

# Hands on Algorithms for robotics

Michal Chovanec, PhD.

August 11, 2024

# Chapter 1

# Hardware description

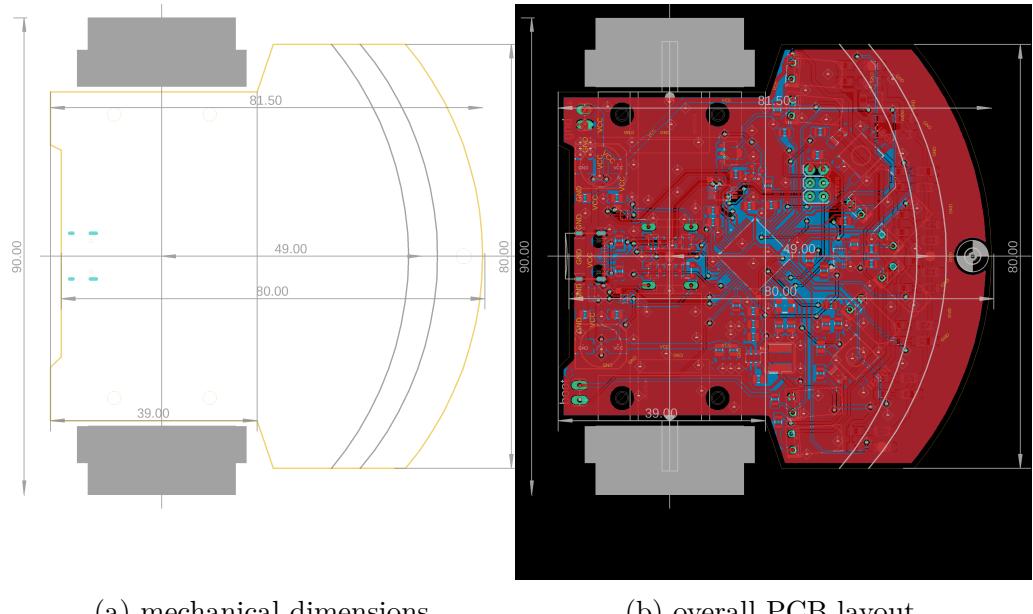


Figure 1.1: Robot PCB



Figure 1.2: Robot photo

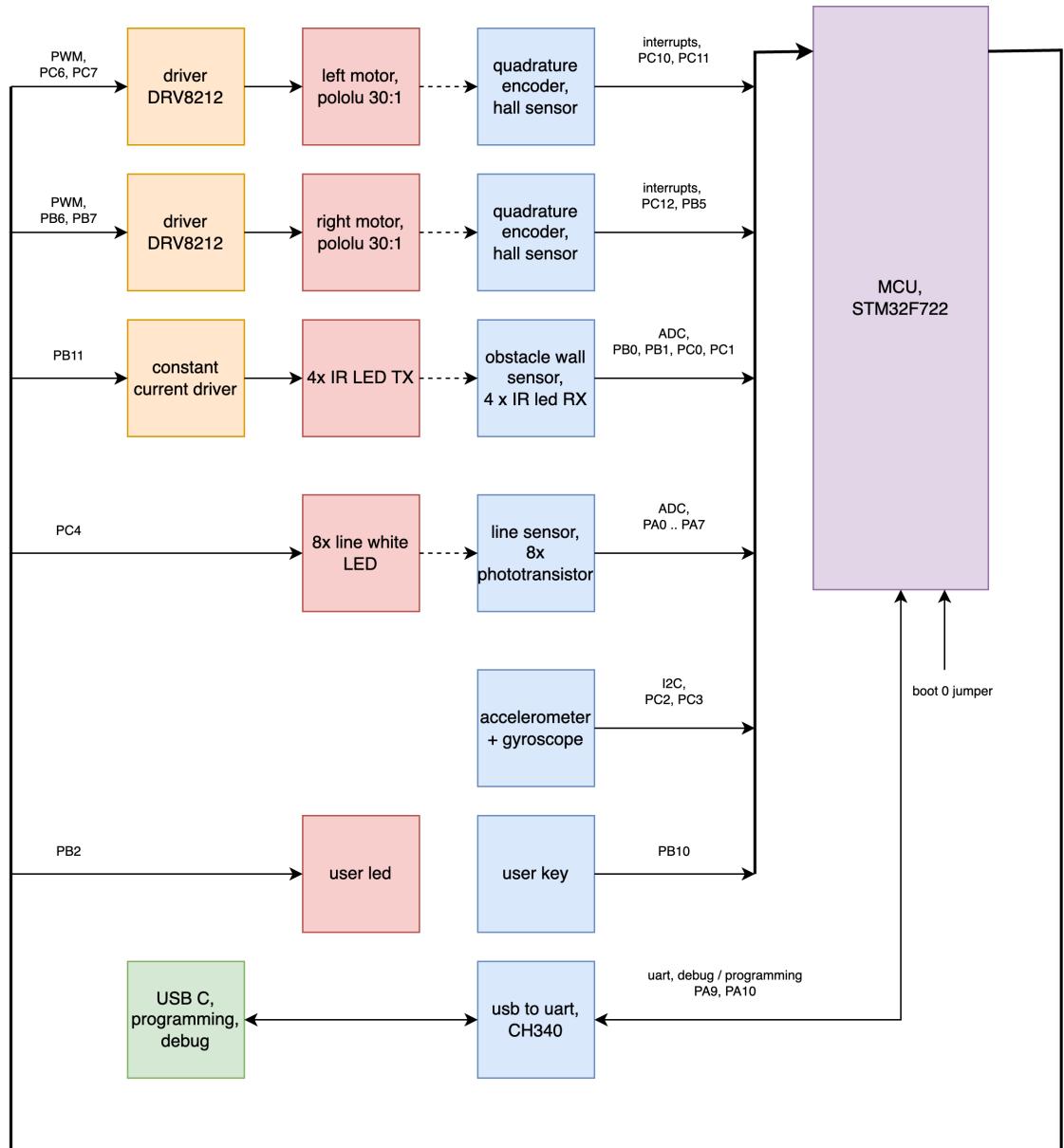


Figure 1.3: block diagram of DC motor version

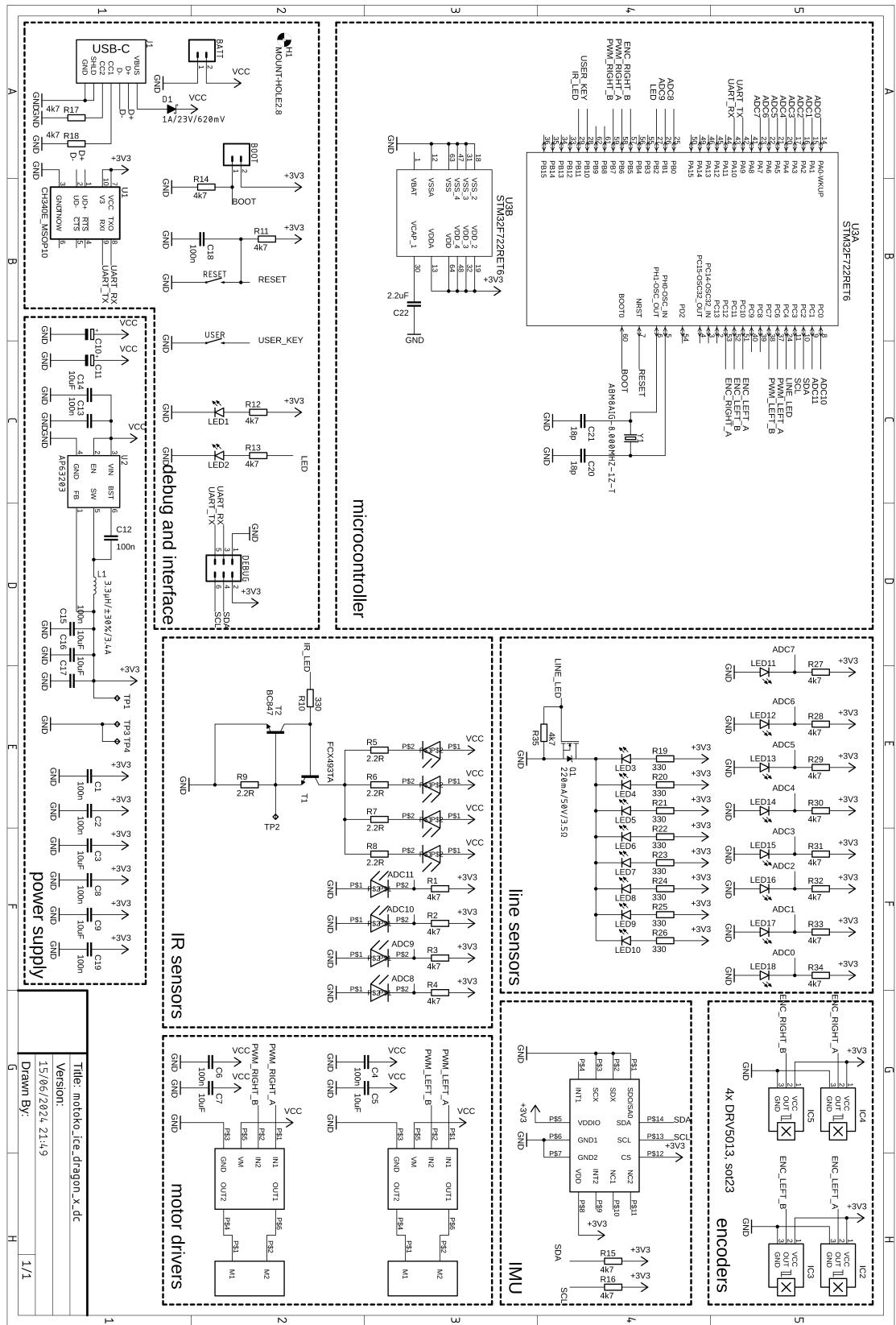


Figure 1.4: schematic diagram of DC motor version

pin number	pin name	function	peripheral	AF number
14	PA0	line sensor ADC0, right	ADC1	analog in
15	PA1	line sensor ADC1	ADC1	analog in
16	PA2	line sensor ADC2	ADC1	analog in
17	PA3	line sensor ADC3	ADC1	analog in
20	PA4	line sensor ADC4	ADC1	analog in
21	PA5	line sensor ADC5	ADC1	analog in
22	PA6	line sensor ADC6	ADC1	analog in
23	PA7	line sensor ADC7, left	ADC1	analog in
25	PB0	IR sensor ADC8, front left	ADC1	analog in
26	PB1	IR sensor ADC9, left	ADC1	analog in
8	PC0	IR sensor ADC10, right	ADC1	analog in
9	PC1	IR sensor ADC11, front right	ADC1	analog in
51	PC10	encoder left A	GPIOC	
52	PC11	encoder left B	GPIOC	
53	PC12	encoder right A	GPIOC	
57	PB5	encoder right B	GPIOB	
37	PC6	PWM left A	TIM3_CH1	AF2
38	PC7	PWM left B	TIM3_CH2	AF2
58	PB6	PWM right A	TIM4_CH1	AF2
59	PB7	PWM right B	TIM4_CH2	AF2
10	PC2	IMU I2C, SDA	GPIOC	
11	PC3	IMU I2C, SCL	GPIOC	
42	PA9	uart TX	UART1	AF7
43	PA10	uart RX	UART1	AF7
27	PB2	user LED	GPIOB	
28	PB10	user Key	GPIOB	
29	PB11	IR LED control	GPIOB	
24	PC4	line LED control	GPIOC	
7	NRST	reset		
60	BOOT0	firmware bootloader control		

Table 1.1: pin mapping and function

# Chapter 2

## System identification

### 2.1 State space models

System state is column vector, with  $N$  rows

$$x(n) = \begin{bmatrix} x_0(n) \\ x_1(n) \\ \dots \\ x_{N-1}(n) \end{bmatrix} \quad (2.1)$$

For single output system this vector contains single value, e.g. the motor velocity  $\omega(n)$

$$x(n) = [\omega(n)] \quad (2.2)$$

If we have 2nd order system, e.g. servo with inertia, the state will contain two elements :

- motor shaft velocity  $\omega(n)$ , measured in  $rad/s$
- motor shaft angle  $\theta(n)$ , measured in  $rad$

$$x(n) = \begin{bmatrix} \omega(n) \\ \theta(n) \end{bmatrix} \quad (2.3)$$

More accurate servo model observes also motor current  $i(n)$ , which gives us three element state vector

$$x(n) = \begin{bmatrix} i(n) \\ \omega(n) \\ \theta(n) \end{bmatrix} \quad (2.4)$$

Robot moving in 2D plane have usually state containing robot position  $x'$ ,  $y'$ , and robot orientation  $\theta$

$$x(n) = \begin{bmatrix} x'(n) \\ y'(n) \\ \theta(n) \end{bmatrix} \quad (2.5)$$

The number of rows in state vector  $x(n)$  is called **system order**.

The dynamical system is controlled with  $M$  inputs, stacked in column input vector  $u(n)$

$$u(n) = \begin{bmatrix} u_0(n) \\ u_1(n) \\ \dots \\ u_{M-1}(n) \end{bmatrix} \quad (2.6)$$

The system with single input, e.g. motor with current control, input vector contains single value

$$u(n) = [i(n)] \quad (2.7)$$

Differential drive robot where are controlled two independent motors have two inputs

$$u(n) = \begin{bmatrix} i_{left}(n) \\ i_{right}(n) \end{bmatrix} \quad (2.8)$$

Relation between control input  $u(n)$ , current state  $x(n)$  and next state  $x(n+1)$  can be modeled using **linear state space model**

$$x(n+1) = Ax(n) + Bu(n) \quad (2.9)$$

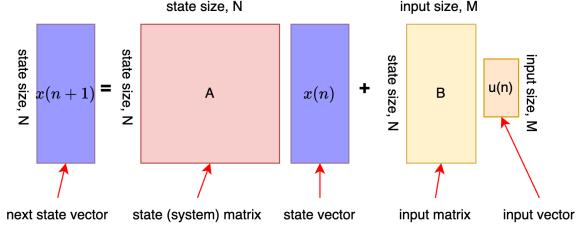


Figure 2.1: State space model matrices shapes

This model can capture dynamics of any linear system. Sometimes the state  $x(n)$  can't be measured directly, and we observe only system output  $y(n) = Cx(n)$ . Where matrix  $C$  have k-rows and n-columns. In our case we consider we have access to full state  $x(n)$ . There is of course the continuous form of state space model

$$\frac{dx}{dt} = A^c x(t) + B^c u(t) \quad (2.10)$$

note : the martices  $A^c$  and  $B^c$  differs from discrete system matrices  $A$  and  $B$ . They are related with bilinear transform as

$$S_a = (I - \frac{\Delta T}{2} A^c)^{-1} \quad (2.11)$$

$$S_b = I + \frac{\Delta T}{2} A^c \quad (2.12)$$

$$A = S_a S_b \quad (2.13)$$

$$B = S_a B^c \Delta T \quad (2.14)$$

where  $\Delta T$  is sampling period. This is mostly used discretisation, from continuous (e.g. physical) model into discrete form. Following pythoncode convert continous system into discrete system. If system contains output matrix  $C$  discretisation doesn't effects its values.

```

1 def c2d(a, b, c, dt):
2     i = numpy.eye(a.shape[0])
3
4     tmp_a = numpy.linalg.inv(i - (0.5*dt)*a)
5     tmp_b = i + (0.5*dt)*a
6
7     a_disc = tmp_a@tmp_b

```

```

8     b_disc = (tmp_a*dt)@b
9
10    return a_disc, b_disc, c

```

Process of system identification is finding matrices  $A$ ,  $B$  (and  $C$ ). From this model can be synthesised controller, or system able to plan multiple steps ahead.

## 2.2 First order identification

Goal is to find parameters of DC motor. Motor velocity model, can be approximated as 1st linear order system. Real system is of course non-linear. The photo of example system is on fig 2.2. Control loop hardware consists of motor driver (H-driver, DRV8212), motor (Pololu HP 1:30 gear), and quadrature magnetic encoder (2x DRV5013), fig 2.3

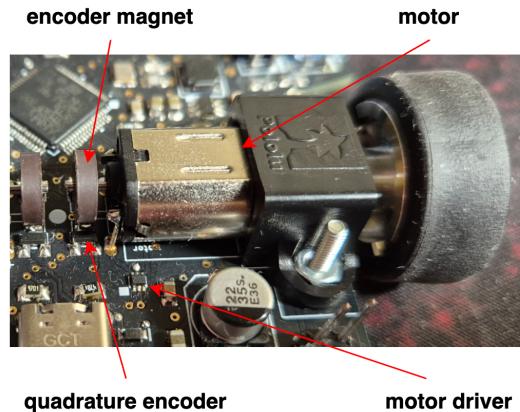


Figure 2.2: Wheel drive hardware detail

Input signal  $u(n)$  can be generated arbitrarily. Mostly used is square wave, to capture high frequency dynamics. Algorithm is from observed motor velocity  $x(n)$  and given input  $u(n)$  estimating model parameters.

Before real motor testing, we focus on simulation example. Our simulated motor will have following parameters

- sampling frequency 4kHz,  $\Delta T = 1/4000$
- maximum control input  $u_{max} = 2$

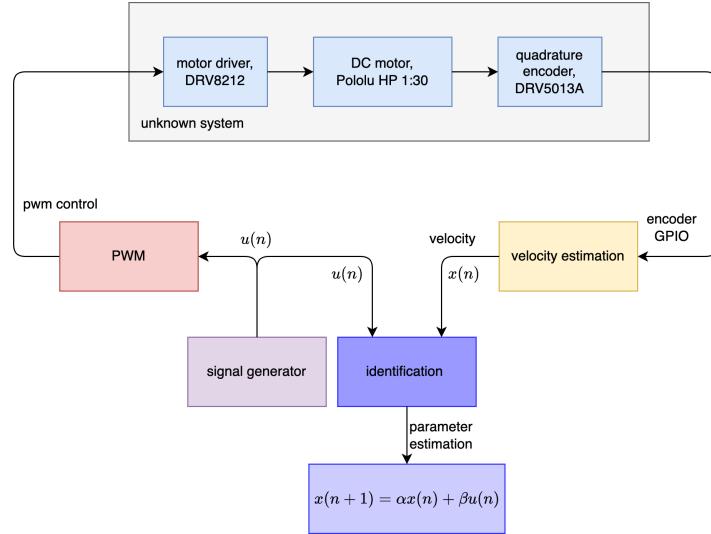


Figure 2.3: Motor identification process

- motor constant  $k = 17$
- motor time constat  $\tau = 29$  milliseconds

Motor state is single variable, angular velocity  $\omega(t)$ . Model have two parameters  $\alpha, \beta$ . First order continuos model becomes

$$\frac{d\omega(t)}{dt} = \alpha\omega(t) + \beta u(t) \quad (2.15)$$

$$\alpha = -\frac{1}{\tau} \quad (2.16)$$

$$\beta = \frac{k}{\tau} \quad (2.17)$$

with given parameters  $k$  and  $\tau$  becomes  $\alpha = -34.48$  and  $\beta = 586.2$ .

Unit step response is obtained by setting  $u(n) = 1$ , and solving using any differential equation solver. The simplest is Euler method, which works well for small  $dt$  and low order systems. Much accurate is commonly used Runge-Kutta 4 method.

Our goal is to generate input  $u(t)$  and by observing system output  $x(t)$  estimate parameters  $k$  and  $\tau$ .

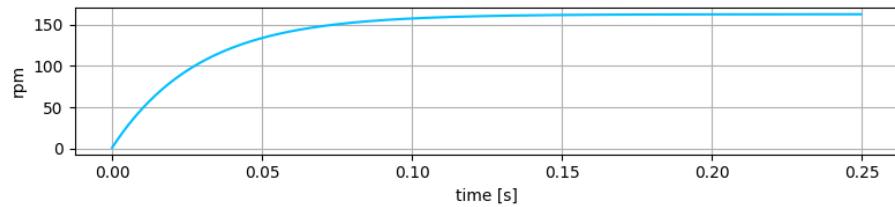


Figure 2.4: Motor step response

We start with simplest identification method, which works only for 1st order non-oscillating systems. Method have two main steps

1. steady state value estimation
2. time constant estimation

### Steady state estimation

We drive motor on different input levels, e.g. ranging from 10% ... 100%. Observing steady state velocity the motor constant  $k$  can be estimated.

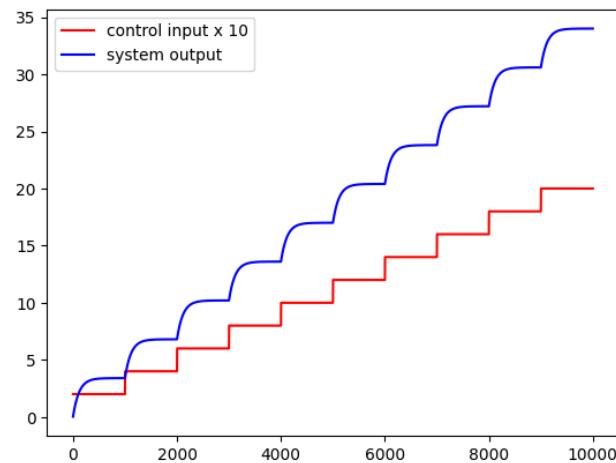


Figure 2.5: Motor constant estimation

Algorithm for estimating motor constant :

1. choose constant motor input  $u \in (0, u_{max})$

2. run motor, and wait for velocity steady state
3. measure velocity
4. estimate  $\hat{k} = x/u$
5. repeats to estimate average  $\hat{k}$

In following code we choose 10 different input levels. After setting, we wait 500steps to steady state, then we measure output velocity. Finally, we average results.

```

1 # input levels
2 u_values = u_max*numpy.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
3
4 # reset dynamical system into zero state
5 ds.reset()
6
7 for j in range(len(u_values)):
8
9     x_mean = []
10    for i in range(1000):
11
12        # convert scalar u to column vector
13        u      = u_values[j]
14        u      = numpy.array([[u]])
15
16        # compute dynamical system one step
17        x, _ = ds.forward_state(u)
18
19        # after steady state, store x
20        if i > 500:
21            x_mean.append(x[0][0])
22
23    x_mean = numpy.array(x_mean)
24    x_mean = x_mean.mean()
25
26    k_est = x_mean/u_values[j]
27
28    print(u_values[j], k_est)
29
30 k_mean = k_est.mean()
31
32 # average k estimate
33 print("k_mean = ", k_mean)
```

In our example the resulted  $\hat{k}$  is 16.995, which very close to real value 17. Accuracy depends on velocity estimation noise (encoder noise) and how many samples we average.

## Time constant estimation

Second parameter is time constant  $\tau$ . Most textbooks uses measuring time at which system reaches 63.2% of  $x_{max}$ . Where  $x_{max}$  is motor velocity on choosen  $u$ , a.k.a. steady state velocity at  $u$ . This method suffers to noise, and precise timing. Here we presents more accurate method, brining system into natural frequency resonance. Algorithm is as follow :

1. choose  $u_{set}$ , and set  $u_{in} = u_{set}$
2. repeat  $n_{steps}$ , each step duration is  $\Delta T$ 
  - a) if  $u_{in} > 0$  and  $x(n) > 0.632ku_{in}$   
increment  $n_{HalfPeriods}$ , and set  $u_{in} = -u_{set}$
  - b) else if  $u_{in} < 0$  and  $x(n) < 0.632ku_{in}$   
increment  $n_{HalfPeriods}$ , and set  $u_{in} = u_{set}$

The time constant  $\hat{\tau}$  can be now estimated by

$$\hat{\tau} = \frac{2}{\pi} \frac{n_{steps}}{n_{HalfPeriods}} \Delta T \quad (2.18)$$

Where  $\Delta T$  is sampling period, ratio  $n_{steps} : n_{HalfPeriods}$  is estimating system frequency. In core, this method is nothing more than period duration estimation. The resulted time plot is on fig. 2.6. Result for our simulated motor is  $\hat{\tau} = 27.44[ms]$ , which is good estimation of real value 29[ms].

Using equations 2.16 and 2.17 we can estimate  $\hat{\alpha}$  and  $\hat{\beta}$  as  $\hat{\alpha} = -36.443$  and  $\hat{\beta} = 619.372$

## Real motor identification

For implementation in real system, the code must runs in real time. With relative high sampling frequency 4kHz, the code is written in c++. Interface is universal, with few key functions.

- `timer.delay_ms(unsigned int time)` - waits given milliseconds
- `float x = motor_control.get_right_velocity()` - return measured wheel velocity

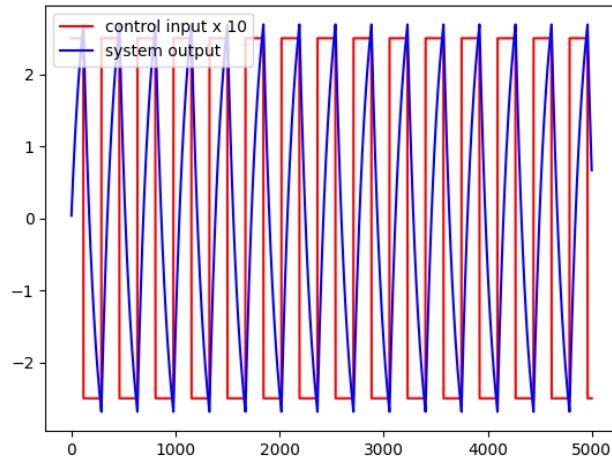


Figure 2.6: Motor time constant estimation

- `motor_control.set_right_torque(float x)` - sets the PWM value, depends on sign the rotation direction can be controlled

This function needs to be customised for user application.

First we estimate motor constant  $k$ :

```

1 //1, estimate motor constant k, on different input values
2 uint32_t n_steps = 500;
3
4 float u_values[10] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0};
5
6 float k_mean      = 0.0;
7 float x_var_mean = 0.0;
8
9 for (unsigned int j = 0; j < 10; j++)
10 {
11     //run motor with desired input and wait for steady state
12     float u_in = u_values[j];
13
14     motor_control.set_right_torque(u_in*MOTOR_CONTROL_MAX_TORQUE);
15     timer.delay_ms(500);
16
17     //estimate average velocity
18     float x_mean = 0.0;
19     for (unsigned int i = 0; i < n_steps; i++)
20     {
21         float x = motor_control.get_right_velocity();
22         x_mean+= x;
23         timer.delay_ms(1);
24     }
25     x_mean = x_mean/n_steps;
26
27     //estimate average variance - encoder noise
28     float x_var = 0.0;
29     for (unsigned int i = 0; i < n_steps; i++)
30     {
31         float x = motor_control.get_right_velocity();
32         x_var+= (x - x_mean)*(x - x_mean);
33         timer.delay_ms(1);
34     }
35     x_var = x_var/n_steps;
36
37     float k = x_mean/u_in;
38     k_mean+= x_mean/u_in;
39     x_var_mean+= x_var;
40
41     terminal << "u_in    = " << u_in << "\n";
42     terminal << "k      = " << k << "\n";
43     terminal << "x_mean = " << x_mean << "\n";
44     terminal << "x_var  = " << x_var << "\n";
45     terminal << "\n\n";
46 }
47
48 //print summary results
49 k_mean      = k_mean/10.0;
50 x_var_mean = x_var_mean/10.0;
51
52 terminal << "k          = " << k_mean << "\n";
53 terminal << "x_var_mean = " << x_var_mean << "\n";

```

Second we estimate motor time constant  $\tau$  :

```

1 //2, estimate time constant by oscilating motor
2 float u_in = 0.25;
3 uint32_t periods = 0;
4
5 n_steps = 2000;
6
7 for (unsigned int i = 0; i < n_steps; i++)
8 {
9     motor_control.set_right_torque(u_in*MOTOR_CONTROL_MAX_TORQUE);
10    float x = motor_control.get_right_velocity()*60.0/(2.0*PI);
11
12    if (u_in > 0.0)
13    {
14        if (x > 0.632*k_mean*u_in)
15        {
16            u_in = -u_in;
17            periods++;
18        }
19    }
20    else
21    {
22        if (x < 0.632*k_mean*u_in)
23        {
24            u_in = -u_in;
25            periods++;
26        }
27    }
28
29    timer.delay_ms(1);
30 }
31
32 float t_period = (2*n_steps/periods)/PI;
33 terminal << "periods = " << periods << "\n";
34 terminal << "tau      = " << t_period << "[ms]\n";
35 terminal << "\n\n";

```

Average motor constant is  $k = 80.056$ , and time constant is  $\tau = 9.230[\text{ms}]$ .  
Terminal output :

```
1 ...
2
3 u_in      =  0.800
4 k          =  87.194
5 x_mean    =  69.755
6 x_var     =  19.236
7
8
9 u_in      =  0.900
10 k         =  86.440
11 x_mean   =  77.796
12 x_var    =  18.124
13
14
15 u_in      =  1.000
16 k          =  85.750
17 x_mean    =  85.750
18 x_var     =  7.554
19
20
21 k          =  80.056
22 x_var_mean =  170.903
23
24
25 periods   =  136
26 tau        =  9.230[ms]
```

# Chapter 3

## Motion control

### 3.1 Motor velocity controller

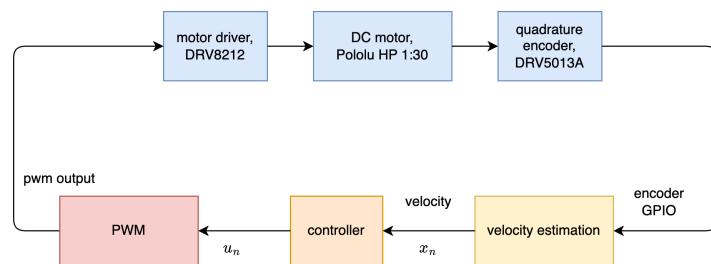


Figure 3.1: Velocity control overview

## 3.2 PID control

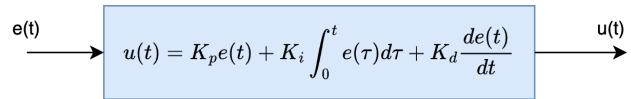


Figure 3.2: Textbook continuous PID controller

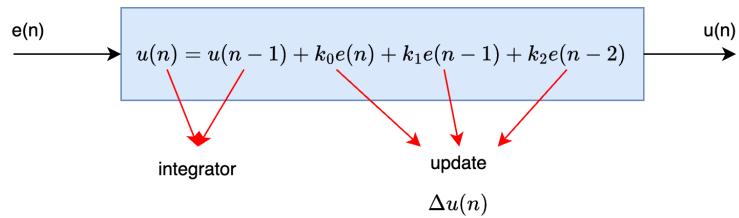


Figure 3.3: Discrete PID controller

$$k_0 = K_p + K_i \Delta T + \frac{K_d}{\Delta T} \quad (3.1)$$

$$k_1 = -K_p - 2 \frac{K_d}{\Delta T} \quad (3.2)$$

$$k_2 = \frac{K_d}{\Delta T} \quad (3.3)$$

### PID control - P only controller

- target value : **1000rpm**
- P-only control causes **steady state error**

$$u(n + 1) = u(n - 1) + k_p e(n) - k_p e(n - 1) \quad (3.4)$$

$$k_p = 0.01 \quad (3.5)$$

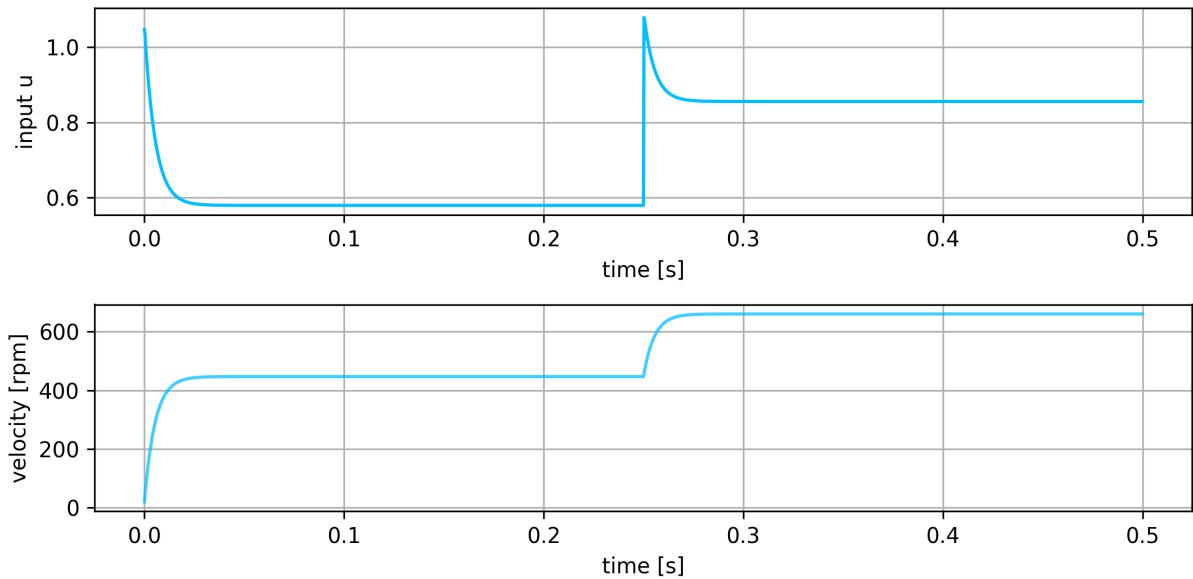


Figure 3.4: P-only controller

## PID control - PI

too high I term

- target value : **1000rpm**
- PI control **removes steady state error**
- too high I term causes **oscilations and overshoot**

$$u(n + 1) = u(n - 1) + (k_p + k_i\Delta T)e(n) - k_p e(n - 1) \quad (3.6)$$

$$k_p = 0.01 \quad (3.7)$$

$$k_i\Delta T = 0.005 \quad (3.8)$$

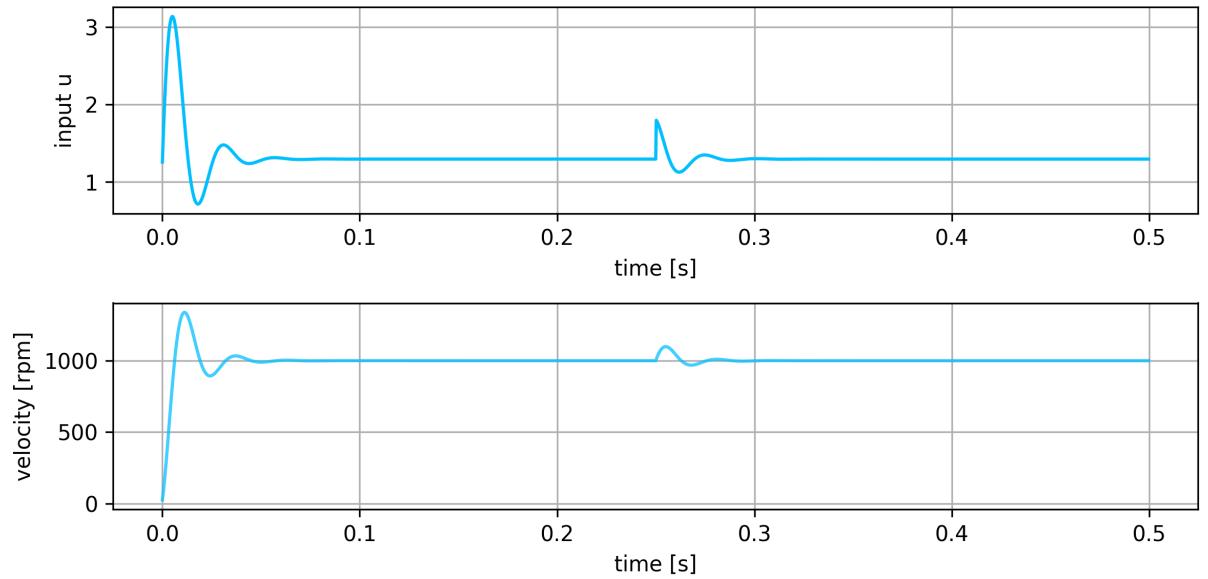


Figure 3.5: PI controller

correct I term

- target value : **1000rpm**
- correct tuned PI controller for 1st order system

- **no overshoot, no steady state error**

$$u(n+1) = u(n-1) + (k_p + k_i \Delta T)e(n) - k_p e(n-1) \quad (3.9)$$

$$k_p = 0.01 \quad (3.10)$$

$$k_i \Delta T = 0.0002 \quad (3.11)$$

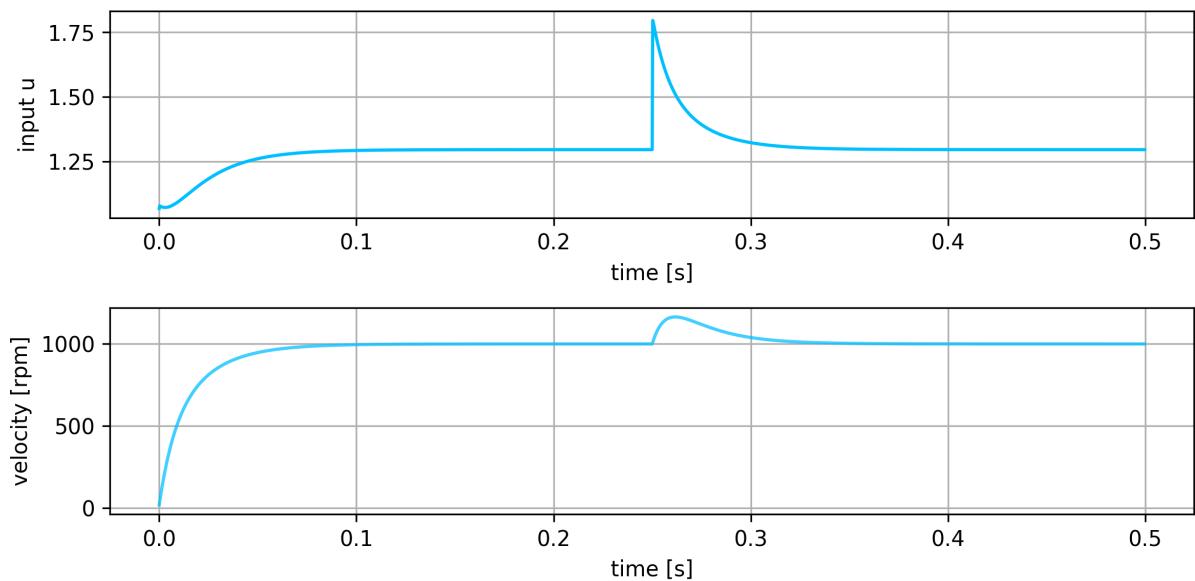


Figure 3.6: PI controller with correct parameters

## Complete discrete PID algorithm

1. calculate u-change candidate :

$$\Delta\hat{u}(n) = k_0e(n) + k_1e(n) + k_2e(n)$$

2. clip maximum allowed u-change, to avoid u-kick :

$$\Delta u(n) = \text{clip}(\Delta\hat{u}(n), -du_{min}, du_{max})$$

3. clip maximum allowed u value, to avoid saturation / windup :

$$u(n) = \text{clip}(u(n-1) + \Delta u(n), -u_{min}, u_{max})$$

Full Python code example for discrete PID controller :

```

1  class PID:
2
3  def __init__(self, kp, ki, kd, antiwindup = 10**10, du_max=10**10):
4      self.k0 = kp + ki + kd
5      self.k1 = -kp -2.0*kd
6      self.k2 = kd
7
8      self.e0 = 0.0
9      self.e1 = 0.0
10     self.e2 = 0.0
11
12     self.antiwindup = antiwindup
13     self.du_max      = du_max
14
15 def forward(self, xr, x, u_prev):
16     # error compute
17     self.e2 = self.e1
18     self.e1 = self.e0
19     self.e0 = xr - x
20
21     # u-change computing
22     du = self.k0*self.e0 + self.k1*self.e1 + self.k2*self.e2
23
24     # kick clipping, maximum output value change limitation
25     du = numpy.clip(du, -self.du_max, self.du_max)
26
27     # antiwindup, maximum output value limitation
28     u = numpy.clip(u_prev + du, -self.antiwindup, self.antiwindup)
29
30     return u

```

### 3.3 Optimal control

Optimal control is solution of following optimisation problem

$$\min_{x(\cdot), u(\cdot)} \sum_{n=0}^{\infty} x^T(n) Q x(n) + u^T(n) R u(n) \quad (3.12)$$

$$s.t. \quad x(n+1) = Ax(n) + Bu(n) \quad (3.13)$$

the cost function is quadratic

$$J = \sum_{n=0}^{\infty} x^T(n) Q x(n) + u^T(n) R u(n) \quad (3.14)$$

where  $Q$  and  $R$  are positive semidefinite square matrices, with shapes  $N \times N$  and  $M \times M$  respectively. The system order is  $N$  and system have  $M$  inputs. Matrix  $Q$  penalises magnitude of  $x(n)$  elements, matrix  $R$  penalises control input  $u(n)$  elements. Solution is obtained by solving algebraic Riccati equation, and results in feedback gain  $u(n) = -Kx(n)$ . This controller is called **Linear Quadratic Regulator - LQR**. This matrix can fully manipulate with system poles and guarantee system stability. However, given structure have steady state error - brings system always into zero state. To eliminate this issue, we introduce integral action term by augmenting system matrices. Resulted controller have two matrices : feedback gain  $K$  and integral action gain  $K_i$  matrix. The modified LQR controller have form on following fig 3.7 with equations 3.15.

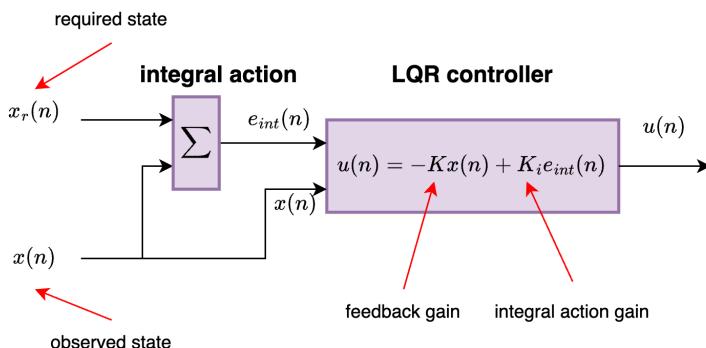


Figure 3.7: LQR controller with integral action

$$u(n) = -Kx(n) + K_i e_{int}(n) \quad (3.15)$$

$$e_{int}(n) = e_{int}(n-1) + x_r(n) - x(n) \quad (3.16)$$

where :

- $x(n)$  is observed state, column vector, with shape  $(N, 1)$
- $x_r(n)$  is required state, column vector, with shape  $(N, 1)$
- $u(n)$  is controller output, column vector, with shape  $(M, 1)$
- $K$  is feedback gain matrix, with shape  $(M, N)$
- $K_i$  is integral gain matrix, with shape  $(M, N)$

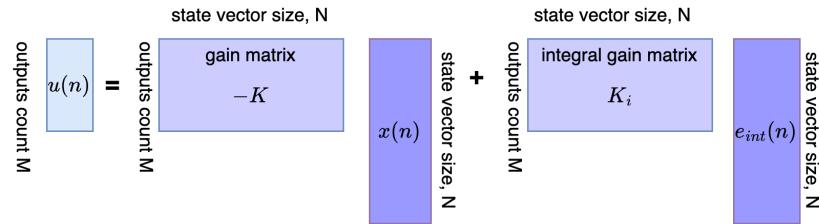


Figure 3.8: Matrices shapes in LQR controller

### 3.4 Model predictive control

Model predictive control is solution of following optimisation problem

$$\min_{x(\cdot), \Delta u(\cdot)} \sum_{h=0}^H (x_r^T(h) - x^T(h))Q(x_r(h) - x(h)) + \Delta u^T(h)R\Delta u(h) \quad (3.17)$$

$$s.t. \quad u(n) = u(n-1) + \Delta u(n), \quad (3.18)$$

$$x(n+1) = Ax(n) + Bu(n) \quad (3.19)$$

where :

- $H$  is prediction horizon steps
- $A$  is matrix,  $n \times n$
- $B$  is matrix,  $n \times m$
- $Q$  is matrix,  $n \times n$
- $R$  is matrix,  $m \times m$
- $\Delta u$  is controller output
- where  $n$  is system orders, and  $m$  system inputs count

Unwrap into form where only  $x(n)$  and  $u(n-1)$  are used to predict all other  $x$

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ &= Ax(n) + B(u(n-1) + \Delta u(n)) \\ x(n+2) &= Ax(n+1) + B(u(n) + \Delta u(n+1)) \\ &= A^2x(n) + (AB + B)u(n-1) + (AB + B)\Delta u(n) + B\Delta u(n) \\ x(n+3) &= Ax(n+2) + B(u(n+1) + \Delta u(n+2)) \\ &= A^3x(n) + (A^2 + AB + B)u(n-1) \\ &\quad + (A^2B + AB + B)\Delta u(n) \\ &\quad + (AB + B)\Delta u(n+1) + B\Delta u(n+2) \end{aligned}$$

...

Rewrite into matrix form

$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \dots \\ x(n+H) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \dots \\ A^H \end{bmatrix} x(n) + \begin{bmatrix} B \\ AB \\ \dots \\ \sum_{i=0}^{H-1} A^i B \end{bmatrix} u(n-1) + \\ + \begin{bmatrix} B & 0 & \dots & 0 \\ AB+B & B & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \sum_{i=0}^{H-1} A^i B & \sum_{i=0}^{H-2} A^i B & \dots & B \end{bmatrix} \begin{bmatrix} \Delta u(n) \\ \dots \\ \Delta u(n+H-1) \end{bmatrix}$$

in compact matrix form

$$\tilde{X} = \Psi x(n) + \Omega u(n-1) + \Theta_{\Delta} \tilde{U}$$

Formulate quadratic loss (cost) function

$$J = {}_{\Delta} \tilde{U}^T \tilde{R}_{\Delta} \tilde{U} + (\tilde{X}_r - \tilde{X})^T \tilde{Q} (\tilde{X}_r - \tilde{X})^T$$

after substitution

$$S = \tilde{X}_r - \Psi \tilde{X} - \Omega u(n-1)$$

we get

$$\begin{aligned} J &= {}_{\Delta} \tilde{U}^T \tilde{R}_{\Delta} \tilde{U} + (S - \Theta_{\Delta} \tilde{U})^T \tilde{Q} (S - \Theta_{\Delta} \tilde{U}) \\ &= {}_{\Delta} \tilde{U}^T \tilde{R}_{\Delta} \tilde{U} + S^T \tilde{Q} S - S^T \tilde{Q} \Theta_{\Delta} \tilde{U} - {}_{\Delta} \tilde{U}^T \Theta^T \tilde{Q} S + {}_{\Delta} \tilde{U}^T \Theta^T \tilde{Q} \Theta_{\Delta} \tilde{U} \end{aligned}$$

find derivative with respect to  $\Delta \tilde{U}$

$$\begin{aligned} \frac{\partial J}{\partial \Delta \tilde{U}} : \\ \frac{\partial \Delta \tilde{U}^T \tilde{R}_\Delta \tilde{U}}{\partial \Delta \tilde{U}} &= 2 \tilde{R}_\Delta \tilde{U} \\ \frac{\partial S^T \tilde{Q} S}{\partial \Delta \tilde{U}} &= 0 \\ \frac{\partial -S^T \tilde{Q} \Theta_\Delta \tilde{U}}{\partial \Delta \tilde{U}} &= -\Theta^T \tilde{Q} S \\ \frac{\partial -\Delta \tilde{U}^T \Theta^T \tilde{Q} S}{\partial \Delta \tilde{U}} &= -\Theta^T \tilde{Q} S \\ \frac{\partial \Delta \tilde{U}^T \Theta^T \tilde{Q} \Theta_\Delta \tilde{U}}{\partial \Delta \tilde{U}} &= 2 \Theta^T \tilde{Q} \Theta_\Delta \tilde{U} \end{aligned}$$

put derivate equal to zero, and solve

$$\begin{aligned} \frac{\partial J}{\partial \Delta \tilde{U}} &= 2 \tilde{R}_\Delta \tilde{U} - 2 \Theta^T \tilde{Q} S + 2 \Theta^T \tilde{Q} \Theta_\Delta \tilde{U} \\ 0 &= 2 \tilde{R}_\Delta \tilde{U} - 2 \Theta^T \tilde{Q} S + 2 \Theta^T \tilde{Q} \Theta_\Delta \tilde{U} \\ (\tilde{R} + \Theta^T \tilde{Q} \Theta)_\Delta \tilde{U} &= \Theta^T \tilde{Q} S \end{aligned}$$

and obtain solution for unconstrained model predictive control

$$\Delta \tilde{U} = (\tilde{R} + \Theta^T \tilde{Q} \Theta)^{-1} \Theta^T \tilde{Q} S$$

**model predictive control - full algorithm**

given matrices :  $\tilde{Q}, \tilde{R}, \Theta, \Phi, \Omega$

initialization (precompute) :

$$\Xi = (\tilde{R} + \Theta^T \tilde{Q} \Theta)^{-1} \Theta^T \tilde{Q}$$

**in loop process controller step :**

$$\begin{aligned} E(n) &= X_r(n) - \Phi x(n) - \Omega u(n-1) \\ \Delta u(n) &= \Xi E(n) \\ u(n) &= u(n-1) + \Delta u(n) \end{aligned}$$