**First source:**

2 From Mathematical Descriptions to Programs  
Programs implementing control systems differ from their corresponding  
discrete-time recurrence equations in several aspects, the first of which is  
not particular to control systems but is concerned with different levels of ab-  
straction in which algorithms can be described. For instance, algorithms for  
searching directed graphs may be defined in terms of the abstract structure of  
the graph, the mathematical G = (V, E), without paying attention to the way  
the graph is stored in memory. An abstract algorithm may contain statements  
such as “for each successor of a vertex v do” without being explicit about the  
way the successors of a node are retrieved from the data structure representing  
the graph. More concrete programs, written in languages such as C, need to  
specify these details. Between these levels and the actual physical realization  
there are many intermediate levels (assembly and machine code, micro code,  
architecture, etc.) and one of the great achievements of computer science and  
engineering is that most of the transformations between these levels are done  
automatically using computer programs.  
As an illustrative example we consider one of the most popular forms of  
control, the PID controller, and see how it is transformed into a program.  
An important feature of feedback functions is that they are typically dynam-  
ical systems by themselves, admitting a state which influences their output  
and future behavior. Fig. 2 shows the Simulink diagram of a typical sampled-  
data PID controller. The annotation of the Simulink blocks is written in the  
z-transform formalism, which is a discrete version of a frequency-domain rep-  
resentation of systems, where delay and memory are expressed using the 1/z  
operator. An explanation of this formalism can be found elsewhere in the  
handbook, and we focus here on a more “mechanical” state-space description  
of the controller. What a PID controller essentially does is to take the input  
signal I, compute its derivative D and integral S and then compute the output  
O as some linear combination of I, S and D. The state variables of the system  
include the integral S and the previous value of the input J, which is needed  
for computing the derivative. The following system of recurrence equations  
defines the semantics of the controller as a set On of output sequences whose  
relation with the input sequence In is defined by  
S−1 = I−1 = 0.0  
Sn = Sn−1 + 0.1 · In  
On = 5.8 · In + 4 · Sn + 3.8 · 10.0 · (In − In−1)  
(1)  
The first line defines the initial values of state variable S and the second line  
defines its subsequent value for every n ≥ 0. The last line determines the  
output, using In − In−1 as the derivative. Since old values of the input are not  
typically kept in memory, we will need to store this information in an auxiliary  
state variable J satisfying Jn = In, and replacing In−1 in the definition of On  
by Jn−1.

6 P. Caspi and O. Maler1  
Out1  
0.1z  
z−1  
Integrator  
5.8  
Gain2  
3.8  
Gain1  
4  
Gain  
z−1  
0.1z  
Derivative  
1  
In1  
Fig. 2. A PID controller represented by a Simulink block diagram  
Before showing the corresponding program, let us note that since (1) in-  
volves memory that has to be maintained and propagated between successive  
invocations of the program, the corresponding programming construct is bet-  
ter viewed as a class in an object-oriented language such as C++ or Java.  
However, since this point of view is probably not so familiar to most readers,  
we will realize it as a C program with global variables. These variables con-  
tinue to exist between successive invocations of the program (like latches in  
sequential digital circuits when the clock signal is low). The program shown  
in Table 1 is a result of a rather straightforward transformation of (1).  
/\* memories \*/  
float S = 0.0, J = 0.0;  
void dispid cycle (){  
float I,O;  
float J 1,S 1;  
I = Input();  
J 1 = I;  
S 1 = S + 0.1 \* I \* 4.0;  
O = I \* 5.8 + S 1 + 10.0 \* 3.8 \* (I-J);  
J = J 1;  
S = S 1;  
Output(O);  
}  
Table 1. A program realizing a PID controller

From Control Loops to Real-Time Programs 7  
The first part of the program is the declaration and initialization of the  
global variables J and S. The second part, the dispid cycle procedure, de-  
scribes the computation to be performed at each invocation of the program. It  
uses auxiliary variables J 1 and S 1 into which the new state is computed. The  
procedure presupposes two auxiliary functions Input and Output provided by  
the execution platform, which take care of bringing (digitized) sensor inputs  
into I and writing O onto the actuators. The implementation details of these  
functions are outside the scope of this article. The computational part of the  
procedure consists of taking the input and propagating it through a network  
of computations to produce the output. We first compute the next values of  
the state variables, then compute the output, write the new state values into  
the global variables and finally write the output and exit.  
Upon closer inspection one can see that we do not really need the auxiliary  
variable S 1 because only the new value of S is used while computing O. Con-  
sequently, we can replace the computation of S 1 by direct computation of S,  
use S in the computation of O and discard the assignment statement S = S 1.  
In fact, we can do similar things with J, by putting the statement J=I after  
the computation of the output, to obtain the optimized program in Table 2.  
/\* memories \*/  
float S = 0.0, J = 0.0;  
void dispid cycle (){  
float I,O;  
I = Input();  
S = S +0.1 \* I \* 4.0;  
O = I \* 5.8 + S + 10.0 \* 3.8 \* (I-J);  
J = I;  
Output(O);  
}  
Table 2. An optimized program for the PID controller  
Saving two variables and two assignment statements is not much, but  
for complex control systems that should run on cheap micro controllers, the  
accumulated effect of such savings can be significant.  
The reader can easily appreciate that the process of writing, modifying and  
optimizing such programs manually is error prone and that it would be much  
safer to derive it automatically from the high-level Simulink model. We have  
derived a program similar to the program in Table 2 from the Simulink model  
of Figure 2 in two steps. First, the Simulink-to-Lustre translator [6] was used to  
transform the model into a program in Lustre, a language [11] which provides

8 P. Caspi and O. Maler  
rigorous syntax and semantics for expressing data-flow equations such as (1).  
Then the Reluc Lustre-to-C code generator [9] produced the program after  
automatic analysis of state variables, dependencies and other optimizations.  
The story does not end with the generation of machine code by the C com-  
piler, as there are some additional conditions associated with the execution  
platform that need to be met. To begin with, the platform should support the  
I/O functions and be properly connected to all the machinery for conversion  
between digital and analog data. Second, the proper functioning of the pro-  
gram depends crucially on its being invoked every T time units, where T is the  
sampling period of the discrete-time system according to which the parame-  
ters of the PID controller were derived. Not adhering to this sampling period  
may result in a strong deviation of the program behavior from the intended  
one. This is a very particular (and rather unexpected) class of software errors  
inherent in control applications.  
To ensure the correct periodic activation of the program we need access  
to a real-time clock that will trigger the execution every T time units. But  
this is not enough due to yet another important difference between an ab-  
stract mathematical function and a program that computes it: the former is  
timeless while the latter takes some time to compute. For a program such  
as dispid cycle to function, the condition C < T should hold, where C is  
its worst-case execution time (WCET). If this requirement is not met, the  
program will not terminate before its next invocation (see the timing diagram  
in Fig. 3). Measuring and estimating the WCET of a program on a given  
architecture is not an easy task, especially for modern processors, and it is  
subject to extensive ongoing research [25].  
Read Compute Idle Read Compute  
C  
T  
. . .IdleWriteWrite  
Fig. 3. The execution of a control program with a period T  
Once these conditions are fulfilled, several implementation techniques can  
be used. Historically, such controllers were first implemented on a bare ma-  
chine, without using any operating system (OS). The real-time clock acts as  
an interrupt that transfers control to the program. If the scheduling condition  
C < T is satisfied, this interrupt occurs after the program has terminated and  
the computer is idle. Hence, unlike preemptive scheduling, there is no need for  
context switching and complex OS services. This implementation technique  
is thus both simple and safe and does not need to rely on a complex piece

From Control Loops to Real-Time Programs 9  
of software like an OS, which is difficult to validate. Much progress has been  
made in real-time OS (RTOS) technology, and today commercial systems are  
available that have been exercised and debugged by a large number of users  
and can be considered quite safe. Hence the role of monitoring the real-time  
clock and dispatching the program for execution can be delegated to an OS.  
This concludes the discussion on the implementation of simple control  
programs where we have tried to touch upon the key relevant computational  
aspects. In the next section we focus on the timing-related aspects of imple-  
menting more complex control loops.  
3 Complex Periodic Controllers  
In many control applications, systems have several degrees of freedom that  
must be controlled simultaneously. Mathematically each controller ci is just  
another recurrence equation that coexists with the other equations. Compu-  
tationally, these loops should be realized on a sequential computer that can  
do one thing at a time. The problem of how to “sequentialize” and schedule  
these parallel processes is one of the major topics in real-time systems. It is  
important that each invocation of a controller will have its relevant inputs  
ready before it starts executing and that the computation of all its outputs  
and their transmission to the outside world terminate in due time. This is the  
basic functional requirement from real-time control software, a fact sometimes  
obscured by details of operating systems and scheduling policies.

Reference:

Paul Caspi and Oded Maler. From Control Loops to Real-Time Programs. Verimag-CNRS, France.[ <http://arpont.imag.fr/~maler/Papers/caspimaler.pdf>]

**Second source:**

# 30 янв., 12.56​.aac

**Speaker1:** [00:00:00] Yes. So it's going to be real time, right?

**Speaker2:** [00:00:06] Because it's typically real time. So you can imagine a robotic arm. It's moving at some velocity and it needs to know how much it's going to overshoot. It can compute that based on the current velocity, but it's not until it observes each error component that it can work out how much to slow down by it. So it's a real time component.

**Speaker1:** [00:00:22] Mainly because why am I? Raspberry Pi and PID regulator in Python are just not well. Understand how it works.

**Speaker2:** [00:00:34] Yes, a python isn't really good a language for this . I would I would say real time.

**Speaker1:** [00:00:38] It's not really real time controller.

**Speaker2:** [00:00:42] No. Right. Mean. Well, but the thing is a Raspberry Pi has an operating system on it. Right. And that's not typically real time. Things can happen. You know, you can get an interrupt to service something, but the Raspberry Pi is not the best solution for a real time. But you can use it because a tremendously fast if it's faster than what you sort of sampling or calculating, then yeah, it's ideal for that. But but Python is not really the fastest or best language, I would say. I would say if you want to avoid an interpreted language or something that's just in time compiled, you want, you want deterministic nature, something like that.

**Speaker1:** [00:01:21] Yeah. Interpret language. Give them time. So you're talking about the real time system?

**Speaker2:** [00:01:27] Yeah.

**Speaker2:** [00:01:33] Yeah. Yeah, well, the thing is, we've just been compiler. You can have things that make, you know, on the fly optimization. So if it can see your for loop is going to last for 10,000 cycles or 10,000 iterations and nothing's going to change there just in time. Compiler can optimize quite a lot and say, Oh, you're not changing during that time. We can just not doing the full loop. But in your mind, if it was to do something like delay for X amount of time and it's just remove that, then you're going to have trouble. So you want the deterministic compiled language for something like a PID controller.

**Speaker1:** [00:02:06] So ideally should be the compile language and real time.

**Speaker2:** [00:02:11] Yes. Ideally, yes. So what project is it you want?

**Speaker1:** [00:02:14] It's like my extra project that I'm doing for fun at Drone.

**Speaker2:** [00:02:19] Okay, awesome.

**Speaker1:** [00:02:20] So it's actually not really a drone. It's actually a platform where the humans can take our controller. So maybe you maybe you care about butterflies or. It's wrong. But the thing is, did you buy the butterfly butterfly controller? And this.

**Speaker2:** [00:02:37] Is you want to make a fortune for.

**Speaker1:** [00:02:40] It. So if you need something else to. Can't do that. You can.

**Speaker2:** [00:02:47] Use. So you want enough speed to sample error times. So your pitch and your write, your role and azimuth. You need enough computational speed to calculate by how much you need to correct. So something in the order of, say, 100 megahertz or higher would be ideal for that. Now, if you imagine your Raspberry Pi, it's got processes in the background that are happening sort of asynchronously, right? And that's not ideal because if one of the threads then suddenly pipes up and it's like, I want to, I want all the CPU resources, then you're controlling the controls, the the correction for the angle momentum is then going to suffer. It's not going to have enough time. So it's then probably going to crash. So we use things like DSP or digital signal processor for real time stuff.

**Speaker1:** [00:03:39] Really good example. Yeah. I mean, I'm working through it.

Reference: Nathan Barbon. What OS type for robust system (robot and drones). 30/01/2023.

**Third source**

After the experiment is completed, the 63.2% method (section 3.3.1.1) is  
implemented. The implementation starts by calculating time delay L. To calculate the  
time delay, variance of the last 5 measured temperatures is used. When a significant  
change on the measured outputs is identified, the time when this change occurs is  
saved. The difference between the time found and the time the step input was  
applied is the time delay. Figure 4.19 shows the system\_response() method and the  
code that uses it to estimate time delay L.

4.3.5 Other practical considerations for PID controller  
In addition to the procedure described in the previous sections, there are also other  
minor details that should be considered while implementing a PID controller. Such  
considerations include the units that should be used, the duration of the time  
sampling period and the integrator windup problem.  
The integrator windup problem is probably the most important of these details  
as if it is not taken under consideration it could vastly increase the oscillations in the  
system control procedure. Integrator windup is a problem that commonly occurs  
when the Ki gain value is considerably large compared to Kp and Kd and the control  
action that needs to be taken exceeds the physical capabilities of the system. As  
described in the previous section the I term of the PID controller is the result of the  
multiplication of the Ki value with the sum of the errors occurred up to that point  
(the Integrator). While the process output is lower than the set point, the error is  
positive and is added to the Integrator (the sum of all past errors). If the process  
output remains lower than the set point for a considerable amount of time the  
Integrator grows larger and larger up to the point that the I term dominates the P  
and D terms and the controller output reaches the maximum or minimum point, e.g.  
100% or 0%. The integrator will continue to grow until the set point has been  
reached even though there is no point of further increasing the integrator as the  
maximum or minimum physical limit has already been reached. Until the time that

66 | Page  
the set point is reached the Integrator will be large and this will result to overshoot  
or response delay as the output of the controller will not reduce to the rate it should.  
After the overshoot or the delay the Integrator will start to wind down as it  
accumulates negative error which will result to the restoration of the Integrator to a  
value close to zero. However, this procedure will take a considerable amount of time  
and the set point has already been passed. This problem is a result of the inability of  
the controller to consider the physical limitations of the system. For instance, if the  
controller calculates that the required output to reach the set point is 2000%, the  
output to the heater will be limited to 100%, as the heaters cannot be powered by  
more than full power. Figure 4.26 shows an example to better illustrate the effect  
and importance of the integrator windup problem.  
In the example, the controller output is limited by the physical system characteristics  
(shown in the bottom graph) and as a result a considerable amount of positive error  
is accumulated in the Integrator (shown in the top graph). The operator using the  
controller sees that the process output will never reach the set point, due to the  
limitation of the controller output, and decides to reduce the set point. When the  
change in the set point occurs there is a significant delay until the controller  
responds by lowering its output. This happens because the integrator already has a  
large positive value that dominates the controller output and it takes some time for  
the negative error to wind down the integrator. The controller responds when  
adequate negative error is summed to the integrator to cancel out the positive error.  
Figure 4.26. Integrator windup effect. Top graph  
shows the process output and bottom graph  
shows controller output, redrawn from ref

67 | Page  
There are several solutions to solve this problem. In this case, clamping of the  
controller output is used as the solution. Using this method, if the controller output  
is larger or smaller than then physical limit, e.g. 100% or 0%, the controller output is  
simply set to that limit. In addition, when the controller output is set to the  
maximum or minimum limit, the integrator stops accumulating any more error. The  
integrator restarts accumulating error when the value drops again between the  
physical range or when the sign of the error is changed (the set point has been  
passed). This way the response of the controller is faster when needed. Figure 4.27  
shows the implementation of anti-windup method.if PID > 100:  
PID = 100  
if self.set\_point > current\_value:  
self.Integrator\_flag = False  
else:  
self.Integrator\_flag = True  
elif PID < 0:  
PID = 0  
if self.set\_point < current\_value:  
self.Integrator\_flag = False  
else:  
self.Integrator\_flag = True  
else:  
self.Integrator\_flag = True  
PID is the controller output value, which is clamped between 0 and 100.  
Other considerations include the time period between the controller output and the  
measurement of the process output (e.g. the temperature). This period is called the  
sampling time or period of the system. The rule of thumb is that the sampling time  
should be equal or smaller than the 1/10th of the time delay T. So for example if  
T is equal to 30 the sampling time should be 3 seconds or less. Generally, at least 10  
measurements should be taken every T seconds.  
Another minor but important issue is the units used for the controller input, the  
controller output, the set point and the process output. The process output and the  
set point should be expressed in the same units, as their difference will produce the  
error value. For example, the process output and the set point can be in degrees  
Celsius. To produce controller output that corresponds to the 0-100% range it is also  
important that the process input used in the system identification experiments is  
also in percentage and also in the same 0-100% range.

Reference:

FEIDIAS IOANNIDIS. INTELLIGENT CONTROLLER BASED ON  
RASPBERRY PI. University of Manchester, SCHOOL OF COMPUTER SCIENCE. 2014. [<https://studentnet.cs.manchester.ac.uk/resources/library/thesis_abstracts/MSc14/FullText/Ioannidis-Feidias-fulltext.pdf>].