IFAC Journal of Systems and Control 8 (2019) 100051

Contents lists available at ScienceDirect

IFAC Journal of Systems and Control

journal homepage: www.elsevier.com/locate/ifacsc

PID control education for computer engineering students: A step to bridge a cultural gap

Alberto Leva

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Via Ponzio 34/5, 20133 Milano, Italy

ARTICLE INFO

Article history:
Received 11 June 2018
Received in revised form 14 January 2019
Accepted 22 March 2019
Available online 8 April 2019

Keywords:
Control education
Computer engineering
PID control
Control of computing systems
Control-based computing systems design

ABSTRACT

In computer engineering curricula, control is typically taught only to students willing to specialise in embedded systems, real-time, and the like. Nowadays, this is becoming a problem. Control-based techniques are gaining importance as a means to manage, optimise and also design computing systems. In such a *scenario*, a lack of control culture is critical. However, a computer engineering curriculum may not have the time and space to introduce a suitably tailored but "complete" course on the principles of systems and control. This paper proposes a solution, based on a PID-centred activity, where the occasion is taken to introduce and stress selected general ideas.

1. Introduction

In a 2005 white paper titled "An architectural blueprint for autonomic computing" (IBM, 2005), IBM defines "autonomic" as "pertaining to an on demand operating environment that responds *automatically* to problems, security threats, and system failures". The same reference [page 2] says "Self-managing capabilities in a system accomplish their functions by taking an appropriate action based on one or more situations that they sense in the environment. The function of any autonomic capability is a *control loop* that collects details from the system and acts accordingly". This loop is formalised [*ibidem*, figure 4] as the so-called MAPE(-K) one, the acronym standing for "Monitor, Analyse, Plan, Execute (based on Knowledge)" — not so different from a sequence like "read from sensor, compare to reference, compute control, write to actuator" (based on knowledge of a model for system, disturbances and requirements). Furthermore, "touchpoint autonomic managers" (MAPE-K loops with a "sensor" and an "effector" touching the controlled system directly, like e.g. a pressure or a flow loop in a plant) are managed by higher-level "orchestrating autonomic managers" that "configure" them, like for example a central MPC could give set points to peripheral PIDs, or a loop analyser could retune them. Indeed, the similarity of Figure 2 in IBM (2005) to a hierarchical plant-wide control system is quite impressing.

Given the above, it should be apparent that computer engineering students have to study control, but quite surprisingly, in practice this does not seem to be so apparent at all. Most computer engineering students do not receive any control education; and those who do, most often take their basic "systems and control" course in a late semester of their BSc, and only if interested in embedded systems, real time, and the like.

As such, many computer engineers need control culture to manage and design their systems, but do not possess it. Present technologies suffer heavily from this gap, as does part of the computer engineering research as well (some evidence is given in the following). In this paper, that builds on the preliminary one (Leva, 2013), the problem is analysed and a solution is proposed, by addressing the research questions below.

RQ1 Do computer engineering students really need control education independently of their specialisation?
RQ2 If so, what is the best pedagogical goal for them?
RQ3 If such a goal can be sketched, is there a means to turn it into a didactic activity effective and compact enough for being inserted in an existing curriculum with feasible effort?

2. Why any computer engineering student needs control education

As discussed more extensively in the following Section 3, computer engineering students either ignore the systems and control theory, or are just taught that computers serve to build controls, not that concepts like dynamics and feedback play a fundamental role in the management and the design of computing systems. The result is that when confronting a problem, even if this should be immediately recognised as a control one, they try to identify a set of cases deemed the most frequent ones, and based on

E-mail address: alberto.leva@polimi.it.

https://doi.org/10.1016/j.ifacsc.2019.100051
2468-6018/

intuition on how to handle these, to "figure out" an algorithm. For which some corner cases will then emerge, leading to another algorithm. For which some further corner cases will then emerge, leading to yet another algorithm. And so on, sometimes even with corner cases re-emerging because solving new ones has broken the solution for some of the older. The resulting mindset and professional attitude may easily prove inadequate to confront the complexity of many modern computing systems.

The reader legitimately suspecting that the last statements are just a consequence of the partial and control-centric viewpoint of the author, is invited to go through a nice recent paper from the computer community, titled "The Linux scheduler: a decade of wasted cores" (Lozi et al., 2016). The paper analyses the Completely Fair Scheduler, spots several logical flaws (at least some of which a model-based design would have avoided, the author bears to add) and lists among their causes that "catering to complexities of modern hardware, a simple scheduling policy resulted in a very complex bug-prone implementation" [Section 7].

In this and other works from the computer community, the perception appears that complexity has grown enough to require a systemic approach. Still missing, however, is a general enough conscience that the solution is to introduce control as a design mindset, not to just take some pre-built algorithms and connect them to a system without a model to reason upon. Besides nice applications, as a consequence, the computer literature also contains examples of how disastrous a distorted idea of control can be. A few such cases, compatibly with the educational focus of this paper, are touched below to support our affirmative answer to RQ1. The reader willing to further investigate can start e.g. from the papers Abdelzaher, Stankovic, Lu, Zhang, and Lu (2003) and Diao et al. (2005) – written respectively for the control and the computer communities – or the books (Hellerstein, Diao, Parekh, & Tilbury, 2004; Janert, 2013; Leva, Maggio, Papadopoulos, & Terraneo, 2013) and their bibliographies.

2.1. Case 1 – a taxonomy on control for self-adaptive software

Intended takeaway: in the absence of system-theoretical education, control ends up viewed as a bookshelf of algorithms rather than a methodological *corpus*.

The survey (Patikirikorala, Colman, Han, & Wang, 2012) on control engineering approaches for self-adaptive software discusses and classifies 161 papers, from 2001 till 2011. It excludes papers not "utilising control-theoretical approaches", hence also "fuzzy logic, neural networks, case based-reasoning and reinforcement learning" [p. 35], thus focusing on "classical" control. However it also excludes works on "hardware [...] or operating system level management" [*ibidem*]. This is surprising: to give just one example, the way a resource is allocated heavily depends on the internals of the operating system. When dealing with *functional* requirements (simplifying for brevity, *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*, e.g. how fast, the software has to do its job) lower layers necessarily come into play. If we were talking e.g. of process control, neglecting such layers would sound more or less like positively refusing to account for the dynamics of actuators—quite often, not a good idea.

Somehow surprising are also the taxonomy axes, i.e., "target system (application domain, performance variable, dimension), control system (model, type, loop dimension, scheme) and validation (simulation, case study)" [Figure 1]. No evidence of how the model structure is chosen, how that of the controller is consequently selected (including whether or not an adaptive one is worth the additional effort), and how the controller is tuned — all symptoms of disregarding the importance of modelling to structure a control strategy. And as for models, these are mostly black-box [Table III], with the definitely aprioristic motivation that "analytical" (we would say "physical" or "first-principle") ones are "not available or significantly complex" [p. 36]. Finally, the variety of control schemes in the classified papers is wide, from PID to LQR, MPC and more [Table III], but the ideas of control *law* and *scheme* are confused with one another — for example, two items in Table III are "LQR" and "cascade".

Despite its undoubted value, the work seems a taxonomy more of how algorithms were taken from a bookshelf and applied, than of what the encountered control problems look like, hence of how control schemes need structuring and tuning.

2.2. Case 2 – a combination of controllers

Intended takeaway: worse still, in the absence of the right education "a control algorithm is just an algorithm", and there is hardly any idea about how strongly it can impact the overall system.

AdaptGuard (Heo & Abdelzaher, 2009) was proposed to "guard systems from instability". The idea is that "adaptation loops" [the authors' term for feedback loops], "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". A "violation" is detected by inferring system causality from I/O measurements: dependency reversals among monitored variables indicate that some feedback has changed sign, causing instability.

In control terms, AdaptGuard is a decentralised state-based switching scheme. The reported examples work, but in the absence of a system-level stability analysis, which could be cumbersome even in the LTI case, the author bears to state that working examples basically mean that the controlled systems are tolerant indeed (which the authors somehow acknowledge by saying that "open-loop actions are stable"). Replacing a controller with another one is *not* the same as replacing a generic algorithm with "another one for the same purpose" like one could do e.g. for sorting. But to understand this, one needs some control theory.

2.3. Case 3 – mutual misunderstanding

Intended takeaway: more in general, in the absence of systems and control education, the ideas of "dynamics" and "dynamic model" are easily misunderstood, so that discussing properly about systems with control is hardly possible.

The survey (Huebscher & McCann, 2008) on autonomic computing focuses on "models" right from the title, but contains no "model" in the sense of a dynamic system drawn from hypotheses on an object to synthesise a control law for it. Sentences like "The model [not some variable] is updated through sensor data" [p. 7:11] denote not distinguishing systems and signals, which makes it difficult to reason properly about feedback loops.

In this respect, it is illuminant to report and annotate a paragraph in the conclusions [Section 8]. "State-flapping is a cross-cutting concern for autonomic computing. This is where oscillation occurs between states or policies [i.e., an oscillating signal is seen the same as a controller cycling among different laws] that potentially diminishes the optimal operation of the element that is being managed. [...] there are many techniques we can borrow from other sciences (e.g., control theory etc.) that can help with dampening and desensitising the adaptivity mechanisms. However, we do not believe that the likes of control theory is a *panacea* to solving all autonomic aspects as it is less able to deal

with systems that exhibit discrete or continuous behaviours or states [unclear whether talking about variables or equations, and whether "discrete" refers to time or value—very different theoretical settings] that can be time varying with less well-defined inputs in an ever-changing systems environment".

Now, such foundations can produce excellent applications. This is not under question. No community can claim any superiority. But apparently, the system/signal distinction, the role of disturbances and uncertainty, and above all the need for a unified formalism to describe all the elements of a control problem mathematically, are missing completely. Such cultural gaps between the computer and the control communities cause a lot of misunderstandings.

2.4. Case 4 – effects on an application

Intended takeaway: as a consequence, when coming to applications, in the absence of control culture one misses the conceptual tools to isolate the phenomena of interest, tailor the model complexity, and formulate consistent specifications on the involved components.

The Linux thermal daemon (Intel Corporation) is a software component to keep the temperature of a processor at a safe level while software introduces a continuously and rapidly varying disturbance by changing the power consumption. The problem is structurally simple: no complicated architectures, no hard-to-formalise requirements. Nonetheless, the daemon is made of thousands of code lines, and the description itself of its strategy (the only information one possesses for maintenance, in the absence of a dynamic control model) is complex indeed. In addition, the daemon uses different sensors and actuators, accessed through operating system modules not designed for control purposes, hence providing no timing guarantee.

Since the time scale of the controlled phenomenon is fast, one should have made timing a requisite for any component of the control system, instead of just taking the existent, and attempting to exploit it by adding complexity to the software. Quite expectedly, the result are temperature swings of tens of degrees, often requiring the hardware protection to intervene — to say nothing about what could happen if the maintainer of some involved operating system module, who knows nothing about its use in the daemon, decides to modify it. Just for comparison, a solution based on a PI with override produces excellent results with about 1/20 of the code, tight timing, clearly interpretable parameters, and formal guarantees verified on the real hardware (Leva, Terraneo, Giacomello, & Fornaciari, 2018).

2.5. Concluding remarks as for RQ1

In the computer community, basic systems and control theory is often just neglected, or overlooked, or applied with poor methodological attention. It is not a criticism. It is just recognising a matter of fact. The reader can verify this through many references other than the few above.

Also, especially in "adaptive" systems, the control theory is viewed as an *alternative* to e.g. machine learning or heuristics, not as a mindset to view problems. Since several computing-related problems require significant skills to cast them into a control paradigm, the conclusion is frequently that "the problem is too complex" for control theory to be applied. Only in the last years 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).

Indeed, the main problem is the lack of a shared culture. Control engineers should probably learn about computers more than they do (material for another paper symmetric to this one) but it is at least equally important that computer engineers learn about control. The examples just seen indicate that this has nothing to do with their specialisation, and also that the point is not to learn procedures but to acquire a mindset. Therefore, the answer to RQ1 is strongly affirmative. Any computer engineering student needs educating about control with the right degree of abstraction – i.e., in general and without delving in the technical details of any application – and properly — i.e., by a systems and control specialist. And the earlier this happens, the better.

3. Mainstream control education practice for computer engineers

Before proceeding, an analysis of the present state of the art is in order. In the apparent impossibility of reviewing all the existing curricula, we consider here as an authoritative source the ACM and the IEEE Computer Society curricula recommendations, available online at ACM and IEEE Computer Society, respectively.

For brevity, we here examine only the joint ACM–IEEE curriculum guidelines for undergraduate degree programs in computer engineering (ACM curriculum). The executive summary says "computer engineering is a discipline that embodies the science and technology of design, construction, implementation, and maintenance of software and hardware components of modern computing systems and computer-controlled equipment". The "background" Section 2.1 mentions control when talking about computer-controlled equipment, and embedded – possibly control-targeted – applications. The "evolution of the field" Section 2.2 envisages "a re-integration with electrical engineering, as computer-based systems become dominant in areas such as control systems and telecommunications". The "breadth of knowledge" section 2.3.4 locates control systems in the "electrical engineering related" coursework.

Coming to knowledge areas, those directly related to systems and control are CE-ESY (Embedded systems, 40 core hours out of 420 total) in unit CE-ESY-8 (Data acquisition, control, sensors, actuators, 4 core hours) and CE-SGP (Signal processing, 30 core hours) in units CE-SGP-6 through 8 (on signal processing – no loop is closed – for a total of 26 core hours) and CE-SGP-11 (Control system theory and applications, no core hours).

Modelling and simulation is considered, but almost exclusively for circuits and devices. For example, CE-CAE (Circuits and electronic, 50 core hours) embodies the supplementary unit CE-CAE-12 (Circuit modelling and simulation methods), CE-DIG (Digital design, 50 core hours) has CE-DIG-2 (Relevant tools, standards, and/or engineering constraints, 2 core hours) and CE-SPE (Systems and project engineering, 35 core hours) has CE-SPE-2 (analogous to CE-DIG-2, 3 core hours). More in general, section 4.5 ("the role of engineering tools") locates simulation within "hardware design tools". Simulation-related topics are also in CE-SRM (System resource management, 20 core hours) in supplementary unit CE-SRM-7 (System performance evaluation).

For a more complete *panorama*, one could of course also observe typical/suggested study plans at university websites. We do not enter such an analysis here because it would be too long, and the choice of the universities inevitably arbitrary. While encouraging the reader to try his/her own, therefore, we just say that going through several sites confirmed the impression gathered from the recommendations above, and the others at ACM and IEEE Computer Society. The conclusion is that with the present practice, when computer engineering students receive systems and control education, this is centred on using computers to make controllers. Hardly any mention is made of computing systems control—let alone of control-based computing systems design. Moreover, taking control courses in late semesters further

strengthens the idea of a unidirectional computers-control relationship: first computers and then control, but only if interested in embedded systems and the like; because computers serve for control, not *vice versa*.

4. Which control pedagogy for computer engineers

We established that any computer engineering student needs "the core ideas of control", that current education practice can be improved in this respect, and that "the earlier the better" (hence in this work we concentrate on the undergraduate level). It is now the time for RQ2, that can be split in two sub-questions. First, do we need a specific pedagogy for computer engineering students, or introducing a "standard" basic control course, maybe just suitably abridged, would suffice? In other words, are computer engineering students somehow "special" – and in the affirmative case, why – with respect e.g. to mechanical, electrical and others? Second, based also on the considerations for the previous sub-question, what are the core ideas to transmit?

4.1. Why computer engineering students are "special"

The roots of the question are historical. When control became a necessity in any engineering domain but the computer one, the required mathematical theory was already there, and developing a control engineering culture was quite straightforward. Nowadays, non-computer engineers get naturally acquainted to using a dynamic system to model an object, simulating such a model to aid design, setting up control if needed, and assessing the whole thing. The technological evolution progressively brought electronics and digital systems into play, adding tools and requiring to extend the theory, but the ultimate foundations and mindset are still the same.

For computer engineering, the story is different. The dawn of computers and of the discrete-time systems theory more or less coincide. As no established theory was ready for use, the computer community developed their own theory and system design mindset, which differ a lot from those of the control community. Instead of using models made of equations, computer engineers got to describe functional entities as algorithms and data structures, or state machines and data paths. Instead of the descriptive and possibly a-causal framework of dynamic systems, they fundamentally adopted the causal and prescriptive one of algorithms. The reader may object to this statement based e.g. on the existence of temporal logic and model checking, and in some sense he/she would be right. However these are advanced topics, taught after the undergraduate level, still largely subject of research, and – apologies for being *tranchant* – less familiar to the average computer professional than dynamics and feedback are to the control one.

In any case, the cultural divergence above – consolidated across half a century – makes today's computer engineering students face a special difficulty. For the others, the concepts learnt in the basic control course do find application in subsequent ones, for example on process or motion control. There are textbooks on e.g. chemical plants or robotics, that are stuffed with control. In one word, when they exit their basic control class, non-computer engineering students encounter a reasonably control-friendly world.

For computer engineering ones, this is not the case. Assuming they receive systems and control theory education properly and as early as possible, in the sequel of their studies they will mostly see design practices involving no control in the sense we mean here, and they will need to relate these practices to their control culture autonomously. This is definitely a less friendly setting, quite likely to make the students rapidly relapse into illiteracy about control, or develop no self-defence against distorted control ideas like those discussed above.

Summing up, as long as the complexity of computing systems allowed to design them without control in mind, teaching control only to students willing to use computers to make controls was fine, and so was for them a "standard" basic control course. Nowadays this is not true anymore. As recognised by several of the quoted works, the complexity of modern system requires a mentality change. To induce this change we necessarily have to start with students, and we must pay special attention to help their control culture withstand the persistence of the old mentality and co-exist with it. Clearly, this requires *ad hoc* teaching.

4.2. Choosing the core ideas to transmit

To strengthen the students enough for the setting just sketched, the ideas to teach must be as general as possible with respect to any application, natural to grasp and to understand firmly, and as mathematically light as possible, to be easily retained and kept alive even when technical details fade away.

Clues for the choice come from the attitudes to promote or prevent. Based on the cultural issues discussed in the previous sections, a brief list of requirements – not exhaustive but enough for our purposes – can be the following.

- When confronted with a (control) problem, the students should avoid to *first* come up with a solution and only *afterwards* possibly discuss its properties.
- They should understand the concept of "property" as something that can be checked formally on a model, before any implementation. Besides for control, this will also help them learn e.g. about temporal logic predicates later on in their studies — and the author adds, when they have been writing programs for so long that in the absence of prior modelling culture, integrating formal design into their habits may not be an easy task.
- They should avoid limiting discussions to "use cases", as already noted.
- When the problem is to reach a goal, they should refrain from seeking the "magic move" to get there in one step, as this entails the extremely questionable assumption that if the goal is missed and another move is required, aiming constantly at the goal will result in the shortest path.

Also, given the students' tendency to jump directly to the algorithm, the peculiarities of control in computers should be discussed with extreme care, and not at the beginning of the activity. The risk is that instead of clarifying the *scenario* as would happen with a more educated audience, such discussions conversely induce the idea that control concepts (not the way they are applied, notice) need "customising" for computing systems, which is exactly the opposite of the educational goal to be attained.

At the Politecnico di Milano, the author teaches a course titled "Fundamentals of Automatic Control" to sophomore students in computer engineering. This is not a standard situation at all. The course has 65 h of lecture, 35 of classroom practice, and two experimental laboratory sessions of 3 h each: far more than usual. The lecture hours distribution for the course subjects is shown in Fig. 1. The part devoted to PID is small, but the amount of hours for general "control synthesis" explains this.

Despite its peculiarity, the course provides an interesting probe. Several students show up some years later for a MSc thesis. These are occasions to first see what they retained of the subject, and then – while working together – observe how they apply

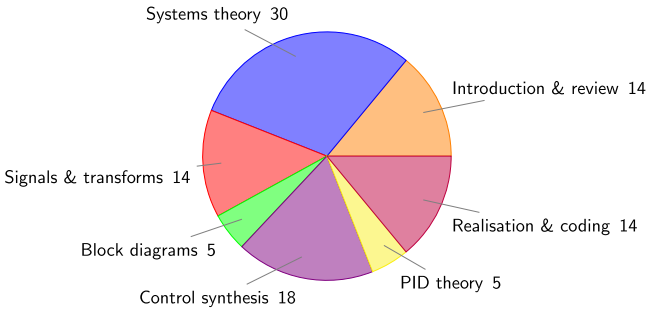

Fig. 1. Hours distribution in the "Fundamentals of Automatic Control" course.

control ideas, what are their strengths and weaknesses, and what is felt as (and in fact turns out to be) "fundamental".

This experience, over more than 15 years, indicates that the really important things are the *general* ideas of dynamics, feedback, set point tracking and disturbance rejection, (internal) stability, performance, and the relative indices. The last point is particularly important because it allows to compare solutions formally, which is inherently preferable to just using "representative" benchmarks.

Concentrating on the matter just evidenced is a good enough basis, and limiting the scope almost exclusively 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.

All this said, the problem remains that in the typical computer engineering curriculum, negotiating the necessary space can be very difficult. Fortunately over the last years there is a growing research interest for the subject, and therefore the argument that "education should eventually start following" may have some appeal. In the rest of the paper we shall assume that some space is available, and address the problem of making the most effective use of this space according to the considerations above. And as discussed in the next section, this is where PID control comes into play.

5. Why centre the activity on PID control

We now come to RQ3. The motivations for centring the activity on PID – actually, mostly PI – control, is twofold. On one hand, it allows to introduce general ideas quite straightforwardly, as is well known to any reader with expertise in control. On the other hand, it fits a large number of problems that arise in computing systems, and involve extremely simple models. We give some details on this second aspect.

In the presentation of the keynote paper (Hägglund, 2012), the author suggested that in modern systems, PIDs relate to the overall control application like ants do to their colony. This can be re-formulated, for the educational scope of this paper, by saying that PIDs act locally and near to the physics of the controlled system, while the quality of their operation appears at higher levels of the control hierarchy. The reader may object that this applies to *any* control hierarchy, not just to computing systems. True, but computers have at least two relevant peculiarities.

First, in any domain but computing, there is a physically defined level below which neither measurements are feasible, nor actions possible. To give a deliberately extreme example, temperature is governed by molecular motion, but one cannot think of measuring and actuating at that scale. In computing systems, the same is not so true. For example, service level agreements are established on an average basis, such as a waiting time below

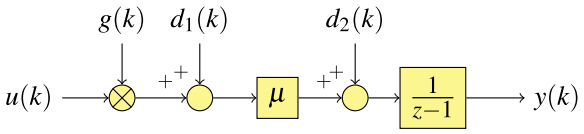

Fig. 2. A simple model fitting several computing-related problems.

100 ms for 99% of the requests to a server; however, not only one *can* think of acting at the level of "molecules" – the individual requests – but in several cases, this is exactly what one *must* do.

Second, outside computing, lower hierarchy levels generally correspond to less arbitrary choices on measurement and actuation. For example, in a "peripheral" pressure loop the choice of sensor and actuator is mostly a matter of technology, and there are established guidelines for it; on the contrary, "central" controls may deal with "product quality" or other management-oriented indicators to be *computed* from measurements, and that can be defined in conceptually different ways. Here too, in computing systems, things are often not so clear. For example, measuring a server's throughput is influenced by the adopted time interval, and indices like the residence time in a queue need discussing right from their definition, if (as is frequently the case) input and output rates vary continuously, so that the system is never at steady state.

Fortunately, the computing domain also brings some good news. A major one is that the deeper one digs into the system, the simpler the dynamic models for the observed phenomena tend to become. As such, great benefits come from endowing systems with a hierarchically "low" layer of PI- or PID-based controls, to mitigate the effects of exogenous disturbances on quite simple dynamics. We now support the ideas above by analysing two examples.

5.1. Integrator-based models

For the first example we consider the discrete-time system

$$y(k) = y(k-1) + \mu\,(g(k-1)u(k-1) + d_1(k-1)) + d_2(k-1), \quad (1)$$

where $u(k)$ is the control signal, $y(k)$ the controlled variable, $g(k)$ a multiplicative disturbance (also interpretable as a time-varying gain), and $d_1(k)$, $d_2(k)$ two additive ones. Since $g(k)$ can frequently be viewed as a variable "efficiency", as discussed below, we assume

$$0 < g_{min} \leq g(k) \leq 1 \qquad \forall k. \quad (2)$$

The system (1), represented as block diagram in Fig. 2 can be viewed at two levels of complexity. Assuming $g(k)$ to be constant – i.e. unitary, given the presence of μ – yields an LTI system, while admitting a variable $g(k)$, and interpreting this as a time-varying gain, gives an LPV one. Some examples of the various interpretations this system can be given, at both levels, are listed below.

1. *LTI — uniprocessor preemptive scheduling:* $g(k) = 1$, $d_2(k) = 0$, $\mu = 1$.
   The processor time $y(k)$ accumulated by a task at the kth intervention of a scheduler equals the time $y(k-1)$ accumulated before that intervention, plus the time $u(k-1)$ allotted by the scheduler, plus the effect $d_1(k-1)$ of any phenomenon making the actually used time to differ from the allotted one, like the task yielding back the processor, or the preemption interrupt undergoing a latency delay.

2. *LTI — master–slave network clock synchronisation:* $g(k) = 1$, $d_2(k) = 0$, $\mu = 1$.
   The error $y(k)$ between the clock of each ("slave") node in a network and a master one at the kth synchronisation event equals the error $y(k-1)$ after the previous such event, plus the integral $d_1(k-1)$ of the slave-to-master clock skew (i.e., the inverse of their normalised frequency difference) over the time elapsed between the two events, minus a correction $u(k-1)$ computed at the previous event.
3. *LPV — batch data processing:* $d_1(k) = 0$, $d_2(k) = 0$.
   The amount $y(k)$ of data processed by a batch task at the kth intervention of a resource allocator equals the one $y(k-1)$ at the previous intervention, plus the data processed between the two. This amount of data nominally equals the inter-intervention period times a nominal resource-to-processing-speed gain μ, times the allotted resource amount $u(k-1)$. However the actually processed quantity of data equals the nominal one above multiplied by the time-varying "efficiency" $g(k-1)$. This efficiency, among other effects, accounts for the fact that different data may stress the allotted resources in a different and *a priori* unpredictable manner.
4. *LPV — queue-based services:* $d_1(k) = 0$.
   The length $y(k)$ of a queue at the kth intervention of a resource manager allotting computational power to its server equals the length $y(k-1)$ at the previous intervention plus the amount $d_2(k-1)$ of jobs arrived between the two, minus the jobs processed in the same time span, which equals the computational power $u(k-1)$ allotted at the beginning of the span times a nominal gain μ representing the maximum server speed (and negative as the server *removes* jobs from the queue) multiplied by the time-varying – or equivalently, data-dependent – efficiency $g(k-1)$.

As long as the control action is computed in the discrete time – a hypothesis to which a first didactic activity on control can well stick – the few examples above should convince that the equation "work done up to now equals work done up to the last step, plus the work nominally accomplished by resources allotted at that step times a possibly varying efficiency, plus the effect of additive disturbances", conveniently re-phrased, fits a large number of physically heterogeneous applications. Adopting this modelling attitude, the *methodological* differences among those applications reside in the presence or absence of disturbances, in the characteristics of those disturbances as signals, and in the aspect of the reference signal. Then there are also technological differences owing to the nature of the involved objects, of course, but these do not affect the modelling and control synthesis activity.

5.2. Low order asymptotically stable models

Sometimes the control action cannot be exerted directly on the physical phenomenon to govern, but has to traverse some management machinery. Our second example refers to the quite frequent case in which a resource is *requested* to an allocator, that enforces the request (if feasible) with some internal control, hence some dynamics.

A notable example is shown in Fig. 3, where all the responses come from models up to the second order and with at most one zero. If the system input is the allotted resource and the output an application performance metrics, the four responses in Fig. 3, top to bottom, could be interpreted as follows.

A. The resource is acquired and exerts immediately its full effect (that is therefore seen at the very next step).
B. The resource acts immediately but takes some steps to yield its full effect (for example because a queue needs emptying).

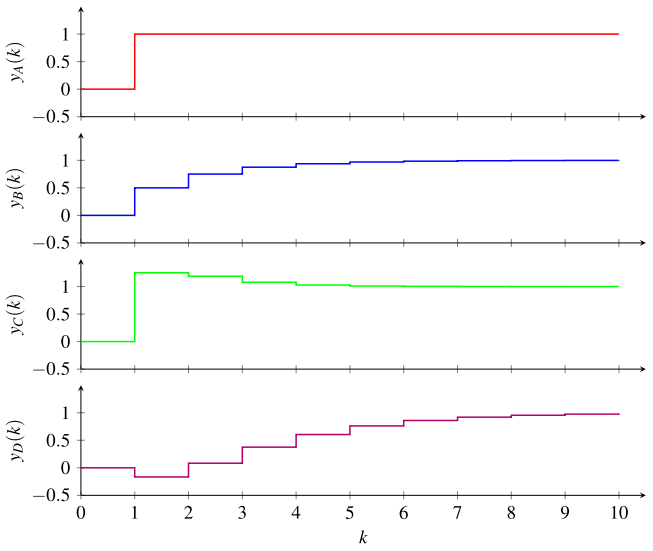

Fig. 3. Example of step responses from dynamics suitable for PI/PID control, for model interpretation.

C. The resource produces a transiently enhanced effect (this is quite typical when the metrics is the *speed* toward a goal).
D. The resource requires some effort to be acquired and thus initially reduces performance, like e.g. a new core that when acquired has a cache with unknown content and thus initially makes a lot of cache misses.

Based on a vast experience and literature (Hellerstein et al., 2004; Janert, 2013; Leva et al., 2013), in most computer-related cases of practical interest, the core phenomena are described well enough by an asymptotically stable model of very low (in general, first or second) order. The aptitude of PI- or PID-based control schemes to a vast set of cases is therefore quite apparent, whence the opportunity of centring the activity on it.

6. Didactic activity

This section gives a sketch of the activity, to illustrate how the ideas previously set forth can be applied. The instructor who agrees with the overall picture, can easily introduce the adaptations deemed convenient for his/her particular teaching context.

In the sketch we suppose to start completely from scratch if not for basic algebra, calculus and procedural programming, which seems a reasonable outset. In such a situation, the activity can span about 16 h. Teaching material, targeted to approximately 8 h of lectures and 8 of classroom practice, can be downloaded at https://github.com/albertoleva/PID4CSE under a Creative Commons licence; LaTeX source files are included, so that the interested teacher can easily adapt the material to his/her needs, insert it in a larger course, and so forth.

Needless to say, the following treatise is based on the author's experience, thus no absolute truth is claimed, and if the 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. Table 1 summarises the activity structure, based on which a few unit-by-unit clues and *caveats*, plus some general remarks, are given.

Table 1
Outline of the activity; L stands for lecture, P for classroom practice. The mentioned software tools are discussed in Section 7.

Unit	Hours	L/P	Content and expected outcome
1	1–2	L	Know and use control terms correctly; understand the idea of state as a reason for a system to react differently to the same input; be aware of the major lexical differences between the computer and the control communities, but at the same time of the presence of many control problems inside computing systems; understand clearly the motivations for the activity.
2	3–4	L	Firmly grasp the general idea of feedback, independently of the mathematical form it can take; be familiar with the main types of dynamic systems; specialising to the discrete time (DT) LTI case, know about the state space and the transfer function representations, about stability and how to check it, and about "hidden parts" due to cancellations.
3	5–6	P	Practice with DT LTI systems and their evolution, relating the various representations to one another; view feedback as a mechanism in nature; understand the meaning of hidden parts in a system; start getting acquainted to block diagrams; familiarise with the wxMaxima tool for symbolic computations.
4	7–8	L	Study the response of simple systems to standard signals; see how many different responses can come from a single model structure; learn to translate time domain requirements into a desired aspect for some transfer function; get familiar with the control loop and its actors; synthesise a controller by cancellation paying attention to all the implications; reason about the system-theoretical view on a control loop as opposite – but necessarily related – to the functional/algorithmic one.
5	9–10	P	Practice with system responses and direct synthesis; introduce the PI law, both intuitively and the control-theoretical way, with examples; abstracting from this experience, re-discuss the algorithmic and the systemic view on a control loop and spot the possible pitfalls of the former as for fully understanding its operation; familiarise with Scilab and OpenModelica.
6	11–12	L	Introduce the PID law and discuss parameter tuning, both in the pure DT and the sampled signals case; present some examples in the computer domain for both situations, reducing continuous-time entities to the bare necessary; discuss windup and antiwindup; write and run a complete PI algorithm in Modelica; give a PID algorithm to the students to examine as a means for them to verify their comprehension.
7	13–14	P	By using wxMaxima and OpenModelica, experiment with simulated PID loops; look at the P, I and D actions and understand their role; concentrate on the second order case to illustrate the main reasons why the PID law is so vastly applicable; experiment with variations on the PI(D) algorithm (positional vs. incremental, different antiwindup methods) to appreciate the importance of a knowledgeable and completely documented implementation.
8	15–16	P	Present some case studies from past experience in the computer domain; show also some cases (e.g., tight clock synchronisation) where a PID is not appropriate, and explain why; give some preliminary ideas about using a PID in the context of a control structure (e.g., an override structure for CPU thermal management); sketch out possible subjects for further control-related studies.

6.1. Unit 1

Unit outline: prerequisites; definitions; terminology and first examples; a minimal control taxonomy; introduction to dynamic systems.

Defining terms is very important, as false friends are ubiquitous. For example, in systems theory a "parameter" physically characterises a system, and is not a variable. In computer engineering "parameter" is understood as in "formal" and "actual parameter" for a function — i.e., it is the computer-ese for "input". References from the computer literature help make the students aware of the "control in/for computers" research and technology domain. Examples help as well but should positively be confined to the same awareness purpose, or in a few minutes the class is lost into irrelevant technological details.

The instructor should speak the language that the students will encounter later on when learning/reading about self-adaptive or autonomic systems and the like, and make this language choice explicit. Contrary to other students, computer engineering ones necessarily need to relate system-theoretical and algorithmic entities, and as long as they can cast ideas into their mindset without misinterpretations, explanations are facilitated. For example, a helpful "control taxonomy" for them could sound like

```
Controller:
   type                      = {modulating,logic}
   timing                    = {continuous,discrete,
                                event_triggered}
   connection_with_system    = {open_loop,closed_loop}
   disturbance_compensation = {present,absent}
```

6.2. Unit 2

Unit outline: benefits of feedback; dynamic systems and their properties, specialised to the DT LTI case.

The students must clearly distinguish the computer and the control sense of "loop". To appreciate the problem, after introducing feedback one can talk about the MAPE-K loop, then ask the audience what the word "loop" in that formalism means, and see how many understand the system-theoretical nature of the concept instead of sticking to its meaning in program flow control, i.e., a cyclically repeated set of operations. More in general, care must be taken to counteract the risk of mixing up feedback and iterative computations, as well as block and flow diagrams.

As for dynamics, take care to call the state in as many ways as possible like "present condition", "internal configuration", and so on. This will help the students 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, we may add) be re-formulated simply as that system being dynamic.

On stability, a fast way to say just the necessary, is to analyse the first order case and then view higher order systems as the series/parallel of first order ones; the role of the poles is then immediately evident. Systems with complex poles (not frequent in computers) can be left as an exercise.

6.3. Unit 3

Unit outline: exercises on dynamic systems, their representations and their properties.

Through the various examples addressed, care should above all be taken that the students understand the concept of structural property (e.g., stability) as a means to draw general conclusions on the behaviour of a system without any attempt to achieve "exhaustive" experimenting. Recall that in computer science "model checking" refers to techniques to verify predicates on a system (over-simplifying for brevity) and is normally extremely complex and computationally intensive. The idea itself of "model" that one needs for control is different from the mainstream computer engineering mindset, and every occasion to deal with his issue should be taken.

6.4. Unit 4

Unit outline: system responses; formalising requirements; the control loop; direct synthesis.

A major goal is to show how many different "use case responses" can 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 some of its parameters. As an example of the possible 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 a supervisor deciding which one to activate, needless to say, and possibly learning from experience) – but just a direction and a speed control receiving the convenient set points. Such attitudes need preventing as early as possible.

A second goal, once the idea of "simple, *control-targeted* model" is grasped, is to show how requirements in the time domain can be translated into a desired aspect of some transfer function, and then (via direct synthesis) how a control law can be obtained non-ambiguously from model and specifications instead of being "figured out". Once again, in few other places may the students see an algorithm arise from a dynamic model. According to experience, at this point computer-related examples are safe enough and help stimulate interest, while the necessary formalisms are normally not critical (e.g., block diagrams are self-explanatory enough).

6.5. Unit 5

Unit outline: exercises on system responses and direct synthesis; the PI law, intuitively and formally.

The most important results to achieve here are first a firm relationship between intuition and control theory, and then – but not less important – the conscience that theory allows to go beyond intuition. Once again, accepting the risk to become boring, the author would like to stress that the students, outside control courses, will frequently hear that "the model is the code", and need to attribute to such statements the correct context and validity limits.

6.6. Unit 6

Unit outline: parameter tuning; the PI law; the control algorithm.

All the treatise refers to discrete-time control. However a very important point is to distinguish pure DT cases (i.e., when the process model is independent of the sampling time) from sampled signals cases, where the same is not true. According to experience this is a difficult subject but also a major source of confusion when controllers are realised in computing systems. Addressing the matter inevitably requires to bring into play a few continuous-time entities, but limiting the scope to Euler-based derivative approximation suffices. Other algorithm-related facts are easier to grasp, with the notable exception of taking care to record, from one step to the following, only the necessary quantities. Stressing that the control algorithm itself is a dynamic system, the above quantities being its state variables, normally helps bridging the two views already mentioned several times.

6.7. Unit 7

Unit outline: exercises on tuning PI(D) controllers and running their algorithms.

The importance of tuning is obvious. Less obvious but equally strong is that of coding, however. In computers there is no control design environment like e.g. those for process control. One has *very* frequently to write control code by hand, in various languages, and integrating with the code of the system hosting the control (e.g., a server written in java). Notice that the network (a primary source of information for students, especially for "code snippets") is often a bad teacher, as for example it provides many applications where not even antiwindup is in place; see Maggio and Leva (2011) for a deeper discussion on this aspect. Also, the students should become aware that nonlinear mechanisms like antiwindup can be realised in different ways, which impact the large signals operation of a system.

6.8. Unit 8

Unit outline: exercises on tuning PI(D) controllers and running their algorithms.

Besides showing the PI(D) applicability limits and carrying out some further exercises, this last session should serve two additional purposes. First, clarify that a controller is part of a system, and quite often its improper setup might cause symptoms visible "elsewhere". Second, make the students aware of further issues – to mention one, robustness – to study. According again to experience, it is a good idea to end the activity by asking the students to formulate some problem of their own interest, and possibly – especially if some computer engineering colleague is willing to cooperate – make this a joint project.

6.9. Some general remarks

Although the organisation above is largely indicative, the proposed activity should inherently be capable of addressing the peculiar issues evidenced in Section 4.2. 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, also sampled signals, for low order asymptotically stable or integrating processes. This is enough in many computing-related cases. Finally, they should be able of detecting that a problem is not tractable with the control tools they learnt, based on the dynamic characteristics of that problem, for example as stemming from process responses—an abstraction capability that only control can teach, and is very precious in many situations. For any detail that could not fit herein, the interested reader can refer to the teaching material made available.

The basic activity here shown could be completed on one side with standard advanced control courses, and on the other side with shorter, specialised modules to be integrated in subsequent computer-centric course. Modules at present under consideration concern task scheduling, memory allocation, shared resource management, task migration, application progress control, synchronisation, adaptive queuing systems, performance- or reliability-driven binding, and combined power/performance /thermal management.

7. A possible suite of supporting software tools

Any control course needs software tools. Here we broadly divide these into "in-class" and "off-class" ones. In-class tools are used by the instructor during lectures. Some hours may be devoted explicitly to the tools, but in general, just watching the instructor use them is enough to start autonomous learning. The most famous such tool is probably MATLAB (The Mathworks).

Off-class tools are meant for autonomous student use. Their story goes from pioneering works like (Johansson, Gafvert, & Astrom, 1998; Mansour & Schaufelberger, 1989) to modern, effective proposals such as Guzmán, Costa-Castello, Dormido, and Berenguel (2016), open source applications for embedded-ready hardware (Hoyo, Guzmán, Moreno, & Berenguel, 2015), or "learning modules" like in Guzmán, Åström, Dormido, Hägglund, and Piguet (2006). Such tools are normally subject-specific (e.g., on PID control), present a user interface beyond which the student is not expected to look, and are built onto various platforms, up to spreadsheets (Aliane, 2010). A good survey on this *scenario* is (Rossiter et al., 2018), but a quite ubiquitous fact, is that the students need some time to familiarise with the tool.

In this respect, the presented activity has two peculiar problems: time is severely limited, and we talk to computer engineering students — not control ones, whom the tools above are mostly for. Hence devoting time to learn any tool is infeasible, and any tool requiring a significantly nonzero prior knowledge of control does not fit.

A solution might be to use only tools and languages that computer students already know. This is good for coding (an important task, as said) but not for designing and analysing a system, in a view to also teaching the idea of employing the right tool for the right purpose—another important and sometimes overlooked educational goal.

Briefly, the set of tools should comprise both "computer" and "control" ones. For the former type, more or less any language fits. For the latter, we jump to the author's conclusions skipping the story that led there. The suggested suite is composed of Scilab (The Scilab Consortium), Maxima plus wxMaxima (Maxima; wxMaxima), and the Modelica (The Modelica Association) translator OpenModelica (The OpenModelica Consortium).

Scilab is useful for defining and analysing dynamic systems, and has an internal programming language suitable for writing small simulators where the control code is replicated exactly and also the controlled system needs representing as code. In Scilab there is only programming (not *modelling* like in Modelica, for instance), thus for computer engineering students the tool is easy to master.

Maxima, with the wxMaxima frontend for a more comfortable use, provides symbolic computation capabilities, about which – at least in the author's experience – many undergraduate students often have simply no idea. Such a tool is useful to enforce the concept that computations can be checked formally and guaranteed correct. The tool can also provide control code without any mistake. For example, the direct synthesis of a controller can be done with the script below, where first the reference-to-output dynamics is prescribed, then a specific case is shown, and finally asymptotic disturbance rejection is checked.

```
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));
Gyd: factor(subst([P=2/(z-0.5),c=C1],P/(1+c*P)));
```

Modelica has here two roles. First, the Blocks package of the Modelica Standard Library is suited to work with block diagrams. Second and most important, the availability of a DAE solver capable of handling events relieves the modeller from the burden of managing time. As a simple example, the model below represents the round-robin scheduling of three periodic tasks with a given workload; the keyword `der` stands for derivative with time, the meaning of the symbol `time` is obvious.

```
model RR
  parameter Real[:]       periods   = {1, 2.2, 0.7};
  parameter Real[:]       workloads = {0.2, 0.3, 0.1};
  parameter Real          quantum   = 0.01;
  final parameter Integer n         = size(periods, 1);

  Real    cpu_times[n] (each start = 0);
  Real    last_act[n]  (each start = 0);
  Integer invoked      (start = 1);
  Boolean running[n]   (each start=false);
  Integer misses[n]    (each start=0);

equation // *** PHYSICS ***
  for i in 1:n loop
    // Task runs if selected by scheduler and still has
    // work to do in period, otherwise yields CPU
    running[i] = (i == invoked
                  and cpu_times[i]<workloads[i]);
    // Time accumulated if running
    der(cpu_times[i]) = if running[i] then 1 else 0;
  end for;

algorithm  // *** CONTROL ***
  // Round robin task selection at each quantum
  // (we neglect context switching delay)
  when sample(0.0, quantum) then
    invoked := invoked + 1;
    if invoked > n then
      invoked := 1;
    end if;
  end when;
  for i in 1:n loop
    // Reset time counter at the end of the period, detect
    // and count deadline misses (workload not completed
    // when period ends)
    when time - last_act[i] >= periods[i] then
      if cpu_times[i]<workloads[i] then
        misses[i]:=misses[i]+1;
     end if;
      last_act[i] := time;
      reinit(cpu_times[i], 0.0);
    end when;
  end for;
end RR;
```

This short model clearly separates "physics" and "control", and allows for a wide set of experiments and additions. There is much more to Modelica than the couple of features just mentioned, but these are enough to motivate its use — and for the curious student, to possibly start a path toward a deeper understanding

Table 2
Data from the evaluation questionnaires for fundamentals of automatic control.

	2012/2013		2013/2014		2014/2015		2015/2016		2016/2017		2017/2018	
	C	S	C	S	C	S	C	S	C	S	C	S
Q1	3.28	3.16	3.35	3.20	3.26	3.22	3.48	3.24	3.23	3.24	3.02	3.25
Q2	3.23	3.00	2.89	2.95	3.03	2.99	3.23	3.02	2.89	3.02	2.93	3.05
Q3	3.53	2.95	3.34	3.00	3.44	3.03	3.63	3.04	3.38	3.06	3.37	3.08
Q4	3.18	3.04	3.24	3.06	3.27	3.09	3.47	3.11	3.11	3.11	3.02	3.13
Q5	3.62	2.98	3.56	3.08	3.66	3.10	3.82	3.12	3.61	3.11	3.53	3.13
Q6	3.55	3.12	3.57	3.12	3.67	3.14	3.77	3.16	3.60	3.15	3.53	3.17
Q7	3.14	3.10	3.12	3.19	3.33	3.23	3.56	3.27	3.45	3.27	3.45	3.31
Q8	3.50	3.01	3.41	3.06	3.51	3.08	3.70	3.10	3.46	3.09	3.39	3.11

C Average result for the course: individual answers are integers from 1 (worst) to 4 (best).
S Average result for all courses in the school.
Q1 Rate your interest in the contents of this course.
Q2 Was knowledge obtained from previous courses adequate to effectively attend this one?
Q3 Was the required effort consistent with the course credits?
Q4 Rate the quality of the didactic material.
Q5 Was the teaching activity motivating?
Q6 Rate the clarity of lectures.
Q7 Rate the effectiveness and usefulness of classroom practice hours.
Q8 Rate your overall satisfaction with this course.

of dynamic modelling. Also, as a further remark on the above computer/control tools balance, though the matter does not fit in this paper, one can couple a Modelica model to a controller written in C.

The tools just listed are the same the author uses in Fundamentals of Automatic Control. According to years of experience, computer engineering students do become capable of employing them by observing the instructor in the class: the same can be assumed to apply to the activity presented herein. Finally, all the tools are free software, with apparent advantages in terms of experience sharing and dissemination.

8. Evaluation of the proposed approach

The proposed activity and material is the result of a long process. At present, an evaluation is only possible based on Fundamentals of Automatic Control, where the computer-targeted approach that grounds the activity – integrated in the wider context of the course – was first introduced in the academic year 2015/2016. Figs. 4 and 5 report respectively the pass/fail distribution and the score breakdown in the A–D range, three years before and after the said introduction. Table 2 reports an excerpt of the questionnaires that the students at the Politecnico di Milano have to fill in at the end of each course.

The score data were taken at the first exam session. Subsequent ones have smaller but more varying numerosities, and tend to contain more students who attended the lectures in the previous years, thus being less reliable for our purposes. Also, it must be noted that in 2015/2016 the BSc course in telecommunications engineering was merged into the computer engineering one, causing a variation in the (sophomore) course population in 2016/2017.

Briefly, the reported results suggest that taking the approach here proposed, does not significantly alter the students' perception of the subject, while the exam results show a visible improvement. This, it is worth repeating, is however an evaluation for the *origin* of the activity here proposed. It is the hope of the author that the said activity and material, distilled from the experience just summarised, can have an analogous beneficial effect.

As for the reception of the proposal by computer engineering colleagues, to date information comes from personal interactions and there is no structured data yet for a systematic evaluation. As a result of the said interactions the author was also invited to give some tutorials on the subject, which is a further signal of interest and positive attitude. All the gathered experience is now contained in the proposed teaching material.

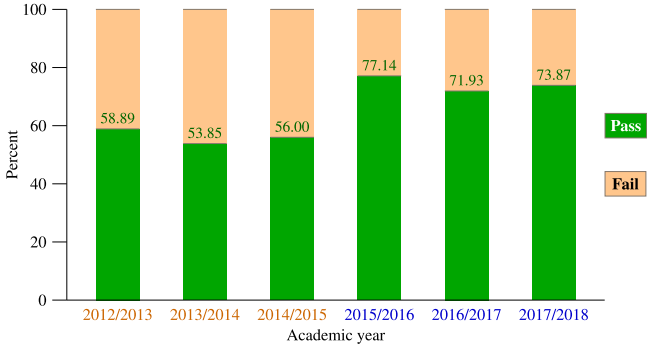

Fig. 4. Results in Fundamentals of Automatic Control — pass/fail.

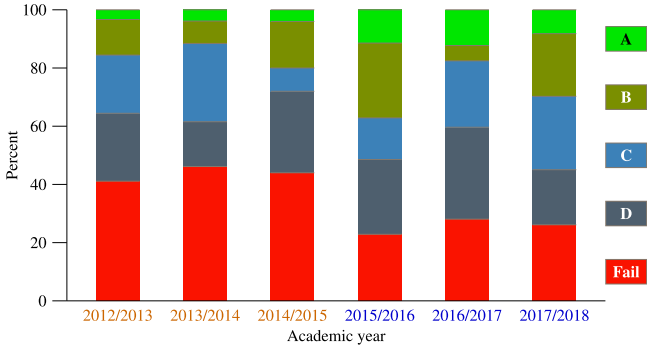

Fig. 5. Results in Fundamentals of Automatic Control — score breakdown.

9. Conclusions and future work

We discussed an important educational issue for computer engineering students, namely the necessity of learning the basics of the systems and control theory. The analysis strongly recommends that such an educational goal be addressed as early as possible, and by control specialists in a strict coordination with computer-centred colleagues.

As a result we devised and motivated a didactic activity, of size hopefully compatible with being inserted in a computer engineering curriculum and proposed to *all* the students. The activity is centred on PID control, but with the underlying *rationale* to provide the students with firm and clear ideas about

what control is, i.e., with a control *culture*. A possible suite of supporting software tools was also proposed, and motivated.

If there is more room than envisaged herein, the activity could be extended with subsequent modules addressing specific aspects, to be possibly connected to courses about software engineering, operating systems and so on.

Material for the interested teacher is available under a Creative Commons licence at https://github.com/albertoleva/PID4CSE, or by contacting the author via e-mail. Future work will consist in refining this material and the didactic approach as well, and addressing the envisaged subsequent activities. 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, experience sharing, and cooperation.

Acknowledgements

The problems addressed herein are delicate in more than one sense. In this difficult navigation the author received help and wise suggestions from colleagues of both communities, to whom he is indebted: V. Cortellessa, A. Filieri, C. Ghezzi, M. Maggio, A.V. Papadopoulos, F. Terraneo, and many others. Particular gratitude is due to computer engineering people, who helped by offering an encouraging and open-minded attitude. We all appreciated mutual constructive criticism and frank scientific dialectic as a means to achieve cultural convergence and to grow together, and I learnt a lot.

References

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

ACM curricula recommendations, http://www.acm.org/education/curricula-recommendations.

ACM curriculum guidelines for undergraduate degree programs in computer engineering, http://www.acm.org/binaries/content/assets/education/ce2016-final-report.pdf.

Aliane, N. (2010). Spreadsheet-based interactive modules for control education. *Computer Applications in Engineering Education*, *18*(1), 166–174.

De Lemos, R., Garlan, D., Ghezzi, C., Giese, H., Andersson, J., Litoiu, M., et al. (2017). Software engineering for self-adaptive systems: research challenges in the provision of assurances. In *Lecture notes in computer science*: *vol. 9640*, *Software engineering for self-adaptive systems III*.

Diao, Y., Hellerstein, J., Parekh, S., Griffith, R., Kaiser, G., & Phung, D. (2005). A control theory foundation for self-managing computing systems. *IEEE Journal on Selected Areas in Communications*, *23*(12), 2213–2222.

Guzmán, J., Åström, K., Dormido, S., Hägglund, T., & Piguet, Y. (2006). Interactive learning modules for PID control. *IFAC Proceedings Volumes*, *39*(6), 7–12.

Guzmán, J., Costa-Castello, R., Dormido, S., & Berenguel, M. (2016). An interactivity-based methodology to support control education: how to teach and learn using simple interactive tools [lecture notes]. *IEEE Control Systems*, *36*(1), 63–76.

Hägglund, T. (2012). Signal filtering in PID control. In *Proc. 2nd IFAC conference on advances in PID control* (pp. 1–10). Brescia, Italy.

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

Heo, J., & Abdelzaher, T. (2009). Adaptguard: guarding adaptive systems from instability. In *Proc. 6th international conference on autonomic computing* (pp. 77–86). Barcelona, Spain.

Hoyo, A., Guzmán, J., Moreno, J., & Berenguel, M. (2015). Teaching control engineering concepts using open source tools on a raspberry pi board. *IFAC-PapersOnLine*, *48*(29), 99–104.

Huebscher, M., & McCann, J. (2008). A survey of autonomic computing – degrees, models, and applications. *ACM Computing Surveys*, *40*(3), 7:1–7:28.

IBM (2005). An architectural blueprint for autonomic computing, IBM White Paper.

IEEE Computer Society curricula recommendations, https://www.computer.org/web/peb/curricula.

Intel Corporation, Linux thermal daemon, https://01.org/linux-thermal-daemon.

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

Johansson, M., Gafvert, M., & Astrom, K. (1998). Interactive tools for education in automatic control. *IEEE Control Systems*, *18*(3), 33–40.

Leva, A. (2013). Teaching PID control to computer engineers: a step to fill a cultural gap. In *Proc. 3rd IFAC conference on advances in PID control*. Gent, Belgium.

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

Leva, A., Terraneo, F., Giacomello, I., & Fornaciari, W. (2018). Event-based power/performance-aware thermal management for high-density microprocessors. *IEEE Transactions on Control Systems Technology*, *26*(2), 535–550.

Lozi, J., Lepers, B., Funston, J., Gaud, F., Quéma, V., & Fedorova, A. (2016). The linux scheduler: a decade of wasted cores. In *Proc. 11th European conference on computer systems* (pp. 1–16). London, UK.

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

Mansour, M., & Schaufelberger, W. (1989). Software and laboratory experiments using computers in control education. *IEEE Control Systems Magazine*, *9*(3), 19–24.

Maxima, A Computer Algebra System, http://maxima.sourceforge.net/documentation.html.

Patikirikorala, T., Colman, A., Han, J., & 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* (pp. 33–42). Zürich, Switzerland.

Rossiter, J., Pasik-Duncan, B., Dormido, S., Vlacic, L., Jones, B., & Murray, R. (2018). A survey of good practice in control education. *European Journal of Engineering Education*, *43*(6), 801–823.

The Mathworks, MATLAB home page, https://www.mathworks.com/products/matlab.html.

The Modelica Association, Modelica home page, https://www.modelica.org.

The OpenModelica Consortium, OpenModelica home page, https://openmodelica.org.

The Scilab Consortium, Scilab home page, http://www.scilab.org.

wxMaxima, A Graphical Frontend to Maxima, https://sourceforge.net/projects/wxmaxima.