p simply overwrites the old one – there is no need for two separate variables.

REPLACING THE SIMPLISTIC APPROACH WITH ONE THAT WORKS MUCH BETTER

The **LeibRampStepper.c** and **LeibRampStepper.ino** programs, available on Moodle, implement the LeibRamp algorithm. Again, the .c file is a simulation of the behaviour of the real sketch behaviour. Use **LeibRampStepperMac.c** if you are using a Mac.

The number of steps still to move S_{tg} is obtained from the function **`getStepsToGo()`** and we will call the initial value of this distance S_{tot}.

To replace the simplistic approach above with the approach described by Eiderman several changes have been made from the simplistic program. Note that all arithmetic, constants and variables are floating point, and only at the final point of use (when it is used for timing the loop or as the register value for a timer) is the step interval p converted to a **`long`** integer.

The changes which have been made in in order to implement the algorithm are described below, starting in function **`main()`** forming the section labelled "Pre-computation code" (the comments STEP 1 etc in the program correspond to the numbering of the points below):

1. The number of steps S (variable name **`accelSteps`**) over which acceleration is to take place must be calculated: this is calculated from equation 4 in the paper:

$$S = \frac{v^2 - v_0^2}{2a} \quad (3)$$

where v is initially assumed to be the maximum permissible speed (termed "slew speed" in the paper), and v_0 is the minimum stepping speed.

2. If twice this value of S exceeds the total number of steps to move S_{tot} then:
   a) The maximum speed v needs to be re-calculated using the following equation (a slight variant of equation 5 from the paper, noting that the total distance S_{tot} is twice the acceleration distance S):
   $$v = \sqrt{(v_0^2 + aS_{tot})} \quad (4)$$
   b) The number of steps for acceleration is now re-calculated as the integer part of half the value of S_{tot}:
   $$S = \text{int}\left(S_{tot}/2\right) \quad (5)$$
   (convert to **`long`** rather than **`int`** to avoid overflow)

3. After the end of the **`if`**-block mentioned above, the initial step interval (in microseconds) is calculated as p_1 from equation 17 in the paper and is a global variable (for compatibility with the Arduino code) so it can be accessed elsewhere:

$$p_1 = \frac{F}{\sqrt{v_0^2 + 2a}} \quad (6)$$

where F is the number of time measurements or "ticks" per second. In the C program we are using milliseconds so F is defined as 10^3; the Arduino sketch however uses microseconds (when using a timed loop) so in that case F is 10^6.

4. The minimum step interval p_s is given by equation 18 in the paper and is a global variable so it can be accessed elsewhere:

$$p_s = \frac{F}{v} \quad (7)$$

5. The constant multiplier R is calculated from equation 19 in the paper:

$$R = \frac{a}{F^2} \quad (8)$$

All of the above pre-computation tasks involve operations which are mathematically tedious such as division, calculation of square roots etc., as well as less onerous tasks such as multiplication and addition. Think why we are not worried about the time taken by these.

6. It is now time to calculate the step rate at each step within the section labelled "On-the-fly step calculation code". The update of the step interval p for each step (in **`computeNewSpeed()`**) is changed from equation (2) above to the "best approximation" theory contained in the paper:
   a) Define a temporary variable m. If in the acceleration phase :
   $$m = -R \quad (9)$$
   Or if the deceleration phase:
   $$m = R \quad (10)$$
   Or if the constant speed phase:
   $$m = 0 \quad (11)$$
   b) Define temporary variable q:
   $$q = m \times p_{old} \times p_{old} \quad (12)$$
   (definition following equation 22 in paper; note that p_{old} is simply the existing value of p in **`computeNewSpeed()`**)

c) Now p is updated as follows (equation 22 in paper)

$$p_{oew} = p_{old}\left(1+q+1.5\times q\times q\right) \quad (13)$$

d) Finally, if p is greater than p_1, set it equal to p_1.

7. Now run the program with the same parameters as previously, and again plot the graph of velocity and acceleration vs. time in Excel. You should find that it gives a reasonable approximation to a linear acceleration and deceleration profile.
8. The changes above have also been implemented in the LeibRampStepper.ino Arduino sketch. If you run your code on an Arduino Mega, you should see the LED flash at a steadily increasing speed, then at constant speed, then at steadily reducing speed.
9. Note that the code implemented in the C program only causes the stepper motor to run in one direction, even though the rest of the program satisfactorily takes account of the direction of movement of the motor via the variable **`direction`** (which is **`true`**, denoted **`FWDS`**, for running forwards and **`false`**, denoted **`BWDS`** for running backwards). In fact, the stepper driver module will run the motor forwards when the line connected to pin **`dirPin`** is high, and backwards when it is low. Note the two lines of code which have been added to the function **`moveOneStep()`** to implement this.

OPEN LOOP CONTROL EXPERIMENT

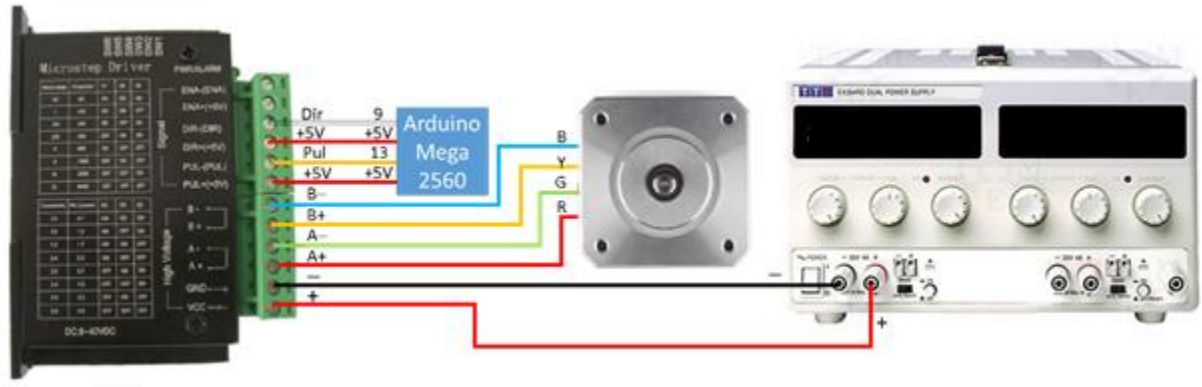


Figure 20 Logical connections to control the stepper motor

1. Leave the power supply settings on 12V, 0.5A.
2. Make the following connections:
    a. **CONN-E** (on Arduino Shield) with the "**Stepper Motor Driver Control**" connector (see Figure 3 a few pages above – this is the plug/connector near to top of the Arduino, WITHOUT the red tap wrapped around the cables)
    b. "**Stepper Motor Driver Control – A**" to **Stepper Motor Driver** (see Figure 21 below)
    c. "**Stepper Motor Supply – B**" to **Stepper Motor Driver** (see Figure 21 below)
    d. "**Stepper Motor Power Supply – C**" to **Stepper Motor Driver** (see Figure 21 below)
    e. **Main Power Supply** (bottom right of the lab kit base board) using the **Power Supply Leads**
3. **Get a demonstrator to check your wiring!**
4. Make the following connections to read the signal on the oscilloscope:
    a. **DIR-** on Stepper Motor Driver with **CH1 Signal** (**Red** crocodile clip) via a jumper cable
    b. **PUL-** on Stepper Motor Driver with **CH2 Signal** (**Red** crocodile clip) via a jumper cable
    c. **GND** on Stepper Motor Driver with **CH1/CH2 Reference** (**Black** crocodile clip) via a jumper cable
5. Connect the Arduino to your PC/Laptop using the USB cable. You should see the light appear on the Arduino. This indicates there is 5V power from the USB.

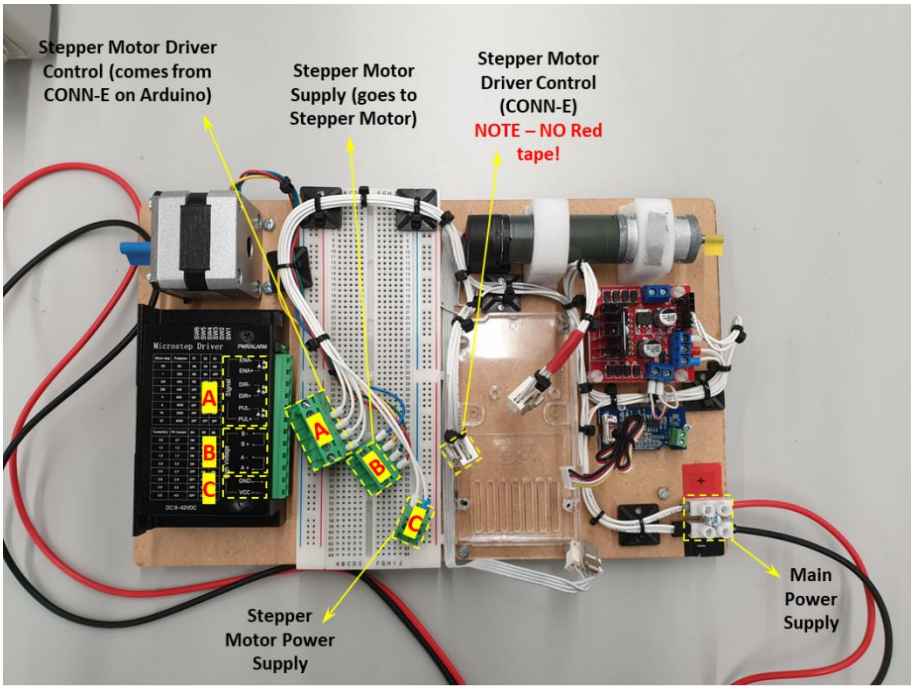


Figure 21 Hardware setup for Lab-2 Exercise-2

6. Load the program **SimplisticRampStepper.ino** into the Arduino IDE.
7. Ensure that the correct port and board type are set by checking **Tool | Board** and **Tools | Port**.
8. Now compile and upload your program. Open the Serial Monitor and check the box in the bottom right-hand corner of the monitor reads "Both NL and CR" to ensure that your program will understand when a line of text has been sent.
9. Enter a move to 5000 steps. **The ramping seems to be very slow, so will the motor manage to complete the move? Does it manage to keep up with the required acceleration?**
10. Now load, compile and upload **LeibRampStepper.ino**. Test with movements in the positive and negative directions and observe the signals on the scope.
11. **What happens to the pulse width as the motor speeds up and slows down?**
12. **What happens to the pulse spacing?**
13. **Does the motor keep up with the desired movements?**
14. **Is something interfering with the smooth movements of your motor?**
15. When you have completed the experiment, please proceed with the shutdown procedure:
    a. Disconnect the Arduino USB cable to the PC/laptop (this removes 5V supply and immediately mitigates against any accidental shorting).
    b. Switch off the PSU, disconnect the power supply leads from the lab kit using the screwdriver.
    c. Disconnect the 3 connectors (A, B, C) from the Stepper Motor Driver.
    d. Disconnect all jumper wires you used for oscilloscope probing.
    e. Disconnect all the connectors that you plugged in on the Arduino Shield (CONN-A to CONN-E).
    f. Remove the Arduino + Arduino Shield from the platform and put it inside the Lab Kit container.
    g. Replace all components in the Lab Kit container as you found it.
    h. Be careful when shutting the lid of the container, make sure no cable gets trapped.

Hope you enjoyed the session! Please work on your lab exercise and submit on Thursday 7th December by 15:00

COURSEWORK SUBMISSION

Well formatted and organised submission: A zip file containing the answers to your questions, ideally as a PDF, and the two Excel files created in the 'Open Loop Control Programs' section. **[10 marks].**

Questions (please limit each answer to a single side of an A4 page – maximum size of submission should not be more than 8 pages):

1. Explain what happens as you increase the gain in the proportional control situation. **[10 marks]**
2. Explain the effect of introducing derivative action to the situation with a gain of 0.02. **[10 marks]**
3. Explain why there are two tests for the control and print loops in the loop() function in ProportionalClosedLoop.ino. What would the effect be if the same variable was used for prevMillisControl and prevMillisPrint? **[5 marks]**
4. Explain how the PID library is used to implement the PID control in the PIDClosedLoop.ino program, including where the PID algorithm is executed and where the new value to be output to the motor is calculated. **[10 marks]**
5. Explain why, in LeibRampStepper.ino, we are not concerned about the presence of mathematically complex operations and functions in the section of `loop()` where parameters such as *R* are set. **[10 marks]**
6. Explain **very briefly** why, by contrast, we have gone to considerable effort to avoid mathematically complex operations and functions in `computeNewSpeed()`. **[10 marks]**
7. What happens when you try to run the stepper motor too fast? **[5 marks]**
8. Create the Excel spreadsheets with data from running the two stepper driver programs as instructed in the 'Open Loop Control Programs' section and plot the graphs of velocity and acceleration vs. time for both. **[10 marks]**