Laboration 1: PID-controls

Sensors and Sensing

Michael Floßmann, Tom Olsson

November 22, 2015

List of Figures

Listings

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 can be seen in (1).

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:

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

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

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

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

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

In order to make this term dependent on T, we must resubstitute:

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

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

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

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:

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

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

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 tics per revolution as well as the maximum velocity of the motor was measured and compared to the datasheet [1],[2]. The

Table 1: Ticks per rotation and maximum ticks per second values

	Ticks per rotation	rps	Ticks per second
Datasheet	230.5	1.35	311
Measured	235.0	1.19	280

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 ticks per second was used. The Ticks per rotation value was measured turning a varying amount of ticks and seeing when one rotation was completed. This method is expected to be relatively uncertain, since the human eye is not prone to spot these characteristics, but the lack of better equipment for angle measurement drove us to this descision. The tics per second value for maximum speed was measured by applying a duty cycle of 100% to the PWM and programatically measuring the ticks each second. This measurement technique 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. 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 TODO, (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 splines should be used instead of the minimun jerk equation.

Listing 1: `Set Position Callback`

```
void setPosCallback(const arduino_pkg::SetPosition::Request &req, arduino_pkg
    ::SetPosition::Response &res) {
    mc1->active_ = true;
    mc1->rf_ = req.encoder;
```

```
    mc1->ri_ = enc1->p_;
    mc1->r_ = enc1->p_;

    mc1->ti_ = micros();


    mc1->T_ = micros() + (abs(req.encoder - enc1->p_))/V_MAX*(15.0/8.0)*1e6;

    res.success = true;

}
```

The position controller is implemented to setup the parameters for PID and MJE. The code for this can be see 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**

```
void positionControl(ControlStates* c_s, EncoderStates* e_s, MotorShieldPins*
    m_pins, PIDParameters* pid_p)
{
    //TODO:
    // -update the setpoint using minimum Jerk DONE
    // -calculate position error DONE
    // -calculate derivative of the position error DONE
    // -update the control states for the next iteration DONE

    // -compute control using pid() DONE

    double setPoint = minimumJerk(c_s->ti_, (double)t_new, c_s->T_, c_s->ri_,
     c_s->rf_);

    c_s->r_ = setPoint;
    double e = (c_s->r_ - e_s->p_); // USED

    double de = (c_s->e_ - e) / dT;


    c_s->e_ = e;
    c_s->de_ = de;

    double ut = pid(e, de, pid_p);

    c_s->u_ = ut;



}
```

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 ticks per second; this therefore becomes $\left\lceil \frac{3}{560} \right\rceil_4 = 0.006$.

Listing 3: `Set Velocity Callback`

```
void initMotor(MotorShieldPins* pins)
{
    pinMode(pins->DIR_,OUTPUT);
    pinMode(pins->BRK_,OUTPUT);
    pinMode(pins->PWM_,OUTPUT);
    pinMode(pins->CUR_,INPUT);

    digitalWrite(pins->DIR_,HIGH);
    digitalWrite(pins->BRK_,LOW);
}

////////////////////////////////////////////////////////////////////
////////////////////////// MAIN LOOP ///////////////////////////
////////////////////////////////////////////////////////////////////
```

The velocity controller is also very similar to the position controller, as can be seen 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 TODO the momentary velocity is smoothed using a sliding integral, and normalized to seconds. The reason for this is that the maximum speed of the motor is 235 ticks per second, while the program operates at 1 kHz. The general rule therefore is that no encoder ticks 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`

```
}
//---------------------------------------------------------------------------
float minimumJerk(float t0, float t, float T, float q0, float qf)
{
    //TODO: calculate minimumJerk set point
    double tbyT = min((t-t0)/(T-t0),1);
    return (q0 + (qf - q0) * (10.0 * pow(tbyT,3.0) - 15.0 * pow(tbyT,4.0) +
   6.0 * pow(tbyT,5.0)));
}
//---------------------------------------------------------------------------
float pid(float e, float de, PIDParameters* p)
{
    //TODO:
    // -update the integral term
    // -compute the control value
    // -clamp the control value and if necessary back-calculate the integral
   term (to avoid windup)
    // -return control value
    p->I_ += e*dT;
    double ut =  p->Kp_*e + p->Ki_ * p->I_ + p->Kd_ * de;
    ut = min(max(p->u_min_, ut), p->u_max_);
    return ut;
```

2.4 PID-controller, minimum jerk and actuation

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

Listing 5: `PID-implementation`

```
    state.current = analogRead(motor1->CUR_);
    state.pwm = analogRead(motor1->PWM_);
    state.encoder = enc1->p_;
}

//////////////////////ROS Services ///////////////////////////////////
void setPosCallback(const arduino_pkg::SetPosition::Request &req, arduino_pkg
    ::SetPosition::Response &res) {
    mc1->active_ = true;
    mc1->rf_ = req.encoder;
    mc1->ri_ = enc1->p_;
    mc1->r_ = enc1->p_;

    mc1->ti_ = micros();
```

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 potentially accumulate an infinite error can lose control. The code for this can be seen in listing 6.

Listing 6: `Mimimum jerk implementation`

```

#if ROS_SUPPORT
//--------------------------------------------------------------------------------
void updateState() {

    state.brake = digitalRead(motor1->BRK_);
```

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 can be seen in listing 7.

Listing 7: `Actuator implementation`

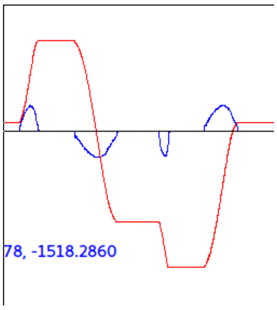
```
}

//--------------------------------------------------------------------------------
void positionControl(ControlStates* c_s, EncoderStates* e_s, MotorShieldPins*
     m_pins, PIDParameters* pid_p)
{
    //TODO:
    // -update the setpoint using minimum Jerk DONE
    // -calculate position error DONE
    // -calculate derivative of the position error DONE
    // -update the control states for the next iteration DONE
```

3 Verification and results

3.1 PID-tuning

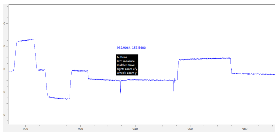
The tuning of the PID-controller was done in three steps. First, the k_p term was found with which reasonable behaviour was seen. Then, the k_i term was estimated to be on the order of 1×10^{-6}, based on $dT = 1000$. This follows from the definition of the integral. When a good value was found, the 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 $k_p = 60$, $k_i = 1 \times 10^{-6}$, and $k_d = 2$ 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 can be seen in figs. 1a and **??** on page ??.



(a) Plot of position=bla bla bla

u



(b) Plot of velocity = bla bla bla.

Figure 2: Plots of controller behaviour for some target values. Asymptote lines added for clarity.

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. For bigger angles to traverse, this timeframe is relatively long. By replacing the minimum jerk equation with other models which also lower the motor jerk but remaining at maximum speed for longer times (i.e. spline interpolation), the efficiency could be vastly increased on this part.

References

[1] RH158 micro motor `http://www.reductor-motor.com/eng-micRH158.htm` (accessed at 2015–11–22)

[2] gear-motors with Hall-effect encoder `http://www.reductor-motor.com/eng-mic_e_data1.htm` (accessed at 2015–11–22)