Laboration 1: PID-controls

Sensors and Sensing

Michael Floßmann, Tom Olsson

November 23, 2015

All code for this exercise can be found at
https://github.com/tgolsson/sensors-arduino-lab1

Code listings

List of Figures

List of Tables

1 Theory and motivation

Control algorithms are important to create predictable, safe, and reliable operation in robotics applications. Two important algorithms/controllers for this purpose is the *PID-controller* and the *mimimum jerk trajectory.*

1.1 PID controller

PID in the name PID-controller is short for *Proportional-Integral-Derivative*-controller. As this implies, the controlling signal is based on a proportion of the current error, the previous error, and the rate of change of the observed error. The mathemathical formulation of this is shown in (1) on the following page.

Let:

$e(t)$	be some error measurement between current state and preferred state
K_p, K_i, K_d	be the respective weights for the proportional, integral and derivate terms
$u(t)$	be the output signal at time t

Then:

$$u(t) = K_\mathrm{p} \cdot e(t) + K_\mathrm{i} \cdot \int_0^t e(\tau) \cdot \mathrm{d}\tau + K_\mathrm{d} \cdot \tfrac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{1}$$

1.2 Minimum jerk

The minimum jerk equation is an important part of creating smooth control. When a rotating actuator such as a motor starts, both the rotor and the stator will be at rest. The momentum generated by the motor can therefore cause movement in either part. As this can create an unwanted jerk while the rotor accelerates, it is important to accelerate slowly so that the stator remains at rest in relation to the reference frame. This can be achieved by the *minimum jerk equation* shown in (2).

Let:

x_i, x_f	be the initial and final states
t, T	be the elapsed time since the action started, and the preferred total time for the action
$x(t)$	be the estimated state at time t

Then:

$$x(t) = x_\mathrm{i} + (x_\mathrm{f} - x_\mathrm{i}) \cdot \left[10\left(\frac{t}{T}\right)^3 - 15\left(\frac{t}{T}\right)^4 + 6\left(\frac{t}{T}\right)^6\right] \tag{2}$$

The T parameter has to be estimated. If T is much larger than the actual time that is needed for the trajectory, the velocity will be very low, and if T is too low $x(t)$ will approach infinity unless $\frac{t}{T}$ is clamped to $[0, 1]$. However, this solution is not optimal. Instead, we choose to calculate the optimal time T_opt as follows.

For finding out the optimal time T_opt, we substitute:

$$\tau := \frac{t}{T} \tag{3}$$

$$(2) \Rightarrow \quad x(\tau) = x_\mathrm{i} + (x_\mathrm{f} - x_\mathrm{i}) \cdot \left(10\tau^3 - 15\tau^4 + 6\tau^6\right) \tag{4}$$

$$\tfrac{\mathrm{d}x(\tau)}{\mathrm{d}\tau} = (x_\mathrm{f} - x_\mathrm{i}) \cdot \left(30\tau^2 - 60\tau^3 + 36\tau^5\right) \tag{5}$$

(5) reaches its' maximum at $\tau = 0.5$ (proof trivial) with the value:

$$\left.\tfrac{\mathrm{d}x(\tau)}{\mathrm{d}\tau}\right|_{\tau=0.5} = \frac{15}{18} \cdot (x_\mathrm{f} - x_\mathrm{i}) \tag{6}$$

In order to make this term dependent on T, we must resubstitute (3) into (6).

$$(3) \Rightarrow \quad \dot{\tau} = \frac{1}{T} \tag{7}$$

$$\Rightarrow \quad \mathrm{d}\tau = \mathrm{d}t \cdot \frac{1}{T} \tag{8}$$

$$(5), (8) \Rightarrow \quad \left.\tfrac{\mathrm{d}x(\tau)}{\mathrm{d}t}\right|_{\tau=0.5} = \frac{15}{18} \cdot \frac{x_\mathrm{f} - x_\mathrm{i}}{T} \tag{9}$$

Now, if we state an optimal maximum velocity v_opt for the minimum jerk equation, we can calculate the optimal time T_opt for this velocity.

$$\Rightarrow \quad \left.\tfrac{\mathrm{d}x(\tau)}{\mathrm{d}t}\right|_{\tau=0.5} = v_\mathrm{opt} \tag{10}$$

$$(9) \Rightarrow \quad T_\mathrm{opt} = \frac{15}{18} \cdot \frac{x_\mathrm{f} - x_\mathrm{i}}{v_\mathrm{opt}} \tag{11}$$

2 Implementation

The purpose of this exercise is to implement a PID-controller using the minimum jerk trajectory, and use this implementation for both a *set-position* mode of operation, as well as a *set-velocity* mode of operation. An important part of this exercise is the tuning of the PID-parameters for either mode of operation.

2.1 Hardware and environment

The laboration is performed using an *Arduino Due* microcontroller, with the *Arduino Motor Shield R3*. These are programmed using Serial-over-USB; with the dedicated IDE. The version of the IDE used is 1.6.5. The exercise also includes usage of the *Robot Operating System* [ROS], version *Indigo Igloo*.

The motor used is the *Micro Motors RHE158 75:1 12V DC*, connected to the motor shield. As the USB-bus cannot supply enough power to drive the motor, an external 12V power adapter was used.

Measurements In order to ensure correct behaviour, the amount of steps per revolution as well as the maximum velocity of the motor was measured and compared to the datasheet [1],[2].

Table 1: Measured and expected motor characteristics

Characteristic	Datasheet	Measured
Steps/rotation	230.5	235.0
Steps/second	311.0	280.0
Rotations/second	1.35	1.19

The maximum speed in the data sheet refers to the motor speed without load. Since there was load present at the laboratory outset, the measured value of steps per second was used. The steps per rotation value was measured turning a varying amount of steps and seeing when one rotation was completed. This method is expected to be relatively uncertain, since there is a large possible error margin without proper measuring equipment. In order to improve the measurement, 10 rotations were made, and the result was verified twice.

The steps per second value for maximum speed was measured by applying a duty cycle of 100% to the PWM and programatically measuring the steps. Ten measurements were made for 10 seconds each, and the average steps per second was chosen as the speed value. This measurement is considered very exact and is used in the rest of this paper.

2.2 Position controller

The first part of the position is the callback for setting a target position. This code is shown in listing 1 on the following page. The code updates the mode of operation, and sets the start and end position for the movement. The integral term of the controller is also set to zero. Tests were made without setting it to zero, but this caused unreliable behaviour in some situations, such as when the target position was moved closer to the current position.

As can be seen on line 319, (11) from subsection 1.2 on the previous page is used to calculate the end time point. This ensures that we reach the maximum speed at $\frac{t}{T} = 0.5$. While this is suboptimal for long trajectories, it makes sure that T is realistic for shorter paths. If long trajectories will be the normal mode of operation; an approach with spline interpolation should be used instead of the minimun jerk equation.

Listing 1: `Set Position Callback`

```
307 void setPosCallback(const arduino_pkg::SetPosition::Request &req, arduino_pkg::
        SetPosition::Response &res)
308 {
309     // Start and end state of encoder
310     mc1->rf_ = req.encoder;
311     mc1->ri_ = enc1->p_;
312     mc1->r_ = enc1->p_;
313
314     // Reset integral
315     pid_mc1->I_ = 0;
316
317     // Start and end times
318     mc1->ti_ = micros();
319     mc1->T_ = micros() + (abs(req.encoder - enc1->p_))/(V_MAX)*(15.0/8.0)*1e6;
320
321     // Activate controller
322     mc1->active_ = true;
323     res.success = true;
324 }
```

The position controller is implemented to setup the parameters for PID and MJE. The code for this is shown in listing 2. The minimum jerk function is called with start-time, current time, and end-time, as well as start and end position. The output of this equation is then used to calculate the momentary error, as well as its derivative. These are then given to the PID-controller.

Listing 2: `Position Controller`

```
221 void positionControl(ControlStates* c_s, EncoderStates* e_s, MotorShieldPins* m_pins
        , PIDParameters* pid_p)
222 {
223     // Sliding sum for encoder speed
224     enc1->dp_ = enc1->dp_ * 0.99 + 0.01*(enc1->p_ - enc1->pp_)/(dT/TIME_SCALE);  //
        dT = 1000 us = 0.001 s
225     enc1->pp_ = enc1->p_;
226
227     // Early escape to prevent "almost-there" PWM-hum
228     if (abs(enc1->p_ - c_s->rf_) < TOLERANCE)
229     {
230         c_s->u_ = 0;
231         return;
232     }
233
234     // Set-point calculation
235     double setPoint = minimumJerk(c_s->ti_, (double)t_new, c_s->T_, c_s->ri_, c_s->
        rf_);
236     c_s->r_ = setPoint;
237
238     // Error and error derivative
239     double e = (c_s->r_ - e_s->p_);
240     double de = (c_s->e_ - e) / dT;
241
242     // Get PID output
243     double ut = pid(e, de, pid_p);
244
245     // Store for next cycle
246     c_s->e_ = e;
247     c_s->de_ = de;
248     c_s->u_ = ut;
249 }
```

2.3 Velocity controller

As with the position controller, an important part of the velocity controller is the callback for setting the target velocity. This code is shown in listing 3 on the next page. The only difference

to the position callback is the T parameter. This was tested to see the minimum acceptable time to go from full forward speed to full reverse speed, and 3 seconds seemed to be a reasonable value. As the difference in speed is 560 steps per second; this therefore becomes $\left\lceil \frac{3}{560} \right\rceil_4 = 0.006$.

Listing 3: `Set Velocity Callback`

```
333 void setVelCallback(const arduino_pkg::SetVelocity::Request &req, arduino_pkg::
        SetVelocity::Response &res)
334 {
335     // Start and end state of encoder
336     mc1->rf_ = req.ticksPerSecond;
337     mc1->ri_ = enc1->dp_;
338     mc1->r_ = enc1->dp_;
339 
340     // Reset integral
341     pid_mc1->I_ = 0;
342 
343     // Start and end time
344     mc1->ti_ = micros();
345     // 0.006  = 3 s / 560 tps -- acceleration constant
346     mc1->T_ = micros() + abs(mc1->ri_ - mc1->rf_) * 0.006*1e6;
347 
348     // Activate correct controller
349     mc1->active_ = false;
350     res.success = true;
351 }
```

The velocity controller is also very similar to the position controller, as shown in listing 4. As before, the minimum jerk function is called with start-time, current time, and end-time. However, the last two parameters are replaced by start and end velocity. As can be seen on line 254-255 the momentary velocity is smoothed using a sliding sum, and normalized to seconds. The reason for this is that the maximum speed of the motor is 280 steps per second, while the program operates at 1 kHz. The general rule therefore is that no encoder steps will happen during one program cycle, therefore causing the momentary speed to be measured as 0. After this, the velocity controller continues as the position controller

Listing 4: `Velocity Controller`

```
252 void velocityControl(ControlStates* c_s, EncoderStates* e_s, MotorShieldPins* m_pins
        , PIDParameters* pid_p)
253 {
254     // Sliding sum for encoder speed
255     enc1->dp_ = enc1->dp_ * 0.99 + 0.01*(enc1->p_ - enc1->pp_)/(dT/TIME_SCALE);  //
        dT = 1000 us = 0.001 s
256     enc1->pp_ = enc1->p_;
257 
258     //Set-point calculation
259     double setPoint = minimumJerk(c_s->ti_, (double)t_new, c_s->T_, c_s->ri_, c_s->
        rf_);
260     c_s->r_ = setPoint;
261 
262     // Error and error derivative
263     double e = (c_s->r_ - e_s->dp_);
264     double de = (c_s->e_ - e) / dT;
265 
266     // Get PID output
267     double ut = pid(e, de, pid_p);
268 
269     // Store for next cycle
270     c_s->e_ = e;
271     c_s->de_ = de;
272     c_s->u_ = ut;
273 }
```

2.4 PID-controller, minimum jerk and actuation

The PID-controller used before matches the equation shown earlier in (1) on page 2. The implementation is shown in listing 5.

Listing 5: `PID-implementation`

```
285 float pid(float e, float de, PIDParameters* p)
286 {
287     // Update integral term
288     p->I_ += e*dT;
289     // Calculate output value
290     double ut =  p->Kp_*e + p->Ki_ * p->I_ + p->Kd_ * de;
291     // Clamp to maixmum and minimum value before returning
292     ut = min(max(p->u_min_, ut), p->u_max_);
293     return ut;
294 }
```

The mimimum jerk equation is also implemented as shown earlier in (2) on page 2. The only difference is that the fraction $\frac{t}{T}$ is clamped to $[0, 1]$. Though this should not be needed with the guarantees made by the calculations of T, it was put in as a safeguard. Otherwise, a delayed controller can cause the setpoint to grow to infinity, which in turn causes the error and PID values to grow to infinity. The code for this is shown in listing 6.

Listing 6: `Mimimum jerk implementation`

```
277 float minimumJerk(float t0, float t, float T, float q0, float qf)
278 {
279     // Calculate t / T. Clamp to [0,1] to prevent value explosion
280     double tbyT = min((t-t0)/(T-t0),1);
281     // Minimum jerk equation.
282     return (q0 + (qf - q0) * (10.0 * pow(tbyT,3.0) - 15.0 * pow(tbyT,4.0) + 6.0 *
        pow(tbyT,5.0)));
283 }
```

Lastly, the actuation function is what actually transforms the control value into a PWM output. The function receives the output from the PID-controller, and clamps it to the allowed range as well as setting the direction of the motor. The code for this is shown in listing 7.

Listing 7: `Actuator implementation`

```
208 void actuate(float control, MotorShieldPins *mps)
209 {
210     // Set motor direction based on sign of control value
211     digitalWrite(mps->DIR_, control < 0 ? LOW : HIGH);
212
213     // Force to positive and make sure it is in allowed control range
214     double controlNew = abs(control);
215     controlNew = max(min(pwm_resolution, controlNew), 0);
216
217     // Write to pin
218     analogWrite(mps->PWM_, controlNew);
219 }
```

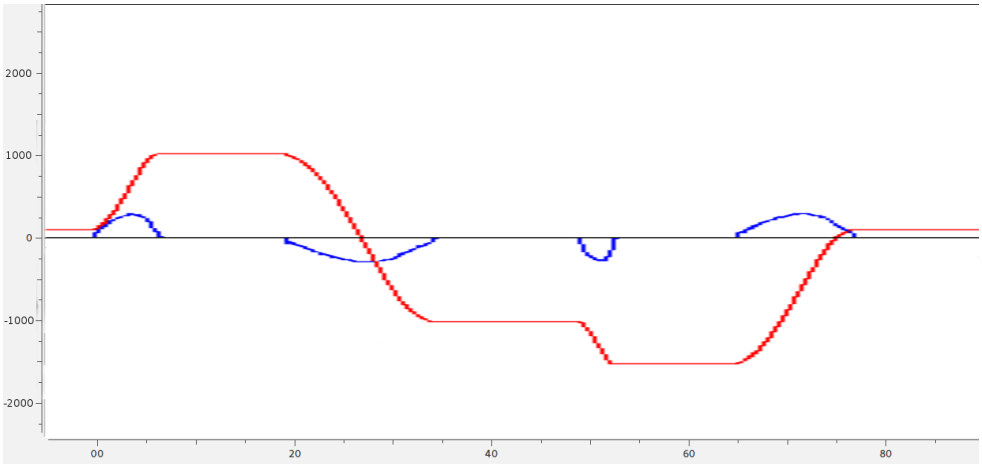
3 Verification and results

3.1 PID-tuning

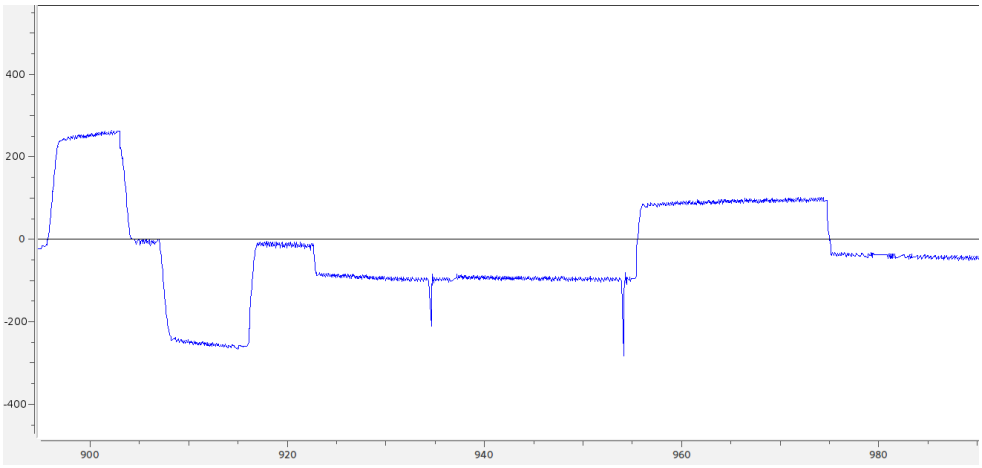
The tuning of the PID-controller was done in three steps. First, the k_p term was found with which reasonable behaviour was seen. Then, the k_i term was estimated to be on the order of 1×10^{-6}, based on $\mathrm{d}T = 1000$. This follows from the definition of the integral. When a good

value was found, the derivative term was increased and was estimated to be on the order of magnitude 1. This follows from the definition of the derivative term and the timestep. Lastly, several tests were ran with the final configuration to make sure that no erroneous behaviour was occuring.

After testing, the values $K_\mathrm{p} = 60$, $K_\mathrm{i} = 1 \times 10^{-6}$, and $K_\mathrm{d} = 2$ was found to provide the best performance. These parameters were found to provide good results for both velocity and position control. Plots of the behaviours are shown in figs. 1a and 1b.



(a) Plot of target position := $[1000, -1000, -1500, 50]$



(b) Plot of target velocity := $[280, -10, -280, -20, -100, 100, -50]$. The spikes are believed to be caused by speed difference between microcontroller execution (1 kHz) and maximum encoder speed (235 steps/second). There is no visible change in output however, and the spikes only last 2-3 ms.

Figure 2: Plots of controller behaviour for some target values.

3.2 Results

A PID controller controlling the angle and angular speed of a DC motor with tick-counter was created. It was implemented using an *Arduino Due* along with ROS. Both angle and velocity control show the expected behaviour, including a minimum jerk motion for the angle control.

The angle control is carried out in the optimal time of the minimum jerk equation, if optimal means reaching maximum velocity at half time. For bigger angles to traverse, this timeframe is relatively long and could be reduced by spending more time at max velocity. This would require replacing the minimum jerk equation with another model; for example a spline interpolation

based on minimum jerk behaviour.

References

[1] RH158 micro motor. *Datasheet*.
`http://www.reductor-motor.com/eng-micRH158.htm`. Last accessed at 2015-11-22.

[2] Gear-motors with Hall-effect encoder. *Datasheet*.
`http://www.reductor-motor.com/eng-mic_e_data1.htm`. Last accessed at 2015-11-22.