Teaching to write control code

Martina Maggio, Alberto Leva

Dipartimento di Elettronica e Informazione, Politecnico di Milano
Via Ponzio, 34/5 - 20133 Milano, Italy
{maggio,leva}@elet.polimi.it

Abstract: Regulators are conceived as dynamic systems, but almost invariably implemented as computer code. In the opinion of the authors the relationships between those two worlds are frequently overlooked in control education, although an insufficient knowledge of the involved concepts can hinder the correct operation of the realised systems. This manuscript presents a didactic activity that aims at closing the mentioned gap. The scope is for now limited modulating control, although extensions can be envisaged.

Keywords: Control Education, Digital control, Modulating control.

1. INTRODUCTION

A number of tools are nowadays available, that take as input a high-level description of a control strategy, e.g. a block diagram, and generate the “corresponding” code (the reason for the quotation marks will emerge in the next paragraph) for virtually any computing architecture: most of those tools are commonly used in control education since many years, see e.g. Bristol (1986), Åström and Lundh (1992), Amadi-Echendu and Higham (1997), Kheir et al. (1997), Dormido Bencomo (2004), Leva (2004), Duro et al. (2008), and many other works. Nonetheless, a control engineer needs to understand the process of transforming high-level description into executable code (Soklic, 2002; Duro et al., 2008; Meza et al., 2009). To say it crudely, as we use to do to the students, whatever controller has been specified. “you must be able to realise it with just a microprocessor and a compiler”.

One major reason for the requirement stated above is that it is inherently dangerous to rely on any code generation tool without fully understanding its operation, but there is more. Several relevant facts regarding industrial controllers do not (an cannot) emerge from their specification as continuous- or discrete-time dynamic systems, whence the quotation marks in the introductory paragraph, but heavily influence the control results as observed on the plant (Shinskey, 2002). The most relevant (but not the only) examples are the antiwindup implementation and the management of saturations and control variation interlocks.

In the authors’ opinion, having the students face controllers as code and not only as e.g. transfer functions is of great help to avoid the most common mistakes committed by the newbie— and according to experience, sadly, sometimes also by the quite seasoned professional. Also, such an activity can pave the way toward a firm understanding of how CACSD (Computer-Aided Control System Design) tools work when it is to generate the control code, to the advantage of a better conscience in the use (and for somebody possibly the development) of such tools.

In one word, facing control realisation at the machine level - if thoroughly related with the underlying theory - enhances the students’ grasp on the overall scientific and technological panorama, so that for example the informative content of works like that by Li et al. (February 2006) be caught completely.

At the Politecnico di Milano, the students are exposed to control code realisation as soon as possible, i.e., at the end of the first course in automatic control fundamentals, when they have already been also taught the basics of programming that are necessary to understand and carry out the activity: a first nucleus of the code shown here is was fact introduced in the course exercise book by Leva and Maggio (2010). This manuscript briefly presents the tasks and pedagogical results expected, together with some remarks on the present state of the activity, and possible future developments. Given the way the activity is conceived and carried out, it was chosen here to report quite long excerpts of C code, to help the reader see how commenting that code can relate it to theoretical facts. Needless to say, the set of C programs listed here will very soon be made available as free software.

2. THE MAIN TOOLS

For the activities presented herein, the gcc compiler and Scilab (both free software tools) were employed—the C language is thus used here, but the presented ideas are general, and consistent with educational trends exemplified by works such as that by Wenjiang et al. (2010). More precisely, a basic set of C functions was created to realise the required dynamic systems, and some Scilab scripts are used to plot and possibly post-process the generated data.

Of course both the C function set and the Scilab scripts are easily extensible, starting from the “core” realised to pursue the activity goals with the minimum necessary complexity, and presented here. In said core, the dynamic systems implemented are limited to a PID controller in the ISA (Åström et al., 2006) form

$$\text{CS}(s) = K \left( bSP(s) - PV(s) + \frac{1}{sT_i}(SP(s) - PV(s)) \right) + \frac{sT_d}{1 + sT_d/N} \left( cSP(s) - PV(s) \right) \quad (1)$$

where SP, PV and CS are respectively the set point, the process (controlled) variable and the control signal, and to a first-order
transfer function in the form

\[ P(s) = \frac{\mu}{1 + sT} \]  

(2)
to conveniently close the loops, so that meaningful experiments (albeit very simple from the control synthesis standpoint so as to concentrate on the specific purpose of the presented activity) can be performed. Incidentally, (2) can be a basic block to build more complex systems when needed.

The code corresponding to (1) and (2) is shown in listing 1. The authors apologise for basing the presentation on a significant amount of C listings, but found no alternative compatible with the available space. The reader who is not familiar with the C language can obviously find plenty of material on the matter. Of course, the code is deeply commented when talking to the students. Notice that backward-difference discretisation is used throughout, while the PID is realised in both the positional and the incremental form. Notice also the presence of antiwindup, realised in different ways according to the two PID implementations, and of the increment/decrement locks provided by the inputs \( F^+ \) and \( F^- \) (\( F^p \) and \( F^m \) in the code) that, when active, prevent the control signal from increasing and decreasing, respectively. Upper and lower saturations are signalled by the logical outputs HI and LO, while the “Track Switch” logical input TS causes the control signal to follow the “Track Reference” one, received via the input TR.

Listing 1 - csys.c

```c
// Input mem updated for next step
pd->PVo = pd->PV;
// State variables stored for next step
pd->CSo = pd->CS;
// Saturation signalling
if (pd->CS<pd->CSo) pd->CS = pd->CSmin;
else { // Tracking mode (can override locks)
    pd->CSo = pd->CS;
    pd->LO = pd->CS<=pd->CSmin;
    pd->HI = pd->CS>=pd->CSmax;
    // Antiwindup
    if (pd->CS>pd->CSo & pd->Fp)
        D = 0;
    else { // Tracking mode (can override locks)
        pd->HI = 0; pd->LO = 0;
        // Automatic mode
        DSP = pd->SP-pd->SPo;
        DSV = pd->PV-pd->PVo;
        P = pd->K*(pd->b*PDSP-DP); D = 0; }
    // Tracking mode (can override locks)
    pd->CS = pd->CSo;
    }
else { // Tracking mode (can override locks)
    pd->HI = 0; pd->LO = 0;
    // Automatic mode
    DSP = pd->SP-pd->SPo;
    DSV = pd->PV-pd->PVo;
    P = pd->K*(pd->b*PDSP-DP); D = 0; }
```
void tf1p0z(TF1P0Z_DATA *td)
{
    float x;
    x = (td->T * (td->xo + td->u)) / (td->T + td->Ts);
    td->y = td->mu * td->Ts / (td->T + td->Ts) * x;
}

void tf1p0z(TF1P0Z_DATA *td)
{
    float x;
    x = (td->T * (td->xo + td->u)) / (td->T + td->Ts);
    td->y = td->mu * td->Ts / (td->T + td->Ts) * x;
}

// Step and ramp functions ---------------------------------------------
float stp(float t) { return t >= 0? 1: 0; }
float rmp(float t) { return t * stp(t); }

3. SOME DIDACTIC EXAMPLES

This section presents some examples of the exercises proposed to the students. The typical assignment consists of tuning and then implementing a given control system, and then verifying the results obtained by the so written (C) code versus the simulations of the same system obtained with environments such as Scilab/Scicos (or equivalently MATLAB/Simulink).

3.1 Example 1: single-loop control

In the first example, the scheme of figure 1 is realised, leading to the program of listing 2; the incremental PID form is used, while the process under control is described by a transfer function in the form (2).

\[
dataPID1.SP = rmp(t-1) - rmp(t-5) - 0.5 \times rmp(t-50) + 0.5 \times rmp(t-60) + 2 \times stp(t-100) + 0.3 \times rmp(t-180) - 0.3 \times rmp(t-190);
\]

(again) the results are redundancies and a consequently error-prone code.

(4) Understand the phenomena relative to saturations (the windup one being the most relevant) and the countermeasures to take in the implementation; here too, no additions must be introduced in the regulator storage.

(5) Understand that positional and incremental realisations have inherently different behaviours, and learn to explain said differences in terms of dynamic systems. Also, learn to forecast (at least in simple cases) which aspects of a non-standard control structure may be affected by the way the single blocks are realised, so as to make the correct choices as for the type of realisation(s) to use.

(6) Learn to translate a control block diagram into a sequence of calls to library functions and data transfer operations that realise the connections, so as to start understanding the very basic operation of control-oriented compilers.

3. SOME DIDACTIC EXAMPLES

This section presents some examples of the exercises proposed to the students. The typical assignment consists of tuning and then implementing a given control system, and then verifying the results obtained by the so written (C) code versus the simulations of the same system obtained with environments such as Scilab/Scicos (or equivalently MATLAB/Simulink).

3.1 Example 1: single-loop control

In the first example, the scheme of figure 1 is realised, leading to the program of listing 2; the incremental PID form is used, while the process under control is described by a transfer function in the form (2).

\[
dataPID1.SP = rmp(t-1) - rmp(t-5) - 0.5 \times rmp(t-50) + 0.5 \times rmp(t-60) + 2 \times stp(t-100) + 0.3 \times rmp(t-180) - 0.3 \times rmp(t-190);
\]

(again) the results are redundancies and a consequently error-prone code.

(4) Understand the phenomena relative to saturations (the windup one being the most relevant) and the countermeasures to take in the implementation; here too, no additions must be introduced in the regulator storage.

(5) Understand that positional and incremental realisations have inherently different behaviours, and learn to explain said differences in terms of dynamic systems. Also, learn to forecast (at least in simple cases) which aspects of a non-standard control structure may be affected by the way the single blocks are realised, so as to make the correct choices as for the type of realisation(s) to use.

(6) Learn to translate a control block diagram into a sequence of calls to library functions and data transfer operations that realise the connections, so as to start understanding the very basic operation of control-oriented compilers.

3. SOME DIDACTIC EXAMPLES

This section presents some examples of the exercises proposed to the students. The typical assignment consists of tuning and then implementing a given control system, and then verifying the results obtained by the so written (C) code versus the simulations of the same system obtained with environments such as Scilab/Scicos (or equivalently MATLAB/Simulink).

3.1 Example 1: single-loop control

In the first example, the scheme of figure 1 is realised, leading to the program of listing 2; the incremental PID form is used, while the process under control is described by a transfer function in the form (2).

\[
dataPID1.SP = rmp(t-1) - rmp(t-5) - 0.5 \times rmp(t-50) + 0.5 \times rmp(t-60) + 2 \times stp(t-100) + 0.3 \times rmp(t-180) - 0.3 \times rmp(t-190);
\]

(again) the results are redundancies and a consequently error-prone code.

(4) Understand the phenomena relative to saturations (the windup one being the most relevant) and the countermeasures to take in the implementation; here too, no additions must be introduced in the regulator storage.

(5) Understand that positional and incremental realisations have inherently different behaviours, and learn to explain said differences in terms of dynamic systems. Also, learn to forecast (at least in simple cases) which aspects of a non-standard control structure may be affected by the way the single blocks are realised, so as to make the correct choices as for the type of realisation(s) to use.

(6) Learn to translate a control block diagram into a sequence of calls to library functions and data transfer operations that realise the connections, so as to start understanding the very basic operation of control-oriented compilers.

3. SOME DIDACTIC EXAMPLES

This section presents some examples of the exercises proposed to the students. The typical assignment consists of tuning and then implementing a given control system, and then verifying the results obtained by the so written (C) code versus the simulations of the same system obtained with environments such as Scilab/Scicos (or equivalently MATLAB/Simulink).

3.1 Example 1: single-loop control

In the first example, the scheme of figure 1 is realised, leading to the program of listing 2; the incremental PID form is used, while the process under control is described by a transfer function in the form (2).

\[
dataPID1.SP = rmp(t-1) - rmp(t-5) - 0.5 \times rmp(t-50) + 0.5 \times rmp(t-60) + 2 \times stp(t-100) + 0.3 \times rmp(t-180) - 0.3 \times rmp(t-190);
\]

(again) the results are redundancies and a consequently error-prone code.

(4) Understand the phenomena relative to saturations (the windup one being the most relevant) and the countermeasures to take in the implementation; here too, no additions must be introduced in the regulator storage.

(5) Understand that positional and incremental realisations have inherently different behaviours, and learn to explain said differences in terms of dynamic systems. Also, learn to forecast (at least in simple cases) which aspects of a non-standard control structure may be affected by the way the single blocks are realised, so as to make the correct choices as for the type of realisation(s) to use.
Simulation data are written in a text file that can be subsequently read by any suitable program (Scilab was used here) to plot and inspect the results. In particular, running the C program of listing 2 and then the Scilab one of listing 3 produces the results of figure 2.

**Listing 3 - plotsingleloop.sce**

```scilab
M = fscanfMat("data.txt");
t = M(:,01); SP = M(:,02); PV = M(:,03); CS = M(:,04);
TS = M(:,05); TR = M(:,06); Fp = M(:,07); Fm = M(:,08);
HI = M(:,09); LO = M(:,10);
```

```scilab
hf = scf(0); clf;
subplot(411); plot(t,SP,'k:',t,PV,'r');
ax = gca();
ax.data_bounds = [t(1),min(PV)-0.1;t($) ,max(PV)+0.1];
subplot(412); plot(t,TR,'k:',t,CS,'b');
ax = gca();
ax.data_bounds = [t(1),min(CS)-0.1;t($) ,max(CS)+0.1];
subplot(413); plot(t,HI,'r',t,LO,'k:');
ax = gca();
ax.data_bounds = [t(1),-0.1;t($) ,1.1];
subplot(414);
plot(t,Fp,'r',t,Fm,'k:');
ax = gca();
ax.data_bounds = [t(1),-0.1;t($) ,1.1];
```

The simulation results are depicted in figure 3. Notice the different behaviours of the two PID forms when the set point weight is changed and the set point is altered while the PID is in tracking mode. Details are omitted here relying on the reader’s expertise, but discussed in detail with the students: such discussion significantly help relating the “block diagram” and the “C code” worlds, and above all understanding that such a relationship can (hence must) contain no ambiguity. In this and the following examples too, plots are obtained with a Scilab program analogous to that of listing 3, omitted for brevity.

**Fig. 3. Positional versus incremental example results.**
3.3 Example 3: cascade control

In this example the scheme of figure 4 is realised, obtaining the program of listing 5 (the type of PID used is inessential for the purpose of the example).

![Cascade example scheme](image)

Fig. 4. Cascade example scheme.

Listing 5 - cascade.c

```c
// Cascade example ----------------------------------------
#include <stdio.h>
#include "csys.c"

int main(void)
{
    TF1P0Z_DATA dataPi;
    TF1P0Z_DATA dataPe;
    ISAPIDinc_DATA dataPIDi;
    ISAPIDinc_DATA dataPIDe;
    float Ts,t;
    int nSteps,k;
    FILE *h;
    Ts = 0.1;
    nSteps = 5000;
    isapidinc_coldinit(&dataPIDi,2,2,0,1,1,0,-2,2,
                        Ts,0,0,0);
    isapidinc_coldinit(&dataPIDe,2,30,0,2,1,0,-10,10,
                        Ts,0,0,0);
    tf1p0z_coldinit(&dataPi,1,3,Ts);
    tf1p0z_coldinit(&dataPe,1,30,Ts);
    h = fopen("data.txt","w");
    for(k=0;k<nSteps;k++)
    {
        t = k*Ts;
        dataPIDe.SP = stp(t-10)+2*stp(t-100)-1.5*stp(t-300);
        // Comment the following two lines to disable the
        // interloop windup prevention logic
        dataPIDe.Fp = dataPIDi.HI;
        dataPIDe.Fm = dataPIDi.LO;
        dataPIDe.PV = dataPe.y;
        isapidinc(&dataPIDe);
        dataPIDi.SP = dataPIDe.CS;
        dataPIDi.PV = dataPi.y;
        isapidinc(&dataPIDi);
        dataPi.u = dataPIDi.CS;
        tf1p0z(&dataPi);
        dataPe.u = dataPi.y;
        tf1p0z(&dataPe);
        fprintf(h,",%f	%f	%f	%f	%f	%f	%f	%d	%d
", t,dataPIDe.SP,dataPIDe.PV,dataPIDe.CS,
                dataPIDi.SP,dataPIDi.PV,dataPIDi.CS,
                dataPIDi.Fp,dataPIDi.Fm);
    }
    fclose(h);
}
```

Notice the logic feedback from the saturation signals of the inner loop to the increment and decrement locks of the outer loop. In the presence of that feedback the outer PID is prevented from further pushing the inner loop into saturation when the inner controlled variable hits one of its limits, while the same “interloop windup prevention” is disabled if said feedback is cut. The two situations correspond respectively to figures 5 and 6, where the different behaviour is apparent.

![Cascade example results with interloop windup prevention](image)

Fig. 5. Cascade example results with interloop windup prevention: notice the behaviour of the $F_+$ and $F_-$ signals of the external regulator, connected to the saturation signals of the internal one as per figure 4.

![Cascade example results without interloop windup prevention](image)

Fig. 6. Cascade example results without interloop windup prevention ($F_+$ and $F_-$ of the external regulator left unconnected, i.e., stuck to “false”).

4. FURTHER EXPERIENCES

Based on the presented code, many other tasks can be proposed to the students. A brief list of possibilities is reported below.

- Add a bias term to the PID regulators presented above, understanding how to manage the integral term correctly depending on the PID form.
- Realise other PID controller forms such as the series one, paying particular attention to the meaning of the parameters.
- Realise other antiwindup types such as that based on the actuation error feedback to the integral term, or that using a read-back of the applied control from the actuator.
- Realise a generic transfer function with saturation and tracking management, and employ that block to obtain generic controllers of both the one- and the two-degree-of-freedom type.
- Employ the so obtained blocks to build the major control structures employed in practice, such as the feedforward compensation of a measured disturbance, and the multivariable control ($2 \times 2$ is enough for didactic purposes) with decoupling.

As can be seen, such activities extend far beyond the fundamental course, comprising more technological ones (e.g. that titled
“Engineering and technology of control systems”). Also, some of said activities involve nontrivial issues that are frequently overlooked in the applications. For example, the forward and backward decoupling structures have very different needs as for the controller mode management when for example one of the loops is set to manual, and the way the bias term is implemented in the PID (or the generic controller) needs considering.

In one word, continuing the activity presented herein allows to make the students’ understanding of control code grow in parallel with that of the theory that grounds control design. As a result, no doubt should survive that when the control code “behaves differently from the block diagram” there surely exist a way to explain and correct the problem in the methodological domain. A professional educated in this way naturally sees and understands the whole design cycle, and while it is clear to him/her that most of said cycle is done with code generation tools for apparent practical reasons, it is equally clear that sometimes manual interventions may be necessary, and that the methods required for said interventions come in the first place from a firm knowledge of the systems and control theory.

As even more advanced activities, the control code presented here could be ported as algorithm in a modelling language allowing to mix equation- and algorithm-based modelling such as Modelica (The Modelica Association, 2010; The Open Source Modelica Consortium, 2010), and applied to arbitrarily complex objects: Åkesson et al. (2009), Martin-Villalba et al. (2010) and similar recent works show how many interesting possibilities can be opened. The direct use of the code within Modelica could also be studied. And needless to say, in very technological courses one could consider porting control code e.g. to microcontroller-based architectures.

Also, the activity could (and will) be expanded to include logic control. Once the students know how to arrive to the code not only from a block diagram but also from say a SFC (Sequential Functional Chart) one, it is possible to treat complete systems, including for example the (sometimes complex and very often worth studying) logic required for plant startup, recipe changes, and so forth. In fact such extensions and activities have already been started in a new course, and will be presented in the near future.

5. CONCLUSIONS AND FUTURE WORK

A set of C and Scilab programs was presented that allows the students to face the realisation of controllers as computer code without the intermediation of any dedicated generation tool. Relating a regulator “as block diagram” and “as code” has already proven to enhance the comprehension of both sides of the story. And what is more, the initiated activity helps inducing a system-theoretical view on the code itself, so that the storage of a past value in a program is instinctively viewed as a state variable in a dynamic system, which is not guaranteed at all if the control synthesis and the programming activities are not treated in a tightly coordinated way, and is on the other hand extremely beneficial for the educational result of both tasks.

Future work will be devoted to extend the scope of the code and the activity, including more advanced modulating controllers, and also including logical control (specified e.g. in the SFC language). Doing so will allow to treat not only modulating loops but also complete, fully functional control systems, always with the coordinated approach presented herein.

REFERENCES


