11
Case Studies

After reading this chapter the reader will:

1. experiment his knowledge in regard to the design and real time implementation of mechatronic systems
2. live the design of mechatronic systems from A to Z
3. be able to perform the different phases of the design of mechatronic systems
4. be able to solve the control problem and establish the control law that we have to implement in real time
5. be able to write programs in C language using the interrupt concept for real time implementation

11.1 Introduction

In the previous chapters we developed some concepts that we illustrated their applications by academic examples to show the readers how these results apply. More specifically, we have seen how to design the mechatronic systems and we have

presented the different steps that we must follow to success the design of the desired mechatronic system. We have presented the approaches that we must use in the design of:

- the mechanical part
- the electronic circuit
- the program in C for the real-time implementation

These tools were applied for some practical systems and more details were given to help the reader to execute his own design.

For the control algorithms most of the examples we presented were academic with perfect models. Unfortunately, for a practical system the model we will have is just a realization that may describe the system in some particular conditions, and for some reasons this model will not work perfectly as expected during the real-time implementation of the algorithms. This can be caused by different neglected dynamics that may change the behavior for some frequencies.

The aim of this chapter is to show the reader how we can implement in real-time the theoretical results we developed earlier in the previous chapters for practical systems. We will proceed gradually and show all the steps to make it easy for the readers. The case studies we consider in this chapter are those that are discussed and designed in the previous chapters.

11.2 Velocity Control of the dc Motor Kit

As a first example, let us consider the speed control of the dc motor driving a mechanical part. The choice of this example is very important since most of the systems will use such dc motor. The dc motor we will consider is manufactured by Maxon company. This motor is very important since it comes with a gearbox (ratio 6:1) and an encoder that gives one hundred pulses per revolution, which gives 600 pulses per revolution that we exploit by using quadrature method to bring it to two thousand four hundred pulses per revolution. The system we are using in this example for the real-time implementation of control algorithms we presented earlier if more flexible and offer more advantages.

The data sheet of this motor gives all the important parameters and therefore the transfer function of this actuator can be obtained easily. The load we are considering in this example is a small disk with graduation that we would like to control in speed and later on in position. This setup is illustrated in Figs. 11.1-(11.2). The disk we are considering has a diameter equal to 0.06 m and a mass equal to 0.050 Kg. With these data and the one of the data sheet of the dc motor, we can get the transfer function between the velocity of the disk and the input voltage.

Let us first of all concentrate on speed control of the load. In this case to establish the transfer function of this system (dc motor actuator and its load) we can either use the data sheet, the information we have on the disk and the results in Boukas [1]

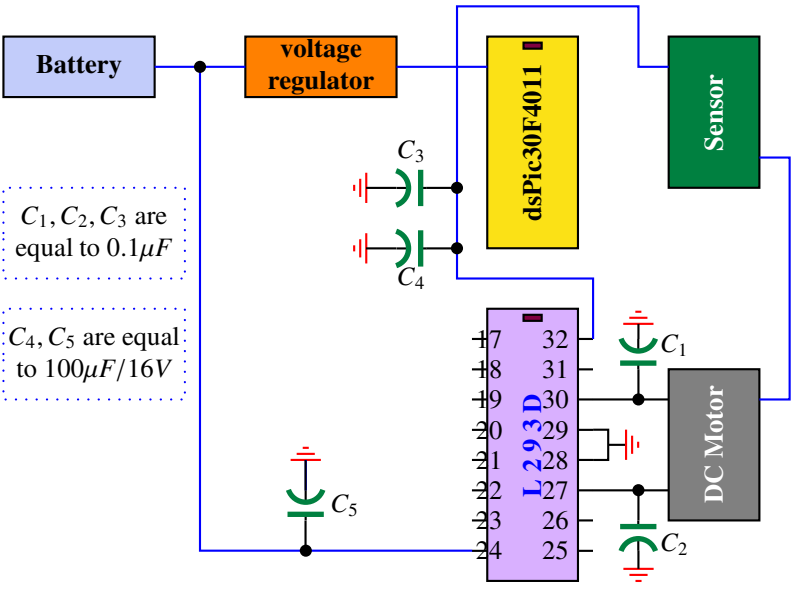

Fig. 11.1 Electronic circuit of dc motor kit

or proceed with an identification. Using the first approach the results of Chapter 2 in [1], we get:

$$G(s) = \frac{K}{\tau s + 1}$$

with

$$\begin{aligned} K &= 48.91 \\ \tau &= 63.921\ ms \end{aligned}$$

For the identification of our system, we can do it in real-time using the real-time implementation setup and appropriate C program. Since the microcontroller owns limited memory, the identification can be done into two steps. Firstly, in a first experiment the gain, K, is determined, then using this gain we compute the steady state value that can be used to compute the constant time τ.

To design the controller we should first of all specify the performances we would like that our system has. As a first performance, we require that our system is stable. It is also needed that the system speed will have a good behavior at the transient regime with a zero error at the steady state regime for a step reference. For the transient we would like that our load has a settling time with 5% less or equal to $\frac{3\tau}{5}$ and an overshoot less or equal to 5%.

To accomplish the design of the appropriate controller we can either make the design in continuous-time and then obtain the algorithm we should program in the software part, or proceed with all the design in the discrete-time directly. In the rest of this example, we will opt for the second approach.

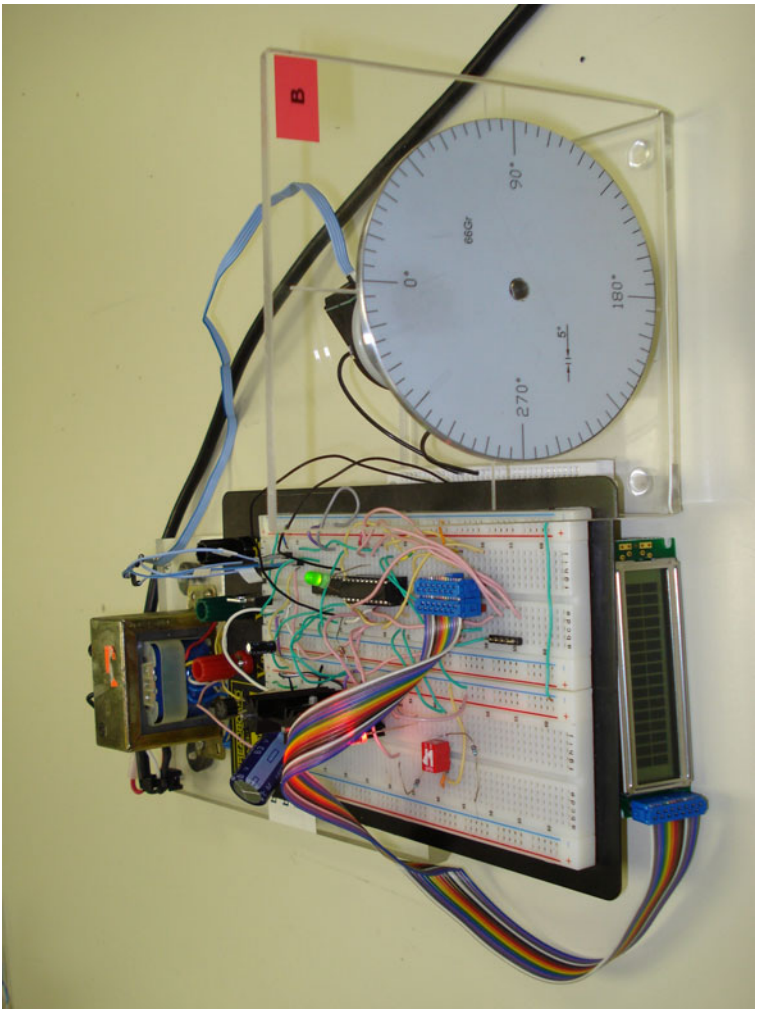

Fig. 11.2 Real-time implementation setup

From the expression of the system transfer function, and the desired performances it results that we need at least a proportional and integrator (PI) controller. The transfer function of this controller is given by:

$$C(s) = K_P + \frac{K_I}{s}$$

where K_P and K_I are gains to be determined to force the load to have the performances we imposed.

Using a zero-order-holder and the $\mathscr{Z}$-transform table we get:

$$\begin{aligned} G(z) &= \frac{Kz\left(1-e^{-\frac{T}{\tau}}\right)\left(1-z^{-1}\right)}{(z-1)\left(z-e^{-\frac{T}{\tau}}\right)} \\ &= \frac{K\left(1-e^{-\frac{T}{\tau}}\right)}{z-e^{-\frac{T}{\tau}}} \end{aligned}$$

For the controller, using the trapezoidal discretisation we get:

$$\begin{aligned} C(z) &= \frac{U(z)}{E(z)} = K_P + K_I\frac{T}{2}\frac{z+1}{z-1} \\ &= \frac{\left(K_P+\frac{TK_I}{2}\right)z+\left(-K_P+\frac{TK_I}{2}\right)}{z-1} \end{aligned}$$

Dividing the numerator and the denominator by z and going back to time, we get:

$$u(k) = u(k-1) + \left(K_P+\frac{TK_I}{2}\right)e(k) + \left(-K_P+\frac{TK_I}{2}\right)e(k-1)$$

Combining the transfer function of the actuator and its load and the one of the controller we get the following closed-loop transfer function:

$$F(z) = \frac{K\left(1-e^{-\frac{T}{\tau}}\right)\left(K_P+\frac{TK_I}{2}\right)z + K\left(1-e^{-\frac{T}{\tau}}\right)\left(-K_P+\frac{TK_I}{2}\right)}{z^2+\left(K\left(K_P+\frac{TK_I}{2}\right)\left(1-e^{-\frac{T}{\tau}}\right)-1-e^{-\frac{T}{\tau}}\right)z + K\left(1-e^{-\frac{T}{\tau}}\right)\left(-K_P+\frac{TK_I}{2}\right)+e^{-\frac{T}{\tau}}}$$

Using now the desired performances, it is easy to conclude that the dominant poles are

$$s_{1,2} = -\zeta\omega_n \pm jw_n\sqrt{1-\zeta^2}$$

where ζ and ω_n represent respectively the damping ratio and the natural frequency of the closed-loop of our system control.

From control theory (see Boukas [1]) it is well known that the overshoot $d\%$ and the settling time t_s at 5% are given by:

$$\begin{aligned} d\% &= 100e^{\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}} \\ t_s &= \frac{3}{\zeta\omega_n} \end{aligned}$$

Using our performances and these expressions we conclude that:

$$\begin{aligned} \zeta &= 0.707 \\ \omega_n &= \frac{5}{\tau\zeta} = 110.6387 rad/s \end{aligned}$$

which give the following dominant poles:

$$s_{1,2} = -78.2216 \pm 78.2452\,j$$

Using the transformation $z = e^{Ts}$ with $T = \frac{\tau}{10} = 0.0064$, we obtain the following dominant poles in the $\mathscr{Z}$-domain:

$$z_{1,2} = 0.5317 \pm 0.2910\,j$$

With these poles we have the following characteristic equation:

$$\begin{aligned}\Delta_d &= (z - 0.5317 - 0.2910\,j)(z - 0.5317 + 0.2910\,j)\\ &= z^2 - 1.0634z + 0.3674\end{aligned}$$

Using now the poles placement technique we get:

$$1 + C(z)G(z) = \Delta_d$$

which implies:

$$\begin{aligned}K_I &= \frac{0.3040}{KT\left(1 - e^{-\frac{T}{\tau}}\right)}\\ K_P &= \frac{-0.4308 + 2e^{-\frac{T}{\tau}}}{2K\left(1 - e^{-\frac{T}{\tau}}\right)}\end{aligned}$$

Using the values of K, T and τ, we get the following expression for the gains K_P and K_I:

$$\begin{aligned}K_P &= 0.1480\\ K_I &= 10.1951\end{aligned}$$

Remark 11.2.1 *Cautions have to be made in this case since we don't care about the positions of the zero of the transfer function and therefore we may have some surprises when implementing this controller. It is clear that the performances we will get (settling time and the overshoot) will depend on the positions of the zero. For more details on this matter we refer the reader to Boukas [1].*

To implement now this PI control algorithm and assure the desired performances we will use a microcontroller from Microship[1]. This choice is due to our experience with this type of microcontroller. The reader can keep in mind that any other microcontroller from other manufacturer with some small changes will do the job. In this example we will use the microcontroller dsPIC30F4011 from Microhip.

The code for our implementation is made in *C*-language. This language is adopted for its simplicity. The implementation has the following structure:

```
//
// Put here the include
//

#include "p30F4011.h"       // proc specific header

//
```

[1] See www.microchip.com

```
 // Define a struct
 //
  typedef struct {
       // PI Gains
       float K_P;      // Propotional gain
       float K_I;       // Integral gain
    //
    // PI Constants
    //
    float Const1_pid;  // KP + T KI/2
    float Const2_pid;  // -KP + T KI/2
    float Reference;   // speed reference

    //
    // System variables
    //
    float y_k;   // y_m[k] -> measured output at time k
    float u_k;   // u[k]   -> output at time k
    float e_k;   // e[k]   -> error at time k

    //
    // System past variables
    //
    float u_prec;     // u[k-1] -> output at time k-1
    float e_prec;     // e[k-1] -> error at time k-1

  }PIStruct;

  PIStruct thePI;

  thePI.Const1= thePI.K_P+T*thePI.K_I/2;
  thePI.Const2=-thePI.K_P+T*thePI.K_I/2;
  thePI.Reference=600;


//
// Functions
//
float ReadSpeed(void);

float ComputeControl(void);

float SendControl(void);

//
// Interrupt program here using Timer 1 (overflow of counter Timer 1)
//
void __ISR _T1Interrupt(void)     // interrupt routine code
  {
```

```
        // Interrupt Service Routine code goes here
        float Position_error;

        //
        // Read speed
        //
         thePI.y_m=ReadSpeed();

        thePI.e_k= thePI.Reference-thePI.y_m;

        //
        // Compute the control
        //
        ComputeContrl();

        //
        // Send control
        //
        SendControl();

        IFS0bits.T1IF=0;    // Disable the interrupt
  }


int main ( void )           // start of main application code
{
// Application code goes here
int i;

// Initialize the variables Reference and ThePID.y_m (it can be read
 from inputs) Reference =  0x8000;  // Hexadecimal number
(0b...  Binary number) ThePID = 0x8000;

// Initialize the registers
TRISC=0x9fff; // RC13 and RC14 (pins 15 and 16) are configured as outputs
IEC0bits.T1IE=1;   // Enable the interrupt on Timer 1

// Indefinite loop
while (1)
{
}
return 0
}


      % ReadSpeed function
      int ReadSpeed (void)
           {
           }
```

```
% ComputeControl function
int ComputeControl (void)
{
thePI.u_k=thePI.u_prec+thePI.Const1*thePI.e_k+thePI.Const2*thePI.e_prec;
}

% SendControl function
int Send Control (void)
{
sendControl()

//
// Update past data
//
thePI.u_prec=thePI.u_k;
ThePI.e_prec=thePI.e_k;
}
```

As it can be seen from this structure, first of all we notice that the system will enter the loop and at each interrupt the call for the functions:

- ReadSpeed;
- ComputeControl;
- SendControl;

is made and the appropriate action is taken.

The ReadSpeed function returns the load speed at each sampling time that will be used by the ComputeControl function. The SendControl function sends the appropriate voltage to the actuator via the L293D chip.

Using the compiler HighTec C to get the hex code and the PicKit-2 to upload the file in the memory of the microcontroller. For more detail on how to get the hex code we invite the reader to the manual of the compiler HighTec C or the compiler C30 of Microchip.

The state approach in this case is trivial and we will not develop it.

11.3 Position Control of the dc Motor Kit

Let us focus on the load position control. Following similar steps as for the load velocity control developed in the previous section, we need firstly to choose the desired performances we would like our system will have. The following performances are imposed:

- system stable in the closed-loop;
- settling time t_s at 2% equal to the best one we can have
- overshoot equal to 5%
- steady-state equal to zero for a step function as input

Using the performances and the transfer function, it is easy to conclude that a proportional controller K_P is enough to satisfy these performances.

In this example, we will use the continuous-time approach for the design of the controller. Based on the past chapter, the model of our system is given by:

$$G(s) = \frac{K}{s(\tau s + 1)}$$

where K and τ take the same values as for the speed control.

Let the transfer controller be give by:

$$C(s) = K_P$$

Using these expression, the closed-loop transfer function is given by:

$$\begin{aligned} F(s) &= \frac{C(s)G(s)}{1 + C(s)G(s)} \\ &= \frac{\frac{KK_P}{\tau}}{s^2 + \frac{1}{\tau}s + \frac{KK_P}{\tau}} \end{aligned}$$

Since the system is of type 1, it results that the error to a step function as input is equal to zero with a proportional controller.

Based on the specifications, the following complex poles:

$$s_{1,2} = -\zeta w_n \pm j w_n \sqrt{1 - \zeta^2}$$

will do the job and the corresponding characteristic equation is given by:

$$s^2 + 2\zeta w_n s + w_n^2 = 0.$$

Equating this with the one of the closed-loop system we get:

$$\begin{aligned} 2\zeta w_n &= \frac{1}{\tau} \\ w_n^2 &= \frac{KK_P}{\tau}. \end{aligned}$$

To determine the best settling time t_s at 2 %, notice that we have:

$$t_s = \frac{4}{\zeta w_n}.$$

Using now the fact:

$$\zeta w_n = \frac{1}{2\tau}$$

we obtain:

$$t_s = 8\tau$$

Therefore the best settling time at %2 we can have with this controller is 8 times the constant time of the system. Any value less than will be attainable. In fact, this is trivial if we look to the root locus of the closed-loop system when varying K_P. This is given by Fig. 11.3. To fix the gain of the controller the desired poles $s_{1,2} = -7.5 \pm j$, we use the figure and choosing a $\zeta = 0.707$. This gives $K_P = 0.1471$.

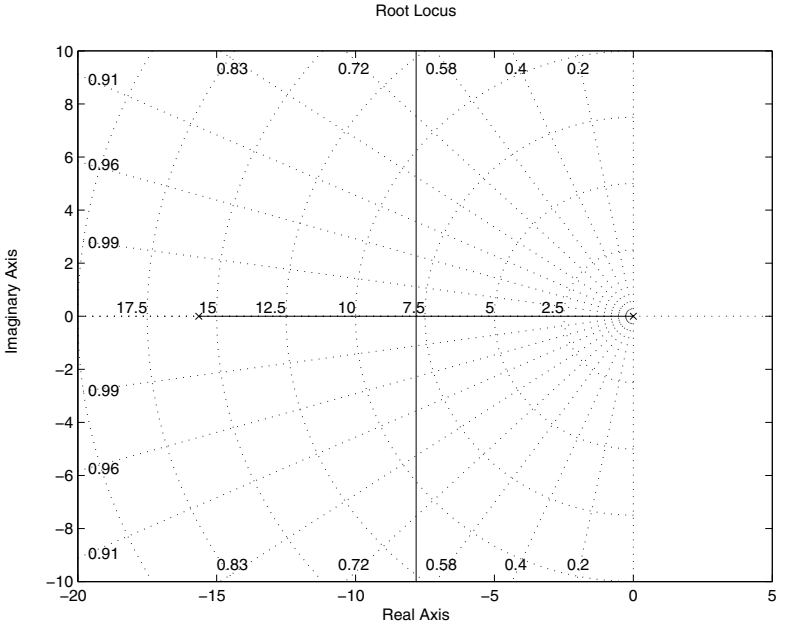

Fig. 11.3 Root locus of the dc motor with a proportional controller

Using this controller, the time response for a step function of amplitude equal to 30 degrees is represented by the Fig. 11.4 from which we conclude that the designed controller satisfies all the desired performances with a settling time at %2 equal to 0.5115 s. But if we implement this controller, the reality will be different from simulation since the backlash of the gearbox is not included in the used model and therefore in real-time the result will be different and the error will never be zero. To overcome this problem we can use a proportional and derivative controller that may give better settling time at %2. Let the transfer function of this controller be given by:

$$C(s) = K_P + K_D s$$

where K_P and K_D are the gain to be determined.

Remark 11.3.1 *It is important to notice that the use of a proportional and derivative controller will introduce a zero in the closed-loop transfer that may improve the settling time if it is well placed. Depending on its position, the overshoot and the settling time will be affected. For more details on this matter we refer the reader to [1].*

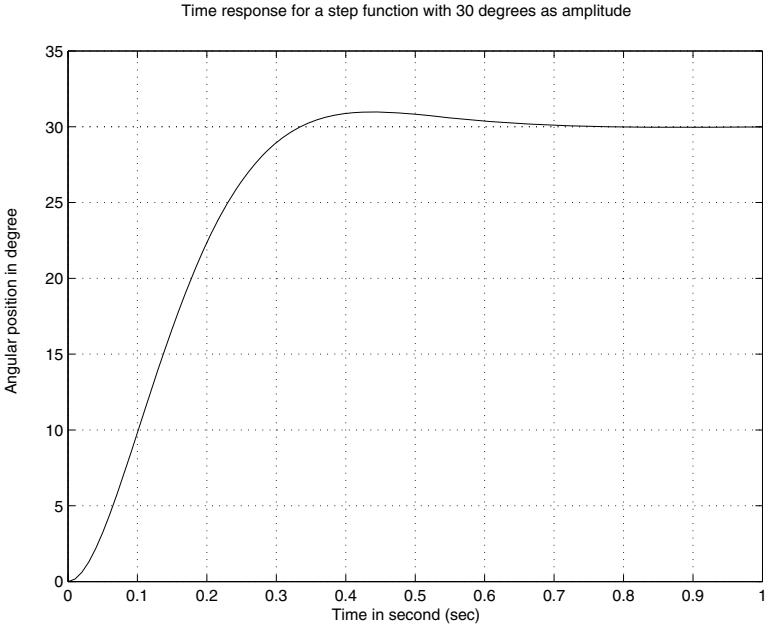

Fig. 11.4 Time response for a step function with 30 degrees as amplitude

With this controller the closed-loop transfer function is given by:

$$\begin{aligned} G(s) &= \frac{C(s)G(s)}{1 + C(s)G(s)} \\ &= \frac{\frac{KK_Ds + KK_P}{\tau}}{s^2 + \frac{1+KK_D}{\tau}s + \frac{KK_P}{\tau}} \end{aligned}$$

As before two complex poles are used for the design of the controller. If we equate the two characteristic equations we get:

$$\begin{aligned} 2\zeta w_n &= \frac{1 + KK_D}{\tau} \\ w_n^2 &= \frac{KK_P}{\tau}. \end{aligned}$$

In this case we have two unknown variables K_P and K_D and two algebraic equations which determines uniquely the gains. Their expressions are given by:

$$\begin{aligned} K_P &= \frac{\tau w_n^2}{K} \\ K_D &= \frac{2\tau\zeta w_n}{K} \end{aligned}$$

Using now the desired performances, we conclude similarly as before that the steady error to an input equal to step function of amplitude equal to 30 degrees for instance is equal to zero and the damping ratio ζ corresponding to an overshoot equal to %5 is equal to 0.707. The settling time, t_s at % 2, that we may fix as a proportion of the time constant of the system, gives:

$$w_n = \frac{4}{\zeta t_s}.$$

Now if we fix the settling time at 3τ, we get:

$$w_n = 29.4985.$$

Using these values we get the following ones for the controller gains:

$$\begin{aligned} K_P &= 1.1374 \\ K_D &= 0.0545. \end{aligned}$$

which gives the following complex poles:

$$s_{1,2} = -28.6763 \pm 6.9163\, j.$$

and the zero at:

$$z = -20.8618.$$

Using this controller the time response for an input with an amplitude equal to 30 degrees is represented in Fig. 11.5. As it can be seen from this figure that the overshoot and the settling time are less those obtained using the proportional controller.

To implement either the proportional or the proportional and derivative controllers we need to get the recurrent equation for the control law. For this purpose, we need to discretize the transfer function of the controller using the different methods presented earlier. Let us use the trapezoidal method which consists to replace s by $\frac{2}{T}\frac{z-1}{z+1}$. This gives:

$$\begin{aligned} C(z) &= K_p \text{ for the proportional controller} \\ C(z) &= K_P + K_D \frac{2}{T}\frac{z-1}{z+1} \text{ for the proportional and derivative controller} \end{aligned}$$

If we denote by $u(k)$ and $e(k)$ by the control and the error between the reference and the output at instant kT, we get the following expressions:

1. for the proportional

$$u(k) = K_P e(k)$$

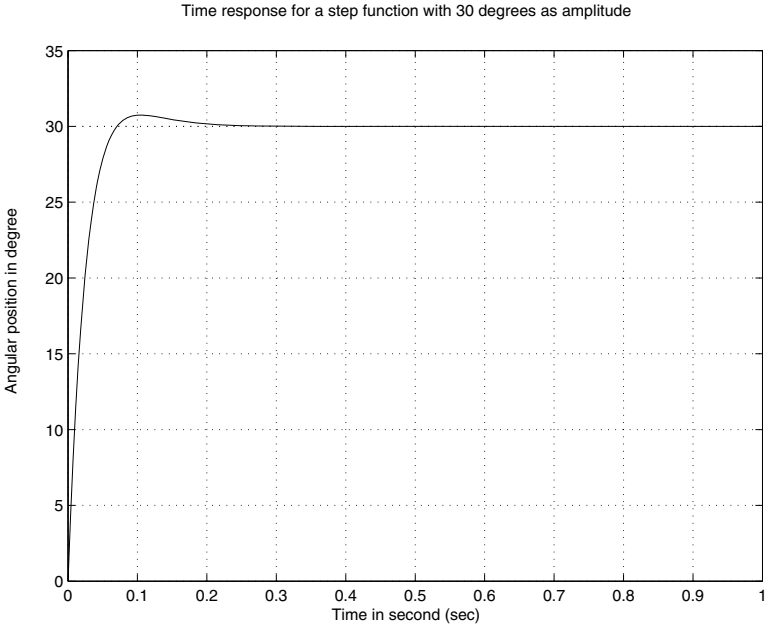

Fig. 11.5 Time response for a step function with 30 degrees as amplitude

2. for the proportional and derivative controller

$$u(k) = -u(k-1) + \left(K_P + \frac{2K_D}{T}\right)e(k) + \left(K_P - \frac{2K_D}{T}\right)e(k-1)$$

The implementation is this controller uses the same function with some minors changes. The listing the corresponding functions is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Main program                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

main
% Data

% Variables

% While loop
  while (1)
     do
        ReadSpeed;
        ComputeControl;
        SendControl;
     end;
```

```
% ReadSpeed function

% ComputeControl function

% SendControl function
```

Let us now use the state space representation of this example and design a state feedback controller that will guarantee the desired performances. For this case we will firstly assume the complete access to the states and secondly we relax this assumption by assuming that we have access only to the position. As we did previously we can proceed either in the continuous-time or in discrete-time.

Previously we establish the state space description of this system and it is given by:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{\tau} \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{K}{\tau} \end{bmatrix} u(t)$$

where $x(t) \in \mathbb{R}^2$ ($x_1(t) = \theta(t)$ and $x_2(t) = \dot{\theta}(t)$), and $u(t) \in \mathbb{R}$ (the applied voltage).

From the desired performances with a settling time at %2 equal to 3τ, we get the same dominant poles as before, and therefore the same characteristic equation, $\Delta_d(s) = s^2 + 2\zeta w_n s + w_n^2 = 0$ (with $\zeta = 0.707$ and $w_n = \frac{4}{3\zeta\tau}$). Using the controller expression the closed-loop characteristic equations given by:

$$\det(s\mathbb{I} - A + BK) = 0$$

By equalizing these two equations we get the following gains:

$$\begin{aligned} K_1 &= 1.1146 \\ K_2 &= 0.0326 \end{aligned}$$

Using this controller the time response for an input with an amplitude equal to 30 degrees is represented in Fig. 11.6. As it can be seen from this figure that the overshoot and the settling time are those we would like to have. It is important to notice the existence of the error at the steady state regime. This error can be eliminated if we add an integral action in the loop. For more detail on this we refer the reader to [1].

For the second case since we don't have access to the load speed we can either compute it from the position or use an observer to estimate the system state. As it was said earlier, the poles that we use for the design of the observer should be faster than those used in the controller design.

Choosing the following poles ($s_{1,2}$ 4 times the real part of those used in the design of the controller):

$$\begin{aligned} s_{1,2} &= -4\zeta w_n \pm j w_n \sqrt{1-\zeta^2} \\ &= -83.4218 \pm 20.8618\, j \end{aligned}$$

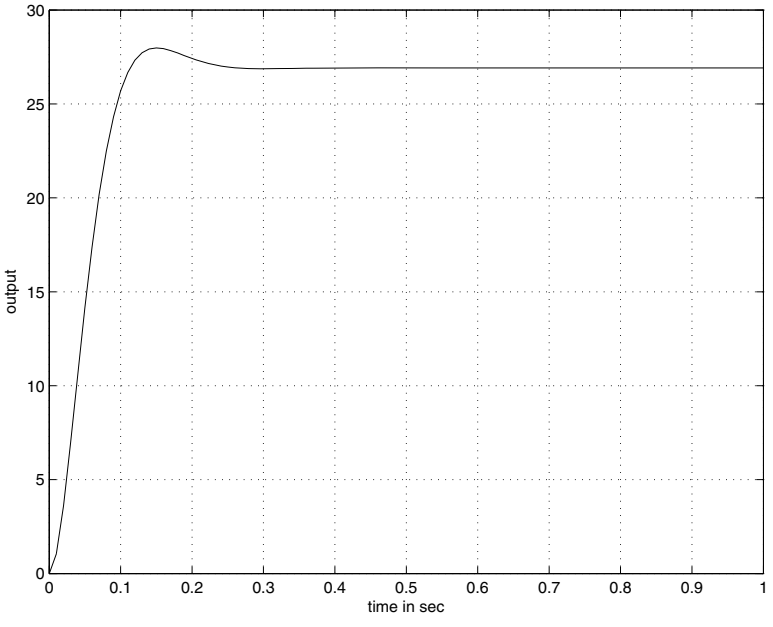

Fig. 11.6 Time response for a step function with 30 degrees as amplitude

we get the following gains for the observer:

$$L_1 = 151.2$$
$$L_2 = 5029.4$$

The gains of the controller remain the same as for the complete access case to the state vector.

In the following Matlab we provide the design of the controller and the observer at the same time and give simulation that shows the behavior of the states of the system and observer with respect to time.

```
clear all

%data
tau=0.064
k=48.9
A = [0 1;0 -1/tau];
B = [0 ; k/tau];
C = [1 0];
D = 0;

% controller design
K = acker(A,B,[-3+3*j -3-3*j]);
L = acker(A',C',[-12+3*j -12-3*j])';
```

```
% Simulation data
Ts = 0.01;
x0 = [1 ; 1];
z0 = [1.1 ; 0.9];
Tf = 2;    %final time

%augmented system
Ah = [A          -B*K;
      L*C    A-B*K-L*C];
Bh = zeros(size(Ah,1),1);
Ch = [C D*K];
Dh = zeros(size(Ch,1),1);
xh0 = [x0 ; z0];
t=0:Ts:Tf;
u = zeros(size(t));
m = ss(Ah,Bh,Ch,Dh);

%simulation
[y,t,x] = lsim(m,u,t,xh0);

%plotting
figure;
plot(t,y);
title('Output');
xlabel('Time in sec')
ylabel('Output')
grid

figure;
plot(t,x(:,1:size(A,1)));
title('States of the system');
xlabel('Time in sec')
ylabel('System states')
grid

figure;
plot(t,x(:,size(A,1)+1:end));
title('states of the observer');
xlabel('Time in sec')
ylabel('Observer states')
grid
```

Figs. (11.7)-(11.9) gives an illustration of the output, the system's states and the observer's states.

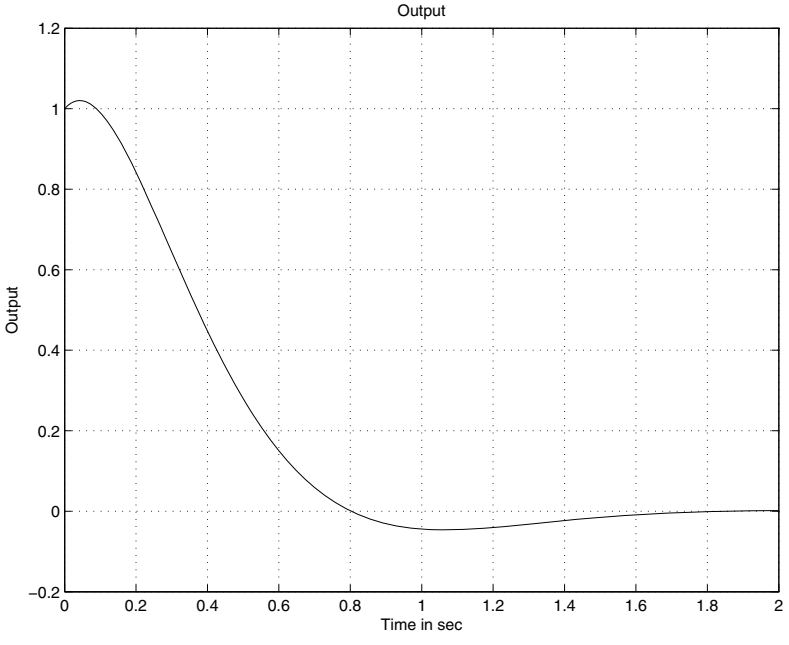

Fig. 11.7 Output versus time

We can also design the state feedback controller using the linear quadratic regulator. In fact, if we chose the following matrices for the cost function:

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix}$$
$$R = 10$$

Remark 11.3.2 *In general, there is no magic rule for the choice the matrices for the cost function. But in general the fact that we use a high value for the control for instance will force the control to take small values and may prevent saturation.*

Using these matrices and the Matlab function, **lqr**, we get:

$$K = \begin{bmatrix} 0.3162 & 0.6875 \end{bmatrix}.$$

We can also design a state feedback controller using the results on robust control part. Since the system has no uncertainties and there is no external disturbance, we can design a state feedback controller for the nominal dynamics. Using the system data and Maltlab, we get:

$$X = \begin{bmatrix} 1.1358 & -0.3758 \\ -0.3758 & 1.1465 \end{bmatrix}$$
$$Y = \begin{bmatrix} -0.0092 & 0.0228 \end{bmatrix}$$

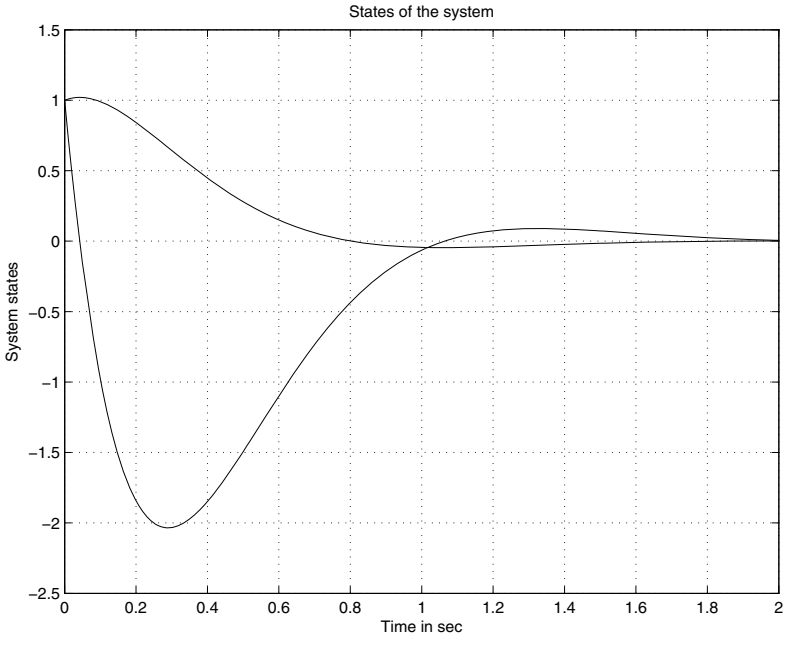

Fig. 11.8 System's states versus time

The corresponding controller gain is given:

$$K = \begin{bmatrix} -0.0017 & 0.0193 \end{bmatrix}.$$

Remark 11.3.3 *Since we have the continuous-time model for the dc motor kit, we have use it to design the controller gain. In this case we have solved the following LMI:*

$$AX + XA^{\top} + BY + Y^{\top}B^{\top} < 0$$

The gain K is given by: $K = YX^{-1}$.

For more detail on the continuous time case we refer the reader to Boukas [2] and the references therein.

11.4 Balancing Robot Control

The balancing robot is a challenging system from the control perspective since it is an unstable system in open-loop. This system has attracted a lot of researchers and many designs have been proposed for this purpose. Here we will present the design developed and tested in mechatronic laboratory at École Polytechnique de Montréal.

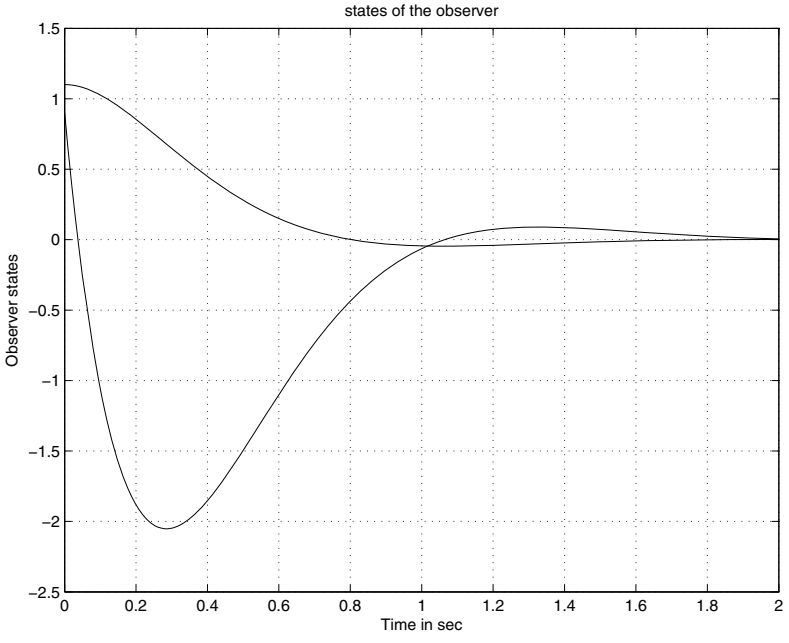

Fig. 11.9 Observer's states versus time

Figs. 11.10-11.11 give an idea of the robot. It was developed for research purpose and to allow the mechatronics students to implement their control algorithm and get familiar with complex system. The robot has two independent wheels each one driven by a dc motor via a gear with a ratio of 1:6. Each motor has an encoder to measure the speed of the shaft. The two motors are attached to the body of the robot. Other sensors like the accelerometer and gyroscope are used to measure the tilt angle. Appropriate filters are introduced to eliminate the noises of the measures and therefore get a useful signal for control.

The brain of the robot is built around a Microchip dsPIC of the family 30F4011. All the programming is done in C and inserted in the dsPIC, after producing the executable code by C30 of Microchip, using the PCKit2.

If we refer to Chapter 4, the mathematical model is given by:

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) \end{cases}$$

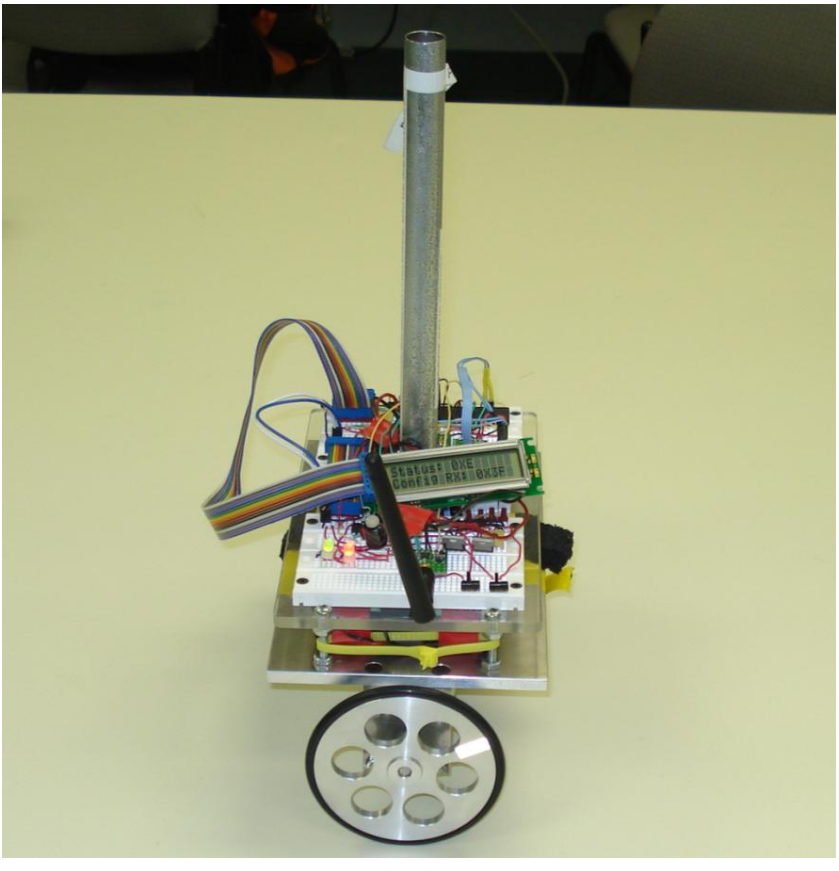

Fig. 11.10 Balancing robot

where

$$
\begin{aligned}
x(t) &= \begin{bmatrix} \psi(t) \\ \dot{\psi}(t) \\ x(t) \\ \dot{x}(t) \end{bmatrix} \\
A &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 147.2931 & -0.4864 & 0 & -10.6325 \\ 0 & 0 & 0 & 1 \\ 0 & -0.0429 & 0 & -0.9371 \end{bmatrix}, \\
B &= \begin{bmatrix} 0 \\ 1.4687 \\ 0 \\ 0.1295 \end{bmatrix}, \\
C &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.
\end{aligned}
$$

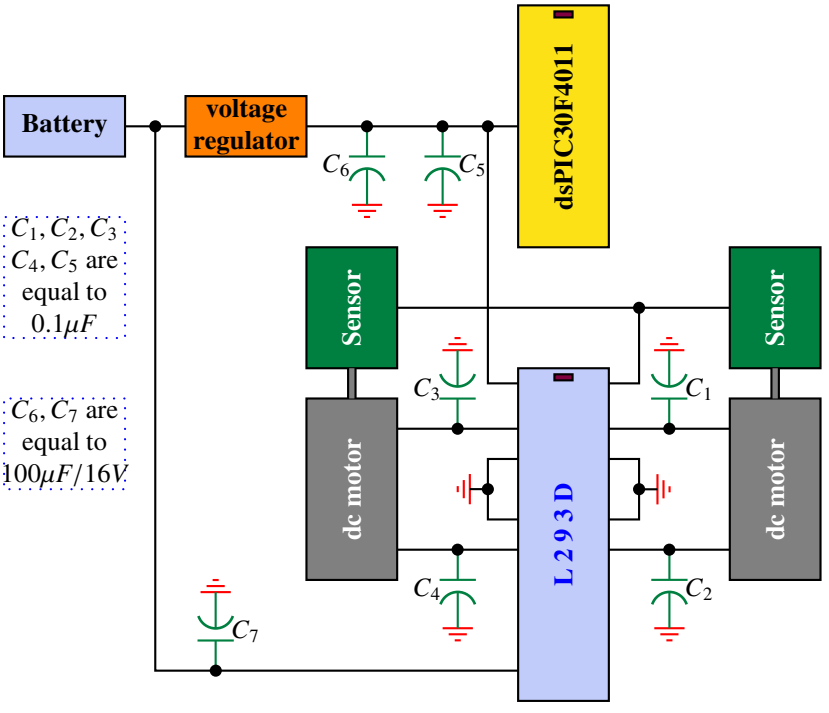

Fig. 11.11 Electronic circuit of the balancing robot

Since the system is unstable in open loop, let us design a state feedback controller that gives the following performances:

1. the system is stable in closed-loop;
2. overshoot less or equal to 5 %;
3. a settling time at 2 % equal to 1.5 s;

From the specifications we get:

$$\begin{aligned} \zeta &= 0.707 \\ w &= \frac{4}{\zeta \times 1.5} = 3.7718 \text{ rad/s} \end{aligned}$$

The corresponding dominant pair of poles is given by:

$$s_{1,2} = -2.6667 \pm 2.6675\, j.$$

Since the matrix A is of rank four, we need to place two more poles to determine the state feedback controller gain, K. Let us chose the following dominated poles:

$$\begin{aligned} s_3 &= -13.3335 \\ s_4 &= -13.3335 \end{aligned}$$

Using the function *acker*, we get the following gain:

$$K = \begin{bmatrix} 339.5604 & 27.5946 & -132.5973 & -76.8450 \end{bmatrix}.$$

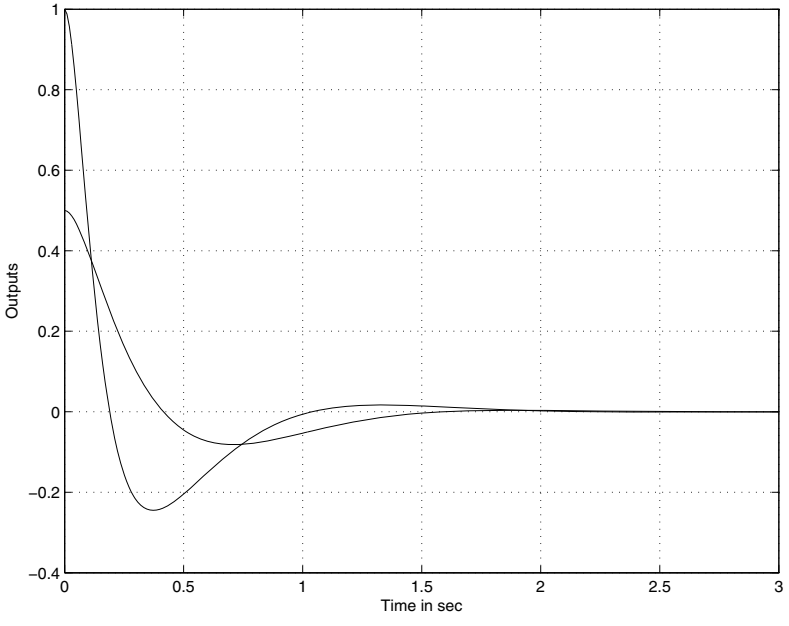

Fig. 11.12 Outputs versus time

The simulation results with this controller are illustrated in Figs. 11.12-11.13. The system starts from an initial condition $x_0^\top = \begin{bmatrix} 1 & 0 & 0.5 & 0 \end{bmatrix}$ with a zero input. If we try to send an input reference we will have errors in states or in outputs. To overcome this an integral action needs to be added. If we denote by $\tilde{x}(t) = \int_0^t (x_r(t) - x(t))\, dt$, with $x_r(t)$ is the position reference and following [1], we get:

$$\begin{aligned} \dot{\eta}(t) &= \tilde{A}\eta(t) + \tilde{B}u(t) \\ y(t) &= \tilde{C}\eta(t) \end{aligned}$$

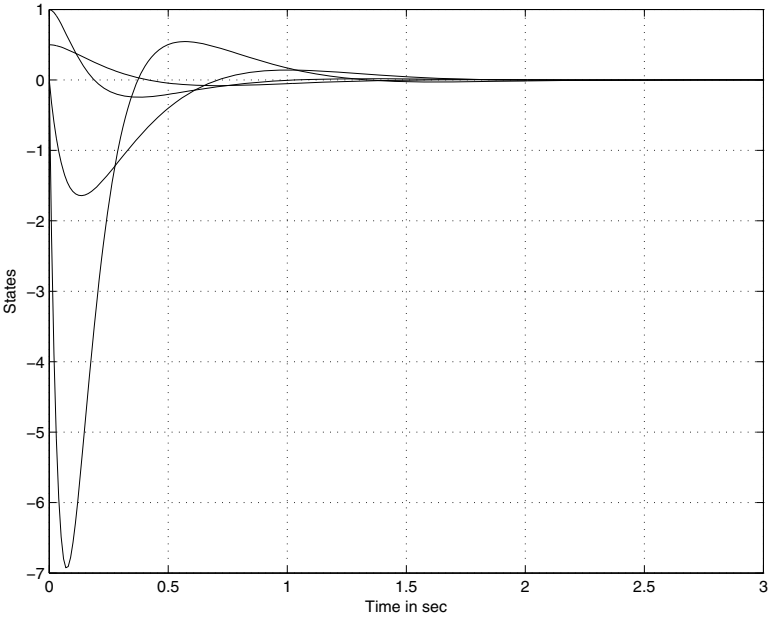

Fig. 11.13 States versus time

where

$$\begin{aligned}
\eta(t) &= \begin{bmatrix} x(t) \\ \tilde{x}(t) \end{bmatrix}, \\
\tilde{A} &= \begin{bmatrix} A & 0 \\ -C & 0 \end{bmatrix}, \\
\tilde{B} &= \begin{bmatrix} B \\ 0 \end{bmatrix}, \\
\tilde{C} &= \begin{bmatrix} C & 0 \end{bmatrix}
\end{aligned}$$

The new dynamics become of order five and we need to fix three dominated poles plus the dominating poles that come from the specifications. These poles are fixed to the following ones:

$$\begin{aligned}
s_{3,4} &= -13.3335 \pm 2.6675\,j \\
s_5 &= -13.3335
\end{aligned}$$

Using the function *acker*, we get the following gain:

$$K = \begin{bmatrix} 339.5604 & 27.5946 & -132.5973 & -76.8450 & 0.1 \end{bmatrix}.$$

We can also design a state feedback controller using the linear quadratic control technique. In fact if we chose the following matrices:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 20 \end{bmatrix}$$
$$R = \begin{bmatrix} 10 \end{bmatrix}$$

and solving the Ricatti equation using the Matlab function *lqr*, we get:

$$K = \begin{bmatrix} 218.9494 & 17.7264 & -1.0000 & -15.7658 \end{bmatrix}$$

The corresponding eigenvalues for the closed-loop are given by:

$$s_1 = -12.7439$$
$$s_2 = -11.5936$$
$$s_3 = -0.9420$$
$$s_4 = -0.1371$$

The simulation results with this controller can be obtained similarly and the detail is omitted.

For this system, we can design also a state feedback controller using the robust control theory. This can be done either in continuous-time or discrete-time. Since our model is in continuous-time we will do the design using the LMI in continuous-time. Solving the appropriate LMI (the one we used for the dc motor kit with different sizes for the matrices, we get:

$$X = \begin{bmatrix} 0.0595 & -0.2156 & 0.1053 & 0.0177 \\ -0.2156 & 1.9238 & -0.0705 & 0.7732 \\ 0.1053 & -0.0705 & 1.4496 & -0.3439 \\ 0.0177 & 0.7732 & -0.3439 & 1.5028 \end{bmatrix}$$
$$Y = \begin{bmatrix} -7.2149 & 27.5211 & -13.6001 & 7.4801 \end{bmatrix}$$

which gives the following gain for the state feedback controller:

$$K = \begin{bmatrix} -174.2563 & -10.5741 & 6.0437 & 13.8536 \end{bmatrix}$$

The corresponding eigenvalues for the closed-loop are given by:

$$s_{1,2} = -6.9770 \pm 6.4079\,j$$
$$s_{3,4} = -0.6028 \pm 0.9598\,j$$

The advantage of this method is that we don't have to provide poles as for the case of pole placement technique. On the top of this the LMI technique can handle more appropriately the presence of saturations in the input if it is the case.

As a lst example, let consider the $\mathscr{H}_\infty$ control problem for the two wheels robot. In this case, we add a term in the state dynamic. This term is $Bw(t)$ where $w(t)$ is

the external disturbance that has finite energy. Solving the appropriate LMI with $\gamma = 0.1$, we get:

$$X = \begin{bmatrix} 1.1233 & -2.7517 & 0.2742 & 1.2667 \\ -2.7517 & 61.0512 & -1.5973 & 14.4055 \\ 0.2742 & -1.5973 & 2.0232 & -3.4932 \\ 1.2667 & 14.4055 & -3.4932 & 16.4930 \end{bmatrix}$$

$$Y = \begin{bmatrix} -143.3219 & 316.5973 & -60.7376 & -33.8808 \end{bmatrix}$$

which gives the following gain for the state feedback controller:

$$K = \begin{bmatrix} -193.7547 & -9.4711 & 39.7088 & 29.5092 \end{bmatrix}$$

The corresponding eigenvalues for the closed-loop are given by:

$$s_{1,2} = -3.8372 \pm 8.9414\,j$$
$$s_{3,4} = -1.9189 \pm 2.0781\,j$$

For the design of other controllers can be obtained easily and we let this as an exercise for the reader since the design is brought to write a program of Matlab similar to those we give in the text.

11.5 Magnetic Levitation System

In this section we will present the magnetic levitation system we presented earlier. This mechatronic system developed in our mechatronic laboratory is composed of two parts: a fixed one that represents the coil and that generates the electromagnetic force and a ferromagnetic object which we would like to place at a certain position by acting on the electromagnetic force generated by the coil. The objective of the system is to control the vertical position of the moving object by adjusting the current in the electromagnet through the input voltage. The object position is measured using a Hall effect sensor. An elecgronic circuit build around a dsPIC30F4011 supplies the coil through an L298, an integrate circuit, with a current that is proportional to the command voltage of the actuator. As the magnetic force can be only attractive, the mutual conductance amplier turns off for negative commands. This system is illustrated by Fig. 11.14.

The mathematical model for this system is given by the following equation:

$$m\ddot{l}(t) = mg - F_1 - F_2$$

where m is the mass of the moving object, $l(t) \in \mathbb{R}^+$ is the distance measured from the electromagnet, F_1 and F_2 are respectively the force generated by the coil when the current is $i(t)$ and the electromagnetic force between the electromagnet and permanent magnet placed of the head of the moving object.

The expression of these force are given by:

$$\begin{aligned} F_1 &= K_1 \frac{i^2(t)}{l^2(t)} \\ F_2 &= K_2 \frac{1}{l^2(t)} \end{aligned}$$

This model in nonlinear that we can linearize to get the following (see Chapter 1):

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

where

$$\begin{aligned} x(t) &= \begin{bmatrix} x_1(t)(position) \\ x_2(t)(velocity) \end{bmatrix} \\ A &= \begin{bmatrix} 0 & 1 \\ \frac{2\left[sign(u_2)k_c u_e^2 + k_p R^2\right]}{mR^2 x_e^3} & 0 \end{bmatrix} \\ B &= \begin{bmatrix} 0 \\ \frac{-2sign(u)k_c u_e}{mR^2 x_e^2} \end{bmatrix} \\ C &= \begin{bmatrix} \frac{-3C_p}{0.032 x_e^4} & 0 \end{bmatrix} \\ D &= \frac{C_b}{0.032R} \end{aligned}$$

The data of the system is given by Table 11.1. Using this data, the matrices are given by:

$$\begin{aligned} A &= \begin{bmatrix} 0 & 1 \\ 2490.8 & 0 \end{bmatrix}, \\ B &= \begin{bmatrix} 0 \\ -1.2711 \end{bmatrix}, \\ C &= \begin{bmatrix} 473.5711 & 0 \end{bmatrix}, \\ D &= \begin{bmatrix} -0.0833 \end{bmatrix}, \end{aligned}$$

It is important to notice that the system is unstable in open loop since it has a pole with a positive real part. This can be checked by computing the eigenvalues of the matrix A.

Let us design a state feedback controller to guarantee the following performances:

1. the system is stable in closed-loop
2. an overshoot less or equal to 0.2 %
3. a settling time at % 2 equal to 0.05 sec

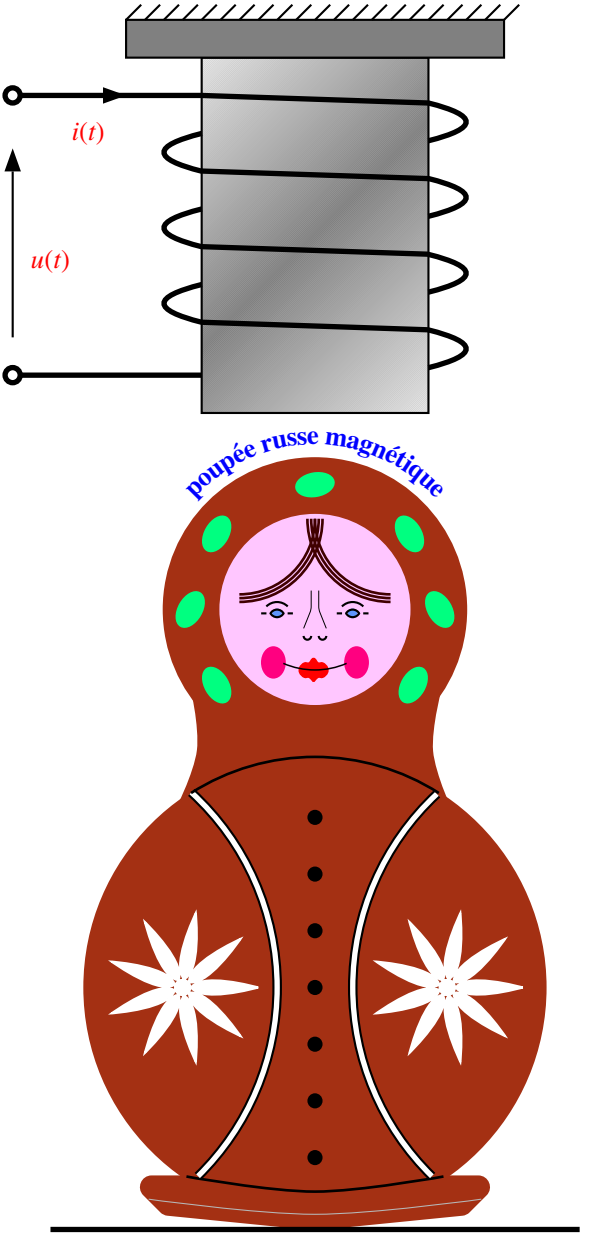

Fig. 11.14 Magnetic levitatios system

Table 11.1 Data of the magnetic levitation system

Variable	value
R	62.7 Ω
L	60 mH
m (object mass)	7.64 g
k_c	$5.9218\ 10^{-4}$
k_p	$4.0477\ 10^{-6}$
C_b	−0.1671
C_p	$-1.9446\ 10^{-8}$
diameter of the permanent magnet	9 mm

Since the overshoot is less or equal to 0.2 %, it results that $\zeta = 0.9$. The time settling time at 2 % is given by:

$$t_s = \frac{4}{\zeta w_n}$$

where w_n is natural pulse.

If we fix the settling time at 0.05 sec we get:

$$w_n = \frac{4}{0.05 \times 0.9} = 88.8889$$

The dominant poles for the design are then given by:

$$s_{1,2} = -\zeta w_n \pm j w_n \sqrt{1-\zeta^2} = -80.000 \pm 38.75j$$

Using this pair of poles we get:

$$\begin{aligned} K_1 &= -175.6 \\ K_2 &= -125.9 \end{aligned}$$

Using this controller the time response starting from given initial conditions is represented in Fig. 11.15. As it can seen from this figure that the overshoot and the settling time are those we would like to have.

For the second case since we don't have access to the load speed we can either compute it from the position or use an observer to estimate the system state. As it was said earlier, the poles that we use for the design of the observer should be faster than those used in the controller design.

Choosing the following poles ($s_{1,2}$ 4 times the real part of those used in the design of the controller):

$$\begin{aligned} s_{1,2} &= -4\zeta w_n \pm j w_n \sqrt{1-\zeta^2} \\ &= -320.0 \pm 38.75j \end{aligned}$$

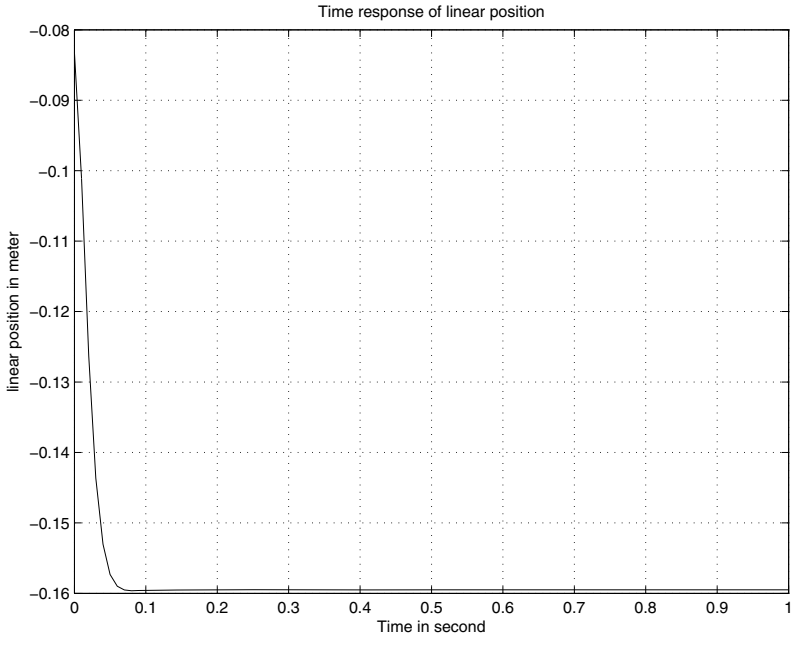

Fig. 11.15 Time response for moving object

we get the following gains for the observer:

$$L_1 = 1.351$$
$$L_2 = 224.5927$$

We can experiment all the other controller as we did for the dc motor kit and the two wheels robot but we prefer let this part as exercise for the reader to practice the tools. Notice we invite him/her to make the design in continuous-time and discrete-time cases and compare the results. The dimension of this system allows that.

A sample for programming the state feedback control of this system is given bellow:

```
 #include <p30fxxxx.h>
#include <stdio.h>
#include <stdlib.h>
#include <adc10.h>
#include <math.h>
#include <uart.h>

//
// Configuration
//
```

```
//  Interne frequency (30 MIPS) instructions/sec
// Number of samples: 7,37*16/4 = 29480000

_FOSC(CSW_FSCM_OFF & FRC_PLL16);
_FWDT(WDT_OFF);
_FBORPOR(PBOR_OFF & MCLR_DIS);
_FGS(CODE_PROT_OFF);
_FICD( ICS_NONE );
__C30_UART=2;

//
//  Variables

    #define Freq_pic 29480000  // PIC Frequency

    #define a11 7.8510061454215840e-001
    #define a12 2.2727760074413661e-004
    #define a21 5.0829248960838420e+001
    #define a22 1.0000643300984893e+000
    #define b11 3.7771272752681438e-005
    #define b12 4.5392069137012870e-004
    #define b21 8.7496385285734044e-003
    #define b22 1.0852723324638587e-001

    #define Ts 2.2727272727272727e-004
    #define u_max 1.1789999999999999e+001
    #define ref_tension 5.0000000000000000e+000
    #define ref_pic 1.0240000000000000e+003
    #define duty_cycle_ref 5.8481764206955049e+001
    #define x_ref 7.8768775539549939e-003
    #define u_ref 2.0000000000000000e+000
    #define y_ref 8.5691877396730676e-001
    #define K0 5.2128678707944724e+004
    #define K1 3.9336557697049994e+002


         double y[2] = {0.0, 0.0};
         double u[2] = {0.0, 0.0};
         double y_tilde[2] = {0.0, 0.0};
         double tension_tilde[2] = {0.0, 0.0};
         double tension = 0.0;
         double duty_cycle_tilde = 0.0;
         double lim_Sup = 0.0;
         double lim_Inf = 0.0;
         double position_tilde[2] = {0.0, 0.0};
```

```
        double vitesse_tilde[2] = {0.0, 0.0};
        double integrale_tilde[2] = {0.0, 0.0};
        double duty_cycle = 0.0;
        double temps_total = 0.0;
        double n = 6553500.0/65536.0;
        int compteur = 0;
        int compteur_freq = 0;
        int uart_flag = 1;
        unsigned long Val_reg = 0;

//
// Functions
//

void init(void){

INTCON1bits.NSTDIS=0; // Activation of the level of interruption
TRISE = 0;  // Configuration of PORTE as output
TRISD = 0;  //Configuration of PORTD as output
PORTEbits.RE8 = 1;
PORTEbits.RE2 = 0;
PORTDbits.RD0 = 1;
ADPCFG= 0xFFFF; // Configuration of the pins of PORTB as digital
I/O
}

void init_ADC (void){

SetChanADC10(ADC_CHX_POS_SAMPLEA_AN3AN4AN5 &
ADC_CHX_NEG_SAMPLEA_NVREF);

ConfigIntADC10(ADC_INT_DISABLE);

OpenADC10(ADC_MODULE_ON & ADC_IDLE_CONTINUE & ADC_FORMAT_INTG
 & ADC_CLK_AUTO & ADC_AUTO_SAMPLING_ON & ADC_SAMPLE_SIMULTANEOUS,
 ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF & ADC_CONVERT_CH_0ABC
 & ADC_SAMPLES_PER_INT_1 & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF,
 ADC_SAMPLE_TIME_1 & ADC_CONV_CLK_SYSTEM  & ADC_CONV_CLK_32Tcy,
 ENABLE_AN4_ANA & ENABLE_AN5_ANA, SCAN_NONE);
}

void init_Timer1 (void){

INTCON1bits.NSTDIS=0; // Activation of mode 16 bits of the Timer1
T1CONbits.TON = 1; // Autorisation du Timer1
```

```
T1CONbits.TGATE = 0; // Dsactivation du mode Timer Gate
T1CONbits.TSIDL=1;  // Synchronisation of Timer1 sur le Idle mode
T1CONbits.TCKPS = 0;   // Choice of the Prescaler 1:1
                          (1=1:8, 2=1:64)
T1CONbits.TCS=0;    // Selection of the interne clock (0=FOSC/4)
IFS0bits.T1IF = 0; // Put to zero the overflow bit for the
                      interrupt of Timer1
IEC0bits.T1IE = 1; // Activation of the interruption of Timer1
PR1 = 6699;  // Sampling frequency at 4400 Hz environ
IPC0bits.T1IP = 5;  // Priority 5 for the interruption of the
                       Timer1

}

/* ROUTINE D'INITIALISATION DE L'UART */

void init_UART (void){

ConfigIntUART2(UART_RX_INT_DIS & UART_RX_INT_PR0
& UART_TX_INT_DIS & UART_TX_INT_PR0);

    // Configuration of the register U2MODE

U2MODEbits.UARTEN = 1; // UART pins controlled by UART
U2MODEbits.USIDL = 0;  // UART communication continue in
                          Idle Mode

U2MODEbits.WAKE = 1;   // Wake up enable in sleep Mode
U2MODEbits.LPBACK = 0; // Loopback mode disabled
U2MODEbits.ABAUD = 0;  // Autobaud process disabled
U2MODEbits.PDSEL = 0;  // 8-bit data,  no parity
U2MODEbits.STSEL  = 0; // 1 stop-bit.

// Configuration du registre U2STA

U2STAbits.UTXISEL = 0;  // Transmission Interrupt Mode
                           Selection bit
U2STAbits.UTXBRK = 0;   // UxTX pin operates normally
U2STAbits.UTXEN = 1;    // Transmit enable
U2STAbits.URXISEL = 1;  // Interrupt occurs when one charater
                           is received
U2STAbits.ADDEN = 0;    // Address detect disabled

U2BRG = 31;             // Value for 57600 bps baudrate
}
```

```
//
// Initialization of the complementary  mode PWM
//
void init_PWM (void){

Val_reg = 1023;  // Frquence de 30000 Hz environ
lim_Sup = (u_max*(2*Val_reg + 1)/(2*Val_reg + 2)) - u_ref;
lim_Inf = -u_max - u_ref;

PTCONbits.PTEN = 1;  // Activation of the time base
PTCONbits.PTSIDL = 1;  // Configuration in Idle Mode
PTCONbits.PTCKPS = 0; // Selection de 4TCY ( Prescale: 00 = 1:1;
                         01= 1:4; 10 = 1:16; 11 = 1:64)
PTCONbits.PTMOD = 0;  // Selection of the free running mode

PTMRbits.PTDIR = 0; // Increment of the time base
PTMRbits.PTMR = Val_reg; // Register value of the Time base

PTPER = Val_reg;  // Value of the signal period

PWMCON1bits.PMOD1 = 0;  // Selection the mode PWM complementary
PWMCON1bits.PEN1H = 1; // Activation of the pins in mode PWM
PWMCON1bits.PEN1L = 1; // Activation of the pins in mode PWM

DTCON1bits.DTAPS = 0; // Time base unit is 1TCY
DTCON1bits.DTA = 0;  // Value of the DT for the unity A
PDC1 = 0; //  zero of the dutycycle

}

void __attribute__((interrupt, auto_psv)) _T1Interrupt (void){

if (IFS0bits.T1IF){

PORTEbits.RE2 = !PORTEbits.RE2;
PDC1 = (2.0 * (Val_reg + 1) * duty_cycle)/100.0; // Calcul de la
          valeur du registre PDC1
y[0] = (ReadADC10(2)*ref_tension)/ref_pic; // Signal of the
                                              sensor in Volts
y_tilde[0] = y[0] - y_ref;


position_tilde[1] = position_tilde[0];
vitesse_tilde[1] = vitesse_tilde[0];
```

```
integrale_tilde[1] = integrale_tilde[0];
y_tilde[1] = y_tilde[0];
tension_tilde[1] = tension_tilde[0];

position_tilde[0] = (a11*position_tilde[1]+a12*vitesse_tilde[1]
                     +b11*tension_tilde[1]+b12*y_tilde[1]);
vitesse_tilde[0] = (-a21*position_tilde[1]+a22*vitesse_tilde[1]
                    +b21*tension_tilde[1]+b22*y_tilde[1]);
tension_tilde[0] = (K0*position_tilde[0]) +
                    (K1*vitesse_tilde[0]);// + N*ref;

if(tension_tilde[0]>lim_Sup){tension_tilde[0] = lim_Sup;}
// Saturation of the tension tilde
if(tension_tilde[0]<lim_Inf){tension_tilde[0] = lim_Inf;}

tension = u_ref + tension_tilde[0];
duty_cycle_tilde = tension_tilde[0]*(50.0/u_max);
duty_cycle = duty_cycle_ref + duty_cycle_tilde; // Computation
                    of the duty cycle in percentage

temps_total += Ts;

compteur_freq = 0;

compteur++;
if(compteur == 10){ // Print data every 1 ms
compteur = 0.0;
uart_flag = 1;
}

IFS0bits.T1IF = 0; // put to zero of the overflow bit
}
}

/* PROGRAMME PRINCIPAL */

int main (void){

    init();
    init_PWM();
    init_ADC();
    init_UART();
    init_Timer1();
    while(1){
        if (uart_flag){
```

```
        printf("%lf %lf %lf %lf %lf %lf %lf\n\r",
        temps_total, tension, u_ref, y[0], y_ref,
        position_tilde[0], x_ref);
        uart_flag = 0;
        }

    }
}
```

11.6 Conclusion

This chapter covered some case studies that were developed in our mechatronics laboratory at École Polytechnique de Montréal. We covered all the steps for the design of a mechatronic system with different kind of details. The emphasis is put on the design of the control algorithm of each presented system.

11.7 Problems

1. Let us consider a dynamical discrete-time system with the following data:

$$
\begin{aligned}
A &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -2 & -3 \end{bmatrix}, \\
B &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \\
C &= \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}, \\
D &= \begin{bmatrix} 0 \end{bmatrix}
\end{aligned}
$$

with the following norm bounded uncertainties:

$$
\begin{aligned}
D_A &= \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}, \\
E_A &= \begin{bmatrix} 0.1 & -0.2 & -0.1 \end{bmatrix}, \\
D_B &= \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}, \\
E_B &= \begin{bmatrix} 0.1 & -0.2 \end{bmatrix}, \\
D_C &= \begin{bmatrix} 0.1 \end{bmatrix}, \\
E_C &= \begin{bmatrix} 0.1 & 0.2 & -0.1 \end{bmatrix},
\end{aligned}
$$

(a) design for the nominal system the following controller:
   i. state feedback controller
   ii. static output feedback controller
   iii. dynamic output feedback controller

(b) design for the uncertain system the following controller:
   i. state feedback controller
   ii. static output feedback controller
   iii. dynamic output feedback controller

(c) if we have an extra term in the state dynamics that add external disturbance: $x(k+1) = [A + \Delta A]\, x(k) + [B + \Delta B]\, u(k) + B_w w(k)$. Design the controllers (state feedback, static output, dynamic output feedback) for the nominal and uncertain system that assure the $\mathscr{H}_\infty$ performance.

(d) design the state-feeback, static output feedback and dynamic feedback controllers that assure the guaranteed cost for all admissible uncertainties.

2. In this problem we invite you to proceed with the design of a small boat that you can control using a joystick to make it moving in a small lac for instance.

   (a) give a schematic design (batteries, motors, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances

3. In this problem we invite you to proceed with the design of a small aircraft that you can control using a joystick to make it flying.

   (a) give a schematic design (batteries, motor, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances

4. In this problem, we ask for the design of a vacuum cleaner. This device should be automatic and avoid obstacle in its environment. It is also important to design a cheap one that can communicate wireless via an emitter and receiver and a joystick
   (a) give a schematic design (electronic circuit, motor, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances
5. In this problem we invite you to proceed with the design of a mechatronic system that control the position of a small ball of ping pong. Pressed air can be used to position the ball.
   (a) give a schematic design (electronic circuit, motor, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances
6. In this problem we would like to design a one leg robot that can move using one wheel while remaining in a vertical position. Provide the design of such mechatronic system.
   (a) give a schematic design (electronic circuit, motor, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances
7. Solar energy is an alternate source of power that can be used. In this problem we ask you to design a solar system that maximize the energy generated by the solar panel.
   (a) give a schematic design (electronic circuit, motor, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances
8. In this problem we ask to design a hoover that can be controlled to seal on water via a emitter and a receiver using a joystick.
   (a) give a schematic design (electronic circuit, motor, etc.)
   (b) establish the mathematical model
   (c) fix the specifications you would like to have and design the appropriate controller that can give such performances

AppendixA
C Language Tutorial

The aim of this appendix is to review the C language to refresh the memory of the reader and help him to start writing his programs without reading huge books on the subject. Our intention is not replace these books and readers that are not familiar with the subject are strongly encouraged to consult book of Kernighan and Ritchie[1].

Firstly we invite the reader to download a C compiler and a text editor. To experiment the different programs in C we will give, the reader must type the programs, save them and then compile them. For more details on how to do this we refer the reader to the manual of the used C compiler.

To start our tutorial, let us consider our first simple program. This program is intended to write "Welcome to mechatronics course". The listing of this program is given by the following lines:

```
#include < stdio.h>

void main()
      {
         printf("\nWelcome to mechatronics course\n");
      }
```

To see the output of this simple program we need to have an editor and a C compiler.

[1] Brian W. C. Kernighan and Dennis M. Ritchie, The C Programming Language– ANSI C, Prentice Hall, 1988

Each C program is composed by variables and functions and must have a "main" function. Except some reserved words all the variables and the functions must be declared before they can be used. The variables can take one of the following types:

- *integer*
- *real*
- *character*
- etc.

The functions specify the tasks the program has to perform and for which it is designed for. The "main" function establishes the overall logic of the code.

Let us examine the previous program. The first statement

```
#include < stdio.h>
```

includes a specification of the C I/O library. The ".h" files are by convention "header files" which contain definitions of variables and functions necessary for the functioning of a program, whether it can be a user-written section of code, or as part of the standard C libaries.

The directive

```
#include
```

tells the C compiler to insert the contents of the specified file at that point in the code.

The notation

```
<...>
```

instructs the C compiler to look for the file in certain "standard" system directories.

The *void* preceeding "main" indicates that main is of "void" type–that is, it has no type associated with it, which means in another sense that it cannot return a result during the execution.

The ";" denotes the end of a statement. Blocks of statements are put in braces $\{\cdots\}$, as in the definition of functions. All C statements are defined in free format, i.e., with no specified layout or column assignment. White space (tabs or spaces) is never significant, except inside quotes as part of a character string.

The statement *printf* line prints the message "Welcome to mechatronics course" on "stdout" (the output stream corresponding to the X-terminal window in which you run the code), while the statement

```
\n
```

prints a "new line" character, which brings the cursor onto the next line. By construction, the function printf never inserts this character on its own and this is let to the programmer.

The standard C language has some reserved keywords that can not be used by the programmer for naming variables or other purpose and he must used them as it is suggested otherwise mistakes will appear during the compilation. These keywords are listed in the Table A.1.

Table A.1 List of C language keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table A.2 Number representations

Base	Representation	Chiffres permis	Example
Decimal (10)		0123456789	5
Binary (2)	0b...	01	0b10101010
Octal (8)	0...	01234567	05
Hexadecimal (16)	0x...	0123456789ABCDEF	0x5A

Table A.3 Integer representations

Type	Size (bits)	Min	Max
char, signed char	8	-128	+127
unsigned char	8	0	255
short, signed short	16	-32768	+32767
unsigned short	16	0	65535
int, signed int	16	-32768	+32767
unsigned int	16	0	65535
long, signed long	32	-2^{31}	$+2^{31}-1$
unsigned long	32	0	$2^{32}-1$
long long, signed long long	64	-2^{63}	$+2^{63}-1$
unsigned long long	64	0	$2^{64}-1$

Table A.4 Decimal representations

Type	Taille (bits)	Emin	Emax	Nmin	Nmax
float	32	-128	+127	2^{-126}	2^{128}
double	32	-128	+127	2^{-126}	2^{128}
long double	64	-1022	+1023	2^{-1022}	2^{1024}

Let us look to the constants and variables. The constants have to be defined before it can be used. The structure we used to define these constant is:

```
#define name value
```

The word name is the one we give to the constant and the value is the value that this constant takes. The following example gives some constants:

```
#define acceleration 9.81
#define pi     3.14
#define mot "welcome to mechatronics course''
#define False 0
```

In general when we manipulate data, we use different number representation. Table A.2 gives the bases we are usually using when programming microcontroller.

For the variables, we must declare them before also. The following syntax is used:

```
type <name>;
```

where the type is one of the following:

- int (for integer variable)
- short (for short integer)
- long (for long integer)
- float (for single precision real (floating point) variable)
- double (for double precision real (floating point) variable)
- char (for character variable (single byte))

Tables A.3 and A.4 gives the values taken in each type. It is important to mention that the compiler checks for consistency in the types of all variables used in the program to prevent mistakes. The following example shows some variables:

```
int i,j,k;
float x,y;
unsigned char var1;
unsigned char var2[10] = "welcome";
```

Once these variables and constants are defined what we can do with? The answer is that we can do many things like:

- arithmetic operations (act on one or multi variables)
- logic operations (act on one or multi variables)
- relational operations (act on one or multi variables)
- etc.

Tables A.5 and A.6 give an idea on the different operations we can do either on the constants or the variables.

Table A.5 Arithmetic operations

Symbol	Meaning	example
+	addition	$u[k] = u_1[k] + u_2[k];$
-	substrate	$u[k] = u_1[k] - u_2[k];$
/	division	$u[k] = u_1[k]/u_2[k];$ ($u_2[k]$ must be different from zero)
*	multiplication	$u[k] = K * x[k];$
%	modulo	$u[k] = u_1[k]\%u_2[k];$

Table A.6 Logic operations

Symbol	Meaning	example
&	ET	$u[k] = u_1[k]\&u_2[k];$
|	OR	$u[k] = u_1[k] \mid u_2[k];$
$\wedge$	XOR	$u[k] = u_1[k] \wedge u_2[k];$
$\sim$	inversion	$u[k] = \sim x[k];$
$\ll$	shift left	$u[k] = u_1[k] \ll u_2[k];$
$\ll$	shift right	$u[k] = u_1[k] \gg u_2[k];$

More often we need to print data wither on the screen or an LCD. For this purpose the *print* function can be used. This function can be instructed to print integers, floats and strings properly. The general syntax is

```
printf( "format", variables );
```

where ”format” specifies the conversion specification and variables is a list of quantities to be printed.

The useful formats are:

```
%.nd   integer (optional n = number of columns; if 0, pad with zeroes)
%m.nf  float or double (optional m = number of columns,
n = number of decimal places)
%ns    string (optional n = number of columns)
```

```
%c        character
\n \t     to introduce new line or tab
\g    ring the bell (‘‘beep’’) on the terminal
```

In some programs, the concepts of loops are preferable to perform calculations that require repetitive actions on a stream of data or a region of memory. There are several ways to loop in the standard C and the following ways the most common used loops:

```
 // while loop
while (expression)
{
        block of statements to be executed
}

 // for loop
for (expression_1; expression_2; expression_3)
{
       block of statements to be executed
}
```

In some cases we need to take an action based on the realization of some condition or dependent on the value of a given variable. In this case the word *if* or the word *if else* and the switch can be used. The following structures are used:

```
if (conditional_1)
{
       block of statements to be executed when conditional_1 is true
}
else if (conditional_2)
{
       block of statements to be executed when conditional_2 is true
}
else
{
       block of statements to be executed otherwise
}
```

Tables A.7 and A.8 show the most used conditional operators that we may use in the expressions.

Table A.7 Logic operations

Symbol	Meaning	example
&&	and	$x\&\&y$
||	or	$x\|\|y$
!	not	$x!y$

Table A.8 Logic operations

Symbol	Meaning	example
<	smaller than	$x < y$
<=	smaller than or equal to	$x <= y$
==	equal to	$x == y$
! =	not equal to	$x! = y$
>=	greater than or equal to	$x >= y$
>	greater than	$x > y$

```
switch (expression)
{
     case const_expression_1:
{
     block of statements to be executed
           break;
}
case const_expression_2:
{
     block of statements to be executed
           break;
}
default:
{
      block of statements
}
}
```

The C language allows the programmer to access directly the memory locations using the pointers. To show how this works, let us consider a variable Xposition defined as follows:

```
float Xposition;
Xposition = 1.5;
```

If for example we want to get the address of the variable Xposition, the following can be used:

```
float* pXposition;
float Xposition;

Xposition = 1.5;
pXposition = &Xposition;
```

In this code we define a pointer to objects of type float, Xposition and set it equal to the address of the variable Xposition.

To get the content of the memory location referenced by a pointer is obtained using the "*" operator (this is called dereferencing the pointer). Thus, *pXposition refers to the value of Xposition.

Arrays of any type play an important role in C. The syntax of the declaration of the arrays is simple and the following gives that:

```
type varname[dim];
```

where dim is the dimension we want to give to the array varname.

In standard C, the array begins with the position 0 and all its elements occupy adjacent locations in memory. Thus, if *matrix* is an array, $*matrix$ is the same thing as *matrix*[0], while $*(matrix + 1)$ is the same thing as *matrix*[1], and so on.