Please do not remove this manual from the lab. It is available via Canvas
DEMO NOTES: *This manual contains demonstrator notes in blue italics*

Servo control: Ball on beam

Aims of this experiment

- Implement a digital feedback system to balance a ball on a beam.
- Investigate the effect of PID parameters and filtering on the transient response of the system.

Balancing a ball at a chosen location on a tilting beam is a classic problem in feedback control, because the system is inherently unstable (the ball runs away unless the beam is perfectly level). This experiment uses a feedback loop programmed in Python on a microcontroller to stabilise the ball.

1 Position sensing

1.1 Linear potentiometer sensor

The position of the ball along the beam is sensed by a linear potentiometer, which you can think of as a one-dimensional resistive touchscreen. A resistive track runs the length of the beam, and a potential gradient is established along it. The weight of the ball pushes a flexible conductive layer into contact with the track, so this second layer acquires the potential of the track at the location of the ball. The circuit is shown in Figure 1.

From the figure, work out how the voltage measured at **`Y12`** depends on the position of the ball, including the possibility that the ball is not on the track. Hence explain the function of the resistors (the 2.2 kΩ is for protection and is not critical to the behaviour). **Q**

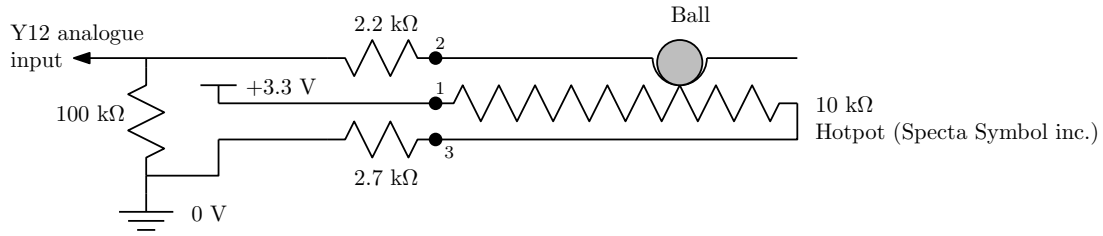

Figure 1: Resistive position sensor

DEMO NOTES: *If the ball is not present, Y12 = 0 V because the 100 kΩ to ground is its only connection; this is the purpose of the 100k. With the ball present, the voltage at Y12 ranges from 3.3 V at one end of the track to 3.3 * 2.7k/(10k + 2.7k) ∼ 0.7 V. The 2.7k is to ensure that the ball-on voltage can never fall to zero, so it can never be confused with the ball missing. The 2.2k is to prevent too much current going through the track if it is wrongly connected. It also interacts with the 100k to slightly reduce the other voltages discussed, but this doesn't really matter.*

1.2 Sensor readout

The circuit of Fig. 1 is built into the sensor cable, so it is only necessary to plug in the sensor so that the red wire goes to +3.3 V and the brown to Y12 on the Pyboard microcontroller. Also connect a 7-segment display, as in Fig. 2, and a 1 kΩ / 1 μF filter, as in Fig. 3, to enable an analogue output from the microcontroller. The completed circuit is shown in Figure 4.

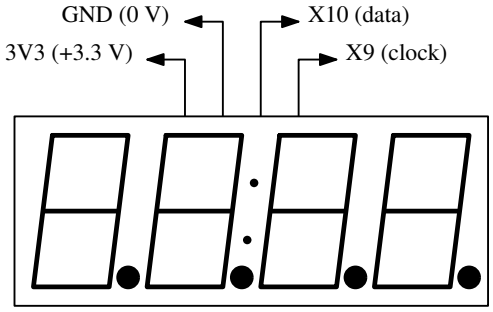

Figure 2: Seven-segment display connections

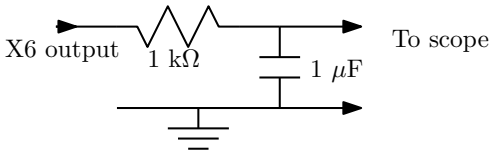

Figure 3: Output filter for the pulse-width modulated digital-to-analogue converter

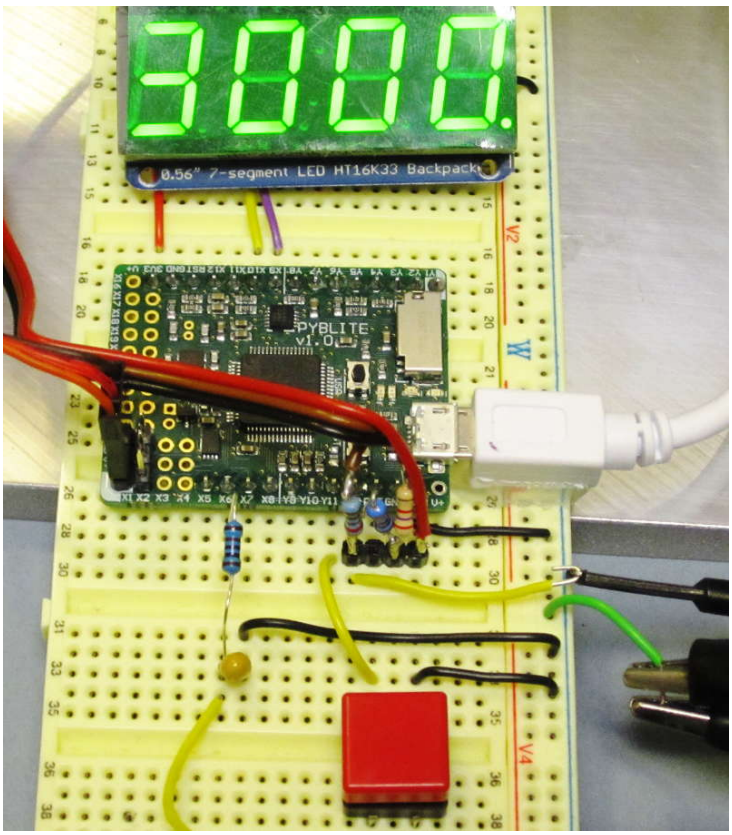

Figure 4: Completed circuit

Connect the USB power to the microcontroller, and using the MuNu environment, run program **`bb1.py`** on the microcontroller. This program reads the voltage from the sensor using an analogue-to-digital converter (ADC), then copies the ADC reading to both the display and a digital-to-analogue converter (DAC) that drives pin **`X6`**. Use one scope channel to monitor the **`Y12`** input and the other the **`X6`** output, at the capacitor.

Manually raise and lower the beam to make the ball roll, and document the scope traces produced. Take note of the approximate ADC readings for key positions of the ball (the ADC noise will make it impossible to get exact values). You will probably see a good deal of noise in the trace, because the ball does not stay in firm contact with the track while it rolls. The first requirement of a feedback loop is a reliable sensor, so the first job is to write some code that recognises and ignores bad data. DEMO NOTES: *both traces should show a lot of glitches to zero volts, as the ball skips on and off the track. There will be fewer glitches on the DAC output, because the microcontroller is sampling the sensor a lot less frequently than the scope.*

1.3 Sensor cleanup

Modify the loop in **`bb1.py`** so that if the ball loses contact with the beam, the position variable **`pos`** does not get modified, but remembers its previous value, ignoring the bad data. Check if your code is effective in removing the glitches in the sensor reading, as judged from the **`X6`** output. DEMO NOTES: *place the*

`pos = sensor` inside an `if sensor > 500` condition (or similar), which requires the ball to be in contact. The DAC output should be a good deal cleaner (there can still be occasional glitches down to levels other than zero). This is done in `bb2.py`, but not mentioned in the student manual

2 The control problem

2.1 Manual feedback

Using a mark roughly half-way along the beam, try to manually roll the ball to the location of the mark by manipulating the angle of the beam. Then knock the ball away from its position and try to bring it back. Make an oscilloscope recording of your attempt, using the **`X6`** DAC output as in section 1.3. This is the process that we want the microcontroller to accomplish. DEMO NOTES: *Make sure students are getting nice transient recordings, using a slow timebase (1 sec, roll, or similar) on the scope. When under software control, they need to keep track of which parameters go with which recording.*

2.2 PID control

The general technique is to continuously calculate the *error* between the actual and target position, and then drive the system with *negative feedback* so as to reduce the size of the error (positive feedback would increase the error). Mechanical systems using negative feedback are known as servomechanisms (latin *servus* = slave). The difficulty lies in how to tailor the feedback to the dynamics of the system (for example, to allow for the inertia of the ball). A flexible method is PID (proportional-integral-differential) control. In PID control, in response to a time-varying error $e(t)$, the output U that the controller feeds back to the mechanical system is the linear combination

$$U(t) = K_P e(t) + K_I \int e(t)dt + K_D \frac{d}{dt} e(t) \tag{1}$$

where K_P, K_I and K_D are constants selected for a particular system. These are normally tuned by hand, based on some physical intuition. This is what we will now do.

2.3 Scale and offset

In our system, the units of U and e in equation 1 are those of the sensor ADC, as noted in section 1.2, whereas the units used by the cam motor are in the range -90 to +90. Moreover, a cam motor setting of zero may not correspond to the beam being level.

Stop the program (ctrl-C), and try typing settings like **`servo1.angle(30)`** until you find the best setting that makes the beam level (so the ball stays still), e.g. 25. Then calculate the scale factor that takes us from the full sensor range to the full cam range, e.g. (90 - (-90)) / (3860 - 610) = 0.055. Use these in the new program **`bb3.py`** to convert **`U`** to the required cam setting in the line
`cam_angle = 25 - 0.055 * U`, but using your own numbers.

2.4 Proportional control

Using **bb3.py**, which contains the beginings of PID control, set the **target** position to the middle of the beam (e.g. $(3860-610)/2 \approx 2200$), and set a reasonably large value such as 2 for the "proportional gain" K_P. Run the code, and see how it performs at bringing the ball to the target position. In particular, give the ball a little displacement from the target position and record its subsequent movements using the DAC output and the scope, as in section 2.1. The behaviour after a sudden upset is known as the transient response, and it is a good way of characterising the system. It is probably not very good at the moment.

You should find that too large a K_P makes the system overcorrect, causing increasingly large oscillations about the target position. Try to find a value for K_P that makes the system converge on the target position. There will still be oscillations, but of decreasing amplitude. Record your results for a range of K_P values. What is the drawback of small values?

DEMO NOTES: *$K_P < 0.5$ was stable in the case I tried. Bigger should give oscillations that get larger on each cycle, until limited by the ball hitting the end of the track. A little smaller than this value (0.4), and the oscillations slowly decay until the ball settles. But small K_P makes the system response very sluggish, and not very discriminating about where the ball settles (when $K_P = 0$, it doesn't care at all, of course)*

2.5 Proportional and differential control

You should be able to show that the position x of a ball rolling on a plane inclined at a small angle α in gravitational field g is given by [1] **Q**

$$x(t) = \frac{5}{7} g \iint \alpha(t)\, dt\, dt \tag{2}$$

DEMO NOTES: *Force down the slope = $mg\sin\alpha \approx mg\alpha$. Effective inertial mass = $m + I/r^2$ where $I = \frac{2}{5}mr^2$ for a ball of radius r, (if you forget the rotation, then the $\frac{5}{7}$ becomes 1). Then $mg\alpha = \frac{7}{5}m\frac{d^2x}{dx^2}$, and integrate twice for x.* The integration amounts to a delay in the response, which gives rise to the overshoot. Adding a differential term to the feedback provides some anticipation ("how fast is it going?") that helps counteract this.

Program **bb4.py** adds a derivative term to the feedback. Explain how the **Derror** term is calculated as an approximation to $\frac{de}{dt}$. Starting with, e.g. $K_D = 0.02$, investigate how an increasing differential term improves the transient response (try 0.05, 0.1, 0.2 ...). The motor will jitter a lot more, because the differential term is sensitive to noise (why?). We will improve this and re-optimise in section 3, so do not spend too long on the remainder of this section. DEMO NOTES: *If students get stuck here, playing with loop constants, try to get them to leave the optimising until they have some filtering, in the next section.*

With a non-zero K_D term, you will be able to increase the K_P term to increase the response speed without paying the price of a massive overshoot. Explore optimising K_D and K_P for a fast transient recovery with minimal overshoot. Keep note of the parameters you try, and make scope recordings of some significant ones.

DEMO NOTES: ***olderror** remembers the error from the previous pass through the loop, so **Derror = (error - olderror)/ dt** is $\frac{\Delta e}{\Delta t} \approx \frac{de}{dt}$. Starting with $K_P = 0.3$, $K_D = 0.05$ already reduces the overshoot, at the expense of some jitter, arising from noise. Increasing K_D to 0.1, 0.2, 0.4 helps a bit*

[1] 5/7 because the ball rotates as well as translates

more, but won't reduce the settling time much, and makes the jitter worse. Noise from the ADC may be quite small in magnitude, but jumps up and down rapidly, hence its differential is large. Or to look at it another way, noise is broadband, and its higher frequency components get a large prefactor (the frequency) when differentiated.

3 Filtering and optimisation

Noise in the position measurement causes the motor to jitter, which will cause wear. Increase the **`pfilter`** coefficient in **`bb4.py`** to apply some filtering to the position measurement. Explain roughly how the line
`Fpos = pfilter * Fpos + (1 - pfilter) * pos` makes **`Fpos`** a smoothed-out version of **`pos`**. This should quieten the motor, but there is an inevitable trade-off between filtering and delay, which affects the feedback, so try a few filtering levels. Adjust the PID parameters to compensate for a little delay, and optimise the response, as suggested at the end of section 2.5. Document your best transient response. Note that the meaning of "best" depends on the application: sometimes the fastest approach to within (e.g.) ±5% of the final value is important; in other cases overshoot must never occur.
DEMO NOTES: *By adding some of the previous values to the current value, **`Fpos`** forms a weighted rolling average of **`pos`** values. Technically it's a single-pole IIR filter, and is equivalent to the low-pass filtering you'd get with a resistor and a capacitor. Too much averaging (pfilter too close to 1) adds too much delay, spoiling the reaction time of the feedback loop.*

4 Suggestions for further experiments

- A good way to show off the response of a system like this is to have it switch backwards and forwards between two targets, so you can see the transient response on each move. If you connect a switch between Y11 and ground, then in the main loop of **`bb4.py`** you can set the target depending on whether the switch is pressed:
  `if switch.value() == 0:  target = 1500; else ...`
- What happens with positive feedback, i.e. if you reverse the sign of U?
- Would an integral term be of any help in this system?
- An improvement of your own.

DEMO NOTES: ***`bb5.py`*** *is a finished program with the switch added. Positive feedback can diverge to + or - infinity, in practice driving the ball to one or other limit. There is so much integration in the system (Eq. 2) that a K_I term is not critical, except for one thing: with zero error, a PI system has no way of producing an output; yet the beam has to be kept level. We are doing this by the constant term hardwired into **`cam_angle`**; if it is wrong, the ball will be kept slightly away from the target in order to generate the necessary output. A small K_I would reduce this long-term error to zero. A new variable is needed in the loop, to keep track of the integrated (summed) error.*

Checklist

Your lab book should contain:

- Diagrams of circuits you built and listings of code that you wrote.
- Waveforms, parameter values, observations and interpretations of your investigations, including those prompted by this manual.
- Enough detail to enable someone to quickly pick up the thread of what you did and observed.

References

- http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/ Starting from the naive approach to PID control used here, develops more professional control software.

MSC 2017-11-12