Rapid Control Prototyping with Python and gEDA tools

Roberto Bucher
University of Applied Sciences of Southern Switzerland - SUPSI
Galleria 2, CH-6928 Manno
roberto.bucher@supsi.ch

Abstract

Rapid Control Prototyping (RCP) methods play an important role in control system design. The most used tools are based on Matlab for the design of the control algorithm, Simulink for the simulation of the controlled system and Real Time Workshop for generating the code for the RT target. Unfortunately such tools are quite expensive and most little companies can’t face the commercial licenses related to these products.

In this paper a free open source alternative is presented: Matlab is substituted by Python, the gEDA tools are used as replacement for Simulink and the control code is automatically generated for a Linux machine running a preempt_rt patched kernel. In particular, the methods used to translate the gEDA schematic into C code are presented.

The new environment has been tested and validated on different plants at the SUPSI laboratory.

1 Introduction

Rapid Control Prototyping (RCP) methods are more and more used in industries and education. Such methods require a Computer Aided Control System Design (CACSD) environment: in particular, universities invested a lot of resources in teaching methods based on Mathworks products: Matlab for the design and the calculation of the control algorithms, Simulink for the hybrid simulation of the designed controller, and Real-time Workshop as tool to automatically generate the code for the control target. In Switzerland, for example, control teachers officially chose Mathworks products [1] without proposing other alternatives. Matlab/Simulink/RTW are de facto the tools for control design. Despite of its popularity in universities, Mathworks products are not widely used in industries, in particular by little companies. In addition, universities are not allowed to use teaching licenses in research projects with industries: they have to purchase expensive commercial licenses, too.

This paper proposes an alternative completely based on open source tools.

Section 2 and 3 introduce Python as control system design tool. In section 4 we propose an open source tool suite to design control block diagrams for hybrid simulation and code generation. Sections 5 explains how the RT code is generated. This paper ends with an example and some conclusions.

2 Python

Python [2] is a general purpose, high level, object oriented programming language, with a wide number of developers. It is distributed under the Python Software Foundation License, similar to BSD.

Python offers a set of functions comparable with Matlab and other environments. Functions are usually integrated in modules (for example “numpy”, scipy”, “sympy”, “matplotlib”). In addition, Python can be easily linked with gtk or qt libraries, in order to create programs with GUI.

3 Control design with Python

In 2009 Richard Murray at Caltech started the develop of the python-control package [3], in order to
implement a set of functions comparable with the control system toolbox of Matlab.

The toolbox is still in development and at present it is possible to:

- implement systems in state space form and as transfer function, in both continuous and discrete time
- work with SISO and MIMO systems
- design controllers using graphical methods like Bode, Nyquist, root locus etc.
- develop state feedback controllers using methods like poles placement, LQR, DLQR etc.
- simulate dynamic systems (controlled or not) with step response, initial value analysis etc.

One of the problems related with the python-control toolbox is represented by the “educational” approach of the proposed functions. Sometimes the results of the functions are mathematically correct, but not usable for developing real controllers. For example, the implemented function for pole placement by MIMO systems returns one of the infinity solutions which is often not well conditioned and consequently not usable for controlling real plants! In order to correct these problems, some functions have been reimplemented by the author into a new package [4].

4 Hybrid simulation

In control design, it is important to simulate the discrete time controller with the continuous plant. At present no tool like Simulink is available in python.

5 Code generation

5.1 Basics

Every element in a block diagram is defined with two functions:

The interface function which describes how the block must be drawn in the block diagram

The Implementation function which describes the operation sequences of the block.

In addition to this two functions we need a description of the I/O connections of the blocks in the schematic and a thread which is responsible to call all the phases of the RT execution in the right way.

5.2 Interface functions

Every block is defined as “symbol” in the gEDA schematic and stored into a gEDA library. This library is loaded at the beginning of the gEDA schematic application. The “symbol” file contains the instructions required to draw the symbol and the default parameters of the block. Using “drag and drop”, symbols can be easily inserted in the block diagram and connected to other elements.

Each block must be renamed with a unique name, and parameters can be modified directly in the gschem application or using the python program “pyAttrib”.

FIGURE 1: Some blocks for control design

In order to perform hybrid simulation, different open source tools have been tested. After this analysis we choose the gEda tool suite [5]. The gEda suite contains a set of applications used for electronic design and it is released under the GPL. The gEDA tool suite offers the possibility to easy create and integrate new blocks. In particular we had the possibility to create a set of new blocks specific for the control design. Some of these blocks are shown in figure 1.

From the gEDA schematic it is possible to generate code for the hybrid simulation. Code for the RT controller can be generated in the same way: user should only substitute the mathematic model of the plant with the blocks of the sensors and the actuators of the real system.

Different template makefiles allow to generate code for simulation or RT executable.
5.3 The implementation functions

In a block diagram, each system can be described with the functions (1) for continuous time systems and (2) for discrete time systems.

\[
y = g(x, u, t) \tag{1}
\]
\[
\dot{x} = f(x, u, t)
\]
\[
y_k = g(x_k, u_k, k) \tag{2}
\]
\[
x_{k+1} = f(x_k, u_k, k)
\]

The \(g(\ldots)\) function represents the static part of the block. This function can be used to read inputs, read sensors, write actuators or update the outputs of the block. All the blocks have this function defined, usually referred as “output update”.

The second function (\(f(\ldots)\)) is only required when the block has internal states, and it is only used by dynamic systems. In addition, every block implements two other functions, one for initializing the system and one to cleanly terminate it.

All these functions are programmed in C-files, compiled and archived into a library.

5.4 I/O connections

The gEDA tool suite offers an application called “gnetlist”, which is used to generate net lists for different targets. In particular we generate a net list for the spice simulator. The resulting net list describes in particular the nodes connected at the input and output of each block.

This net list can be translated into a list of objects of a new python class “RCPblk”. This class contains the following fields:

- **fcn** is the name of the C-Function used to handle this block
- **pin** is an array containing the id of the input nodes
- **pout** is an array with the id of the output nodes
- **nx** is the number of internal states (continuous or discrete)
- **uy** is a flag which indicates a direct dependency between input and output signals
- **realPar** is an array containing the real parameters of the block
- **intPar** is an array containing the integer parameters of the block
- **str** is character array

As example we can draw the diagram of figure 2

**FIGURE 2: Simple block diagram**

This block diagram is translated to the following net list

* Spice netlister for gnetlist
  DISPLAY 1 3 nin:2|nout:0|printBlk
  PLANT 2 3 nin:1out:1|cssBlk|System: tf(1,[1,1,1])|Initial conditions: 0
  SUM 1 3 2 nin:2|nout:1|sumBlk|Gains: [1,-1]
  REF 1 nin:0|nout:1|stepBlk|Step Time: 1|Step Value: 1
  .END

In this example the block “PLANT” have one input connected to the node 2 and one output connected to the node 3, it is a continuous transfer function (\(cssBlk, 1/(s + 1)\)) with zero initial conditions. The “SUM” block has 2 inputs connected to node 1 and 3, one output connected to node 2 and performs a subtraction of the input signals (Gains: [1,-1]).

This net list is parsed and translated into four objects of type “RCPblk:

- **DISPLAY** = printBlk([1, 3])
- **PLANT** = cssBlk([2],[3], tf([1,1,1]), 0)
- **SUM** = sumBlk([1, 3],[2], [1,-1])
- **REF** = stepBlk([1], 1, 1)

**blks** = [DISPLAY, PLANT, SUM, REF]

5.5 Translating the block list into C-code

Before the generation of the C-code we have to get the correct execution sequence of the blocks in the diagram. A simple method to perform this task is to generate a square matrix of dependency filled with 0 and 1 values where rows and column are related with the blocks. The matrix of the diagram of figure 2 is

\[
\begin{bmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]
First line and first column refer to block “DISPLAY”, the second line and column to block “PLANT” etc.

The second and fourth line only contain 0 values. This means that the output of “PLANT” and “REF” can be calculated without knowing the outputs of the previous blocks. We can now begin to fill a new ordered block list by inserting these two blocks.

```
ordered_list = [PLANT, REF]
```

The output of these two blocks is now known and we can put 0 values in the columns corresponding to these blocks. The dependency matrix becomes now

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

The blocks “DISPLAY” and “SUM” have only 0 values in its rows now, and they can be appended to the ordered list.

In [6]: ordered_list = detBlkSeq(3, blks)
In [7]: for n in ordered_list:
... print n
...:
Function : css
Input ports : [2]
Outputs ports : [3]
Nr. of states : [2 0]
Relation u->y : 0
Real parameters : [[ 0. 0. -1. 1. -1. -1. 0. 0. -1. 0. 0. 0.]]
Integer parameters : [2 1 1 1 5 7 9 10]
String Parameter :
Function : step
Input ports : []
Outputs ports : [1]
Nr. of states : [0 0]
Relation u->y : 0
Real parameters : [1 1]
Integer parameters : []
String Parameter :
Function : print
Input ports : [1 3]
Outputs ports : []
Nr. of states : [0 0]
Relation u->y : 1
Real parameters : []
Integer parameters : []
String Parameter :
Function : sum
Input ports : [1 3]
Outputs ports : [2]
Nr. of states : [0 0]
Relation u->y : 1
Real parameters : [1 -1]
Integer parameters : []
String Parameter :
```

If the block diagram contains algebraic loops it is not possible to zero the matrix.

Starting from this new ordered list of blocks, it is now possible to generate C-code.

The code contains 3 functions:
- The initialization function
- The termination function
- The periodic task

5.5.1 The init function

In this function each block is defined as a python_block structure

```
typedef struct {
    int nin;              /* Number of inputs */
    int nout;             /* Number of outputs */
    int *nx;              /* Cont. and Discr states */
    void **u;             /* inputs */
    void **y;             /* outputs */
    double *realPar;      /* Real parameters */
    int *intPar;          /* Int parameters */
    char *str;            /* String */
    void *ptrPar;         /* Generic pointer */
} python_block;
```

The nodes are defined as “double” variables and the inputs and outputs of the blocks are defined as vectors of pointers to them.

```
/* Nodes */
static double Node_1[] = {0.0};
static double Node_2[] = {0.0};
static double Node_3[] = {0.0};
/* Input and outputs */
static void *inptr_0[] = {0};
static void *outptr_0[] = {0};
static void *outptr_1[] = {0};
static void *inptr_2[] = {0,0};
static void *inptr_3[] = {0,0};
static void *outptr_3[] = {0,0};
```

```
inptr_0[0] = (void *) Node_2;
outptr_0[0] = (void *) Node_3;
```

```
block_test[0].nin = 1;
block_test[0].nout = 1;
block_test[0].nx = nx_0;
block_test[0].u = inptr_0;
block_test[0].y = outptr_0;
```

After this initialization phase, the implementation functions of the blocks are called with the flag “INIT”.

```
css(INIT, &block_test[0]);
step(INIT, &block_test[1]);
pprint(INIT, &block_test[2]);
sum(INIT, &block_test[3]);
```

5.5.2 The termination function

This procedure call the implementation functions of the blocks with the flag “END”
5.5.3 The ISR function

This procedure represents the periodic task of the RT execution. First of all, the implementation functions are called with the flag “OUT”, in order to perform the output update of each blocks. As second step, the implementation functions of the block containing internal states \( (nx \neq 0) \) are called with the flag “STUPD”.

```c
... css(OUT, &block_test[0]);
step(OUT, &block_test[1]);
print(OUT, &block_test[2]);
sun(OUT, &block_test[3]);
... css(OUT, &block_test[0]);
css(STUPD, &block_test[0]);
...```

5.6 The main file

The core of the RT execution is represented by the “python_main_rt.c” file. During the RT execution, the main procedure starts a high priority thread for handling the RT behaviour of the system. This main file is used to launch the executable in a Linux pre-empt_rt environment.

```c
void *rt_task(void *p)
{
    ...
    param.sched_priority = prio;
    if(sched_setscheduler(0, SCHED_FIFO, &param)==-1){
        perror("sched_setscheduler failed");
        exit(-1);
    }
    ...
    double Tsamp = NAME(MODEL,_get_tsamp);()
    ...
    NAME(MODEL,_init);()
    while(!end){
        /* wait untill next shot */
        clock_nanosleep(CLOCK_MONOTONIC,
            TIMER_ABSTIME, &t, NULL);
        ...
        /* periodic task */
        NAME(MODEL,_isr)(T);
    }
    NAME(MODEL,_end)();
}
```

5.7 The pyCodeGen application

A python application (pyCodeGen) is used to manage the full development (Figure 3), from schematic to C-code.

**FIGURE 3:** The pyCodeGen environment

The user should provide a schematic file, a template makefile (for simulation or RT executable) and a python initialization script, if needed.

6 Example

One of the educational plants in the SUPSI laboratory is the system shown in figure 4

**FIGURE 4:** The disks and spring plant

Two disks are connected by a spring. The goal for the students is to control the angle of the disk on the right by applying an appropriate torque to the disk on the left.

The physical model of this plant can be directly calculated in python using for example the *sympy* toolbox. Numpy can deliver a symbolic description of the system and using python “dictionary” we can easily get the numerical matrices of the state space form of the plant.
In [4]: A
Out[4]:
matrix([[0, 0, 1, 0],
        [0, 0, 0, 1],
        [-c/J1, -c/J1, (-d - d1)/J1, -d/J1],
        [-c/J2, -c/J2, -d/J2, (-d - d2)/J2]])

In [5]: B
Out[5]:
matrix([[0, 0],
        [0, 0],
        [kt1/J1, 0],
        [0, kt2/J2]])

The control system toolbox and the additional “yottalab.py” package contain all the functions required for the design of the controller. In this case we design a discrete state feedback controller, with integral part for eliminating steady state errors. The states are estimated with a reduced order observer. In addition an anti-windup mechanism has been implemented. The sampling time is set to 10 ms.

```python
# Sampling time
ts = 10e-3

gss1 = ss(A,B,C,D)
gss = ss(A,B,C2,D2)
gz = c2d(gss,ts,'zoh')

# Control design
wn = 10
xi1 = np.sqrt(2)/2
xi2 = 0.85

cl_p1 = [1,2*xi1*wn,wn**2]
cl_p2 = [1,2*xi2*wn,wn**2]
cl_p3 = [1,wn]
cl_poly1 = sp.polymul(cl_p1,cl_p2)
cl_poly = sp.polymul(cl_poly1,cl_p3)
cl_poles = sp.roots(cl_poly) # Desired continuous poles
cl_polesd = sp.exp(cl_poles*ts) # Desired discrete poles

# Add discrete integrator for steady state zero error
Phi_f = np.vstack((gz.A,-gz.C*ts))
Phi_f = np.hstack((Phi_f,[0,0,0,0,1]))
G_f = np.vstack((gz.B,zeros((1,1))))

# Pole placement
k = placep(Phi_f,G_f,cl_polesd)

# Observer design - reduced order observer
poli_o = 5*cl_poles[0:2]
poli_oz = sp.exp(poli_o*ts)

disks = ss(A,B,C,D)
T = [0,0,0,1,0,0,0,0,0,0,0,0]

# Reduced order observer
r_obs = red_obs(disksz,T,poli_oz)

# Controller and observer in the same matrix - Compact form
contr_I = comp_form_i(disksz,r_obs,k,ts,[0,1])

# Implement anti windup
(gss_in,gss_out) = set_aw(contr_I,[0,1,0,1,0,1])
```

We can perform the simulation of the discrete controller with the continuous mathematic plant using the block diagram of figure 5.

FIGURE 5: Block diagram for the simulation

The plant is represented by a continuous state space block with 1 input and 2 outputs. The controller implement the state feedback gains and the state observer and it has been splitted into a CTRIN block and a CTRFBK block in order to implement the anti-windup mechanism.

We can now generate the code for the simulation and launch the generated executable. The template makefile is “sim.tmf”.

The result is shown in figure 6.

FIGURE 6: Simulation of the plant

For generating the RT controller for the real plant, we first have to substitute the plant with sensors and actuators.

The sensors and actuators have been substituted with blocks that send and receive CAN message using a USB dongle of Peak System. The template makefile for this system is now “rt.tmf”. The block diagram for the real time controller is represented in figure 7.
FIGURE 7: Block diagram for the RT implementation

The motor position can be represented in real time into a web browser, using the XML protocol developed by Klaus Weichinger [6], or plotted in python at the end of the execution (see figure 8).

FIGURE 8: RT execution

7 Conclusions

This paper demonstrated a way to implement a simple CACSD environment which can be successfully used to generate RT code for controlling real plants. Python can be an interesting alternative to Matlab. The gEda tool suite is not the best solution to implement a Simulink alternative: for example it is not possible to perform some plausibility checks in gschem, as for example check if outputs are connected together. But this tool demonstrates how a graphic block diagram can be easily translated into C-code.

References