Department of Mechanical, Materials and Manufacturing Engineering

The University of Nottingham

Computer Engineering and Mechatronics MMME3085

Laboratory Exercise 2: Motion Control (preparatory work)

---

1 Objectives

The objectives of this exercise are:

- To give some practical experience of how servomotors are used in closed loop mode in order to achieve accurate motion control
- To give experience of using objects in the form of Arduino libraries
- To give experience of how ramping of step rates can be achieved without the use of division.

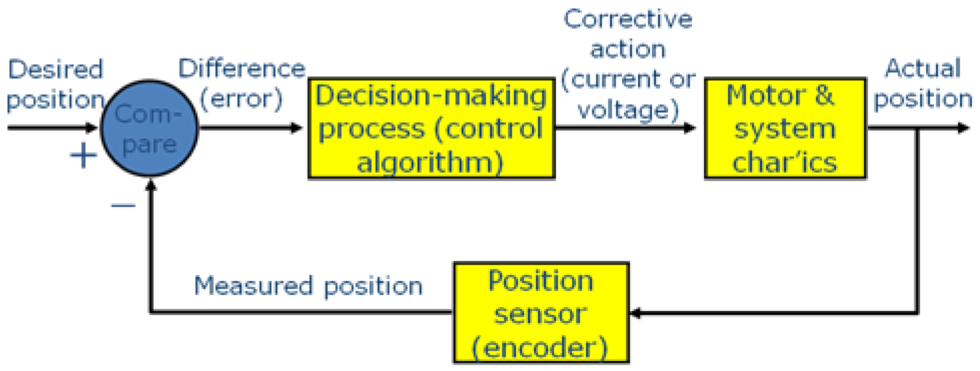
2 The control loop in the context of a servo mechanism

Fundamental to the operation of a servomotor is the concept of closed loop control – this was covered mathematically in an elementary manner in MM2DYN (MMME2046), and forms the basis of more advanced study in MM3CAI (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 follows:


Desired position
+
Com-pare
−
Difference (error)
Decision-making process (control algorithm)
Corrective action (current or voltage)
Motor & system char'ics
Actual position
Measured position
Position sensor (encoder)

Within your preparatory work you will be going from "open loop control" (where you vary the motor signal directly, with no attempt at controlling position) to "closed loop control", where you will implement an extremely simple control algorithm (proportional control) and incorporate a VI implementing a more sophisticated (PID) algorithm.

You will also be examining the software used to control 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.

3 A hint of C++: classes and objects to implement libraries

The programming language taught by Louise Brown in the software engineering part of the module is ISO C, sometimes known informally as "pure C" to distinguish it from an enhanced version of C known as C++. The main difference between the languages is that C++ is designed to be used for object-oriented programming, in which data and functions are no longer separate concepts but are combined in the form of "objects", which are instances of user-defined variable types known as classes. A `class` is essentially a `struct` with member functions as well as member data. For instance, the encoder sketch you used in Lab 1 uses the SPI class. This ad-hoc use of classes in what is essentially a C program structure is a characteristic of "the Arduino way of programming" and is not generally regarded as good practice, not least because pure C simply does not allow this. By contrast, C++ programs are often written using classes and objects throughout, following an approach known as "object oriented programming" which is not covered in this module.

4 Preparatory work

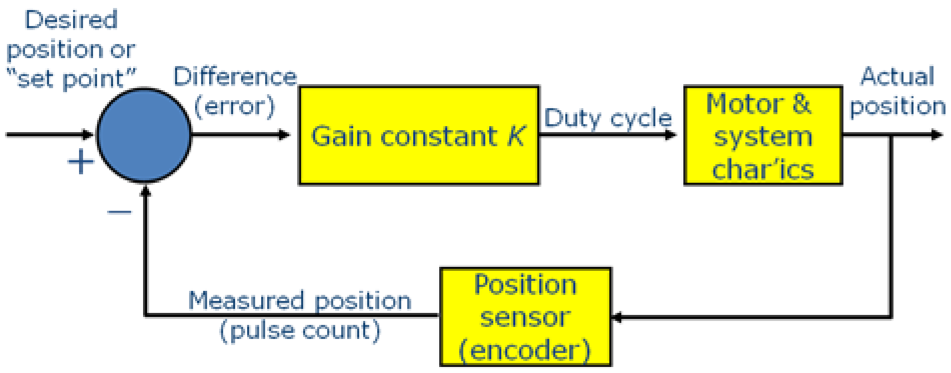
This involves creating **two separate Arduino sketches** which implement closed loop control: one involves a "home-made" proportional-only control algorithm; the other makes use of a ready-written "PID" (proportional-integral-derivative) library object. **You are strongly advised to make an early start on the preparatory work** so you do not run out of time to ask for help if required

4.1 Exercise 1: Closing the Loop: Proportional Control

1. Download a slightly modified version of the open loop PWM and position sensing project which you used in Lab 1, now called **MotorControlSkeleton2.ino**: this may be downloaded from Moodle in the Lab and Programming Exercises section. In brief, this drives the motor with a PWM signal of varying duty cycle and of positive or negative polarity – the present version differs very slightly from the version used in Lab 1 in that the state machine and internal counter code has been stripped out and some variables renamed. If you have an Arduino Mega, try compiling and running the example, but remember to set the Serial Monitor to 9600 baud rate and to send "Both NL & CR" on pressing return. If you type a number

representing the percentage PWM value, you should see the LED glow brightly or dimly accordingly.

2. You will notice how the printing takes place within a timed loop which cycles every 1000 milliseconds, and calls the function **`printLoop()`**. Just as you did within the introductory "blink without delay" exercise, create another timed loop and loop function, called the control loop, with a period of 20 ms. We suggest you name this function **`controlLoop()`**.
3. Create a global variable to store the measured position from the encoder.
4. Within this loop, obtain the value of measured position from the encoder (as counted using the LS7366R counter) just as is done in the print loop, but store it in the global variable you have just created.
5. Create a global constant called **`Kp`** representing the proportional gain, and initialise it to a value of 0.02.
6. At this stage, check that your sketch compiles correctly and correct any problems making use of any compile error message. If you have an Arduino Mega you should still be able to observe how the LED changes in intensity as you change the entered value between -100 and 100.
7. Now change the way the program behaves as follows. Create a new global floating-point variable called **`positionSetPoint`** and arrange so that this variable, instead of the variable **`percentDutyCycle`**, is set by the user input from the serial monitor.
8. Within your control loop function, implement the proportional closed loop control algorithm shown in the diagram below by setting the value of **`percentDutyCycle`** to the proportional gain multiplied by the value of error (the difference between the position set point and the position measured by the encoder; make sure you get the difference the right way around as shown in the diagram).


Desired position or "set point"
Difference (error)
+
−
Gain constant K
Duty cycle
Motor & system char'ics
Actual position
Position sensor (encoder)
Measured position (pulse count)

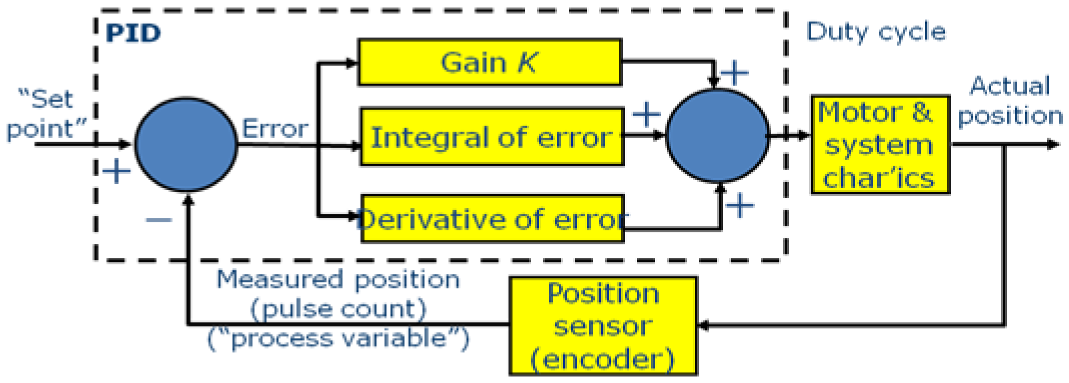
.

9. Within the control loop function, remember to call the function to update the motor speed with the updated value of **`percentDutyCycle`**.
10. Enhance the contents of **`printLoop()`** to print out the setpoint and the error (which you will need to calculate here again) as well as the measured position.
11. Try compiling your code; ensure there are no errors. If there are, look at the error messages and make the necessary corrections. If you don't know

what's wrong, please ask for help (so leave enough time for sorting out problems!).

12. Save your sketch as **ProportionalClosedLoopXXX.ino**.
13. Save an additional copy of your sketch and call it **PIDClosedLoopXXX.ino** ready for the next exercise (so you don't overwrite your ProportionalCloseLoopXXX.ino sketch and end up without that work!).

4.2 Exercise 2: Closing the Loop: Proportional-Integral-Derivative (PID) Control



1. 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.

2. 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 address 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). Two other aspects also need to be set: the PWM duty cycle can be set from −100% (implying the motor in reverse) to +100%, and the PID controller needs to be informed of the actual control loop interval, so you will actually need to include the following three lines of code within **`setup()`**, noting that the percentage limits are (or should be) already set as constants in your program:

```
myPID.SetOutputLimits(minPercent, maxPercent);
myPID.SetSampleTime(controlInterval);
myPID.SetMode(AUTOMATIC);
```

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.

3. Now alter your **PIDClosedLoopXXX.ino** sketch as follows:
   - Defining two additional global variables **`Ki`** and **`Kd`**; at present set these to zero.
   - Define the global PID object so its input is the measured position, the output is the PWM percentage, and the set point is the position set point as before.
   - Initialise the set point to zero in **`setup()`**
   - Remember to make the PID controller active. Two other aspects also need to be set: the PWM duty cycle can be set from −100% (implying the motor in reverse) to +100%, and the PID controller needs to be informed of the actual control loop interval, so you will actually need to include the following three lines of code within **`setup()`** noting that the percentage limits are (or should be) already set as constants in your program:

```
myPID.SetOutputLimits(minPercent, maxPercent);
myPID.SetSampleTime(controlInterval);
myPID.SetMode(AUTOMATIC);
```

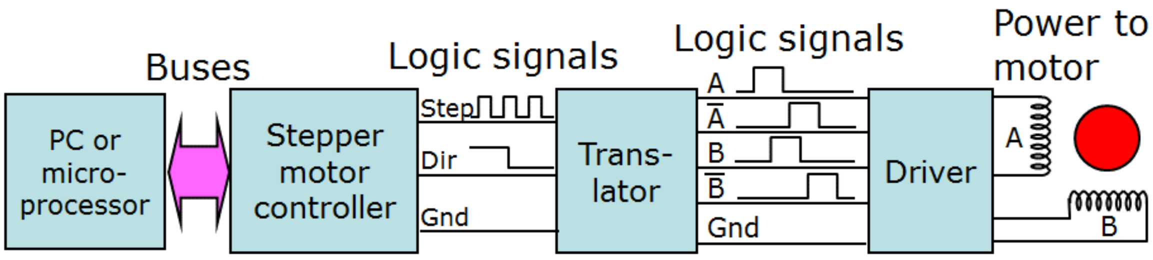
   - Replace the proportional action in the control loop with the PID action in the same way as in the PID example.

4. Save **PIDClosedLoopXXX.ino** and try compiling. If there are problems, please seek help, allowing enough time to do so.

4.3 One step at a time

4.3.1 Overview of stepper motors and ramping

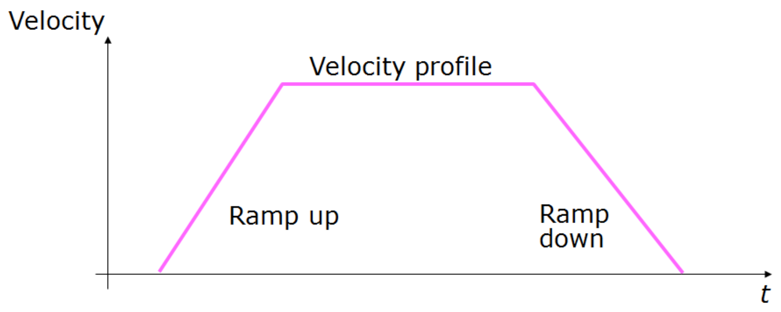
Part of the laboratory exercise involves controlling a stepper motor. These devices are motors which operate under (usually) open-loop control and produce a fixed angle of rotation for each increment (step) of their movement. The basics of stepper motors were covered in Electromechancial Devices and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



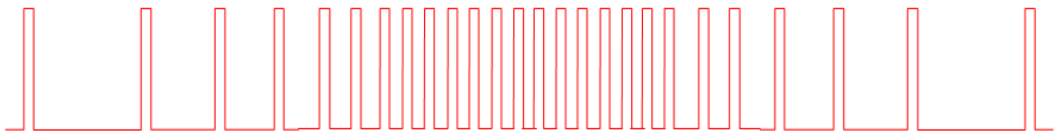
In the present case, the stepper motor controller is a combination of a timer and various input-output lines rather than a separate device. Stepper motors are driven by circuitry (translator and driver –covered in the Electronics in Mechanical Engineering laboratory as an example of digital logic) which 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.

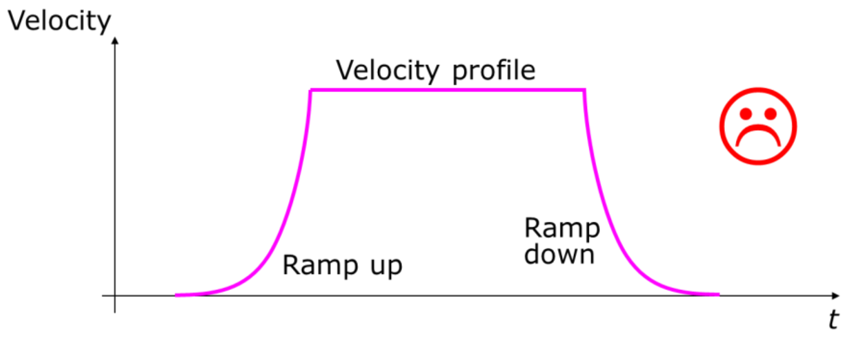


This profile involves generating trains of pulses which are initially widely spaced in time, then gradually become more closely spaced as the motor is accelerated, and eventually become more widely spaced again and cease altogether as the motor slows down to a halt.



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 that 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 the 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.



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.

4.3.2 Existing program – a simplistic approach that nearly works

The scenario for your exercise is that an Arduino sketch has been written by a competent programmer to implement a ramping profile. Let's assume that the writer had a basic understanding of stepper motors but assumed that the generation of a ramping profile is a trivial exercise (which of course it isn't)

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

and has implemented a ramping profile which works, but not very effectively.

The starting point for your sketch is **SimplisticRampStepper.ino** which you can download from Moodle; however, as debugging an Arduino program is difficult, you will develop your code first within a specially-provided C test program which also 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!)** Before starting work on the updated algorithm first take a look at the **SimplisticRampStepper.c** program. As set 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. However, the code includes some "hidden" options for using interrupts – you may safely ignore these for the present time, but don't be confused by (or try to delete) lines such **`#ifdef HARDWARETIMING`** as you will use these in the lab.

Firstly, try running SimplisticRampStepper.c 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.

At this, 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 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:

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

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

4.3.3 Replacing the simplistic approach with one that works much better

Now use the **LeibRampSkeleton.c** program to develop the new algorithm. This is very similar to the SimplisticRampStepper.c program but has the simple algorithm stripped out and comments to indicate where the changes outlined below should be inserted.

The minimum stepping speed of the motor, the maximum permissible speed **`maxPermisSpeed`** and the acceleration **`maxAccel`** are already defined as global constants so you don't need to redefine these. The number of steps for the acceleration phase **`accelSteps`** is also specified; this is no longer going to be constant so is not defined as a constant.

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}.

<u>However, to replace the simplistic approach above with the approach described by Eiderman you will need to make some changes</u>. Note that all your arithmetic, constants and variables must be 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) should your step interval *p* be converted to a **`long`** integer.

Starting with the changes to be made in function **`main()`** forming the section labelled "Pre-computation code" (the comments STEP 1 etc in the skeleton 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:

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

   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*):

      $$v = \sqrt{(v_0^2 + aS_{tot})} \qquad (4)$$

   b) The number of steps for acceleration is now re-calculated as the integer part of half the value of S_{tot}:

      $$S = \mathrm{int}\left({S_{\mathrm{tot}}}\big/{2} \right) \qquad (5)$$

      (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 should be made a global variable so it can be accessed elsewhere:

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

   where *F* is the number of time measurements or "ticks" per second. In the C program we will be using milliseconds so *F* is defined for you as 10^3; the Arduino sketch however uses microseconds (when using a timed loop) or system clock ticks (which occur at 16 MHz, for use with the interrupts and timer counters) so in those cases *F* is 10^6 or 1.6×10^7 respectively.

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:

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

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

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

   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 change the calculations of 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()`**) can now be 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 :

      $$m = -R \qquad (9)$$

      Or if the deceleration phase:

      $$m = R \qquad (10)$$

      Or if the constant speed phase:

      $$m = 0 \qquad (11)$$

      (see definition following equation 20 in the paper; noting that the three situations are already correctly identified in **`computeNewSpeed()`** but you will need to add your code to them).

   b) Define temporary variable *q*:

      $$q = m \times p_{\text{old}} \times p_{\text{old}} \qquad (12)$$

      (definition following equation 22 in paper; note that p_{old} is simply the existing value of *p* in **`computeNewSpeed()`**)
   c) Now update *p* as follows (equation 22 in paper)

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

Note that this will be represented in your program as a good example of "replace existing value of p with a new value calculated from old value of p" – as before, you don't need separate variables for p_{old} and p_{new}, they are the old and new values of p.

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

7. Save your code as **LeibRampStepperXXX.c**. Compile your code and check for issues such as undefined variables.

8. Now try running your program with the same parameters as previously, and similarly 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. As a check, the first four values of step time (p) should be 218.218, 188.528 and 157.245 ticks (here representing milliseconds).

9. Once you are happy that your code is working correctly, you should be able to incorporate your changes into the Arduino sketch **LeibRampStepper.ino**. Simply replace the pre-computation section of the Arduino sketch with the pre-computation section of the C program, and similarly replace the on-the-fly calculation section of the Arduino sketch with that from the C program. Save it as **LeibRampStepperXXX.ino** then compile and run it.

10. 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. If you are not able to get your code to work, please ask for help (in good time!). Note that the code will only work on an Arduino Uno if **`USEINTERRUPTS`** is not defined as we use more timers than are available on the Uno.

11. Finally, it has been noted that the current version of the 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. Add two lines of code to the function **`moveOneStep()`** to implement this. (Indeed, you can do it in one just line of code if you really want!).

12. When you are satisfied that it is correct, save it.

5 Submission

You will need to submit your preparatory work online. Zip up **ProportionalClosedLoopXXX.ino**, **PIDClosedLoopXXX.ino** and **LeibRampStepperXXX.c** and **LeibRampStepperXXX.ino** along with the spreadsheet with graphs of step rate vs. time, and submit this to the appropriate Moodle submission box no later than **18:00 on 22nd November 2021**.

You will need to bring your programs to the laboratory

L. P. Brown
Revised 3 November 2021

I. A. Jones
5 November 2019