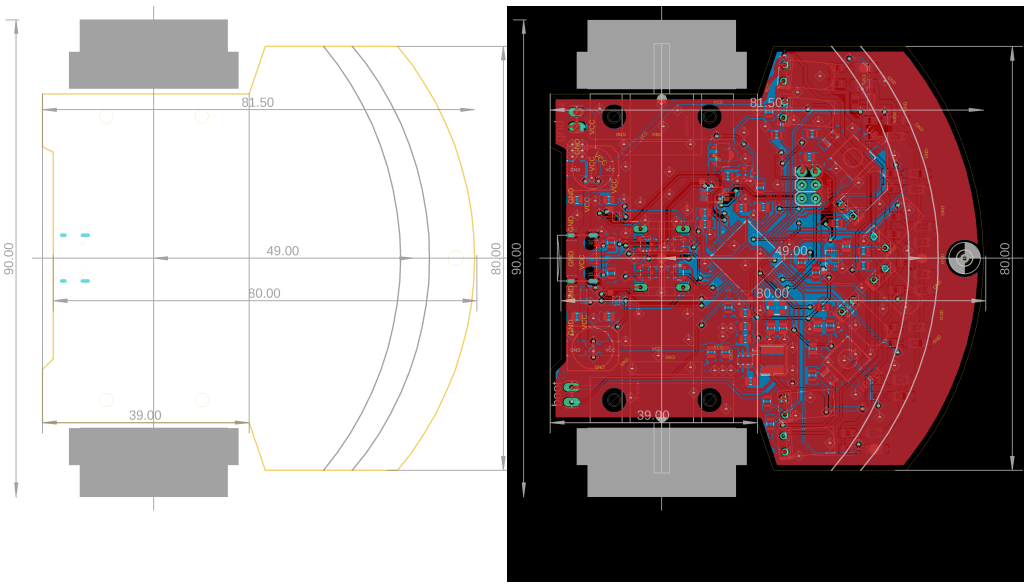
Hands on Algorithms for robotics

Michal Chovanec, PhD.

August 11, 2024

Chapter 1

Hardware description



(a) mechanical dimensions (b) overall PCB layout

Figure 1.1: Robot PCB

Figure 1.2: Robot photo

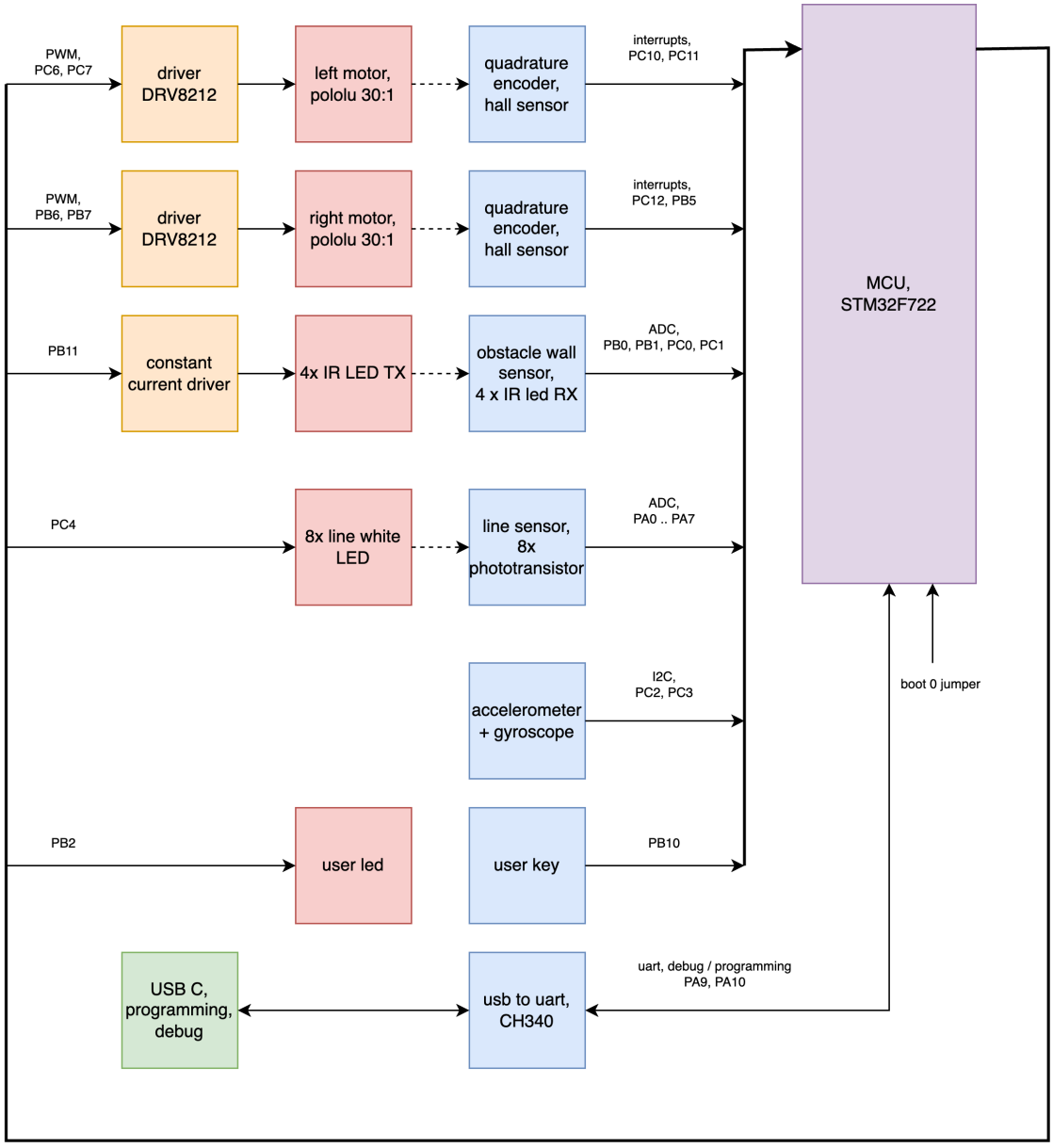


Figure 1.3: block diagram of DC motor version

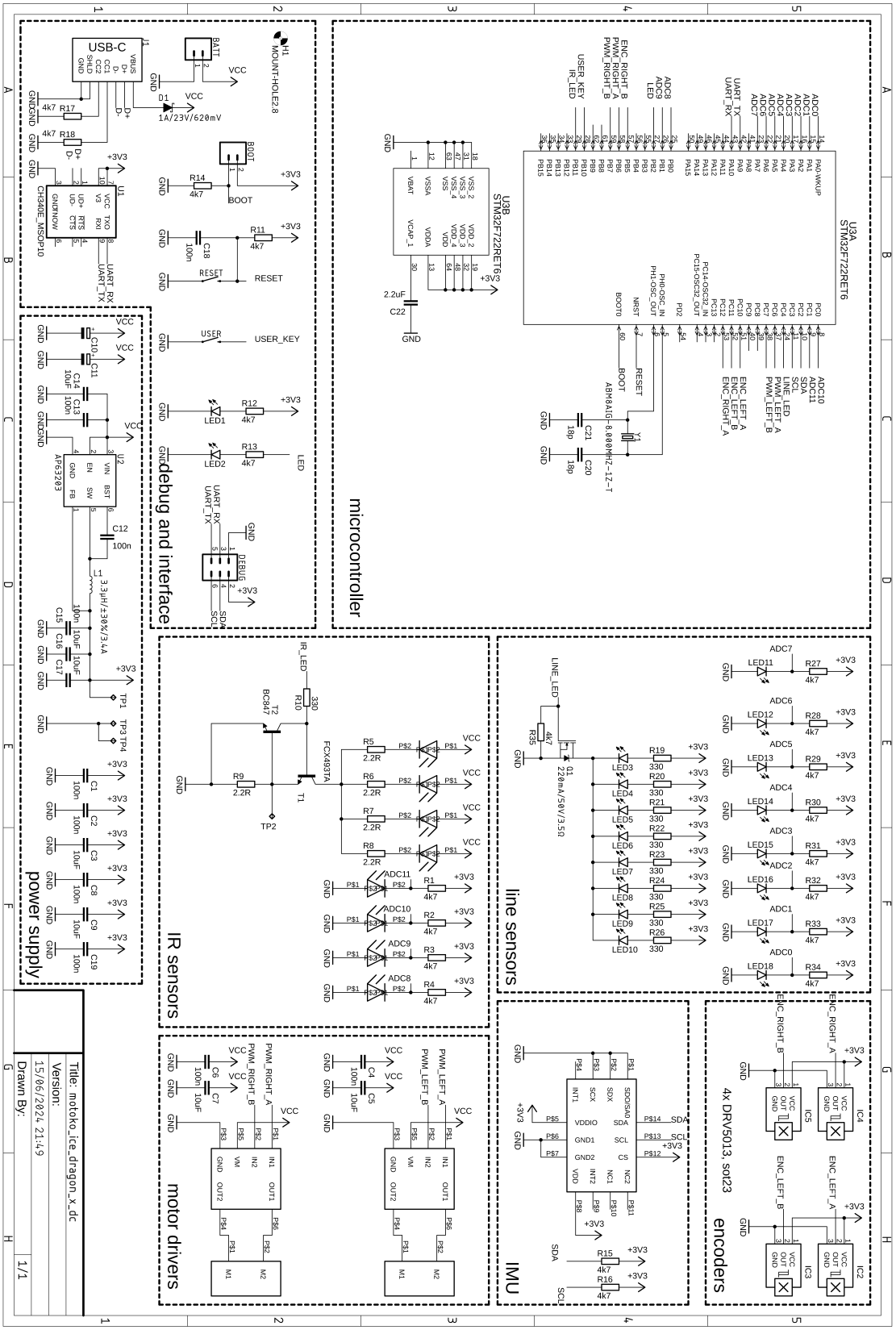


Figure 1.4: schematic diagram of DC motor version

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

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

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

$$x(n) = \begin{bmatrix} \omega(n) \end{bmatrix} \tag{2.2}$$

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

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

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

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

Robot moving in 2D plane have usualy state containing robot position x', y', and robot orientation θ

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

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 $u(n)$

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

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

$$u(n) = \begin{bmatrix} i(n) \end{bmatrix} \tag{2.7}$$

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

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

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

$$x(n+1) = Ax(n) + Bu(n) \tag{2.9}$$

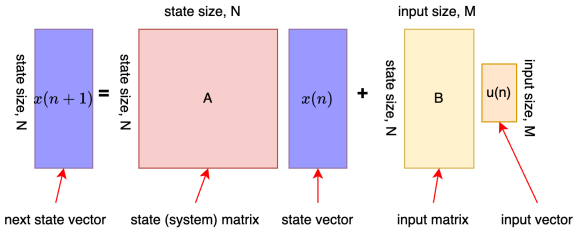


Figure 2.1: State space model matrices shapes

This model can capture dynamics of any linear system. Sometimes the state $x(n)$ can't be meassured 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

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

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

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

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

$$A = S_a S_b \tag{2.13}$$

$$B = S_a B^c \Delta T \tag{2.14}$$

where ΔT 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 sythetised 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

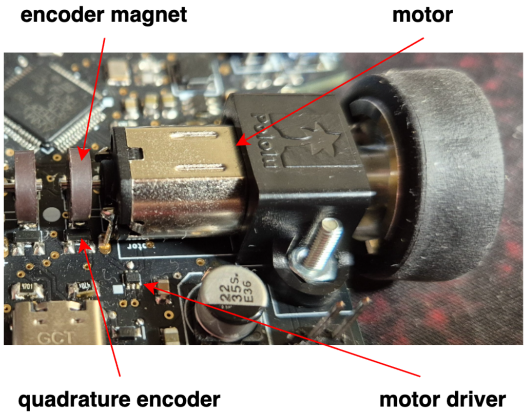


Figure 2.2: Wheel drive hardware detail

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 $u_{max} = 2$

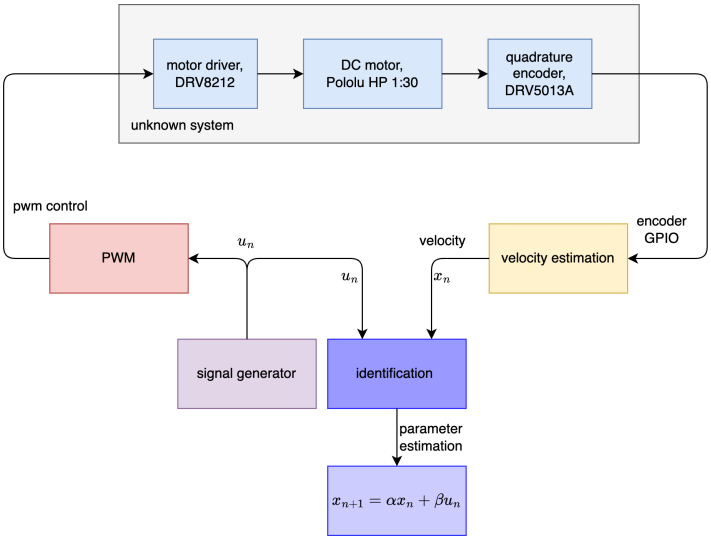


Figure 2.3: Motor identification process

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

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

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

$$\alpha = -\frac{1}{\tau} \tag{2.16}$$

$$\beta = \frac{k}{\tau} \tag{2.17}$$

with given parameters k and τ 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 τ.

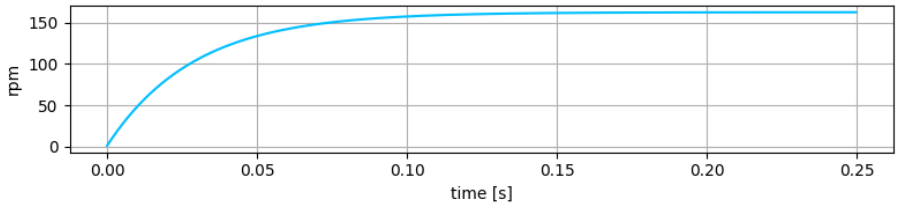


Figure 2.4: Motor step response

We start with simplest identification method, which works only for 1st order non-oscilating 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.

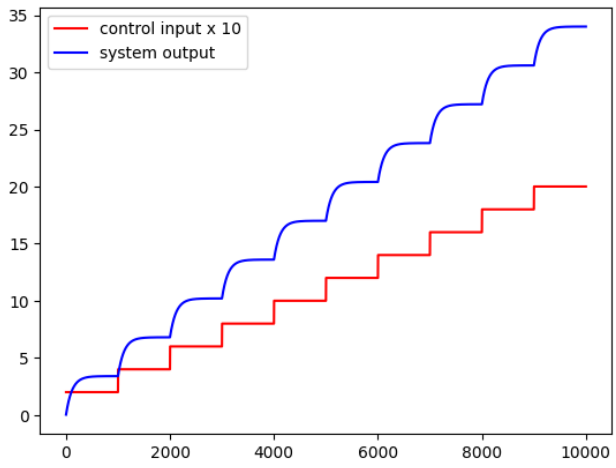


Figure 2.5: Motor constant estimation

Algorithm for estimating motor constant :

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

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.

```
# input levels
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])

# reset dynamical system into zero state
ds.reset()

for j in range(len(u_values)):

    x_mean = []
    for i in range(1000):

        # convert scalar u to column vector
        u    = u_values[j]
        u    = numpy.array([[u]])

        # compute dynamical system one step
        x, _ = ds.forward_state(u)

        # after steady state, store x
        if i > 500:
            x_mean.append(x[0][0])

    x_mean = numpy.array(x_mean)
    x_mean = x_mean.mean()

    k_est = x_mean/u_values[j]

    print(u_values[j], k_est)

k_mean = k_est.mean()

# average k estimate
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 τ. 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, briging 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 ΔT
   a) $if\ u_{in} > 0\ and\ x(n) > 0.632ku_{in}$
   increment $n_{HalfPeriods}$, and set $u_{in} = -u_{in}$
   b) $else\ if\ u_{in} < 0\ and\ x(n) < 0.632ku_{in}$
   increment $n_{HalfPeriods}$, and set $u_{in} = -u_{in}$

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

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

Where ΔT 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

TODO

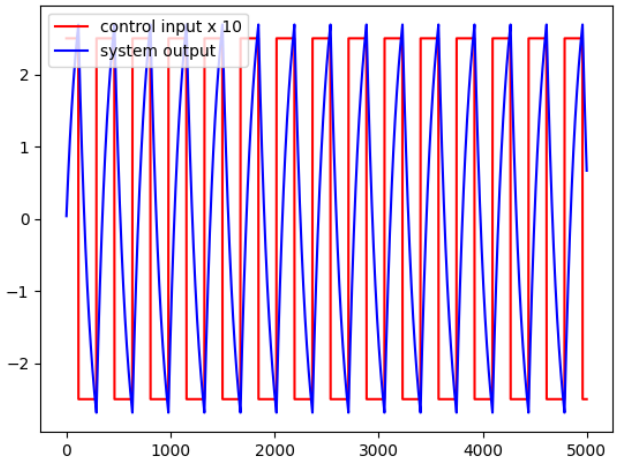


Figure 2.6: Motor time constant estimation

Chapter 3

Motor control

motor velocity controller

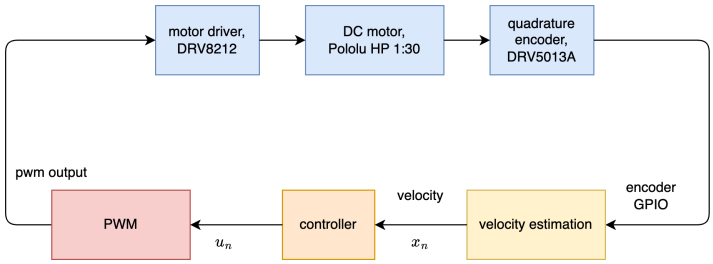


Figure 3.1: Velocity control overview

3.1 PID control

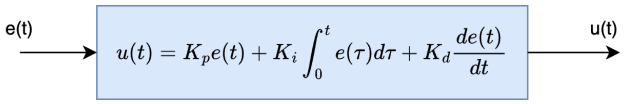


Figure 3.2: Textbook continuous PID controller

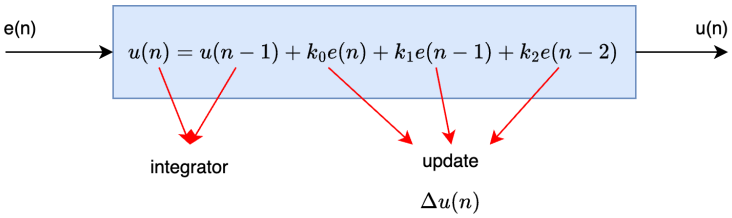


Figure 3.3: Discrete PID controller

$$k_0 = K_p + K_i \Delta T + \frac{K_d}{\Delta T} \tag{3.1}$$

$$k_1 = -K_p - 2\frac{K_d}{\Delta T} \tag{3.2}$$

$$k_2 = \frac{K_d}{\Delta T} \tag{3.3}$$

PID control - P only controller

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

$$u(n+1) = u(n-1) + k_p e(n) - k_p e(n-1) \tag{3.4}$$

$$k_p = 0.01 \tag{3.5}$$

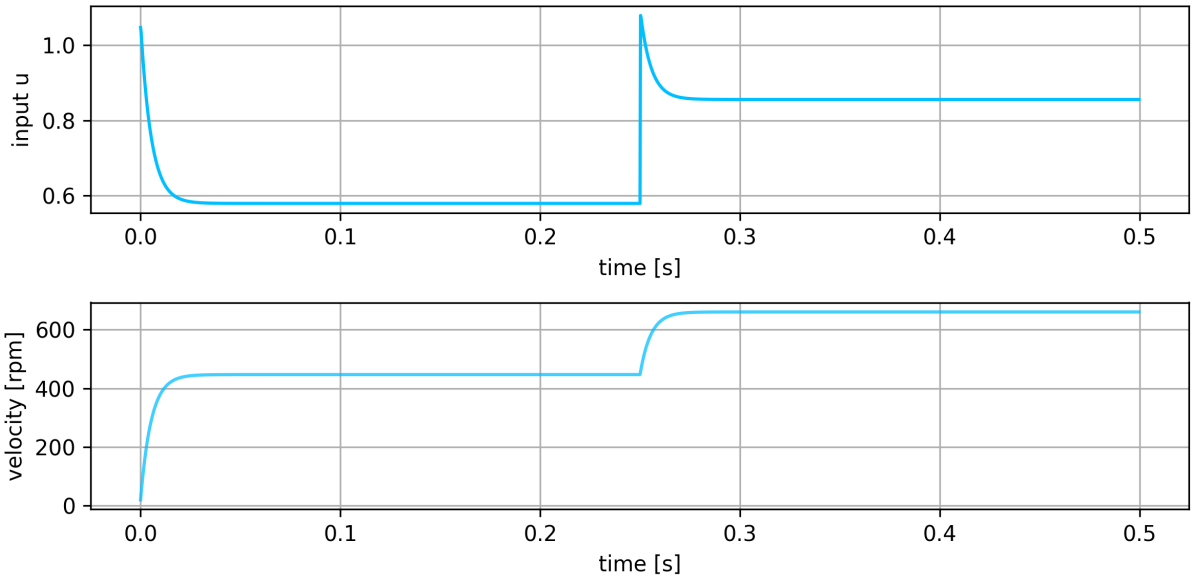


Figure 3.4: P-only controller

PID control - PI

too high I term

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

$$u(n+1) = u(n-1) + (k_p + k_i\Delta T)e(n) - k_p e(n-1) \tag{3.6}$$
$$k_p = 0.01 \tag{3.7}$$
$$k_i\Delta T = 0.005 \tag{3.8}$$

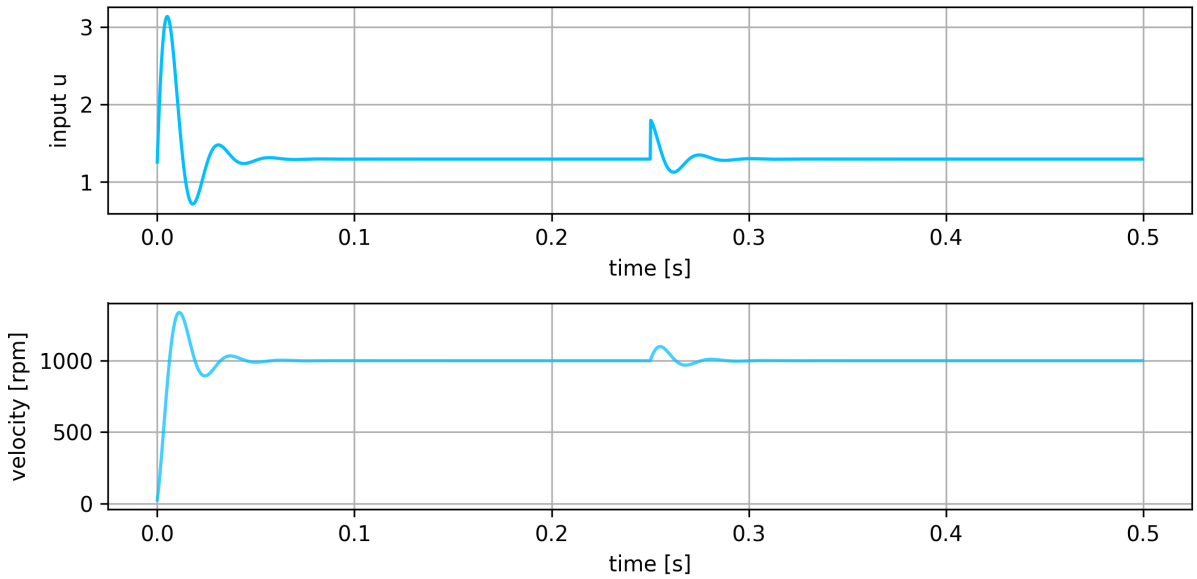


Figure 3.5: PI controller

correct I term

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

- **no overshot, no steady state error**

$$u(n+1) = u(n-1) + (k_p + k_i\Delta T)e(n) - k_p e(n-1) \tag{3.9}$$
$$k_p = 0.01 \tag{3.10}$$
$$k_i\Delta T = 0.0002 \tag{3.11}$$

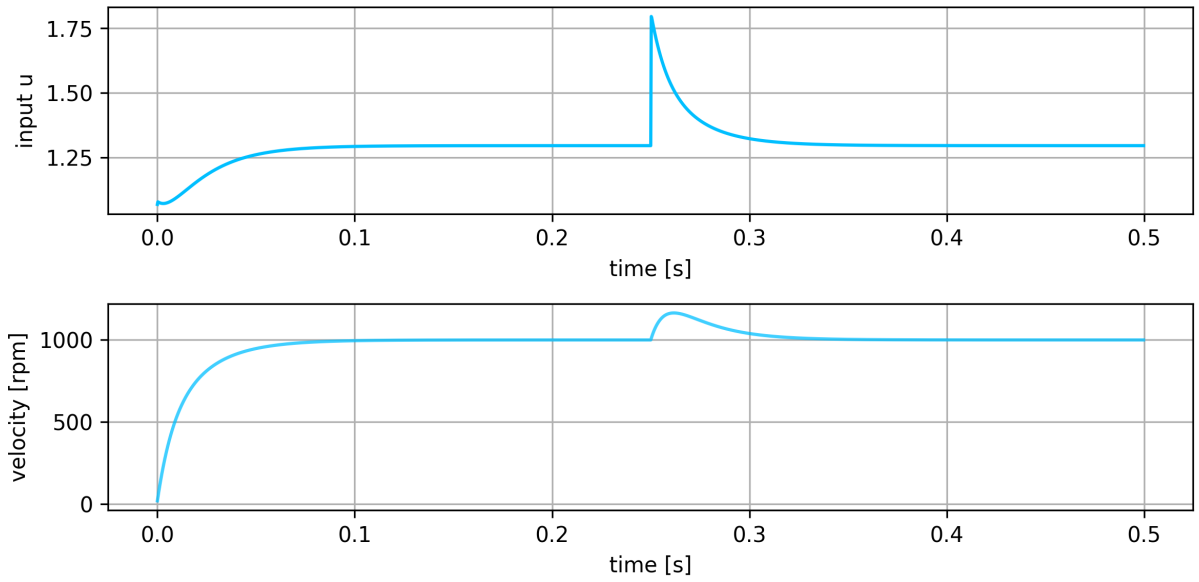


Figure 3.6: PI controller with correct parameters

complete discrete PID algorithm

1. calculate u-change candidate :

$$\Delta\hat{u}(n) = k_0 e(n) + k_1 e(n) + k_2 e(n)$$

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

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

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

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

Full Python code example for discrete PID controller :

```
class PID:

def __init__(self, kp, ki, kd, antiwindup = 10**10, du_max=10**10):
    self.k0 = kp + ki + kd
    self.k1 = -kp -2.0*kd
    self.k2 = kd

    self.e0 = 0.0
    self.e1 = 0.0
    self.e2 = 0.0

    self.antiwindup = antiwindup
    self.du_max     = du_max

def forward(self, xr, x, u_prev):
    # error compute
    self.e2 = self.e1
    self.e1 = self.e0
    self.e0 = xr - x

    # u-change computing
    du = self.k0*self.e0 + self.k1*self.e1 + self.k2*self.e2

    # kick clipping, maximum output value change limitation
    du  = numpy.clip(du, -self.du_max , self.du_max)

    # antiwindup, maximum output value limitation
    u   = numpy.clip(u_prev + du,  -self.antiwindup, self.antiwindup)

    return u
```

3.2 Optimal control

Optimal control is solution of following optimisation problem

$$\min_{x(.),u(.)} \sum_{n=0}^{\infty} x^T(n)Qx(n) + u^T(n)Ru(n) \tag{3.12}$$

$$s.t.\ x(n+1) = Ax(n) + Bu(n) \tag{3.13}$$

the cost function is quadratic

$$J = \sum_{n=0}^{\infty} x^T(n)Qx(n) + u^T(n)Ru(n) \tag{3.14}$$

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.

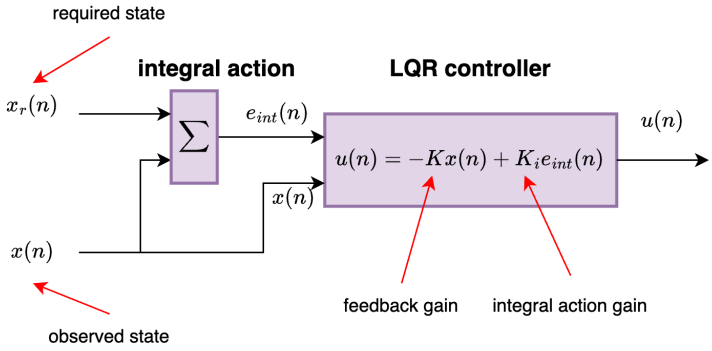


Figure 3.7: LQR controller with integral action

$$u(n) = -Kx(n) + K_i e_{int}(n) \tag{3.15}$$
$$e_{int}(n) = e_{int}(n-1) + x_r(n) - x(n) \tag{3.16}$$

where :

- $x(n)$ is observerd 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)

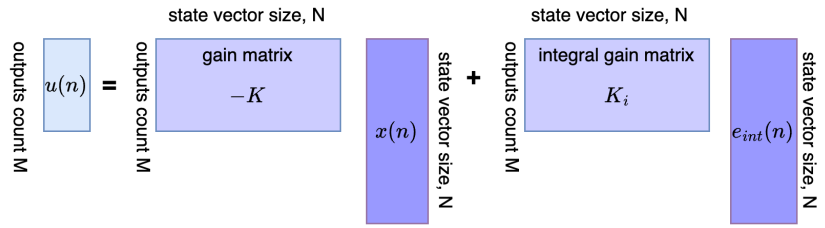


Figure 3.8: Matrices shapes in LQR controller