Teaching PID control to computer engineers: a step to fill a cultural gap

Alberto Leva

Dipartimento di Elettronica e Informazione e Bioingegneria, Politecnico di Milano, Italy (e-mail: alberto.leva@polimi.it).

Abstract Many computer engineers either do not receive any control education, or are led to think that studying control is necessary only for those who want to specialise in embedded systems, real-time, and so forth. This causes a cultural gap that can have highly undesired consequences, since control-based techniques are gaining importance as a means to manage and optimise computing systems. Given the conflicts for time and space that are typical of articulated curricula like computer engineering, however, a full course on the principles of systems and control, tailored to computer engineering, is quite often an infeasible solution. This paper presents an alternative – or maybe better, a workaround – based on a suitably tailored PID-centred activity, where the occasion is taken to introduce and stress selected general ideas.

Keywords: PID control education; Control of computing systems.

1. INTRODUCTION

There is a high interest – both academic and technological – and a vast literature on the use of control, often in the form of simple structures based on PID loops, to manage and optimise the behaviour of computing systems, see e.g. the blueprint by IBM (2003), the papers by Abdelzaher et al. (2003) and Diao et al. (2005) – written respectively to address the control and the computer communities – or the books by Hellerstein et al. (2004); Janert (2013); Leva et al. (2013), covering almost a decade, and their bibliographies.

However, this apparent interest for control in the computer community, still does not reflect into education as strongly as it should. Many computer engineering – let alone computer *science* – students, simply do not receive any control education. And those who do, most often take some "systems and control" courses quite late in their curriculum, only if they are interested in embedded systems, real time, and the like. This way, computer engineering students either just ignore the systems and control theory, or are implicitly taught that computers serve to build controls, but the core ideas behind control – dynamics, feedback, and so forth – have hardly anything to do with the design of computing systems.

Such a philosophy is enforced by most curricula recommendations at the undergraduate level, where the foundations for the students' mentality are laid, see e.g. the document by the Joint ACM/IEEE Task Group on Computer Engineering Curricula (2016). Hence it is definitely a mainstream philosophy, and proposing a significantly different viewpoint requires thorough justification. Sticking to the subject of this paper, we therefore need in the first place to ask ourselves whether or not computer engineers need control education independently of their specialisation. Then, since the answer is (quite expectedly) affirmative but at the same time the computing domain is peculiar indeed, a second question immediately arises as to what is the best pedagogical goal for those students. And finally, as in the addressed domain PI(D) control emerges quite naturally as a useful tool, we come to wonder whether a purpose-specific didactic activity based on it, can provide some countermeasure for the cultural gap sketched above.

The rest of the paper is devoted to provide a tentative answer to the questions just posed, and is organised as follows. Section 2 is devoted to showing why all control engineering students strongly need control education, and as a consequence, Section 3 expresses some ideas – based on the author's experience – about what the pedagogical approach should be. A possible activity, centred on the basics of control and taking profit of PID control to convey the most important ideas according to the mentioned approach, is sketched out in Section 4. Finally, Section 5 draws some conclusions and outlines future work.

2. WHY ANY COMPUTER ENGINEERING STUDENT NEEDS CONTROL EDUCATION

Premising that this topic would deserve a paper by itself, here we start from the opposite side with respect to undergraduate education, i.e., from the effects of the mentioned cultural gap on research. First, let us make and discuss a few statements based on the survey by Patikirikorala et al. (2012) on control engineering approaches for self-adaptive software, where 161 papers published between 2001 and 2011 are classified and analysed.

- The survey excludes works not "utilising control-theoretical approaches" , which "also excludes the control solutions primarily based on fuzzy logic, neural networks, case based-reasoning and reinforcement learning" (Patikirikorala et al., 2012, p. 35).
- Works on "hardware [...] or operating system level management issues" are also excluded (*ibidem*, p. 35).

- The taxonomy axes are "Target system (application domain, performance variable, dimension), Control system (model, type, loop dimension, scheme) and Validation (simulation, case study" (*ibidem*, Fig. 1). Models are mostly black box (*ibidem*, Table III) as "analytical" ones are "not available or significantly complex" (*ibidem*, p. 36).
- The variety of applied control schemes is wide, from PID to LQR., MPC and more (*ibidem*, Table III).

While the first statement seems to focus precisely enough on "classical" control, the second is quite surprising in a domain where for example the way a resource is allotted to an application may *heavily* depend on the internals of the operating system. When dealing with *functional* requirements (over-simplifying, *what* software has to do) taking the lower layers of a system just as a matter of fact is correct, but for *non-functional* requirements (*how*, for example how fast, the software has to do it) interactions *necessarily* come into play. And since the role of (classical feedback) control is mainly to enforce properties of the second type, for example to ensure quality of service, computer engineers cannot only possess the "layered" view of a system that is quite typical (over-simplifying again) of software engineering; they also need – it is adding, not replacing – the idea of interacting subsystems that is typical of control.

Coming to the other statements, taxonomy axes – again surprisingly – do not include how the model structure is chosen, how the controller structure is consequently selected (including e.g. how to decide where the additional complication of an adaptive one is worth the effort), and how the controller is tuned. Given also the variety of laws and schemes – two ideas sometimes confused with one another, as for example two items in Table III are "LQR" and "cascade" – the work (despite its undoubted value) ends up, viewed from a control standpoint, as a taxonomy more of how algorithms were taken from a bookshelf and applied, than of what control problems look like, and thus how control schemes need structuring and tuning.

On this last aspect, a second paradigmatic work is that by Heo and Abdelzaher (2009) on AdaptGuard. The idea is that "adaptation loops" [for which the authors just mean feedback loops, incidentally], "implicitly assume a model of system behaviour that may be violated; [...] in the absence of an *a priori* model of the adaptive software system, [AdaptGuard] has to anticipate system instability, attribute it correctly to the right runaway adaptation loop, and disconnect it, replacing it with conservative but stable open-loop control until further notice". The way to detect a "violation" is to infer the system causality from I/O measurements, and identify reversals in the dependencies among the monitored variables to conclude that some feedback has changed its sign, therefore provoking instability. In control terms, this means taking a decentralised control scheme, and open or close some of its loops on an observation basis—i.e., turning that scheme into a state-based switching one. No doubt the reported examples work, but in the absence of an *overall* stability analysis, which could be extremely cumbersome even in an LTI setting, the author bears to state that this basically means that the controlled system is tolerant indeed (which is somehow acknowledged in the paper by saying that "open-loop actions are stable"). In general, as the lesson to learn here, we could say that replacing a controller with another one, especially in a multivariable and interacting system, is *not* the same as replacing a generic algorithm with another one for the same purpose. But again, to appreciate why, one needs to know about the basics of control.

Sticking to the idea that there is much more to a controller than its algorithm, one can further observe that in most works on "adaptive" computing systems, control theory is viewed as an *alternative* to other techniques like e.g. machine learning or heuristics, not as a mental framework to view problems, and since several computing-related problems do require significant skills to find the right abstraction and cast them into a control paradigm, the conclusion is frequently taken that "the problem is too complex" for control – in the classical sense – to be applied. As a result, for many computer engineers and scientists, control is really hardly anything more than a bookshelf to take pre-built algorithms from when these seem (no matter why) to fit the problem, with the detrimental effects above.

Only recently it has been recognised that "even for software systems that are too complex for direct application of classical control theory, the concepts and abstractions afforded by control theory can be useful" (De Lemos et al., 2017). Based also on the very few examples just quoted, we state that the answer to the first question is positive: the earlier computer engineering students are exposed to the core ideas behind systems and control, no matter what specialisation they will subsequently choose, the better.

3. CONTROL PEDAGOGY FOR COMPUTER ENGINEERS

Having established that computer engineers need "the core ideas of control", the question is now what those core ideas are. The matter is far from trivial, for at least two reasons: first, the computing domain is highly peculiar with respect to any other control one; second, assuming that the students receive systems and control theory education as early as possible, most likely they will subsequently be left basically alone as for establishing relationship between that theory and the design practices they will be taught. This is not an impossible assignment, but no doubt it is not easy at all. As such, and specifically in a view to strengthening the students enough for the said assignment, the ideas to convey to them must be as general as possible with respect to any specific application, easy to grasp and understand firmly, and as mathematically light as possible so as to be easy to retain and keep alive even when technical details get lost.

In particular, at least according to the author's experience, attempting to evidence and discuss the peculiarities of the computing domain should be avoided. No doubt they are relevant: to quote just two, in no other domain the controller and the controlled objects are as homogeneous as two pieces of software, nor are they manufactured together and often by the same developer, and in no other domain can sensing and actuation depend on component (like operating systems modules) that were not necessarily designed for the purpose of control—see Leva et al. (2013) for details impossible to report here. However, discussing these facts at the undergraduate level is definitely prema-

ture, and instead of clarifying the *scenario* as it would do with a more educated audience, may on the contrary induce the idea that control concepts (not the way they are applied, beware) need somehow "customising" for computing systems, which is the exact opposite of the intended goal.

At the Politecnico di Milano, the author teaches a course titled "Fundamentals of automatic control" to sophomore computer engineering students. This is not a standard situation, as the course comprehends 65 hours of lecture, 35 of classroom practice, and two laboratory sessions of 3 hours each, which is far more than usual; the percent distribution of the lecture hours among the course subjects is shown in Figure 1. Note the apparently small part devoted to PID, but consider the hours for "control synthesis" in general.

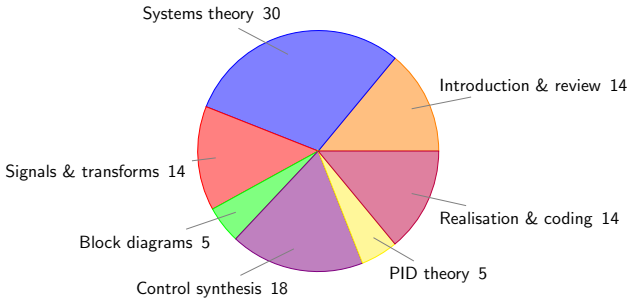

Figure 1. Percent hours distribution in the "Fundamentals of automatic control" course.

Despite its peculiarity the course provides an interesting probe, because several students show up some years later for a MSc thesis, most frequently involving control and computing systems. That is a good moment to first see what they retained of the subject, and then – during the activities carried out in cooperation – to observe how they apply control ideas, what are their weaknesses, and what is felt as really "fundamental".

The result of this experience over more than 15 years clearly indicates that for computer engineering students, at the undergraduate level, the really important things are the *general* ideas of dynamics, feedback, disturbance rejection, (internal) stability, and performance indices. Also, limiting the scope to the discrete time domain is feasible. More advanced concepts are surely necessary e.g. for researchers willing to cooperate with control scientists, but these can be quite easily learnt later on, if the basis is solid enough.

In a generic computer engineering curriculum, however, only a limited amount of hours (say 20 or less) could possibly be devoted to teaching control to everybody, either as a standalone activity or joined e.g. to some software engineering course: according again to the author's experience it can be very difficult to negotiate such a space, but in the last years the interest for the subject is growing, and the argument that "sooner or later education should start following" may work. Hence in the following we assume that some space is available, and address the problem of making the most effective use of this space in the light of the considerations above. And this is where PID control comes into play.

4. DIDACTIC ACTIVITY

The aim of this section, more than describing an activity in detail, is to just give a sketch of it, to illustrate the way it should be designed for the purpose stated above, and in particular, why and how PID control can be exploited. Once the why and the how are established, in fact, the detailed activity design is largely adaptable to the instructor and the class peculiarities—possibly even revolutionising the presented sketch completely.

To provide some operational figures, supposing to start completely from scratch if not for basic algebra, calculus and procedural programming, which seems a reasonable outset, the activity can span about 16 hours. A set of slides, targeted to approximately ten hours of lectures and six of classroom practice, is at present being prepared and will be offered as soon as ready as a possible teaching material (or better, as an example of how the concepts exposed hereafter can be applied). Needless to say, also the following treatise is based on the author's experience, thus no absolute truth is claimed, and if the following presentation may seem a bit prescriptive, this is only for compactness. In fact, the author has far more doubts than certainties: discussions, criticisms and alternative proposals would be highly beneficial and therefore appreciated.

This said, a good way to start the treatise is to point out some characteristics of computer engineering students that further justify the quest for a specific manner to introduce systems and control to them, but at the same time ground the proposed activity. The list is not exhaustive, of course, and also a bit *tranchant* for brevity, but enough for our purposes.

- When confronted with a problem, computer engineering students tend to *first* come up with a solution and *then* discuss its properties.
- In doing this, like in virtually any reasoning, they like to base the discussion on "use cases" wherever this makes sense.
- When the problem is to reach a goal, they tend to seek the "magic move" to get there in one step, under the implicit assumption that if the goal is missed and the move needs repeating, aiming constantly at the goal will result in the shortest path.
- They are acquainted to describing functional objects as algorithms and data structures, or state machines and data paths, rather than with models made of equations.

There is a vast literature supporting the idea of designing systems by "optimising the most frequent use case", and such an approach has already proven to fit a number of designs. But for learning control, sticking only to this *forma mentis* also causes problems. Evidence is (briefly) provided in the following. Now, for the activity sketch.

4.1 Incipit

A possible introduction could sound like this. If you (the students) search the literature for "control in computers" you will find that (i) there is a lot of material, and (ii) the PID controller is one of the most widely employed. Since an incorrect use of that object can be as disastrous

as a good use beneficial, however, you must understand the underlying principles, that is, the essentials of systems and feedback control theory. This activity will make you capable of using PID control knowledgeably, and as a by-product, will teach you what a dynamic model of a system is, which in turn is useful e.g. to detect when a problem *cannot* be tackled with a PID and thus you need to learn more control, or seek advice from a specialist, or both.

4.2 Step 1

This step, say two hours long, is about generalities and formalisms. Start with a very colloquial introduction to "control", make a lot of examples, in the continuous and the discrete time. and possibly also some simple event-driven cases. Provide literature references, also from the computer literature, but essentially to make the students aware of the existence of the "control in/for computers" research and technology domain. Most important, however, if not for the purpose just mentioned, positively refrain from approaching examples in the computing domain, or in a few minutes the class is lost into irrelevant technological details.

On this front, speak the language they will encounter later on when learning/reading about self-adaptive systems and the like, which is basically the way the computer community calls systems endowed with control, and make this language choice explicit. Introduce the ideas of "objective" and tell them that when this is representable with a signal (and it is in more cases than one may expect) we call it set point or reference, of "outcome" or "quality metrics" (controlled variable), "action" or "tuning knob" or "parameter" (control signal), and finally "exogenous" or "environmental" or "external action" (disturbance). Pay attention to false friends, that are ubiquitous. For example, in systems theory "parameter" is a quantity characterising a system, not a variable; in computer engineering "parameter" is understood as in "formal" and "actual parameter" for a function in a program, i.e., it means "input".

Then introduce the idea of dynamic system, the basic control-to-system connections, and the ways the controller can be invoked (no words required here on this). Recall that computer engineering students start out with a prescriptive and algorithmic – rather than a-causal and equation-based – mindset, and as long as they can cast ideas into that mindset without misinterpretations, following explanations is facilitated. A possible outcome of this step is a brutally simplified taxonomy, like

```
Controller:
   type                      = {modulating,logic}
   timing                    = {continuous,discrete,
                                event-triggered}
   connection_with_system    = {open-loop,closed-loop}
   disturbance_compensation  = {present,absent}
```

Finally, restrict the focus to the modulating discrete-time closed-loop case, with the motivation that (i) it is enough to explain the general ideas and (ii) it allows for particularly standard and general-purpose (the magic word for them is "reusable") solutions.

4.3 Step 2

This, say two hours as well, is about feedback and dynamic systems. In this order, because the former has to be understood independently, or the risk of mixing up feedback and iterative computations, as well as block and flow diagrams, is emphasised when on the contrary it needs strongly preventing, as the mindset of computer engineering students is particularly keen to such conceptual mistakes. Just describing, needless to say—no intention to judge. Figure 2 reports three slides devoted to introducing feedback in general, no further explanation is needed here.

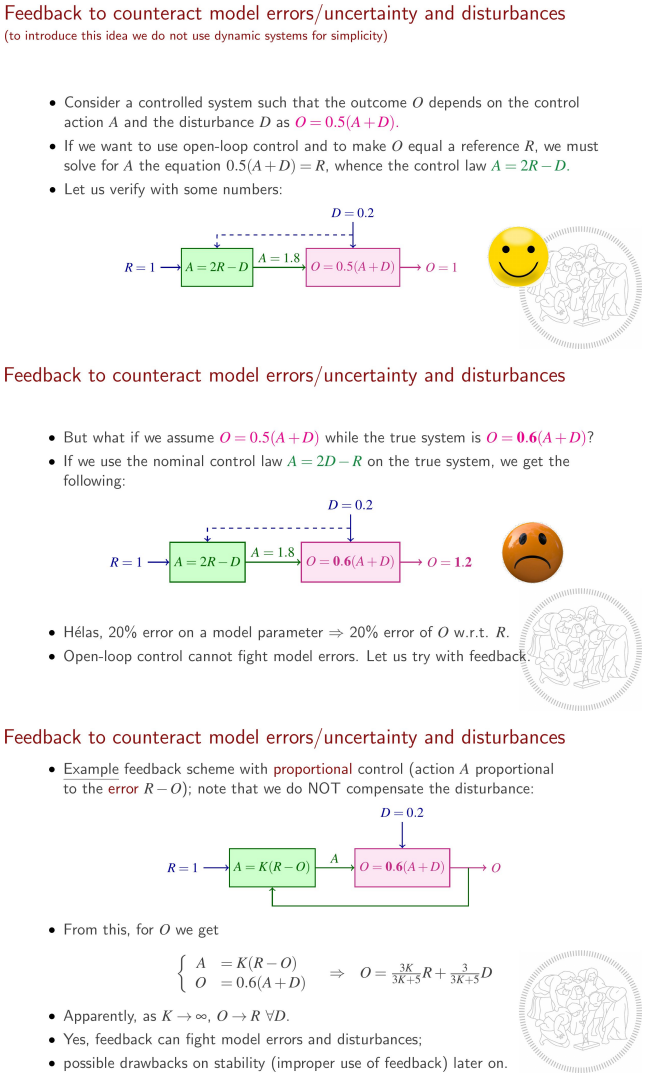

Figure 2. Teaching material example – slides to introduce feedback.

As for dynamics, one can go the usual way to explain that some systems remember the past. Just take care to call the state in as many ways as possible like "present condition", "internal configuration", and so forth. This, if firmly grasped, will allow the students, later on in their studies, to understand that the idea of "adaptiveness" as "the ability of a system to respond in different manners to the same stimulus depending on some internal condition" can *sometimes* (but not so infrequently, one may add) be re-formulated simply as that system being dynamic.

This said, write some example in the time domain, introduce the one-step advance operator z (without talking about transforms, at least in the author's opinion) and show that any input-output model can obviously be written as a "compound" operator, that we call "transfer function", constructed with the elementary one z.

Finally, just a few words on stability. A fast way to say just the necessary, is to analyse the first-order case, which is trivial, and then view higher-order systems as the series/parallel of first-order ones; the role of the poles is immediately evident. Systems with complex poles can be left as an exercise. At this point, it is convenient to have the first couple of classroom practice hours.

4.4 Step 3

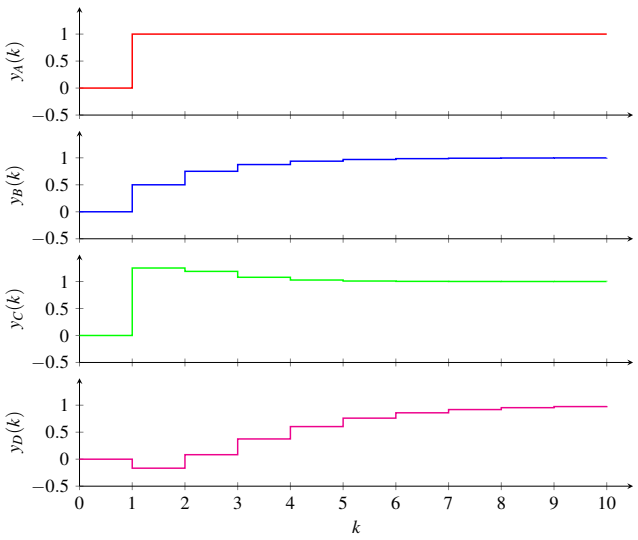

Figure 3. Example of step responses from dynamics suitable for PI/PID control, for model interpretation.

This is about describing system and goals. Limit the scope to the asymptotically stable case, it suffices here. Show how to compute a step response (first by hand and then e.g. with Scilab [1]), point out the role of the gain, and introduce other characterising quantities like settling time, maximum overshoot, undershoot. Now, show some responses – Figure 3 is an example – and provide some "use case" interpretation. For example, if the input is an allotted resource and the output a performance metrics for an application, the four responses shown could be explained as follows.

A. The resource is acquired and immediately exerts its effect (thus seen at the very next step).
B. The resource acts immediately but takes some steps for its full effect (for example because a queue needs emptying).
C. The resource produces transiently enhanced effect (typical when the metrics is the *speed* toward a goal).
D. The resource requires effort to be acquired and initially reduces performance, like e.g. a new core that initially makes a lot of cache misses.

Apparently, all responses come from a second-order transfer function, possibly with one zero (which is well suited for PID control, incidentally). Stress as strongly as possible that different "use cases" come from *one* model. This is very important, because in hardly any other place may the students get the idea that a single object can produce so different behaviours by just changing parameters. As an example of the resulting damage, the author once had a very hard time convincing (maybe) some non-newbie computer people that an automated vehicle does not need one controller "per behaviour" – i.e., one for turning, one for accelerating, one for braking and one for cruising (plus of course a supervisor deciding which one to activate, and possibly learning from experience) – but basically (accepting here a high-level viewpoint) a direction and a speed control receiving the convenient set points. Such attitudes need preventing as early as possible.

Carrying on, show how requirements in the time domain can be translated into a desired reference-to-output or disturbance-to-output transfer function, and here too, do rely on computer-related examples like varying the CPU share of a task, recovering a response time in the face of a load increase, and so on: in the quoted books there are a lot of suggestions. Then, draw the block diagram of a closed-loop system (according to experience the diagram formalism is self-evident enough) and show how the required transfer functions depend on that of process and controller by loop balance equations. Finally, learn to obtain a controller by solving the required closed-loop equation (first by hand and then e.g. with wxMaxima [2] , see the example script below.

```
kill(all);
C  : rhs(solve(c*P/(1+c*P)=To,c)[1]);
C1 : factor(subst([P=2/(z-0.5),To=0.2/(z-0.8)],C));
D2Y: factor(subst([P=2/(z-0.5),c=C1],P/(1+c*P)));
```

In the script first a controller is computed to assign the reference-to-output dynamics, then a specific case is shown, and finally it is verified that here asymptotic set point tracking requires an integrator and implies asymptotic disturbance rejection.

4.5 Step 4

Specialise the treatise to 1st/2nd order case processes, possibly with an integrator, and revisit examples to see how wide the coverage is. Derive controllers and introduce the PI/PID, which is now straightforward. Isolate and interpret the actions in common language (prompt response, zero steady-state error, anticipation). Introduce windup and antiwindup. Write a PI algorithm step by step and commenting. Give a PID one to study at home. This may be a good point for two further hours of practice, paying attention to the code. Do not disregard this as in computers one has *very* frequently to write it, and the network (a primary source of information for students) provides many examples where not even antiwindup is in place; see Maggio and Leva (2011) for a deeper discussion on this aspect.

To end this part, compute and plot some responses, including the control signal. Stress that (i) the "solution" was not

[1] http://www.scilab.org

[2] https://sourceforge.net/projects/wxmaxima

figured out directly as algorithm but through a model that *generated* the algorithm *unambiguously*, (ii) feedback leads the system to the desired state with no attempt to find any "magic move", and (iii) when properties are assessed *formally* on models, they are guaranteed. Extensive testing is useful to check the correctness of the code, and for other issues not pertaining to basic teaching, but not to verify the properties of a dynamic system.

4.6 Step 5

As the final step, explain why the PI(D) fits so many problems nut also what are its limitations. Stress the importance of figuring out the structure of the controlled dynamics. Show some examples when the desired TF cannot be obtained with a PI(PID. Carry out some tuning procedures and stress the importance of doing it properly (for example, deliberately make the integral action insufficient and see how long it takes to get to steady state after a fast initial transient, or tune for reference tracking and see how sluggish the disturbance response can be, and so forth). Bring in some process/model mismatch, explain its possible meaning (for example, a gain reduction may be a processor running slower because the overheating protection has intervened) and see the results; take the occasion to say a very few words about robustness, and sketch out open problems (no need for a list here) to stimulate further learning. Two final hours of practice, allowing the students to propose examples, can conclude the activity.

4.7 Some general remarks

Although the organisation is largely indicative, the proposed activity should inherently be capable of addressing the issues stemming from the bullet list at the beginning of this section. The students should at least perceive the power and usefulness of not attempting to find a solution without going through a formalisation (modelling) of the problem, of not relying excessively on any set of use cases (no matter how wide), of not seeking the "one-step" path to the optimum, whatever it is, but rather letting feedback do its work, and finally of assessing anything in the world of models, when possible, and letting algorithms follow. The students should also reasonably master basic PID control, in the SISO discrete time case and for asymptotically stable or integrating low-order processes, which is enough in many computing-related case. Finally, they should be able of detecting that a problem is not tractable this way based on the dynamic characteristics of that problem, for example as stemming from process responses—and abstraction capability that only control can teach, and is precious in many situations.

5. CONCLUSIONS AND FUTURE WORK

An activity was presented, of size compatible – hopefully, at least – with being inserted in a computer engineering curriculum and proposed to *all* the students. The activity is based on PID control, but not totally centred on it. In fact, the underlying *rationale* is to provide the students with firm and clear ideas about what control is, as early as possible in their education.

Apart form presenting a sketch of the activity, the focus was here set on (some of) the main cultural obstacles to a correct understanding – and therefore an effective adoption – of control concepts and methods in the computing domain.

As anticipated, some teaching material is being prepared, and will be made available – in the form of slides and scripts for free software tools – as soon as possible. Future work will consist in refining this material, and learning from experience to refine the didactic approach as well. The author hopes in the first place that the ideas expressed herein, and the material just mentioned, will be helpful to both the control and the computer engineering communities, and then, more in perspective, that all of this can stimulate discussions and cooperation.

REFERENCES

Abdelzaher, T., Stankovic, J., Lu, C., Zhang, R., and Lu, Y. (2003). Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3), 74–90.

De Lemos, R., Garlan, D., Ghezzi, C., Giese, H., Andersson, J., Litoiu, M., Schmerl, B., Weyns, D., Baresi, L., Bencomo, N., et al. (2017). Software engineering for self-adaptive systems: Research challenges in the provision of assurances. *Software Engineering for Self-Adaptive Systems III. Lecture Notes in Computer Science*, 9640.

Diao, Y., Hellerstein, J., Parekh, S., Griffith, R., Kaiser, G., and Phung, D. (2005). A control theory foundation for self-managing computing systems. *IEEE journal on selected areas in communications*, 23(12), 2213–2222.

Hellerstein, J., Diao, Y., Parekh, S., and Tilbury, D. (2004). *Feedback control of computing systems*. John Wiley & Sons, New York, NY, USA.

Heo, J. and Abdelzaher, T. (2009). AdaptGuard: Guarding adaptive systems from instability. In *Proc. 6th International Conference on Autonomic Computing*, 77–86. Barcelona, Spain.

IBM (2003). An architectural blueprint for autonomic computing. *IBM White paper*.

Janert, P. (2013). *Feedback control for computer systems*. O'Reilly Media, Sebastopol, CA, USA.

Joint ACM/IEEE Task Group on Computer Engineering Curricula (2016). Curriculum guidelines for undergraduate degree programs in computer engineering. Technical report, Association for Computing Machinery and IEEE Computer Society.

Leva, A., Maggio, M., Papadopoulos, A., and Terraneo, F. (2013). *Control-based operating system design*. IET, London, UK.

Maggio, M. and Leva, A. (2011). Teaching to write control code. *IFAC Proceedings Volumes*, 44(1), 7292–7297.

Patikirikorala, T., Colman, A., Han, J., and Wang, L. (2012). A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Proc. 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 33–42. Zürich, Switzerland.