C libraries for Digital Controllers implementation (α version 1.0)

Ivan Furlan

July 19, 2023

Contents

Chapter 1

Introduction

This document introduces the libraries developed for facilitating the implementation in c of control system algorithms. If more details about the created data types and functions I/O are needed, please consult the **Doxygen** doc generated in the `doc` folder in the library directory.

1.1 Library descritpions

The main library is called `control_system.h` which contains some of the most popular control system architectures. Other libraries have been created in this work, and are:

- The `lti_system.h` which contains the basic blocks (transfer-function and state-space system) that are used in the the library `control_system.h`.
- The `lin_algebra.h` which defines some basic linear algebra operators which are used in the previous two libraries.

Th2 last 2 libraries are instrumental to the main one, but can of course be used by themselfs, for implementing other kind of controllers or any kind of LTI system not included in the main library, such as: LP filters, mathematical models of a system, etc.

In the next chapters the usage of the libraries will be introduced by means of examples

1.2 Data types

The type of variable used on this library package is `t_system_var`. The typedef which defines `t_system_var` is located in the file `type_defs_control_system_libs.h`. The default type is `float`, but also any other kind of type can be adopted.

1.3 Unit tests

The libraries have been tested by means of units tests in simulink. The unit tests can be found in the library directory, under the `unit_tests` directory. The libraries, subject of this document, are provided 'as is'.

Chapter 2

Library control_system.h

This chapter introduces the elements and the usage of the main library `control_system.h`.

2.1 State-feedback controller

This section introduces the commands for the implementation of a state feedback MIMO controller

$$u_k = r_k - K \cdot x_k$$

where K is the state feedback matrix

$$K := \begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & \dots & k_{1,n} \\ \vdots & & & & \\ k_{n_u,1} & k_{n_u,2} & k_{n_u,3} & \dots & k_{n_u,n} \end{bmatrix}$$

and

- r_k are the reference signals (a vector of n_u signals),
- x_k are the states of the system (a vector of n signals),
- u_k are the actuation signals (a vector of n_u signals).

The architecture of the controller is shown in figure (2.1).

As a matter of example, the pseudocode for implementing a state-space feedback controller with the folloiwng state-feedback matrix (4th order system with 2 inputs)

$$K := \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \end{bmatrix}$$

follows.

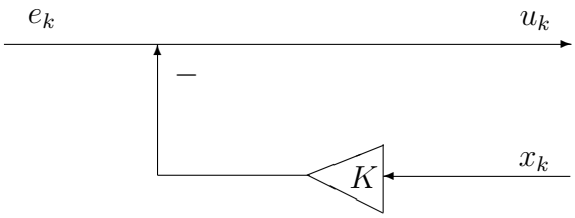

Figure 2.1: State-feedback controller

```
#include "control_systems.h"
#define ORDER 4 // order of the system (i.e. number of states)
#define N_U 2 // nummber of system inputs

t_state_feedback_controller sfb;

init_func() {
    t_lti_initialization_errors init_err;
    // definiton of teh state-feeback matrix
    t_system_var K[N_U][ORDER] = {{k_11,k_12,k_13,k_14},
                                  {k_21,k_22,k_23,k_24}};

    // creation of the object
    init_err = state_feedback_controller_create(ORDER,
                                                N_U,
                                                K,
                                                &sfb);
    if(init_err != init_successful) {
            printf("Init error\n"), while(1);
    }
}

loop()
{
    wait_sync();
    ref_k = read_ref();
    x_k = read_system_states();
    // calculation of the controller output (actuation)
    state_feedback_controller_output_calc(ref_k,
                                          x_k,
                                          u_k,
                                          &sfb);
    write_out() = u_k;
}
```

2.2 State observer

This section introduces the commands for the implementation of a state observer

$$\hat{v}_{k+1} = A_{\text{obs}} \cdot \hat{v}_k + B_{\text{obs}} \cdot [y_k, u_k]$$
$$\hat{y}_k = B_{\text{obs}} \hat{v}_k + D_{\text{obs}} \cdot [y_k, u_k]$$

where $A_{\text{obs}}, B_{\text{obs}}, C_{\text{obs}}$ and D_{obs} are the observer state-space matrices, and

- y_k are the output of the observed system (a vector of n_y signals),
- u_k are the actuation signals (a vector of n_u signals),
- v_k are the states of the observer (a vector of $n_v \leq n$ signals),
- $\hat{x}_k$ are the state estimations (a vector of n signals)

As a matter of example, the pseudocode for implementing a state observer with the folloiwings state matrices (observer for observing a 2th order system)

$$A_{\text{obs}} := \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B_{\text{obs}} := \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{13} \end{bmatrix} \quad C_{\text{obs}} := \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \quad D_{\text{obs}} := \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \end{bmatrix}$$

follows.

```
#include "control_systems.h"

#define OREDER 2 // order of teh system to be observed
#define N_U 2 //number of system inputs
#define N_Y 2 //number of system outputs (available measurements)

t_state_feedback_controller_wfullobs sf_contr_wfobs;

init_func() {

    t_lti_initialization_errors init_err;

    // definition of the observer matrices
    t_system_var Aobs[order][order] = {{a_{11},a_{12}},
                                       {a_{21},a_{22}};
    t_system_var Bobs[order][n_y+n_u] = {{b_{11},b_{12},b{13}}},
                                         {b_{21},b_{22},b{23}}};
    t_system_var Cobs[order][order] = {{c_{11},c_{12}},
                                       {c_{21},c_{22}};
    t_system_var Dobs[order][n_y+n_u] {{d_{11},d_{12},d{13}}},
```

```
                                        {d_{21},d_{22},d{23}}};

    // creation of the object
    init_test = state_observer_create(order, n_u, n_y,
                                      (t_system_var*)Aobs,
                                      (t_system_var*)Bobs,
                                      (t_system_var*)Cobs,
                                      (t_system_var*)Dobs,
                                      &test_observer);

    if(init_err != init_successful) {
        printf("Init error\n"), while(1);
    }

}

loop()
{
    wait_sync();
    u_k = read_system_in();
    y_k = read_system_outputs();

    // calculation of the obeserver state estimation
    state_space_observer_estimation(hat_x_k,
                                      y_k,
                                      u_k,
                                      &test_observer);

    write_states_estimations() = hat_x_k;
}
```

2.3 State-feedback controller with full-state observer

In some situations, may be convenient to have the state-feedback controller and the observer in an monolithic code block. This section introduces this controller variant, i.e. a state feedback MIMO controller

$$u_k = r_k - K \cdot \hat{x}_k$$

where K is the state feedback matrix

$$K := \begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & \dots & k_{1,n} \\ \vdots & & & & \\ k_{n_u,1} & k_{n_u,2} & k_{n_u,3} & \dots & k_{n_u,n} \end{bmatrix}$$

and

- r_k are the reference signals (a vector of n_u signals),
- $\hat{x}_k$ are the estimations of the states of the system (a vector of n signals),
- u_k are the actuation signals (a vector of n_u signals),

where the states are calculated by the full-state observer

$$\begin{aligned} \hat{x}_{k+1} &= A_{\text{obs}} \cdot \hat{x}_k + B_{\text{obs}} \cdot [y_k, u_k] \\ \hat{x}_k &= I\hat{x}_k + 0 \cdot [y_k, u_k] \end{aligned}$$

where $A_{\text{obs}}, B_{\text{obs}}$ are the observer state-space matrices, and

- y_k are the output of the observed system (a vector of n_y signals),
- $\hat{x}_k$ are the state estimations (a vector of n signals).

The architecture of the controller is shown in figure (2.2).

As a matter of example, the pseudocode for implementing a state-space feedback controller with the folloiwng state-feedback matrix (4th order system with 2 inputs)

$$K := \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \end{bmatrix}$$

and the following observer

$$A_{\text{obs}} := \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B_{\text{obs}} := \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{13} \end{bmatrix}$$

follows.

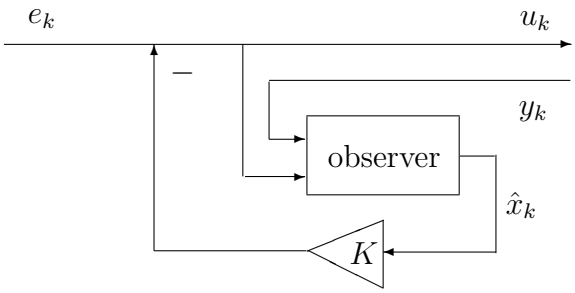

Figure 2.2: State-feedback controller with full-state observer

```
#include "control_systems.h"

#define OREDER 2 // order of teh system to be observed
#define N_U 2 //number of system inputs
#define N_Y 2 //number of system outputs (available measurements)

t_state_observer test_observer;

init_func() {

    t_lti_initialization_errors init_err;

    // definiton of teh state-feeback matrix
    t_system_var K[N_U][ORDER] = {{k_11,k_12,k_13,k_14},
                                  {k_21,k_22,k_23,k_24}};

    // definition of the observer matrices
    t_system_var Aobs[order][order] = {{a_{11},a_{12}},
                                       {a_{21},a_{22}};
    t_system_var Bobs[order][n_y+n_u] = {{b_{11},b_{12},b{13}}},
                                         {b_{21},b_{22},b{23}}};
    t_system_var Cobs[order][order] = {{c_{11},c_{12}},
                                       {c_{21},c_{22}};
    t_system_var Dobs[order][n_y+n_u] {{d_{11},d_{12},d{13}}},
                                       {d_{21},d_{22},d{23}}};

    // creation of the object
    init_test = state_feedback_wfullobs_controller_create(order,
                                                          t_system_var*)K_test,
                                                          n_u,
                                                          n_y,
                                                         (t_system_var*)Aobs,
```

```
                                                        (t_system_var*)Bobs,
                                                         &sf_contr_wfobs);


    if(init_err != init_successful) {
        printf("Init error\n"), while(1);
    }

}

loop()
{
    wait_sync();
    ref_k = read_ref();
    y_k = read_system_outputs();

    // calculation of the obeserver state estimation
    state_feedback_controller_wfullobs_output_calc(ref_k,
                                                   y_k,
                                                   u_k,
                                                   &sf_contr_wfobs);

    write_states_estimations() = u_k;
}
```

2.4 State-feedback controller with internal model (under dev.)

(...Under development)

2.5 State-feedback controller with internal model and full-state observer (under dev.)

(...Under development)

2.6 Controller in state-space compact form (under dev.)

(...Under development)

2.7 PID controller (under dev.)

(...Under development)

2.8 Polynomial controller

This section introduces the commands for implementing a polynomial controller

$$C(z) := \frac{N(z)}{D(z)} = \frac{a_m \cdot z^m + a_{m-1} \cdot z^{m-1} \ldots + a_0}{z^n + b_{n-1} \cdot z^{n-1} \ldots + b_0},$$

where $m \leq n$. The controller input signal is the error signal e_k, the controller output, u_k (actuation signal), if needed, may be saturaed in the set $[l_{\text{low}}, l_{\text{up}}]$ by means of the anti-wind up saturation feedback architecture. The architecture of the controller, with anti-windup, is shown in figure (2.3). If the anti-windup is not present, the

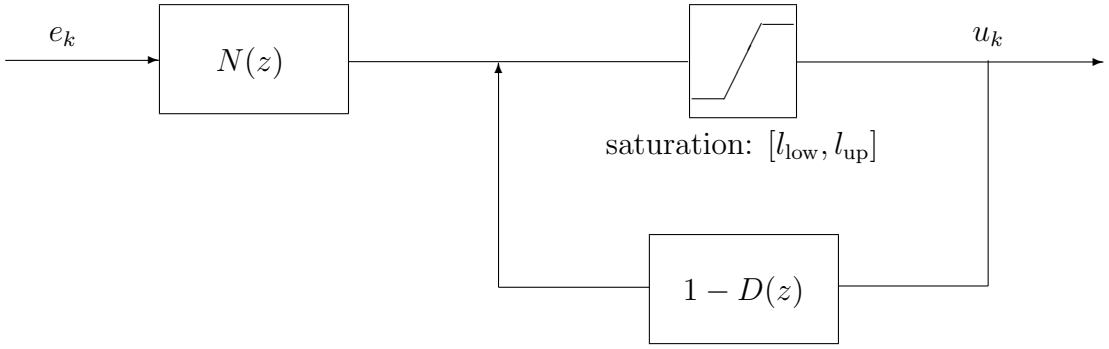

Figure 2.3: Polynomial controller in antiwid-up form

controller becomes a standard transfer-function.

As a matter of example, the pseudocode for implementing the following second order polynomial controller with antiwid-up

$$C(z) := \frac{N(z)}{D(z)} = \frac{a_2 \cdot z^2 + a_1 \cdot z + a_0}{z^2 + b_1 \cdot z + b_0},$$

follows.

```
#include "control_systems.h"

#define POLY_CTR_ORDER 2 // order of the controller
#define L_UP (10.0) // AW limit up
#define L_LOW (-10.0) // AW limit low

t_polynomial_controller_antiwindup_form poly_ctr;

init_func() {

    t_lti_initialization_errors init_err;

    // numerator and denominator definition
```

```
    t_system_var num_poly_ctr[POLY_CTR_ORDER+1] = {a_2, a_1, a_0};
    t_system_var den_poly_ctr[POLY_CTR_ORDER+1] = {1, b_1, b_0};

    // anti-windup limits definition
    bool anti_windup_form = true;

    // creation of the object
    init_err = polynomial_controller_object_create(POLY_CTR_ORDER,
                                                   anti_windup_form,
                                                   num_poly_ctr,
                                                   den_poly_ctr,
                                                   &poly_ctr);

    if(init_err != init_successful) {
        printf("Init error\n"), while(1);
    }
}

loop()
{
    wait_sync();
    in_k = read_in();

    // calculation of the controller output (actuation)
    // if no antiwind-up, parameters L_UP and L_LOW have no effects
    out_k = polynomial_controller_output_calc(in_k,
                                              L_UP,
                                              L_LOW,
                                              &poly_ctr);

    write_out() = out_k;
}
```

Chapter 3

Library lti_system.h

This chapter introduces the library `lti_system.h`. As will be shown, this library, can be stand-alone used for implementing any kind of LTI digital dynamic systems, such as: filters and so on.

3.1 LTI state-space

This section introduces the commands for the implementation of a state-space equation

$$\begin{aligned} x_{k+1} &= A \cdot x_k + B \cdot u_k \\ y_k &= C \cdot x_k + D \cdot u_k \end{aligned}$$

where A, B, C and D are the state-space matrices, and

- u_k are the inputs of the state-space equation (a vector of n_u signals),
- x_k are the states of the state-space equation (a vector of n signals),
- y_k are the state outputs estimationsd of the state-space equation (a vector of n_y signals).

As a matter of example, the pseudocode for implementing a state observer with the folloiwng state matrices (observer for observing a 2th order system with 3 inputs and 2 outputs)

$$A_{\text{obs}} := \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B_{\text{obs}} := \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{13} \end{bmatrix} \quad C_{\text{obs}} := \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \quad D_{\text{obs}} := \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \end{bmatrix}$$

follows.

```
#include "lti_systems.h"

#define ORDER 2
```

```
#define N_U 3
#define N_Y 2

t_system_state_space G_ss;

init_func() {
    t_lti_initialization_errors init_err;
    // definition of the state-space matrices
    t_system_var A[order][order] = {{a_{11},a_{12}},
                                    {a_{21},a_{22}};
    t_system_var B[order][n_y+n_u] = {{b_{11},b_{12},b{13}}},
                                      {b_{21},b_{22},b{23}}};
    t_system_var C[order][order] = {{c_{11},c_{12}},
                                    {c_{21},c_{22}};
    t_system_var D[order][n_y+n_u] {{d_{11},d_{12},d{13}}},
                                    {d_{21},d_{22},d{23}}};

    // creation of the object
    init_err = state_space_system_create(ORDER,
                                         N_U,
                                         N_Y,
                                        (t_system_var *)A,
                                        (t_system_var *)B,
                                        (t_system_var *)C,
                                        (t_system_var *)D,
                                        &G_ss);
    if(init_err != init_successful) {
            printf("Init error\n"), while(1);
    }
}

loop()
{
    wait_sync();
    in_k = read_in();

    // calculation of the state-space output
    state_space_linear_filter(&in_k,
                              &out_k,
                              &G_ss);

    write_out() = out_k;
}
```

3.2 LTI transfer-function

This section introduces the commands for the implementation of a transfer-function

$$G(z) := \frac{N(z)}{D(z)} = \frac{a_m \cdot z^m + a_{m-1} \cdot z^{m-1} + \ldots + a_0}{z^n + b_{n-1} \cdot z^{n-1} + \ldots + b_0},$$

where $m \leq n$.

As a matter of example, the pseudocode for implelentig the following second order polynomial controller

$$C(z) := \frac{N(z)}{D(z)} = \frac{a_2 \cdot z^2 + a_1 \cdot z + a_0}{z^2 + b_1 \cdot z + b_0},$$

follows.

```
#include "lti_systems.h"

#define ORDER 2 // order of teh system

t_system_transfer_funtion G_tf;

init_func() {

    t_lti_initialization_errors init_err;

    // numerator and denominator definition
    t_system_var num_G_tf[ORDER+1] = {a_2, a_1, a_0};
    t_system_var den_G_tf[ORDER+1] = {1, b_1, b_0};

    // creation of the object
    init_err = transfer_function_system_create(G_tf_order,
                                               num_G_tf,
                                               den_G_tf,
                                               &G_tf);

    if(init_err != init_successful) {
        printf("Init error\n"), while(1);
    }
}

loop()
{
    wait_sync();
    in_k = read_in();
```

```
    // calculation of the transfer-fucntion output
    out_k = transfer_function_linear_filter(in_k,
                                            &Gtf);

    write_out() = out_k;
}
```

Chapter 4

Library lin_algebra.h

This chapter introduces the library `lin_algebra.h`. As for the `lti_system.h`, this library, can be tand-alone used, for implementing some matrix/vectors operations.

4.1 Vector/matrix operators

Three function have been created

- `mat_vect_mult` for performing the matrix vetor multiplication,
- `mat_vect_sum` for performing the sum of two vectors,
- `mat_vect_sub` for performing the subtraction of two vectors,
- `vect_el_sign_change` for changing the sign of the elemnts of a vector.

Follows a pseudocde which details the usage of those operators.

```
#include "lin_algebra.h"

#define N_ROW 3
#define N_COLUMN 3

main() {

    t_system_var vect_1[N_COLUMN] = {1,3,4};
    t_system_var vect_2[N_COLUMN] = {5,6,1};
    t_system_var mat[N_ROW][N_COLUMN] = {{1,6,1},{7,6,3},{4,6,1}};

    t_system_var vect_result_1[N_COLUMN];
    t_system_var vect_result_2[N_COLUMN];
    t_system_var vect_result_3[N_COLUMN];
```

```
    // Matrix vector multiplication
    mat_vect_mult(vect_result, mat, vect_1, N_ROW, N_COLUMN);

    // Sum of two vectors
    vects_sum(vect_result_2, vect_1, vect_2, N_ROW);

    // Subtraction of two vectors
    vects_sub(vect_result_2, vect_1, vect_2, N_ROW);

    // Change of the sign of a vector
    vect_el_sign_change(vect_result_2, vect_1, N_ROW);

}
```

4.2 Identity and zero matrix creation

Two function have been created

- `eye_matrix_filling` for filling an existing matrix of zeros,
- `zero_matrix_filling` for converting an existing square matrix in an identity matrix.

Follows a pseudocde which details the usage of those operators.

```
#include "lin_algebra.h"

#define N_ROW 3
#define N_COLUMN 3


main() {

    t_system_var mat[N_ROW][N_COLUMN];

    // Conversion of an existing square matrix in an identity matrix
    eye_matrix_filling(mat, N_COLUMN);

    // Filling an existig matrix of zero elements
    zero_matrix_filling(mat, N_ROW, N_COLUMN);
}
```